Sapienza
Università di Roma

ASiMOV: Microservices-Based Verifiable Control Logic with Estimable Detection Delay against Cyber-Attacks to Cyber-Physical Systems

PhD School in Computer Science

Dottorato di Ricerca in Computer Science – XXXII Ciclo

Candidate

Gabriele Gualandi
ID number 802464

Thesis Advisor

Prof. Luigi Vincenzo Mancini

Co-Advisor

Prof. Emiliano Casalicchio

April 2020

Thesis defended on 28 Feb 2020
in front of a Board of Examiners composed by:

Prof. Andrea Torsello (chairman)

Prof. Francesco Lo Presti

Prof. Raffaele Montella

ASiMOV: Microservices-Based Verifiable Control Logic with Estimable Detection Delay against Cyber-Attacks to Cyber-Physical Systems

Ph.D. thesis. Sapienza – University of Rome

© 2019 Gabriele Gualandi. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: April 20, 2020

Author's email: gabriele.gualandi@gmail.com

Abstract

The automatic control in Cyber-Physical-Systems brings advantages but also increased risks due to cyber-attacks. This Ph.D. thesis proposes a novel reference architecture for distributed control applications increasing the security against cyber-attacks to the control logic. The core idea is to replicate each instance of a control application and to detect attacks by verifying their outputs. The verification logic disposes of an exact model of the control logic, although the two logics are decoupled on two different devices. The verification is asynchronous to the feedback control loop, to avoid the introduction of a delay between the controller(s) and system(s). The time required to detect a successful attack is analytically estimable, which enables control-theoretical techniques to prevent damage by appropriate planning decisions. The proposed architecture for a controller and an Intrusion Detection System is composed of event-driven autonomous components (microservices), which can be deployed as separate Virtual Machines (e.g., containers) on cloud platforms. Under the proposed architecture, orchestration techniques enable a dynamic re-deployment acting as a mitigation or prevention mechanism defined at the level of the computer architecture. The proposal, which we call ASiMOV (Asynchronous Modular Verification), is based on a model that separates the state of a controller from the state of its execution environment. We provide details of the model and a microservices implementation. Through the analysis of the delay introduced in both the control loop and the detection of attacks, we provide guidelines to determine which control systems are suitable for adopting ASiMOV. Simulations show the behavior of ASiMOV both in the absence and in the presence of cyber-attacks.

Acknowledgments

To my family, friends and colleagues.

I express gratitude to my co-workers and supervisors, Prof. Emiliano Casalicchio, Emanuele Gabrielli, and Prof. Luigi Vincenzo Mancini. Prof. Giuseppe Accascina gave me some great little suggestions. The two external reviewers Professors Mauro Caporuscio and Alessandro Vittorio Papadopoulos have given me extremely timely and valuable suggestions.

Contents

1 Introduction **1**
1.1 Cyber-Physical Systems 3
1.2 Security of Cyber-Physical Systems 4
1.2.1 Classification of IDS for CPS 4
1.2.2 Classification of Attacks to CPS 7
1.2.3 Control-Theoretic IDS (CT-IDS) 7
1.2.4 Detection delay in control-theoretic techniques 8
1.3 Research problem and contributions 10
1.4 Thesis organization 12

2 Background **15**
2.1 Feedback Control Systems 15
2.2 Computing Background 17
2.2.1 Virtualization and Cloud Platforms 17
2.2.2 Microservices oriented architectures 19
2.2.3 Event Sourcing Pattern 21
2.2.4 Security of microservices oriented architectures 22
2.2.5 Automated deployment tools 23

3 ASiMOV **27**
3.1 Detection requirements 27
3.2 Introduction to ASiMOV 31
3.3 ASiMOV Deployment model 32
3.3.1 Deployment model for a CPS 32
3.3.2 The deployment process for a model 33
3.4 Considered attacks 35
3.4.1 Security assumptions in ASiMOV 35
3.4.2 Model of the Attacker 36

4 Model and Architecture **39**
4.1 Model of ASiMOV 39
4.1.1 Motivations for the replication of the control logic 40
4.1.2 Model of a verifiable Control Application 41
4.1.3 State of the Controller 44
4.2 Architecture of ASiMOV 46
4.2.1 Control Application 46

4.3 Detection of Cyber-attacks 47
4.3.1 Compare task 48
4.3.2 Match task 50
4.4 Mitigation and Prevention of attacks 51
4.4.1 Controller state migration 52

5 Implementation 55
5.1 Implementation of ASiMOV 55
5.1.1 Implementation of the CA and IDS models 56
5.1.2 Microservices Choreography 57
5.1.3 Microcyle routine 60
5.2 Mitigation and prevention of cyber-attacks 62
5.2.1 Controller state migration 62
5.3 Dynamics of an attacked System 66
5.4 Deployment-based Mitigation and Prevention 67
5.4.1 Deployment Manager 68

6 Performances analysis 75
6.1 Effects of delay on control systems 75
6.2 Delays introduced by ASiMOV 77
6.2.1 Delay model 78
6.2.2 Approximations 81
6.2.3 Performances when CA is at rest 82
6.2.4 Performances when CA is not at rest 84
6.2.5 Simulation results 84

7 Related works 89
7.1 A reference ICS scenario: the smart factory 89
7.2 Enabling technologies and security of the smart factory 90
7.2.1 Enabling technologies for the smart factory 91
7.2.2 Intrusion detection, prevention and mitigation 92

8 Conclusions and future works 95

9 Annex: Pseudo-code 99

Chapter 1

Introduction

This thesis focuses on the **cyber-security** of **Cyber-Physical Systems**. The term Cyber-Physical System (**CPS**) is a generic way of referring to an integration of computation, networking, and physical processes. Typically, in a CPS there are physical systems that are remotely controlled, resulting in a distributed feedback control system.

CPS can be found in critical infrastructure control (e.g., electric power, water resources), distributed robotics (telepresence, telemedicine, automated manufacturing), healthcare systems, assisted living, environmental control, traffic control and safety, advanced automotive systems, unmanned vehicles, and others. Depending on the specific context, the literature also refers to CPS as Power Networks, Industrial Control Systems, Mass Transport Networks, Networked Control System, Sensor Actuator Network, Wireless Industrial Sensor Network, or SCADA (Supervisory Control And Data Acquisition). In some cases, both literature and common language refer to CPS as to "Smart" or "IoT" scenarios (e.g., Smart Manufacturing or Industry 4.0, Smart Cities, Smart Grids).

CPS finds application in many critical aspects of human life. Hence their **security** is a crucial aspect for all governments. In 2010 Stuxnet[29] damaged as many as one-fifth of the nuclear centrifuges in Iran, making aware the community of the fact that attacks to control systems can cause substantial damage. In the successive years, twenty or so cases followed[38]. In 2014 Havex/Dragonfly successfully attacked Power Networks. This kind of attack has the potential to disable primary services in large geographical areas. Other attacks involved Industrial Control Systems [108], with the potential of impairing the products' quality and therefore damaging the customers and the reputation of the manufacturer. According to Kaspersky Lab, 41.2% of factories were attacked by malicious software at least once in the first half of 2018. For these reasons, there is an urgent need for new approaches that improve the resiliency of CPS to cyber-attacks.

Ideally, all the devices of a CPS are protected by the strongest security measures possible. In the real world, to prevent attacks to all the devices of a CPS may not be feasible. Devices may have limited computational resources, maybe physically exposed, or could be tampered by malicious insiders.
A cyber-attack that is not *prevented*, become a *successful attack*. Intrusion Detection

Systems (**IDS**) are techniques for the detection of successful attacks. The field of CPS security [43] is a multidisciplinary area, and involves sciences oriented to both the "cyber" (e.g., Information Security, Computer Science) and "physical" (e.g., Control Theory) processes of a CPS. Independently on the disciplinary area, an IDS works according to the concept of *anomaly*. An anomaly can be defined in different ways, e.g., the behavior of an application (for the "cyber") part, or the violation of a physical constraint. After an attack is detected, its effects should be actively *mitigated* as soon as possible. The different disciplinary area takes different approaches, e.g., an intervention may concern a component at any layer of computer architecture, or appropriate management of physical systems realized trough control action.

Among the “cyber” processes of a CPS, the most relevant one is that carried out by the so-called **control logic**. A successful attack on the control logic (we call Internal attack) gives control over the “physical” processes of a CPS - therefore, it can deliver damage. The so-called **external attacks** target sensors and actuators in the attempt to deceive the control logic, and they may be detected by considering physical constraints. For this type of attack, there exist IDS which are based on Control Theory (CT-IDS). CT-IDS employs knowledge of physical laws. Therefore they can detect a broader spectrum of attacks then IDS based on Information Security or Computer Science, e.g., a CT-IDS working at the level of the dynamics of a physical system may detect both Internal and External attacks - since the dynamics of a control system deviates in both case. However, since the typical CT-IDS employs a measure of anomaly that belongs to a metric space, to employ a CT-IDS against Internal attacks introduces an unnecessary uncertainty. Tampering of an execution environment should be exactly determinable.

CT-IDS are typically formulated as an abstract model in which the I/O of the control logic (sensing and actuation) gets intercepted and analyzed. This thesis investigates what happens when a CT-IDS is implemented and deployed. There are discussed the following possibilities.

In case the CT-IDS is part of the control logic (i.e., it is host-based), it disposes of exact knowledge about the state of the control logic. Therefore, the only uncertainty in the detection process comes from the state of the controlled system. However, being host-based, the CT-IDS is not reliable against unrestricted Internal attacks (i.e., the device executing the control logic and the CT-IDS cannot be trusted).

In case the CT-IDS is network-based (i.e., it is in a different device then the control logic), there is increased security against unrestricted attacks. However, without proper coordination between the CT-IDS and the control logic, the CT-IDS can only estimate the state of the control logic - hence detection accuracy is reduced compared to the host-based case.

In our proposal, we investigate the possibility of gaining the benefits of the two scenarios (i.e., host-based and network-based). A CT-IDS can be part of the control logic (host-based), but the latter gets replicated and verified into a different host (network-based). In order to make the proposed verification process useful, special care is taken not to interfere (i.e., bottleneck) with the control process.

Optimization-based techniques enable active control strategies that can reduce, or even prevent, damage caused by attacks. If a control logic also implements active defense strategies, having an estimate of the (maximum) time needed to detect an attack (*detection delay*) is an advantage, because it opens up the possibility of interfering as little as possible with the control process, i.e., only as much as is necessary.

This thesis proposes a software-engineering oriented solution for increasing the resiliency of CPS to cyber-attacks. In particular, we propose a microservice architecture to implement an event-based controller which outputs are *verified* through redundancy. That is, we revisit the classic modular redundancy to be employed in distributed control schemes. The introduced technique is named Asynchronous Modular Verification (ASiMOV), and realizes both a controller and an IDS, the latter having: ideal detection accuracy (no false negatives or positives) against Internal attacks, and an estimable detection delay. We propose a model describing the state of the controller at the level of an execution environment. The model is implemented using an *event sourcing* pattern to allow the reconstruction of the state of a controller from a history of events. The latter feature is exploited to implement a *mitigation* and *prevention* strategy based on virtualization technologies.

The proposed solution is an approach of Information Security and Software Engineering towards the security of control systems. Attention is paid to the requirements of control systems, i.e., the need to employ state estimators, and the need to avoid that the verification process increases the feedback-loop delay.

1.1 Cyber-Physical Systems

Cyber-Physical Systems are made of different enabling technologies, which generate an autonomous, intercommunicating and intelligent system and, therefore, are able to integrate different and physically distant entities. In a CPS there are the following processes:

- **Physical processes**: are carried out by physical systems which are part of feedback control systems.
- **Computation processes**: are carried out by so-called Control Applications, having the goal of controlling the physical systems. The Control Applications, together with the physical systems, form feedback control systems. The computation processes require to collect, store and analyze histories of data from/to the physical systems (i.e., actuation/sensing). The computation processes may require to consider the notion of physical time in which data from (to) systems is collected (produced), and to realize real-time control i.e., the timing in which actuation are produced for the physical system may be critical.
- **Communication processes**: there are communication processes enabling communication between Control Application(s) and physical system(s).

The computation processes of a CPS can be divided into real-time and offline. The real-time processes realize direct control of systems employing techniques from

control system engineering (e.g., time and frequency domain methods, state space analysis, filtering, prediction, optimization, robust control). Offline processes (sometimes referred as to big-data analysis) involve aggregation and analysis of collected data to support the human decision-making process and improve the direct control processes e.g., define optimal strategic business decisions, to improve direct control by refining the model employed, bug fixing. Techniques employed in offline process can include system identification, data mining, neural networks.

1.2 Security of Cyber-Physical Systems

Cyber-Physical Systems (CPS) are modeled using the fundamentals principle of feedback loop from traditional control theory, i.e., there are *controllers* producing *actuations* signals upon receiving *sensing* signals, with the goal of directing the state of *systems* toward a desired state.

A CPS is more complex than a traditional control system. The “cyber” processes part of a CPS, containing the control logic, are complicated in their theoretical definition. A CPS contains many physical systems, possibly of a different kind, and distributed in a large geographic area. Therefore there is the need to adopt distributed control schemes, and to deal with uncertainties arising both in the time domain and in the utilized models.

Additional aspects of complexity of a CPS are related to the management of the controllers, which are the applications executing the control logic. Controllers were originally developed as ad-hoc hardware devices attached to the physical systems to be controlled. In a CPS controllers are complex applications, possibly subjected to the practice of software evolution, which instances have to be managed automatically or semi-automatically. To this end, different technological solutions are required for the execution environments supporting a controller. Different layers of computer architecture (e.g., Operating System) are involved in the management of instances of so-called **Control Applications**. An automated deployment, possibly based on virtualization, may be required for a fault-resilient, evolvable and scalable control infrastructure.

The described complexity introduces new challenges in the protection of a CPS from cyber-attacks, as its surface of attack is enlarged compared to traditional control systems. The first line of defense against cyber-attacks is implemented by solutions from Information Security (e.g., authentication) aiming to reduce drastically the probability that an attack becomes successful. However, there is always the possibility that an attack becomes successful. This may happen trough newly discovered vulnerabilities (zero-days) affecting any of the architectural layers of the devices in a CPS, disclosed administrative credentials, or from the activities of human insiders.

1.2.1 Classification of IDS for CPS

A successful attack modifies, or injects, data on the *devices* of a CPS, i.e., networked devices realizing controllers, sensors and actuators. IDS are applications which

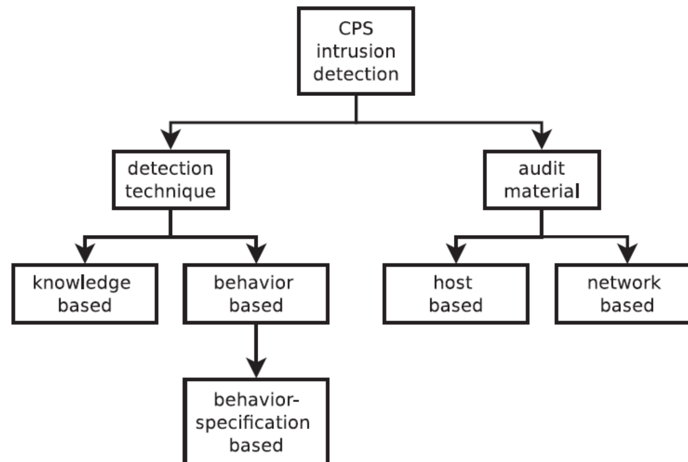


Figure 1.1. Classification of CPS-IDS detection technique. Image taken from work[76].

seek for anomalies that may be interpreted as cyber-attacks. There are extensive literature reviews about generic IDS [51, 40, 10] and IDS specific for CPS[41, 71, 76].

CPS is an interdisciplinary field. Therefore there are several different techniques to realize an IDS. To provide a complete and coherent classification of all the possible CPS-IDS techniques is not an easy task. We adopt the classification of work [76] relatively to the existing class of CPS-IDS. In the cited work, the authors discuss the advantages and disadvantages of different approaches. The conclusions drawn are that the so-called *Behavior-Specification-Based* techniques, which we briefly describe in this Section, are the most effective for realizing CPS-IDS since they provide better accuracy, i.e., false-negative (FN) and false-positive (FP) ratios. Furthermore, the cited work underlines the importance of introducing new metrics for the performances of CPS-IDS, such as the detection delay.

Behavior-Specification-Based IDS for CPS

A CPS-IDS can be classified accordingly to two criteria (Figure 1.1): the *Audit Material*, defining how the IDS collects data before data analysis, and the *Detection Technique*, determining “what” is an anomaly.

- *Audit Material*: “how” data is collected
  - **Host-Based**: the IDS is executed by the same device to be monitored. Audit data may consist of any data regarding the activity of the device, e.g., OS system calls, application dataflow, or the content of the network traffic of the device.
  - **Network-Based**: the IDS is executed by a different device than the one to be monitored, and intercept and analyze the network traffic of the monitored device.
- *Detection Technique*: “what” is an anomaly
  - **Knowledge-Based**: the IDS looks for runtime features that match a specific pattern of misbehavior. By definition, these approaches only react to known bad behaviors.

- **Behaviour-Based**: the IDS looks for runtime features that are out of the ordinary. The ordinary could be defined with respect to the history of the test signal (unsupervised) or with respect to a collection of training data (semisupervised). This class of IDS includes, but is not limited to, Statistics, Machine Learning, Clustering, and Neural Network approaches. One particular sub-class of the Behaviour-Based IDS is the **Behavior-Specification-Based**. In this case, the IDS is unsupervised, contains a formal definition of legitimate behavior, and detects an intrusion when a system departs from this model.

Audit Material

Host-Based approach has the advantage of distributing the computational load over the devices of a CPS. Another advantage of this approach is the ability to access each of the internal aspects of a device (e.g., OS) for a more in-depth inspection. However, if a device is totally compromised (that is, captured by an attacker), the IDS functionalities may be compromised, and deliver only FN.

Network-Based approach is the dual approach of the Host-Based in terms of advantages and drawbacks. In particular, a network-based IDS could detect an attack even in case of the device executing the controller is totally compromised.

Detection technique

Knowledge-based approach is effective for unsophisticated attacks and known attacks, but ineffective for sophisticated attacks and zero-days. Therefore, given the complexity of a CPS, it is not effective.

Behaviour-Based approach eliminates the need to specify all possible attacks, hence may provide a good FN ratio, even against sophisticated attacks or zero-days.

- **Behaviour-Non-Specification-Based** approach is susceptible to FP as the models of attacks could be too general. Semi-supervised methods are susceptible to attacks during training phases. In some cases, the requirements of these techniques in terms of memory and computation are prohibitive.
- **Behavior-Specification-Based** approach has the potential to be the most effective technique for CPS intrusion detection[76]. The effort required by human experts in defining formally a legitimate behavior is the only disadvantage. Once implemented, there is not the need of any training phase, and can deliver good performance in terms of both FP and FN.

Detection delay

The performances of an IDS for an Information and Communication Technologies (ICT) system are typically expressed in terms of accuracy, i.e., false-negative/positive ration of a confusion matrix. Differently from the case of ICT systems, in CPS the values of a confusion matrix alone do not fully characterize the actual performances of a CPS-IDS. The **detection delay**[101], sometimes referred as detection latency, is the amount of time required to detect a successful attack. More specifically, the detection delay Υ^{Det} is the time interval starting when a tampered actuation is injected into a controlled system, ending when such malicious activity is detected. Υ^{Det} is a fundamental metric to quantify the effectiveness of a CPS-IDS since the ultimate goal of an attacker is to provoke (physical) damage. Intuitively, the amount of damage

provoked by an attack is proportional to the time in which the attack remains undetected. Any physical system requires some time to transit between two states. Therefore some time is always necessary for a system to reach a state causing damage.

The definition of new IDS performances metrics for CPS-IDS is still an open problem[76], and to the best of our knowledge, there are not CPS-IDS capable of estimating, or better guaranteeing, a detection delay. We underline that estimating the detection delay should not simply consist of averaging the time measured in a series of tests or simulations. Rather, the detection delay should be expressed as a function valid in the general case.

1.2.2 Classification of Attacks to CPS

In this work, *successful attacks* are modeled according to the compromise of devices. A device is a computer architecture realizing one of the fundamental components of a CPS, i.e., a controller, a sensor or an actuator.

Successful attacks

- **Internal attack**: the attack is originated from the device realizing a Controller. Therefore the Control Logic may produce tampered actuation signals.
- **External attack**: There is one, or multiple, of the following conditions:
  - the attack involves actuators devices or the network between the controller and the actuators. As a result, the actuators provide tampered quantities to the controlled system(s).
  - the attack involves sensors devices or the network between the sensors and the controller. As a result, the data from the sensor is tampered.

This work deals with **Internal attacks**, while External Attacks are kept into consideration for the proposal of an IDS compatible with correspondent solutions from control-theoretical methods. External attacks may be particularly dangerous since, in certain conditions, they can become not detectable (stealth attacks[85]). However, to provoke damage by means of an External attack is more difficult then with an Internal attack. Indeed, an External attack requires the knowledge of a model of the attacked System and its state, i.e., trivial attacks such as randomizing the values of control signals are commonly considered ineffective. Conversely, in an Internal attack the attacker can simply change the desired state, and let the controller cause the damage.

1.2.3 Control-Theoretic IDS (CT-IDS)

IDS based on control-theoretic methods define an *anomaly* at the level of the (physical) state of a feedback control system. We refer to these applications as **CT-IDS** to distinguish them from the conventional IDS based on Information Security, like the one proposed by this thesis. CT-IDS exploit knowledge of physical laws (e.g., conserved quantities or known constraints) hence they can detect attacks that have escaped to the detection of conventional IDS.

CT-IDS include methods to perform attack *identification*, i.e., to tell what control signals are causing an anomaly. Without these techniques, one could identify an attack but not tell what control signals are providing bad data. In fact, the same anomaly on the system's state or on the system's dynamics may be caused interchangeably by tampered actuation or sensing signals. One example of well established method for the *detection* and *identification* of attacks is based on the so-called time-invariant descriptor representation[84, 86]. In this formulation, there are so called *monitors*, which are mathematical functions utilizable to identify the particular control signals (i.e. actuation or sensing) provoking an anomaly. In this formulation, a successful attack is modeled as:

Control-theoretical IDS

- *State attack*: the state of the physical system is affected by means of bad actuation(s), i.e., the actuators are delivering quantities subjected to tampering;
- *Output attack*: sensing signal(s) are bad, i.e., they do not transport the physical quantities actually present in the system.

From this definition, it follows that monitors or similar techniques, do not explicitly distinguish between Internal and External attacks (Section 1.2.2). As an example, the identification of a specific set of compromised actuation signals could be originated interchangeably by an External attack (i.e., the actuation devices are tampered) or by an Internal attack (the control logic is tampered).

As already introduced at the beginning of Chapter 1, a concrete implementation of network-based CT-IDS could not dispose of an exact model of the control logic - thereby decreasing its accuracy.

To sum up, CT-IDS may detect and identify attacks based on the knowledge of physics laws and may tell what control signals are involved. However, they cannot distinguish between Internal and External attacks, and in case they are network-based, the synchronization between their state and that of the control logic should be considered.

1.2.4 Detection delay in control-theoretic techniques

The control logic may include solutions from control theory implementing run-time optimization-based techniques for mitigating attacks to a CPS. As an example, in [88], the Control Logic reacts accordingly to the detection of compromised sensors, minimizing the damage sustained by a power grid.

There are also ways, at least in principle, to *prevent* damage from attacks by means of control action. An intuitive example in this sense is given by the *system reachability*, which is a fundamental concept in Control Theory. Broadly speaking, the reachability of a system consists of all possible states that the system can reach, starting from a given state and disposing of a given time. In works [28, 78] the authors utilize reachability to assess the impact of different attacks to a CPS. By computing the reachability of a controlled system at run-time, the control logic could implement decisions that expose the system to lesser risks in the event of a future successful attack.

Detection delay and prevention

The idea that a control logic can mitigate or prevent the damage from attacks is based on general concepts from adaptivity and planning. These concepts, extensively

studied by the Control Theory and applied to dynamical systems, led to the definition of Autonomous Control Systems [6]. More recently, this kind of approach has laid the foundations for the field of Timed Automata [5], which deals with systems having both continuous and discrete state components (i.e., hybrid systems). The need for equipping computer systems with the self-managing and self-healing capabilities led to the Autonomic Computing [42], which deals with complex, distributed computer systems. The common denominator of solutions from these scientific fields is the ability of a control system to predict its future states, and thus to plan appropriate sequences of commands and controls.

Predicting future states within physical systems cannot be separated from the concept of physical time. We denote with **detection delay** the time required to detect a successful attack. An estimable detection delay is fundamental information in the protection of physical systems, as it enables to carry out effective prevention strategies.

For readers who are familiar with dynamical systems, the following example provides an intuitive way to visualize the importance of the detection delay in the protection of physical systems. Consider a robotic arm or vehicle starting from state x_1 with the task of reaching state x_2. Usually, in the phase of *trajectory planning*[30] the control logic of a robotic system is allowed to choose one of many trajectories in the state space leading to x_2 (e.g., a specific motion for an arm, or a path for a vehicle). The control logic will choose an optimal trajectory in the state space accordingly to a criteria, e.g., the one that leads to x_2 in minimum time. Tipically, the control logic is instructed with hard or soft constraints defined in the state space, i.e., to avoid the system passes through individual states along the trajectory (*forbidden* states). In robotics, this problem takes the name of *obstacle avoidance*, and the solution is a sub-optimal trajectory in the state space, which avoids passing trough forbidden states. In this example, the forbidden states are those providing a Euclidean distance with a value of 0 between any part of the robot and any obstacle.

Prevention using known detection delay

Consider the following scenario:

- an attacker has the purpose of causing a collision with an obstacle;
- the attacker can take direct control of the robot at any time;
- there is an IDS with ideal detection accuracy (i.e., no FP or FN) against any kind of attack;
- in case the IDS detect an attack, the robot is immediately stopped trough a fail-safe command.

Under these conditions, the control logic can prevent any collision, given that: all states belonging to the planned trajectory gives a sufficiently great Euclidean distance with any obstacle. In order to let the planner find such trajectory, the detection delay should be known (or estimated with sufficient precision). In particular, the trajectory to reach x_2 must be chosen such that, for each point of the trajectory, there not exist any trajectory in the state space of the robot reaching an obstacle in an amount of time smaller of Υ^{Det} (the detection delay).

There are not many literature works that use techniques from Control Theory, Timed Automata, or Autonomic Computing for the mitigation or prevention of cyber-attacks to CPS. Possibly, this is due to the complexity of CPS, which makes it difficult to estimate the actual performance of such techniques in the real world. However, this author believes that there will be extensive developments in this direction. As the complexity of the controller process will increase, the control logic will have to actively contribute to the mitigation and prevention of cyber-attacks. A complete examination of the aspects related to the autonomicity of control systems is outside the scope of this thesis. What is essential for the contributions of this thesis is that a control logic should be verifiable with an estimable Υ^{Det}.

1.3 Research problem and contributions

In this thesis, there are considered the following research questions relative to the detection, mitigation, and prevention capabilities of a security system for CPS:

- Detection of Internal attacks:
  - **RQ1**: how to detect Internal attacks with ideal accuracy even in case of a total compromise of the device executing the controller?
- Mitigation and prevention of Internal attacks:
  - **RQ2**: how to prevent and mitigate Internal attacks, at the level of the computer architecture of the device executing the control logic?
- Detection, Mitigation and prevention of External attacks:
  - **RQ3**: how to increase the level of security coming from control-theoretical IDS (CT-IDS), and optimization-based prevention and mitigation methods against External attacks?

The investigation of these questions led us to the proposal of ASiMOV (ASinchronous MOdular Verification), which is based on our previous works [35, 36].

ASiMOV is a self-protection mechanism [45, 44, 50] for industrial control applications capable of detecting and reacting to attacks that modify the outputs of the control logic. ASiMOV includes an IDS with ideal accuracy performances (no false positives or negatives) and response mechanisms that automatically restores a clean instance of the control logic. Our solution is inspired by Triple Modular Redundancy (TMR) [72] and realizes unsupervised behavior-specification-based detection [76].

ASiMOV enhances the state-of-the-art literature by proposing a self-protection mechanism that:

- does not require knowledge about attack signature, or training on historical attack traces or human-assisted decisions;
- has ideal accuracy in detecting tampering of the control logic;
- does not significantly increase the delay between the control application and a controlled system (negligible increase in the control loop delay);

- detects tampering in an estimable time (estimable detection delay);
- applies to any CPS that includes control applications and specifically to cloud-based industrial control systems like in works [...];
- enables a deployment-based mitigation and prevention management capable of reconstructing the state of control application into a different instance.

ASiMOV consists of a Control Application interfaced with a network-based behavior-specification-based IDS (from now referred to as ASiMOV IDS, or IDS). The proposed architecture for a Control Application can be used to implement distributed (i.e., modular), event-based schemes for the control of physical systems. The intrusion detection mechanism is inspired by the classic modular redundancy, in which a Control Application and its replica(s) are used to detect inconsistencies in their outputs. An implementation of the ASiMOV architecture is adapt to be deployed and orchestrated on virtual resources, which enable deployment-based mitigation and prevention strategies. To the best of our knowledge, the cited features have never appeared together in an CPS-IDS.

ASiMOV's answers to research question **RQ1** is to verify each of the outputs of the Control Application using a dedicated device (i.e., different from the device subjected to attacks). The proposed method realizes an IDS with ideal accuracy against Internal attacks even in case of total compromise of the device executing the Control Application.

The answers of ASiMOV to research question **RQ2** is to enable a deployment-based strategy based on virtualization technologies to mitigate or prevent Internal attacks. ASiMOV separates by design the state of the Control Application from its execution environment, i.e., it is possible to reconstruct an instance of the Control Application from a history of control signals after an attack is detected. The following strategies are employed:

- in case of attack, a fresh instance of Control Application can be provisioned with a *verified* state, allowing to continue in the control operations, hence to *mitigate* an attack;
- the state of a Control Application can be periodically transferred between different devices, to *prevent* Control Logic attacks from becoming Successful.

Answers to **RQ3** are investigated as follows.
Considering an IDS included into the control logic i.e., there is a CT-IDS sharing data-structure and system clock with the algorithms that are actually controlling a system. The control logic is executed on a device (*field device*), that is used for direct control and is subjected to attacks. Assuming the presence of a different device (i.e., *security device*), which has a reduced surface of attack (i.e., un-attackable). Assuming that by means of ASiMOV the security device has a perfect knowledge (i.e., replicates) the state of the control logic in the field device. Therefore, on the replica on the security device, the CT-IDS maintains the same accuracy then that of the field device. As a consequence, there is an increase in security of the detection

process (since the CT-IDS is un-attackable). In alternative, there is an increase in accuracy in the detection process, compared to the case in which the CT-IDS was installed in a security device without a mechanism like ASiMOV to manage its state synchronization.
In other words, ASiMOV answers to **RQ3** if the following scenario is true. Assuming a generic CT-IDS tested in a real-world scenario. The CT-IDS employs a dynamic model to estimate the state of the control logic, because this is required by the detection process. Defining a measure of error E in the detection process (e.g., vector norm of the distance of the estimated and actual state of the control logic). Then, there is a relation between the average norm of E and the detection accuracy, i.e., when E grows, the detection accuracy goes to zero.

The introduced features of ASiMOV are further developed in Section 3.1 "Requirements".

The second answer to **RQ3** comes from the fact that ASiMOV IDS has an estimable detection delay. Therefore, it enables optimization-based prevention techniques to make decisions that increase security.

The last answer to **RQ3** is given by the fact that the typical CT-IDS does not distinguish between Internal and External attacks (i.e., identification at the level of devices), but only between state and output attacks (i.e., identification at the level of control signals, Section 1.2.3). Assuming the security devices un-attackable, the accuracy performances of ASiMOV IDS is ideal against Internal attack. Therefore, an attack identified at the level of the control signals by a CT-IDS can be mapped, by exclusion, with an attack identified at the level of the devices (i.e., an Internal or External attack).

1.4 Thesis organization

Chapter 2 (Background) provides backgrounds for the concepts required in the rest of this thesis.

Chapter 3 (ASiMOV) introduces to the main characteristics of ASiMOV. The adopted deployment model and the model of an attacker are defined.

Chapter 7 (Related Works) discuss how ASiMOV improves the current state-of-the-art solutions for secure controllers in CPS, and in particular for the Smart Factory.

Chapter 4 (Model and architecture) proposes a formal model of the verifiable controller realizing ASiMOV, a component-based description of its architecture, and sufficient conditions for the verification of its outputs.

Chapter 5 (Implementation) details a micro-service implementation of ASiMOV, accompanied with pseudo-code. Additionally, the Chapter describes how virtualization technologies can be used to realize a deployment-based mitigation and prevention strategy.

Chapter 6 (Performances analysis) analyzes the performance of ASiMOV both in the absence and in the presence of attacks. The effects of an attack on a dynamical system are simulated. The delay introduced by ASiMOV in a feedback control loop, and the detection delay estimated analytically and through simulation. This Chapter also provides an analysis of how, in general, the introduction of a delay in a feedback loop may affect the performances of a simple control system.

Chapter 8 (Conclusions) draw the conclusion and future developments of the topics of this thesis.

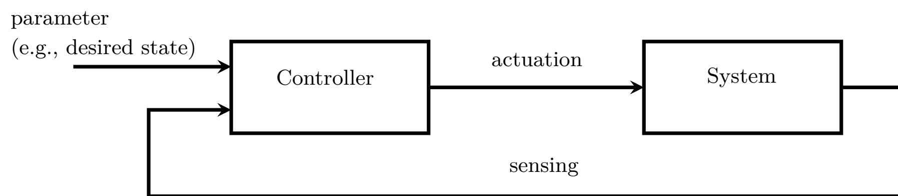


Figure 2.1. A control system is determined by a Controller and a System in a feedback loop relation.

Chapter 2

Background

2.1 Feedback Control Systems

A **System** to be controlled (or *plant*) is an entity having an internal state, inputs (*actuation signal*) and outputs (*sensing signal*). A **Controller** is an entity connected in a *feedback loop* with a System i.e., the actuation and sensing signals are respectively output and input for the Controller (see Figure 2.1). A feedback loop realizes a *feedback control system*, where the sensing, actuation and parameters signals are called **control signals**. The state of the System changes over time, i.e., it is subject to dynamics that depend on the actuation signals. The task of the Controller is to generate actuation signals that make the System's state approach a **desired state**. The desired state (or *setpoint*) is an input for the Controller, i.e., a parameter for the control system. The primary metric to describe the performances of a control system is the time required for the Controller to execute its task.

Desired state

Conventional control theory dates from the 19th century and utilizes the concept of *transfer function*. In a nutshell, the transfer function describes the effects on the output of the System when the inputs are pure sinusoidal signals (*frequency-domain analysis*). Conventional control theory is limited to linear, time-invariant, single-input, single-output systems. Complex systems may have multiple inputs and multiple outputs and may be time-varying. Because of: the necessity of meeting increasingly stringent requirements on the performance of control systems, the increase in system complexity, and easy access to large scale computers - modern control theory has been developed since around 1960. This new approach is based on the concept of **state**. Modern control theory is a generalization of conventional

control theory and also applies to multiple-input, multiple-output systems, which may be linear or nonlinear, time-invariant, or time-varying [82]. In modern control theory, the time-domain analysis describes the effects on the output of a System against any type of input (i.e., not necessarily sinusoidal).

The Controller contains a **logic** which employ a **model** of the System. An appropriate representation of a state is utilized by the logic, depending on the particular control system considered. The most utilized approach is the *state space representation*, which makes use of Euclidean spaces and, therefore, is based on linear algebra. Other state representations are employed by sciences deriving from control theory, e.g., graphs are typically employed in the fields of Timed Automata or Autonomic Computing to represent each possible state for a System. A controller employ a *dynamic system* to describe the dynamics of a control system (i.e., controller and system in a feedback loop). Based on its own state, and the input (e.g., sensing), a dynamic system describe: the output (i.e., actuation); and the transition for its own state (i.e. the *transition function*).

Transition function

The Controller produces actuation signals over time by considering the **current state** of the System, i.e., the state of the System, as known by the Controller at the time of production of an actuation. Ideally, the current state reflects exactly the actual state of the System. However, the state may be not exactly determined, due to numerical noise in the value transported by control signals, or inaccuracies in the System's model. **State estimation** techniques aim in determining the most probable state of a System at a given time. A well-known state estimation technique is the Kalman filter. In its original formulation, a Kalman filter is recursive and based on sensing, and thus it updates the estimate for each new sensing in a sequence. More generally, *time series analysis* utilizes *sequences* of control signals. Time series analysis may require multiple sensing and actuation, and their time-stamp, data to provide a single state estimation with reasonable accuracy. An even more upstream problem than state estimation is the problem of **model estimation** (e.g., the problem of system identification). Model estimation is typically required in the presence of complex systems. The concept of time series analysis introduces the definition of a **knowledge** of the Controller, which consists of observations accumulated over time. In a controller, the knowledge differs from the logic in that the former depends on the behavior of the System, while the latter depends on the Controller's current design choices or the current value of its parameters.

State estimation

The **control scheme** of a control system is the particular logic adopted by the Controller. The most basic control scheme is the *proportional feedback* control scheme, which was initially defined by the traditional control theory. Despite its simplicity, the proportional control scheme can be used as an example to describe the essential ingredients of the process used by a modern control system. Being x the current state of the System, $\hat{x}$ the desired state, then actuation a is:

Proportional control scheme

$$a = K(\hat{x} - x) \tag{2.1}$$

where $K > 0$ is a constant parameter influencing the performances of the control system. In this case, $\hat{x}$ and K are the two inputs for the control system, i.e., they are parameter signals. The logic estimates x by considering an arbitrary history of sensing and actuation signals, then determines the *error* $\hat{x} - x$, and finally produces a using Eq.2.1.

The Controller (the System) is an entity producing (consuming) actuation signals and consuming (producing) sensing signals. The **control frequency** is the rate in which control signals are produced and consumed. The **control loop delay** is the time interval starting from the time in which a sensing signal is produced, to the time in which a correspondent actuation reaches the System. In networked control systems, the control frequency and the control loop delay find an analogy, respectively, in the bandwidth and the latency between two hosts.

Control frequency

Control loop delay

A fundamental characteristic of a control system is its **stability**. A stable control system maintains the state of a System in limited regions, with no possibility that the state ceases to be under control. A control system that is not stable could lead to unexpected and unrecoverable behaviors, possibly damaging the System. Before it can be used, a control system must be analyzed in order to determine its stability and performance. In traditional control theory, control systems are analyzed using mathematical frameworks in the frequency domain (i.e., in the so called Laplacian domain). Typically, the analysis is idealized with respect to a concrete implementation of a control system because it assumes an infinite control frequency and a zero control loop delay. This assumption is reasonable in the case of a Controller attached to the System, and working at a sufficiently high control frequency. Conversely, in the case of networked control Systems, a concrete implementation may affect both the performances (i.e., time to execute a task) and the stability of a control system. For the problem of a limited control frequency, modern control theory studies **event-based control**, which is a generalization of a periodic-based control, i.e., there is not a constant control frequency but rather an event-based production and consumption of control signals. For the problem of control loop delay, modern control theory studies **delayed control systems**, in which appropriate control schemes tries to restore the performance and the stability in the presence of delay.

Stability

2.2 Computing Background

2.2.1 Virtualization and Cloud Platforms

A computer architecture is conceptually modeled as a stack of layers, where at each layer there is a service interface providing functionalities. In standardized computer architecture, the layers are (from top to bottom): Application, Library, Operating System (OS), Hardware Abstraction Layer, Instruction Set - the latter providing an interface to the hardware.

An interface of a computer architecture provides an abstraction over functionalities to entities (*users*) located at higher layers, e.g., the Operating System (OS) offers to applications the allocation of RAM without the need to deal with low-level details. *Virtualization* is a concept intrinsically linked with the architecture of computers. As an example, an OS implements *virtual memory* to offer applications a more considerable amount of (virtual) RAM than the physically available. For doing so, the OS utilizes lower interfaces to aggregate physical RAM and hard disk storage memory as a single resource.

Architectural interfaces

A **virtualization layer** is an overriding of one or more (concrete) architectural interfaces to traduce them into different (virtual) interfaces. Trough a virtualization

layer, a (concrete) interface becomes usable even if the user (e.g., an application, an OS) is not designed for that interface. As an example, by using a virtualized Hardware Abstraction Layer, an obsolete OS can run on modern hardware. The role of a virtualization layer is to traduce each invocation to its interface into an appropriate (sequence of) invocation(s) to lower interfaces. Virtualization layers exist at any of the architectural layers of computer architecture.

Virtualization Layers

An Hypervisor realizes virtualization at the Hardware Abstraction Layer. A hypervisor is an OS (type-1 hypervisor) or an application (type-2 hypervisor), providing an execution environment to a guest OS. A guest in execution on a supervisor is a Virtual Machine. For historical reasons, nowadays, the term Virtual Machine (VM) refers to a guest OS (e.g., Linux) together with libraries and applications. However, a Virtual Machine is an abstract concept whose meaning does not depend on the particular level of virtualization employed. Real-world data centers show that virtualization increases fault tolerance and enables distributed resource scheduling. An example of highly innovative functionality introduced by system-level VMs is their capability of being “migrated”, i.e., the execution of a VM is paused, then the VM is transferred to a different data-center, the execution of VM is finally resumed. It is pointed out that, in case a cyber-attack has compromised any functionality above a virtual service interface, the threat continues to exist in the new VM.

Virtualization at the OS level is realized by so-called virtualization engines, which allow the existence of multiple isolated user-space instances on the same OS kernel. The most diffused virtualization engine is Docker, which defines a *container* as a set of libraries and applications required to run into an isolated execution environment. Compared to system-level VMs, containers have a smaller footprint, which brings considerable improvements in terms of deployment time and execution performances.

Unikernels is a relatively new kind of Virtual Machine, in which software is directly integrated with the kernel it is running on. In a nutshell, a unikernel is a system-level VMs where the OS kernel is customized and compiled together with a specific application. Preliminary studies of Unikernels show that under certain conditions, they can exceed containers in terms of pure speed and response time[33].

Virtual Machines

In the definition of a "Virtual Machine" there are many different names to refer to something conceptually similar. For example, from the point of view of functionality, a container can be understood as a system-level VM, though defined at a different virtualization layer. Even within the same virtualization layer, there exist different names to indicate a Virtual Machine, e.g., Solaris OS refers to "Zones" as an analog of Containers.

A Virtual Machine (and not necessarily a system-level VM) has the following characteristics:

- before being started, a VM consists of a *image*, which is a file containing everything required by a virtualization layer to start the execution of the VM.
- when a VM is started, a *configuration*, can be *provisioned* to the VM. The configuration characterizes the initial state of the VM after its execution begins.

Cloud Platforms

A cloud platform [65] is a service that, in turn, renders the services belonging to its user available. A cloud platform provides an abstraction over one or more

layers of computer architecture, thus greatly simplifying the deployment of services. Depending on what are the layers are managed, a cloud service takes a different name. As an example, in an IaaS (Infrastructure as a Service) model, the user instantiatesits own VMs at Hardware Abstraction Layer. Higher levels of abstraction are possible. As an example, in a CaaS (Container as a service), the VMs are containers. In a PaaS (Platform as a Service) model, the user composes its service utilizing modules offered by the cloud platform. The IaaS model allows the maximum level of flexibility, but also the higher degree of complexity in its use, as the cloud platform provides fewer high-abstraction tools for service deployment.

Pool of resources

An IaaS platform introduces the concept of *Pool of resources* which is an abstract entity representing the aggregation of multiple physical resources, seen as a single set of virtual resources. A Pool of resources finds different implementation names in different cloud solutions. As an example, in Openstack [12] Pool of resources is a "project". There exist virtualization technologies which allow to interface with a cloud platform, abstracting the user from the particular implementation of a cloud platform. For example, in the case of an IaaS model, Apache LibCloud translates high-level commands (e.g., to start a specific VM image) into commands and controls for multiple implementations of cloud platforms (e.g., Apache CloudStack, Microsoft Azure, Amazon Elastic Compute Cloud).

2.2.2 Microservices oriented architectures

Service-oriented architecture (SOA) is a concept for the design of applications based on replaceable, decoupled components that possess unified interfaces for standard protocol communication[9]. The origin of SOA is found in complex web applications such as Google, Amazon, and eBay - and nowadays is at the basis of cloud-based services[107]. The usage of SOA and cloud platforms for CPS is nowadays a well-established topic[55], particularly for Industrial Control Systems[25].

Microservices

Microservices [105] is a term describing the idea of manage growing complexity by functionally decomposing large systems into a set of asynchronous services. Microservice architectural style is an approach for developing a single service as a set of small services components, each running within a process, and communicating via messages[46]. A microservice is an autonomous process collaborating with other microservices for a common goal. A *Microservice Architecture* (MSOA) is an SOA where all components are microservices. Some studies[4, 27] define MSOA as the most promising evolution of SOA. As discussed in work [113], MSOA is highly modular, distributed, and made of reusable components through network-exposed APIs. Being a distributed system, an MSOA has the following characteristics: 1) the overall application state is unknown to individual nodes (i.e., microservices); 2) individual nodes make decisions based on the locally available information; 3) failure of one node should not affect other nodes.

A *message channel* is established between a producer microservice and one or more consumer microservice(s). Message channels can be established with a point-to-point or publish-subscribe model. In point-to-point, a producer microservice specifies the identity of the consumer. In a publish-subscribe model, a producer can deliver the messages to one or more consumers without having to know their

Message Broker

existence, i.e., to discover their location. The publish-subscribe model is enabled by a ***message broker***, which is an intermediary service managing all message channels. An essential benefit of using a message broker is that the broker buffers messages until the consumer(s) are able to process them[91]. This approach is well suited to the usage of **queues** in message channels, which is commonly considered a best practice for MSOA.

An MSOA can adopt two approaches for the coordination of microservice components: *orchestration-based* or ***choreography-based***. Orchestration requires a central microservice sending requests to other microservices and oversee the process by receiving responses. Choreography, on the other hand, assumes no centralization and is based on a publish-subscribe mechanisms in order to establish collaboration. The concept of choreograpy was defined prior to microservices to describe the global behavior of SOA. In Distribuited Programming, choreographies are high-level descriptions of expected interactions between components.

Choreograpy

In Database theory, an ACID transaction is a sequence of operations updating the state of an application while enforcing Atomicity, Consistency, Isolation, e Durability. ACID transactions are essential to every enterprise application to maintain consistency of its state. In a monolithic application with a single database, the enforcement of ACID transaction is straightforward. In a microservice architecture, however, it is possible that a transaction has to update the state of multiple microservices. It is challenging to employ distributed transaction models (e.g., X/Open XA) in an MSOA, as these conflict with the principles of mutual decoupling and asynchronicity of components. The main problem with the standard distributed transaction model is that all the microservices have to be available to complete a distributed transaction. Another problem is that many technologies that are fundamental to microservices, such as NoSQL databases (e.g., MongoDB, Cassandra) or message brokers (e.g., RabbitMq, Kafka), do not implement distributed transactions models. Although workarounds are possible in this sense, the microservice community defines the concept of *sagas* to replace the functionalities provided by distributed transactions. A saga is a sequence of local transactions (i.e., local to a microservices) that are coordinated using asynchronous messaging. Each local transaction updates the state of a single microservice using the familiar ACID transaction. Sagas maintain data consistency across microservices, i.e., a saga defines a sequence of actions that are performed atomically. Simple sagas can use a choreography-based approach, but the orchestration is usually a better approach for complex sagas[92].

MSOA simplifies an automated deployment. Each microservice of an MSOA is deployed as a separate VM, e.g., a container. Since microservices are distinct and autonomous processes, they can be deployed independently with minimal centralized management. An important concept related to the deployment of microservices is the concept of **immutable deployment**. An architecture with immutable deployment is entirely made of *immutable microservices*. That means that the instance of an immutable microservice can be replaced (i.e., re-deployed) without any special care in the management of its internal state. The state of the execution environment of a microservice is not necessary to the consistency of an application. In case the microservice needs to maintain a permanent state, an external database is used. The term immutability originated from Functional Programming to describe the

Immutable deployment

behavior of a system not in terms of in-place mutation of objects, but in terms of the immutable input and output values of pure functions[23]. Differently from pure functions in Functional Programming, immutable microservices have a state, and the immutability is defined only at the level of their deployment.

In a MSOA adopting the immutability principle, there are two kinds of microservices:

- *immutable microservice*: it is dedicated to computing. In case it has a state, it is memorized in a datastore microservice.
- *datastore microservice*: it is dedicated to memorization, and realize a data store. It is a network database, or a network disk volume, offered to immutable microservices.

The combination of the content of all the datastore microservices in an MSOA fully captures the state of an instance of an MSOA.

There is an overhead introduced by the adoption of an MSOA. Being a distributed architecture, the development of an MSOA is more difficult than in the case of a monolithic application, and possibly require the usage of formal tools [87, 79] to verify the absence of bugs. Bugs includes possible deadlock between microservices (*communication deadlock*), and possible unexpected order in the reception of messages (*race*). Having the components of an MSOA to communicate through network messages (rather than shared memory), the computational performances decreases. This problem is further complicated by the fact that, like any other SOA, the microservices should employ end-to-end encryption in each component[13].

2.2.3 Event Sourcing Pattern

Event Sourcing is a pattern for capturing all changes to the state of an application as a sequence of *events*. A *event store* is a data store containing a sequence of events that can be used to determine the state of an application. An MSOA architecture adopting the principle of immutability combines elegantly with the event-sourcing pattern. The event store contains sufficient information to determine the content of the datastore microservices which, in turn, fully describes the state of an application. Event Sourcing simplifies the usage of choreography-based sagas since the participant to a saga reacts to events by producing new events. Therefore, to reproduce the effect of a sequence of events, only require the provisioning of the same sequence of events.

Immutability and Event Sourcing

Event sourcing simplifies the following activities:

- debug, by reproducing the effects of each of the events leading to a problematic state, e.g., an inconsistency in the application model;
- testing, by simulating the behavior of an application through the provisioning of an appropriate sequence of events;
- state reversing, by reversing the effects of the final part of a sequence of events, or by provisioning of a truncated sequence of events;

- evolution, by provisioning a historical sequence of events to a new version(s) of component(s) of an application.

2.2.4 Security of microservices oriented architectures

Security of microservices

Virtualization can be a mechanism to increase the security of an application. An attack performing *lateral movement* has the ability to spread among different components of a system. Consider an application exposing endpoints, e.g., REST API. Each of the endpoints introduces a potential vulnerability at the application level, e.g., it is vulnerable to code injection. In the case of monolithic deployment of the application, the modules communicate through shared memory. Therefore, the compromise of a single endpoint renders lateral movement an easy task. i.e., the attack may spread trough other modules using the shared memory. Lateral movement can be rendered more difficult by deploying each module on a separate Virtual Machine, like in microservice oriented application (MSOA) - because there is an increased isolation between the components. In MSOA, lateral movementis possible if (at least one of the following):

- the attack can spread through the internal APIs utilized by the microservices
- the attack can surpass the virtualization layer(s)

The difficulty of surpassing a virtualization layer depends on the adopted virtualization layer(s), and from the additional security measurements employed. Considering system-level VMs, an attacker have to surpass the guest OS kernel *and* the hypervisor. In the case of containers, however, the same OS kernel is shared among virtual machines. For this reason, Unikernels provides better isolation than containers since they do not share the same OS kernel [16]. In any case, the described isolation of an MSOA can be further strengthened using dedicated frameworks (e.g., SCONE [8] for containers).

Works [113] and [68] examines MSOA with a particular focus on its security implications, and highlight the relation between immutability and security in MSOA. As an example, in [113] the authors describe the automated ***immutable deployment*** property as: "*services should be immutable: to introduce permanent changes to microservices, services should be rebuilt and redeployed. Microservice's immutability improves overall system security since malicious changes introduced by an attacker to a specific microservice instance are unlikely to persist past redeployment. Automation should be leveraged in maintaining the security infrastructure. Immutability aids the security of microservices similarly to how immutability promotes correctness in programming languages.* "

Relation between immutability and security in MSOA is found in technical guides related to the security of MSOA, e.g.in [22] the author define an immutable infrastructure as: "*immutable infrastructure consists of immutable components that are replaced for every deployment, rather than being updated in place. Those components are started from a common image that's built once per deployment and can be tested and validated. Image integrity is a core security requirement for an immutable*

infrastructure.”

2.2.5 Automated deployment tools

To “deploy” a service means to render the service available. This is typically realized by starting a VM containing an application. The operations required to deploy a service can be automated using appropriate tools and virtualization technologies. Automation involve operations aiming to maintain acceptable QoS levels after the initial deployment of a service. Automated deployment operations go under the name of *orchestration* [7], which has become particularly useful with the advent of lightweight VM such as containers. Cloud platforms utilize orchestration to provide acceptable QoS of deployed user services, and at the same time to optimize the usage of computational resources. Among the several functionalities enabled by orchestrations, the main are: *i*) react to hardware failures by deploying new instances of services; *ii*) optimize the usage of computational resources by deploying replicas of service only when required (so-called "horizontal scaling"); *iii*) manage the evolution of services without causing temporary disruption to the services (so-called “continuous integration and delivery” in DevOps[48]).

In this thesis, we are particularly interested in orchestration because it enables a mitigation and prevention strategy against cyber-attacks. There are two important technologies related to automated deployment:

- A *Configuration Management Engine* (**CME**) maintains available services at the OS layer in machines, e.g., ensures that specific packages are installed, and specific OS services are running. *CME*
- A *Container Cluster Manager* (**CCM**) orchestrates the instances of containers among the Nodes of a cluster. Orchestration activities have the final goal of exposing application services realized by the containers while respecting given QoS levels. *CCM*

CME and CCM are commonly employed “under the hood” by the implementation of cloud platforms. There exist implementations of CME and CCM, which allow specifying in a declarative manner what *a desired state* for their clusters is, and have the ability of planning and actuating opportune actions when the desired state is not met [58]. To be more specific, the *current state* of a cluster may be undesirably changed by events that not depends on its management. When such disturbances occur, a dedicated controller tries to manage the cluster such that the desired state is met again, by means of a sequence of internal commands and controls. This approach is reminiscent of the concepts of state of a System as understood in Control Theory or Autonomic Computing. We are particularly interested in this kind of approach since it enables to realize a self-healing system at the level of the deployment, as proposed by the prevention and mitigation approach of this theis. In what follows, we describe existing implementations of CME and CCM having the common characteristic of accepting a declarative specification for their task, i.e., they accept a desired state for their cluster.

CME Master, Minion

SaltStack is a CME based on a master-slave model [94]. A **Master** is an application responsible of configuring the OS of the **Minion** nodes. For each Minion, the desired state consists of OS packages that must be installed, and OS services that must be running. SaltStack periodically checks that all the Minions are in a desired state. Consequently, SaltStack may decide to install packages and start OS services, or re-start them if they are sensed to be not running (e.g. when a service crashes). Among other configuration management engines, SaltStack is characterized by the usage of asynchronous message queues.

CCM Master, Nodes

Kubernetes is a CCM based on a master-slave model [39]. The Kubernetes **Master** is an application realizing a scheduler to manage the deployment of applications. Kubernetes provides mechanisms for deployment, maintenance, and scaling of containers. In Kubernetes terminology, a **Cluster** is composed of a set of **Nodes**, each Node can host one or more **Pods**, a Pod represents a service made of components. A Node is associated with a particular host machine (e.g., a physical machine or system-level VM). A Pod is modeled as a group of Docker containers having shared namespaces and filesystem volumes. The concept of a Pod allows informing the scheduler that a group of containers should be treated as a single unit of deployment, i.e., co-located on the same Node and share the same resources such as network, memory, and storage. Each of the container in a Pod is a single microservice realizing part of the functionalities of an application service. Each Pod gets a dedicated IP address that's shared by all the containers belonging to it. A Kubernetes **Service** is an invariant endpoint for Pods. These endpoints remain the same, even when the Pods are relocated to different Nodes by the orchestration activities.

CCM Labels

Kubernetes allows to specify user-level constraints for the deployment, which are considered during the orchestration activities. The constraints allow to specify, for example, what Nodes are possible candidates to host a certain Pod. The cited functionality is realized by allowing the user to attach **Labels** to Nodes, and then define constraints associated to a Pod which restrict the deployment to Nodes having certain Label(s). Other user-level constraints are related to the minimum computational capabilities (e.g., number of CPU-cores) a Pod should have reserved.

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of a Cluster. Example of Kubernete's objects are: `Node`, `Pod` or `Service`. Instances of these object can describe:

- what containerized applications are running (and on which Nodes);
- the resources available to those applications (e.g., CPU, RAM);
- the policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance;

Kubernetes spec, status

Kubernetes API allows to interact with objects, i.e., create, modify, or delete them. Kubernetes API makes a distinction between the specification of a desired state of an object using a field called *spec*, and its current state using field *status*. As an example, a Service is specified to be deployed on a certain subset of the available Nodes by specifying a certain *spec* for a `Service` object. In case the state

of the cluster don't met the desired state, or the desired state is changed by the user, Kubernetes automatically recover by re-deploying the Pods accordingly to the `specs`.
From the Kubernetes documentation[1]:
" *when a new version of an object is POSTed or PUT, the "spec" is updated and available immediately. Over time the system will work to bring the "status" into line with the "spec". The system will drive toward the most recent "spec" regardless of previous versions of that stanza. In other words, if a value is changed from 2 to 5 in one PUT and then back down to 3 in another PUT, the system is not required to 'touch base' at 5 before changing the "status" to 3. In other words, the system's behavior is level-based rather than edge-based. This enables robust behavior in the presence of missed intermediate state changes.*"

[1] `https://github.com/Kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#typical-status-properties`

Chapter 3

ASiMOV

The purpose of a defense system can be the prevention, detection, or mitigation of attacks. Ideally, a defense system does not omit any of the above functionalities. This thesis focuses on detection aspects, with a detailed model and implementation (ASiMOV). Moreover, we discuss a mitigation strategy, with a high-level of abstraction architecture of a dedicated application (Deployment Manager). Aspects of preventions are, in this thesis, are only hinted at, by considering the possible advantages coming from the ASiMOV architecture for third-party systems. As an example, deployment-based processes could employ N-version programming[81], i.e., a functionally equivalent CA replaces a tampered CA, using a different language, however. This strategy is easily applicable through the provisioning of different VM images.

Our main goal is to define an Intrusion Detection System (IDS) against *Internal attacks*, that are tampering of the control logic in Cyber-Physical systems (CPS). Section 3.1 provides and motivates the requirements for the proposed solution, named ASiMOV. Section 3.2 introduces to ASiMOV (ASynchronous MOdular Verification). Section 3.3 describes the characteristics of an infrastructure executing ASiMOV, and a deployment model. The attacks that we consider are defined more formally in Section 3.4.

3.1 Detection requirements

In our threat model, the devices executing the control logic (*field devices*) may be successfully attacked in any of their functionalities (*unrestricted* attack), so that they produce tampered actuation commands. Moreover, we assume the presence of devices (*security devices*) in which attacks are prevented from becoming successful.

Table 3.1 provides the list of requirements for the proposed solution. Each of the requirement traduces into a specification (Table 3.2), as detailed in what follows.

Requirement R1 comes from research question **RQ1** of Section 1.3. As an answer, we propose a *network-based* IDS executed by security devices. The motivation, is that we are not allowed to trust any process carried out by field devices. Therefore,

	Requirements for the detection process
R1	detection of unrestricted attacks to the control logic of a CPS
R2	detection is behaviour-specification-based
R3	detection has ideal accuracy (i.e., 1)
R4	detection does not increase the control loop
R5	detection delay is estimable

Table 3.1. Requirements for an IDS against Internal attacks

	Specifications from requirements of the detection process
from R1	ASiMOV is a network-based IDS
from R2,R3	ASiMOV employs an *exact model* of the control logic defined at the level of the execution environment (i.e., both the model and its current state are known without any uncertainty)
from R4	detection processes of ASiMOV are decoupled from direct control
from R5	detection delay of ASiMOV is estimated trough a model

Table 3.2. Specification of the prosed IDS from the requirements

we let security devices to intercept and inspect the network traffic of the field devices. Assuming that the field devices have a different human administrator than security devices, having a replica of a field device into security devices may protect against malicious insider attacks - that accordingly to [43] are one of the challenges for CPS.

Figure 3.1 places our solution (ASiMOV) in a classification of IDS, considering the employed audit material (host/network-based) and the detection's target (internal/external attacks).

We require a behavior-specification-based solution (R2) because, as described in Section 1.2.1, its is currently considered the preferable approach in the protection of CPS, that are complex systems (e.g., highly nonlinear, with a hybrid state), and

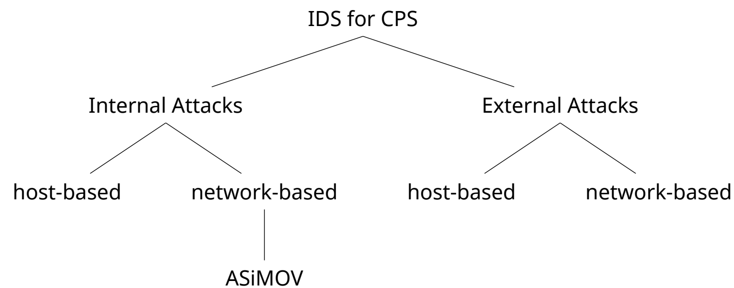


Figure 3.1. IDS for Internal (External) attacks aim to detect tampering of the control logic (sensors and actuators). Host-based (Network-based) IDS are executed on the same device (on a different device) of that executing the control logic.

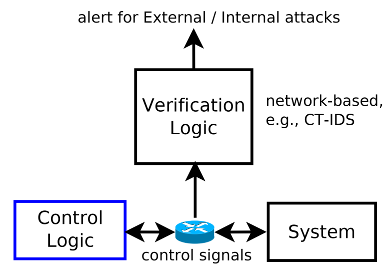


Figure 3.2. Generic representation of a network-based Intrusion Detection System (IDS). The control logic and the verification logic are in two different execution environments, therefore their state may be not synchronized.

therefore difficult to protect with knowledge-based approaches.

In the presence of a network-based IDS, there are two distinct logics: the *control logic* **CL**, that forms a feedback loop with a System; and the *verification logic* $\mathbf{CL}^v$, that inspects network traffic. In general, $\mathbf{CL}^v$ needs to consider both the influence of the "cyber" processes (i.e., **CL**) and the "physical" processes (i.e., system(s) under control) of a CPS (Figure 3.2). A known problem in literature regards the undesired effects of having an inexact model of **CL** in $\mathbf{CL}^v$. In a trivial example, if a parameter for **CL** (e.g., setpoint, or desired state) needs to change, $\mathbf{CL}^v$ needs to be informed. In their proposal of a network-based IDS against External attacks, the authors of work[103] say: " *The results do not consider the influence of the feedback controller. However, the results can be generalized by considering the augmented system composed by the plant and controller dynamics, which is subject to future work.*". In a nutshell, the authors consider to let $\mathbf{CL}^v$ track the dynamics of **CL** (i.e., the state of **CL** can be seen as a feed-forward signal). Requirement R3 is the answer to the research question **RQ3**. The motivation follows. We aim to avoid any uncertainty in the detection process, to obtain an ideal detection accuracy against Internal attacks. The reason for the requirement is that in the presence of complex control systems, e.g., highly-nonlinear - that are common in real-world CPS, the slightest inaccuracy in a model may lead to very diverted dynamics in a short time. Control-Theoretic IDS with a geometric approach, as well as knowledge-based IDS, typically raise an alert when a particular value exceeds a given threshold (e.g., the norm of a vector). Such threshold is a parameter for $\mathbf{CL}^v$ that is chosen to be an optimal trade-off between detection and false alarm rates - and accounts for any modeled uncertainty (e.g., noise on sensors, numerical inaccuracies, and so on).

To better explain the considered problem, we consider a generic CT-IDS against External attacks (e.g., attacks to sensors). We assume that the accuracy of $\mathbf{CL}^v$ is affected by two unrelated causes of inaccuracy: inaccuracy in determining the state of the controlled system (e.g., sensors noise), and inaccuracy in determining the state of **CL**. We argue that if the latter inaccuracy increases, the accuracy in the detection process of $\mathbf{CL}^v$ decreases. Our answer to research question **RQ3** is based on the intuition that the more a detection process has an accurate model of the control system (i.e., including **CL**), the better is the detection accuracy (i.e., lower false postitve/negative). Without a formal proof, we assume that the following comparison between the two scenarios is true:

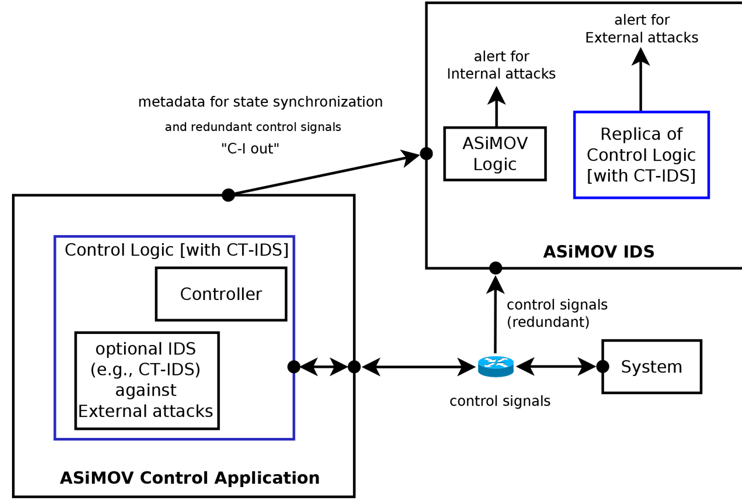


Figure 3.3. Network-based IDS realized by ASiMOV. The ASiMOV's Control Application and IDS are synchronized. The control logic can optionally include an IDS for the detection of External attacks, e.g., a Control Theoretic IDS (CT-IDS) protecting the sensors. In this case, and differently from Figure 3.2. the CT-IDS has an exact model of the state of the control logic, since they are on the same execution environment.

- Without ASiMOV: the CT-IDS is network-based, and deployed in an infrastructure without the features introduced by ASiMOV (Figure 3.2). The production and communication of control signals are subjected to network conditions and independent timing policies (e.g., event-based control). Therefore $\mathbf{CL}^v$ does not employ an exact model of **CL**. The IDS displays accuracies of $\chi_E < 1$ and $\chi_I < 1$, respectively against External and Internal attacks.

- With ASiMOV: the same IDS of the previous scenario is now a sub-component of the control logic, and therefore disposes of an exact model of **CL**. ASiMOV replicates the control logic, that include the CT-IDS, using a host we assume un-attackable (Figure 3.3). Therefore, detection performances against Internal attacks are ideal (i.e. 1). Assuming that the network conditions remain the same of previous scenario, in this scenario the performances of the ASiMOV IDS are $\bar{\chi}_E \geq \chi_E < 1$ and $\bar{\chi}_I = 1$.

Section 4.1.1 "Motivations for the replication of the control logic" deepens the problem of an exact state estimation of **CL**.

We require that ASiMOV does not increase the control loop delay (requirement R4) of the control system. ASiMOV is a network-based IDS. Therefore, we intend to avoid the latency between **CL** and $\mathbf{CL}^v$ to be a bottleneck for real-time control of physical systems. Therefore, we require that **CL** never waits for $\mathbf{CL}^v$. In particular, we propose a *delayed synchronization* of the $\mathbf{CL}^v$ over **CL**, $\mathbf{CL}^v$ is informed about the exact state of **CL** but at a later time.

We require that the detection delay, i.e., the time required to detect an attack, is estimable trough a model (requirement R5).

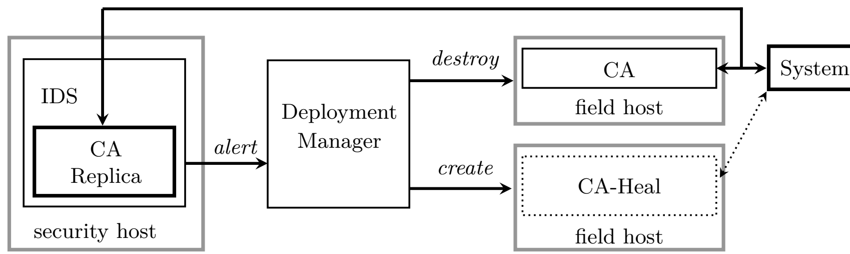


Figure 3.4. High-level representation of the ASiMOV architecture.

3.2 Introduction to ASiMOV

ASiMOV (ASynchronous MOdular Verification) is a reference architecture for :

- an event-based Control Application (**CA**) for Cyber-Physical Systems (CPS), that enables a deployment-based mitigation and prevention mechanism against Internal attacks.
- an Intrusion Detection System (**IDS**) detecting Internal attacks.

This Chapter describes the scenario in which ASiMOV operates. Section 3.2 introduces, at a high level of abstraction, the protection mechanism realized by ASiMOV. Section 3.3 describes the adopted deployment model. Finally, Section 3.4 defines the model of the attacker considered by ASiMOV.

Figure 3.4 represents a single module in ASiMOV, that is composed by two applications: a CA and an IDS. Multiple ASiMOV modules are interconnected to realize a distributed control scheme governing a whole CPS. The IDS contains a replica of the CA, which is provisioned with the history of the control signals to verify runtime features that are out of the ordinary (i.e., unsupervised detection method). The proposed detection technique requires that the inputs and outputs of

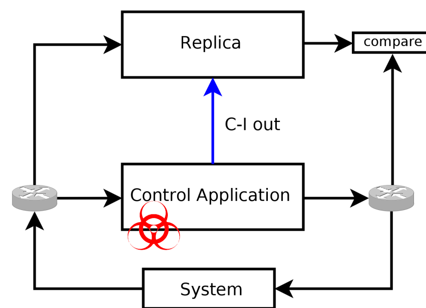


Figure 3.5. Logical representation of the dataflow between a System, a CA and a replica of a CA.

the CA are replicated at the level of the network. Figure 3.5 represents the dataflow between the CA and a replica. The detection consists in comparing the outputs of the replica with the outputs of the CA: any discrepancy may be interpreted as the consequence of a cyber-attack.

The state of the CA and the replica are kept consistent by providing the same inputs, i.e., control signals. The verification and direct control processes are asynchronous to let the CA and the IDS work independently, hence to avoid the introduction of a delay in the control loop. Being the CA and the replica asynchronous, CA needs to push synchronization data toward the replica (*C-I out* arrow in Figure 3.5).

If an attack is detected, the mitigation mechanism consists of the redeployment of a new instance of the CA (i.e., the CA-Heal in Figure 3.4), which is provisioned with a previously verified state contained in the IDS. The proposed mitigation mechanism can also be used as a preventive strategy, since periodic redeployment and rejuvenation of the host executing the CA may prevent silent attacks not yet detected to become successful.

3.3 ASiMOV Deployment model

A CPS contains systems controlled by CAs, where each CA implements a specific control scheme, e.g., a proportional controller. A control scheme of a CPS corresponds to the control schemes of the individual CAs and their interconnection. An instance of a *deployment model for a CPS* contains sufficient information to deploy a control scheme for a CPS, and the corresponding set of IDS (i.e., one IDS for each CA).

3.3.1 Deployment model for a CPS

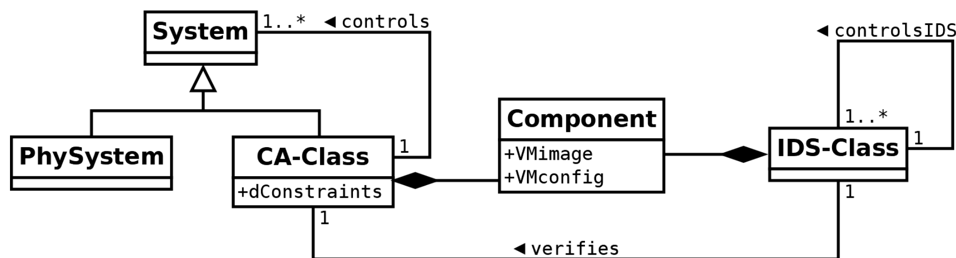


Figure 3.6. UML deployment model for a CPS. A CA-Class represents a control scheme assigned to the control of a System (relation `controls`) A System may be a physical System or a CA-model (i.e., for a hierarchical control schemes). An IDS-Class represents and IDS assigned to the verification of a CA-Class.

Figure 3.6 is the UML of the deployment model for a CPS. A CA is architected using microservices and adopts an immutable deployment model. The **CA-Class** is a generic a controller accordingly to the ASiMOV architecture, while an instance of CA-Class characterizes a specific control scheme to be deployed. Before its deployment, a CA is described by an instance of CA-Class, which contains everything required by an execution environment to execute the CA, and characterizes the state of a CA at the time of the deployment. A CA-Class consists of different components (class **Component**). Each Component corresponds to a microservice, and an instance of Component is characterized by a specific image of a Virtual Machine (attribute `VMimage`), for example a container image; and attribute `VMconfig`, for example a

configuration file. The attribute `dConstraints` contains the value for constraints relative to the deployment, e.g., it allows to restrict the deployment of a CA to a particular device. This aspect is clarified in Section 5.4.

In the UML model, the association `controls` represents the assignment of a Controller to one or more Systems. The class **System** is a generalization of both CA-Class and class **PhySystem**, the latter representing a physical System equipped with a network interface. The introduced generalization allows describing a distributed control scheme, i.e., where a CA controls a different CA. In a hierarchical control scheme, the set of associations `controls` defines a tree where nodes are instances of System, and leafs are instances of PhySystem.

IDS-Class represents and IDS assigned to the verification of a Controller. In an instance of the model, the set of associations `controlsIDS` is consistent with the set of `controls`, i.e., the IDS of two CA in a `controls` relation are in a `controlsIDS` relation. As already mentioned, an instance of a System could represent a physical system or a CA. However, in order to avoid disambiguation, we refer to a System as a controlled entity. We refer to a controlling entity as a CA-Class, CA, or Controller.

3.3.2 The deployment process for a model

An instance of deployment model for a CPS is said *deployed* when:

- each instance of the CA-Class is deployed as a CA;
- for each CA, there is deployed a correspondent IDS;
- CAs, Systems, and IDSs are configured to establish the required communication channels (e.g., a CA and a System are configured with their reciprocal
- the network enables the required communication channels (i.e., between CAs, Systems, and IDSs);network addresses).

The deployment aspects regarding the achievement of the above conditions are part of an automated or semi-automated process.

Devices can be either *computing devices* (e.g., a general-purpose computer, or a micro-computer like a Raspberry PI) or *networking devices* (e.g., a router or a switch). Utilizing virtualization technologies, both computing and networking devices may be virtual. Depending on their purpose, devices are classified as:

- *Field devices*: realize the infrastructure dedicated to the direct control of the Systems, i.e., executes CAs and realizes the network topology required by an instance of the deployment model;
- *Security devices*: realize the infrastructure dedicated to the detection, mitigation, and prevention of attacks (i.e., including the IDS).

Some of the devices are remotely configured e.g., a field device is configured to hosts a CA, a software-defined network switch is configured to enable the required communication channels.

A specialized application, we name *Deployment Manager* (DMAN), is in execution on the security devices. The DMAN realizes automated deployment and orchestration of applications (both CAs and IDSs), and possibly the network.

The management is realized trough the provisioning of *configurations* to appropriate management client application(s) that are executed on the configurable devices. As an example, consider a physical machine dedicated to the execution of CA. The device is a Minion of a Configuration Management Engine (CME), hence it is possible to deploy multiple CAs trough the management of the layers of the computer architecture of the device. The management operated by the DMAN may involve different layers of computer architecture and may require multiple steps to be completed. In particular, it is possible that the DMAN needs to prepare the device for the deployment (pre-deployment operations). After the devices are prepared, they can be used to deploy a CA.

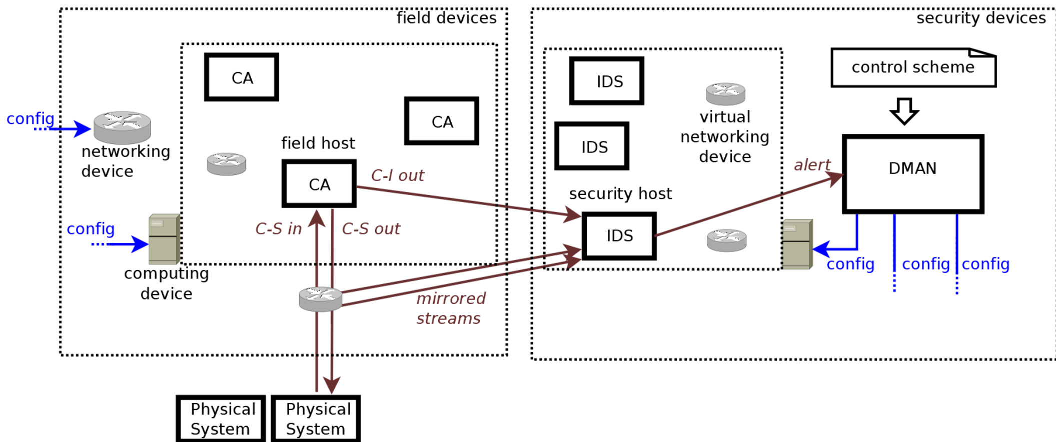


Figure 3.7. Representation of a CPS once a control scheme is deployed. For each CA there is an IDS. There are streams a CA-System stream and a CA-IDS stream for each CA.

Figure 3.7 represents the devices of a CPS in the deployment model proposed by ASiMOV. A *host* is an environment executing and exposing the endpoints of a service-oriented application. We assume that physical systems are equipped with a networked application interface, hence they are hosts. A CPS contains computing devices which, after a remote configuration, become hosting services. Hosting services can be used to deploy VMs (e.g., containers). The hosting services are classified depending on their purpose:

- *Field hosts*: used to deploy CAs into field devices
- *Security hosts*: used to deploy IDSs into security devices

Inter-host communication takes place entirely through separate, non-synchronized data streams. Considering a single CA, there is a communication with each of the systems under control, and with the associated IDS. In particular, a CA participates in the following communications:

- Communication with System(s): for each System, there is a bilateral communication made of streams containing control signals. Considering a single System, there are the following streams:
  - *C-S out*: a stream produced by the CA containing actuation for the System

  - *C-S in*: a stream produced by the System for the CA containing sensing from the System

- Communication with the IDS: a set of streams named *C-I out* is produced by the CA for the IDS, carrying data required to synchronize the state of the CA and its replica (i.e., control signals *metadata*).

The streams between the CA and the System (i.e., C-S out, C-I in) are subjected to mirroring toward the IDS, i.e., the streams are intercepted, copied and forwarded to IDS. Depending on the network infrastructure, the required mirroring functionalities may be realized employing configurable networking devices (e.g., CISCO router with SPAN technology) or employing dedicated physical devices (e.g., network TAPs).

3.4 Considered attacks

An attacker has the goal of providing tampered actuation to System(s). We do not consider timing attacks, i.e., to slow down the provisioning of actuation signals. An attacker is capable of tampering the device(s) which execute the CAs. We assume that an attack can only occur on the field computing devices. All the networking devices and security devices are assumed safe from attacks, i.e., each IDS receives the same streams produced and consumed by System(s), and cannot be tampered. An *Attack* is any tampering of the computer architecture of field computing devices(s), i.e., the tampering may occur at any layer of the computer architecture. Indipendently from the specific technique employed by the attacker (e.g., code injection, execution of malware, tampering of the memory, substitution of code by a malicious insider), an Attack leads to the production of malicious actuation.

ASiMOV protects each CA separately and independently. Therefore, in what follows the description of an Attack is relative to a single field computing device.

3.4.1 Security assumptions in ASiMOV

The detection of attacks in ASiMOV is based on the concept of modular redundancy, which is commonly employed against random faults, i.e., when the probabilities of failures of instances (replicas) of an application are independent. Differently from traditional modular redundancy, we differentiate the reliability of two instances of a controller based on the surface of attack of computing devices hosting a CA and a replica, i.e., respectively field and security devices. This assumption is based on the fact that, under the conditions that follow, a security device has a smaller surface of attack than a field device. For the security devices, we assume:

- an attacker does not have physical access to a security device, i.e., the device is in a *segregated environment*;

- a security device adopts strong security policies, e.g., strong encryption, host hardening, SGX enclaves, and the likes.

The above conditions do not apply to a field device, because:

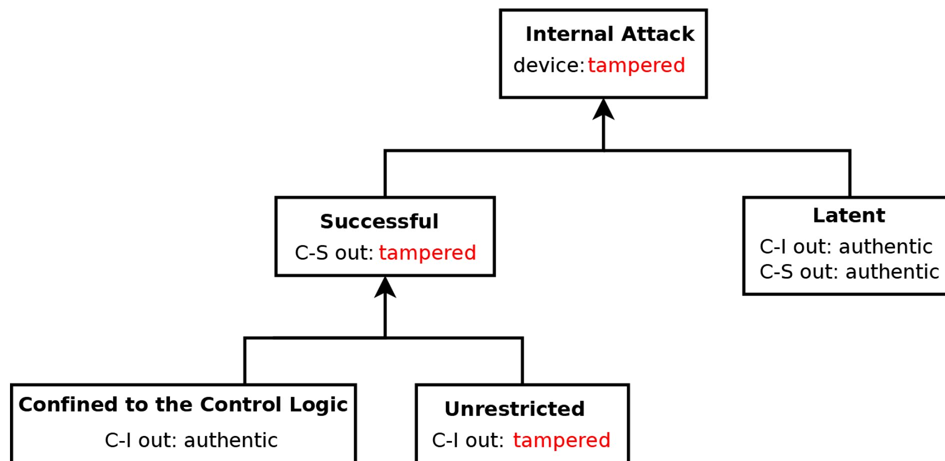


Figure 3.8. UML model representing the specializations of an Attack into different kinds.

- to maximize the performances of direct control, the communication delay between the controlled System and the CA should be minimized, specifically for soft and hard real-time control. Therefore, the CA must be near the System, i.e., in terms of network latency. It is unfeasible to segregate each of the field devices, in particular considering the vast amount of controlled physical Systems and their localization in an IoT scenario;

- strong security policies may introduce performance degradation (e.g., [57]), which may be variable and difficult to be estimated. Hence, the highest possible degree of security policies may introduce a (variable) delay within the control loop, which is problematic primarily for real-time control.

Attacks to networking devices (e.g., switches) are outside the scope of ASiMOV. A tampered networking device may result in an External attack (i.e., attack to the sensor and actuation data), which may be detected by control-theoretical IDS (CT-IDS) integrated into the control logic and therefore protected by ASiMOV.

3.4.2 Model of the Attacker

A tampered device may experience different malfunctions that may not be observable from the outputs. An attack is specialized in different kinds depending on what can be determined from the outputs of a CA (Figure 3.8).
An attack is *successful* when it tampers the C-S streams, i.e., the stream from a CA to a System. The main task of ASiMOV is to detect and possibly mitigate successful attacks.

Depending on what functionalities of a device have been tampered, also the C-I stream(s) may get tampered as well. Therefore we distinguish between two kinds of successful attacks:

- *Confined to Control Logic*: the C-S streams have been tampered, while the C-I streams are authentic.

- *Unrestricted*: both the C-S and the C-I streams have been tampered.

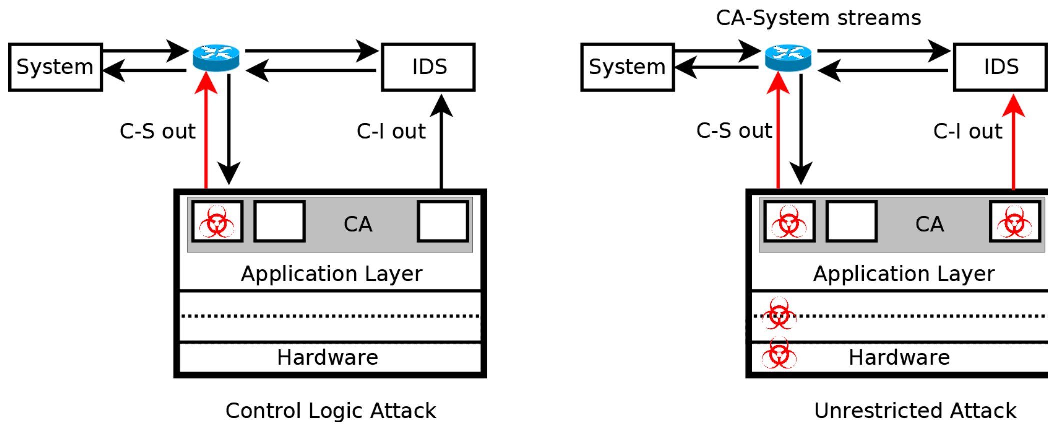


Figure 3.9. Any of the multiple architectural layers of a device can be compromised, as can any sub-module of a CA. Depending on what outputs of a device get tampered (red arrows) a Successful Attack is Unrestricted or to the Control Logic.

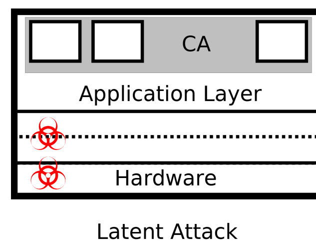


Figure 3.10. In a Latent Attack a device is tampered, but there is not an observable malfunctioning.

Figure 3.9 represents the two introduced kinds of a successful Attack. When an Attacker can do nothing more than an attack confined to the Control Logic, this means that the attack is confined to the subcomponent(s) of the CA dedicated to the computation of actuation signal. An attacker capable of carrying out an unrestricted attack may have full control over the tampered device. Such an attack could be caused by vulnerabilities at the OS level (e.g., administrative escalation), hypervisor (e.g., compromised isolation), or at the hardware level (e.g., CPU microcode vulnerability).

An attack that is not (yet) successful is referred as a *latent attack* (Figure 3.10), which is undetectable by ASiMOV. However, ASiMOV may be able to prevent a latent attack from becoming successful. Although a latent attack has no impact on a System at the current time, it is dangerous as it could become successful in the future. In particular, a coordinated attack on multiple devices could inflict significant damage.

as a possible advantage within strategies employed by leveraging the functionalities introduced by ASiMOV.

Chapter 4

Model and Architecture

This Chapter details the design of ASiMOV, to realize the following functionalities:

- **Control** of Systems, possibly employing event-based and state estimation control schemes.
- **Detection** of Internal attacks, by verifying the outputs of the Controller.
- **Mitigation** and **Prevention** of cyber-attacks, by reconstructing the state of the Controller from a history of control signals.

Section 4.1 provides the model of a verifiable Controller. Section 4.2 defines an architecture to implement the proposed model and the detection of successful Internal attacks. Section 4.4 describes how the proposed architecture can use a state-preserving deployment-based mitigation and prevention mechanism.

The usefulness of the protection mechanism of ASiMOV in a real-world scenario derives from the assumption that the *security devices* can be made much more difficult to tamper, then the *field devices*. For economic, computational, or logistical constraints, it is typically impossible to equip all the devices in a CPS with the most robust security measures, e.g., physical surveillance, strong cryptography, and the likes. Indeed, the devices are typically too many, may have reduced computational capabilities, and some of them must work in real-time with the physical systems. As a difference, we assume it is much easier to protect few, powerful security devices, which are not required to work in synch with the controllers or the physical systems. To provide a concrete example, a single data center could execute the replicas of thousands of controllers, while employing the most robust security measures (e.g., against code injection, exploit O.S. vulnerability, etc..).

4.1 Model of ASiMOV

Section 4.1.1 motivates the needs for an exact model of the state of a Controller. In particular, we discuss the usefulness of ASiMOV in avoiding loss of accuracy in the detection of External attacks when a Control-Theoretic IDS (CT-IDS) is rendered network-based. Section 4.1.2 proposes the conceptual model of a verifiable

Controller. The model includes a definition of the State of the Controller, presented in Section 4.1.3.

4.1.1 Motivations for the replication of the control logic

ASiMOV describes the control logic (**CL**), and the verification logic (**CL**^{v}), as two systems which state include that of an execution environment. We aim to realize the operative condition in which **CL**^{v} disposes of an exact model of **CL**. Not only the mentioned condition enables an ideal detection accuracy against Internal attacks but - as already introduced in Sections 3.1 (requirement R3) and 1.3 (research question **RQ3**) - it may prevent decreasing of detection accuracy of an IDS due to a lack of coordination between **CL** and **CL**^{v}.

In a conventional CT-IDS, **CL** and **CL**^{v} are two dynamic time-invariant systems. We argue that, in the case of an event-based implementation of **CL**, and a network-based implementation of **CL**^{v}, the two instances require a sharing of information, i.e., what are the input sensing consumed by the two models during the time, what is their notion of the current time. Without such information, the usage of a CT-IDS is limited to the case of trivial control timing policies, e.g., an actuation is always produced using one sense, and the notion of current time is not used. In an event-based control scheme, without proper coordination between **CL** and **CL**^{v}, the CT-IDS may divert from its ideal dynamic trajectory, therefore not representing anymore the real cyber-physical system. In this case the detection accuracy of the CT-IDS may decrease (i.e., increase in false positives, negatives, or both).

In the proposed model for a Controller (Section 4.1.3), the equivalent of a transition function is named, to disambiguate from the control theoretic terminology, *update of the Execution State* function. Differently from a conventional dynamic system, such function takes as an input *a set* of timestamped control events (i.e., a piece of information of a knowledge).

The first considered cause for **CL**^{v}–**CL** state inconsistency is a difference in the notion of the current time, as measured by two different hosts. Control techniques such as state estimators (e.g., Kalman filters) may require as an input time-stamped control events (possibly remote in the past), and the notion of the current time. Besides, even assuming that the clocks of two hosts can be exactly synchronized, two instances of a transition function could be invoked at different time instants. The second cause for state inconsistency comes from the adoption of an event-based control scheme. In this case, **CL** adopts self-triggering policies for the production of actuation, that may depends on factors unknown to **CL**^{v} e.g., the time in which a sensing payload became available, the set of sensing events available at a specific time.

In what follows, we assume, without explicitly mentioning it, the presence of an IDS against External attacks (e.g., a Control-Theoretic IDS, or CT-IDS) inside **CL**. The mentioned possibility does not require any further mention, because whatever is the mode of communication between a CT-IDS and the Controller, we assume that there is a shared execution environment between the two (e.g., they are implemented as the same component **CL**).

System's Model	System's State	State Estimation
Dynamic system	Point in a state-space representation	Kalman filter
Finite state machine	Graph node of a state diagram	Markov model
Knowledge-based	Set of facts	Probabilistic logic

Table 4.1. Examples of System's models, System's state and estimation techniques implemented by a logic.

4.1.2 Model of a verifiable Control Application

Actuation, *sensing* and *parameters* are *control signals*. A Controller is an entity connected in a feedback loop with a System, i.e., the actuation and sensing signals are respectively output and input (input and output) for the Controller (for the System). An entity containing the payload of a control signal accompanied by a timestamp value is a *control event*. A Controller utilizes both control events and control logic to produce actuation for the System. We aim to define the model for a Controller capable of describing a broad spectrum of control schemes. In particular, we consider the following features required by a control scheme:

F1: the logic utilizes the notion of current time and prior knowledge made of control events to produce actuation for the System;

F2: the logic is subjected to time-varying parameters, e.g., the desired state for the System is time-varying.

The first two columns of Table 4.1 shows examples of models employed by a logic accordingly to feature F1 and the corresponding representation of a System's state in the System's model.

A control scheme utilizing a state space and state estimation, e.g., Kalman filter, may require feature F1. The proposed model envisages control schemes not strictly requiring feature F1. As an example, a conventional Proportional Integral Derivative controller (PID) reacts to sensing without considering the notion of the current time. However, the timestamps associated with sensing events are still required to enable the functionalities of ASiMOV.

ASiMOV defines a Control Application (CA) containing the following components:

- *Control Knowledge* (**CK**): a data store containing control events (e.g., timestamped sensing).
- *Control Logic* (**CL**): an executable software implementing a control scheme (e.g., a PID).

There are defined the following abstract entities.

Knowledge : a set of control events k , which models the data store CK. A control event can be a timestamped sensing, actuation, or parameter (e.g., desired

state for the System). The Knowledge contains the complete raw information available to the CA i.e., it is the history of events to be consumed, already consumed and produced by the CA. A query to the CK data store is modeled with a function q.

Algorithm : represents the stateless part of CL, which is logic without memory. Actuation payloads are produced based on the notion of the current time and a subset of the Knowledge $x \subseteq k$. The Algorithm corresponds to the program memory of software CL.

Execution State : represents the state of the execution environment hosting CL. Precisely, the Execution State models the stateful part of CL, which contains, for example, data-structures storing the values of the variables used by the Algorithm. The Execution State is the result of the processing of the Knowledge as done by the Algorithm in a sequence of iterations. A single iteration is modeled with a function c usable to produce a single actuation payload. The Execution State corresponds to the data memory of software CL.

State of the Controller : the combination of the Execution State and the Knowledge.

The production of actuation is a recurring operation carried out by a task of the CA, called *Actuate task* hereafter. An iteration of the Actuate task produces a single actuation payload a, and is defined by the following workflow:

1. a subset x of the Knowledge k (e.g., recent sensing events) and the notion of the current time r are provisioned to the Algorithm;

2. the Algorithm updates the Execution State (e.g., estimates the current state of the System) then computes an actuation payload.

Figure 4.1 (a) shows the abstract entities of a CA, and x, r input to CL used to produce actuation a.

Figure 4.1 (b) shows the interaction between CK and CL during the i-th iteration of the Actuate task, which spans the time interval $[t_i, t_{i+1})$, where t_i is the time in which the iteration begins, $i = 1, 2 \cdots$.

Depending on the adopted process to produce actuation (e.g., event-driven), an iteration is initiated by a notification event ("notif." in the Figure), which is fired when new events enter the Knowledge (event-driven), or periodically (time-driven). The sequence of operations leading to the production of a_i, i.e., the i-th actuation payload, is:

1. CL receives the set $x_i \subseteq k$ from CK i.e., CL selects a subset of the Knowledge k by means of query function q

2. CL receives the notion of current time r_i

3. CL uses x_i, r_i for updating the Execution State and producing a_i by means of function c

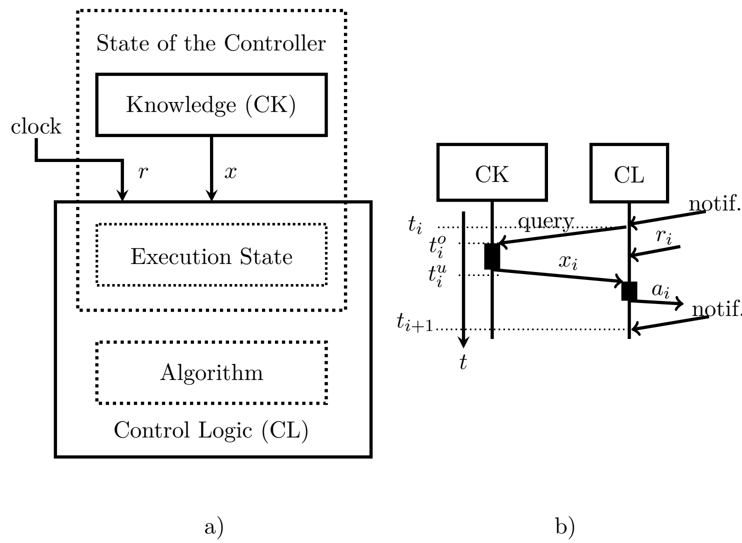


Figure 4.1. (a) The abstract entities (dotted boxes) and the components of a Controller, and the inputs to the CL component. (b) Interaction diagram between CK and CL.

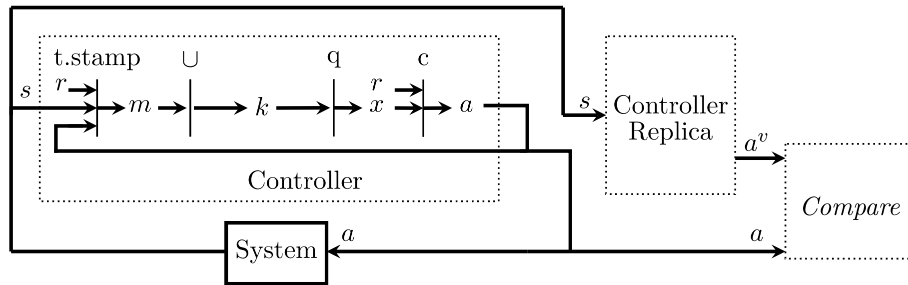


Figure 4.2. Dataflow to produce a control event from a control signal ("t.stamp"), to store a control event m in k, to select $x \subseteq k$ (function q), and to produce actuation a (function c). The outputs a of the controller and a^v of its replica are compared to detect cyber-attacks to the control logic. The diagram is a simplification that considers synchronized the system clock and the I/O ports of the Controller and its Replica. For the sake of simplicity, the parameter signal p is not shown.

The State of the Controller (i.e., Knowledge and Execution State) changes during the i-th iteration. We assume that:

- the selection criteria for obtaining x_i (i.e., any parameter for function q) depends solely on the Execution State at time t_i
- a_i and any changes to the Execution State depend exclusively on the Execution State at time t_i, and on the inputs to the Algorithm x_i, r_i

Figure 4.2 shows how a controller and its replica act and communicate. The Knowledge k is accessed to store (unite operator ∪ in the Figure) any control event m. In turn, control events m are timestamped version of control signals a, s, or p (the latter not shown in the Figure for the sake of simplicity). When an iteration

of the Actuate task begins, the function q selects x. In turn, x and the notion of current time r are used by the function c to update the Execution State and produce actuation a. The outputs a of the controller and a^v of its replica are compared to detect successful cyber-attacks to the controller.

The comparison $a \stackrel{?}{=} a^v$ is effective in detecting cyber-attacks depending on the assumptions relative to the synchronization of the controller and its replica. Ideally, the two are always synchronized over time. However, in ASiMOV the controller and its replica as asynchronous - because the controller never waits for the replica. Therefore, at a given time t the controller and its replica have, in general, a different Execution State. Moreover, the controller and the replica have two different notions of the current time (i.e., we do not assume clock synchronization). The diagram in Figure 4.2 represents a simplified version of ASiMOV, since it assumes clock and I/O synchronization between the internal component of the controller and its replica. Without the definition and implementation of appropriate sufficient conditions, the controller and the replica may produce different outputs in the absence of cyber-attacks. In what follows, we provide a model representing the time evolution of the State of the Controller, which is used in §4.3.1 to provide sufficient conditions for the verifiability of the controller's outputs using an asynchronous replica. In particular, we realize the following conditions, for each iteration of the controller (Actuate task) and the correspondent iteration in its replica:

- the two Execution State coincide
- functions c, q are provided with the same inputs x, r

4.1.3 State of the Controller

We define E the set of all possible Execution States and M the set of all possible control events. A control event $m \in M$ could carry the timestamped payload of an actuation, sensing, or parameter. $M^a \subseteq M$ is the set of possible actuation events, M^e its complement, i.e., sensing and parameter. $\mathcal{P}$ indicates the powerset operator. The following functions of time $t \in \Re$ describe the inputs for the controller:

$r(t) \in \Re$: a measure of current time

$g(t) \in M^e$: a sensing s or parameter p event

$a(t) \in M^a$: an actuation event a

The following functions describe the State of the Controller over time:

$k(t) \in \mathcal{P}(M)$: the events in the Knowledge

$e(t) \in E$: the Execution State

A selection of an element of the Knowledge is modeled by the function q defined as:

$$q : \{m \in M\} \times E \rightarrow \{m\} \cup \emptyset \tag{4.1}$$

$q(m, e)$ provides m or the empty set, depending on m and on the content of $e(t)$, the latter determining a selection criteria. By definition:

$$\forall t, \quad q(k(t), e(t)) \subseteq k(t) \cup \emptyset$$

We are interested in describing the State of the Controller only at the time instants t_i in which the i-th iteration of the Actuate task begins. For that reason, we consider the following discretized version of introduced entities:

$$
\begin{aligned}
r_i &\coloneqq r(t_i) \\
k_i &\coloneqq k(t_i) \\
e_i &\coloneqq e(t_i)
\end{aligned}
$$

Considering the initial time t_0 and the time interval:

$$\Delta_i \coloneqq [t_{i-1}, t_i)$$

Defining $g_i \in \mathcal{P}(M^e)$ the set of sensing and parameter control events entering the Knowledge during Δ_i:

$$g_i \coloneqq \bigcup_{t \in \Delta_i} g(t)$$

The Algorithm is modeled with the function $c(x, r, e)$:

$$c : \mathcal{P}(M) \times \Re \times E \to M^a \tag{4.2}$$

Function 4.2 provides the payload of an actuation event based on a set of control events (i.e., selected knowledge x), the notion of current time (i.e., r) and an Execution State (i.e., e). The update of the Execution State is the function $u(x, r, e)$:

$$u : \mathcal{P}(M) \times \Re \times E \to E \tag{4.3}$$

Function 4.3 represents each of the changes in e_i introduced by the Algorithm during the i-th iteration of the Actuate task, and it provides a new Execution State e_{i+1}. The update of the State of the Controller is:

$$
\begin{aligned}
k_1 &\coloneqq g_1 \\
x_i &\coloneqq q(k_i, e_i) \\
a_i &\coloneqq c(x_i, r_i, e_i) \\
k_{i+1} &\coloneqq k_i \bigcup g_{i+1} \bigcup \{a_i\} \\
e_{i+1} &\coloneqq u(x_i, r_i, e_i)
\end{aligned}
\tag{4.4}
$$

where e_1 is the initial Execution State i.e., when iteration 1 of the Actuate task begins, $\mathcal{S}_i \coloneqq [k_i, e_i]$ is defined as the State of the Controller when i-th iteration of the Actuate task begins.

The key idea of the proposed model is that an Execution State is reproducible using an initial Execution State and the sequence of inputs to the Algorithm i.e., the following mapping is realizable:

$$q, c, e_1, x_{1 \cdots i}, r_{1 \cdots i} \;\mapsto\; e_{i+1}$$

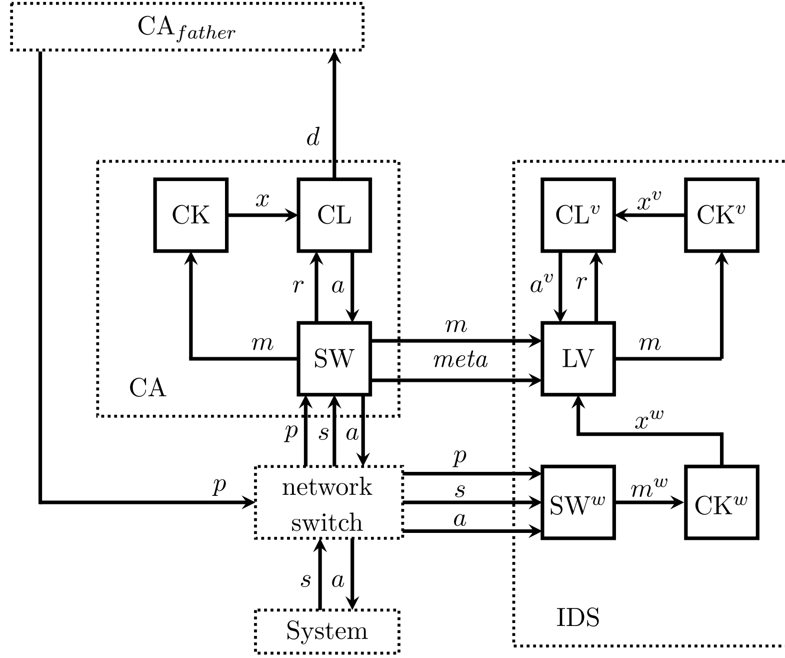


Figure 4.3. The architecture of ASiMOV and the related exchange of messages between its components. Symbols are defined in Section 4.1 and Section 4.3.

4.2 Architecture of ASiMOV

This Section proposes the architecture of a verifiable Controller, implementing the model of the previous Section. The architecture consists of two application: the CA (§4.2.1) and the IDS (§4.3). Section 4.3.1 details the detection of attacks confined to the Control Logic, while Section 4.3.2 describes, at a high-level of abstraction, the detection of unrestricted attacks.

4.2.1 Control Application

Figure 4.3 shows the architecture of ASiMOV. The CA and the IDS applications (dashed boxes) include respectively, the controller and the replica. In the Figure, each arrow is a data stream, where all data streams are reciprocally asynchronous. The streams from the CA to IDS, i.e., m and $meta$ (*C-I streams* of Figure 3.7), enable the IDS to reproduce the same subset of Knowledge x and the notion of current time r utilized by the controller at each iteration of the Actuate task. In particular, control events m carries the same payload and timestamp utilized by the controller, while $meta$ enables the replica to reproduce the same x_i and r_i of the i-th iteration of the Actuate task, and therefore to verify a_i without the need to inspect the state of the CA.

ASiMOV assumes the presence of a network device (e.g., a switch or Terminal Access Point device) replicating control signals p, s, a to the IDS. Our choice depends on the fact that a network device is assumed more secure than a computing device. A software-defined switch decouples a particular instance of a CA and the System [56],

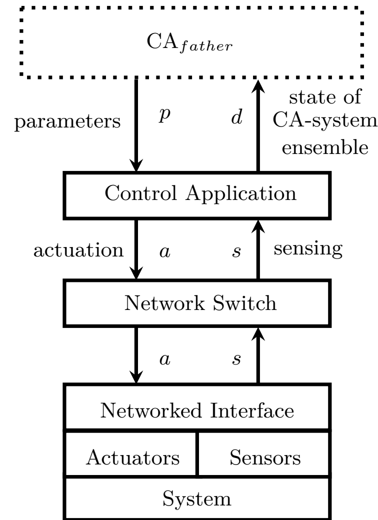


Figure 4.4. Architecture of a CA connected with a networked system and a father CA. The CA and CA_{father} are protected by 9 two distinct IDSs.

which allows performing a hot-swap of the CA as foreseen by our mitigation strategy. The data streams carrying p, s, a are employed by the IDS to verify the consistency of the data stream carrying m.

It makes sense to consider a hierarchical control scheme in which two CAs may be in a father-child relationship. In this case, CA_{father} produces actuation that is parameters p for CA_{child}, and CA_{father} receives as sensing a view d over the state of CA_{child}. In Figure 4.3, CA plays the role of a CA_{child}.

Figure 4.4 shows a service-oriented view of the architecture. The CA offers to a human user or a CA_{father} the following services: the realization of a control action driving the state of the System (or the CA_{child}) toward a desired state specified by parameters p; and the provisioning of a view d over the state of the System (or the CA_{child}).

4.3 Detection of Cyber-attacks

In what follows, a superscript ^{v} or ^{w} indicates an entity belonging to the IDS. The IDS consists of: replicas of CK, CL, and SW, here referred to as CK^v, CK^w, CL^v and SW^w; and the Logic Verifier (LV). The SW is configured to push the same control events to both CK and LV. The latter, in turn, forwards the messages to CK^v.

The IDS performs two tasks using control events in CK^v and CK^w:

- *Compare task*: LV compares the actuation produced by CL and CL^v to detect tampering of $m \in M^a$ i.e., actuation produced by CL (§4.3.1).

- *Match task*: LV verifies the consistency of $m \in M$ received by LV with the signals p, s, a received by SW^w i.e., to detect tampering of signals and events produced by SW (§4.3.2).

The two tasks correspond to two tests: the failure of one or more tests is interpreted as the consequence a tampering of the Knowledge, the Algorithm, or the Execution State of the controller.

4.3.1 Compare task

The detection of tampering to the actuation produced by CL is based on sufficient conditions for interpreting $a \neq a^v$ as a cyber-attack. The conditions are summarized in Table 4.2 and detailed in what follows.

Condition	Description
C1	the controller and the replica have the same initial Execution State
C2	the replica verifies the i-th actuation using the same r_i used by the controller during the i-th iteration of the Actuate task
C3	the replica verifies the i-th actuation using the same x_i used by the controller during the i-th iteration of the Actuate task (holds if C4-C6 holds)
C4	control events enter in k and k^v in the order of their timestamps
C5	the timestamp of events in k is unique
C6	at the time in which q selects $x_i^v \subseteq k^v$ on the Replica side, k^v contains an event having timestamp τ_i^M (i.e., the maximum timestamp in x_i)

Table 4.2. Sufficient conditions for an asynchronous detection of attacks to the logic

The Compare tasks of the IDS corresponds to the Actuate task of the CA. At the i-th iteration of the Compare task, the LV interacts with CL^v and CK^v to verify a_i, which is the payload of the actuation produced by CL during the i-th iteration of the Actuate task.

Being e_1 and e_1^v the initial Execution State of respectively the controller and its replica, we define the following conditions:

$$e_1 = e_1^v \tag{C1}$$

$$\forall i, r_i = r_i^v \tag{C2}$$

$$\forall i, x_i = x_i^v \tag{C3}$$

If conditions C1–C3 holds, and assuming that the controller and the replica implement identical functions c and q, from Eq. 4.4 we have:

$$\forall i, e_i = e_i^v$$

and therefore:

$$\forall i, a_i = a_i^v \tag{4.5}$$

The case where Eq. 4.5 is not true is interpreted as a cyber-attack.

We assume condition C1 always true, i.e., the components CL, CL^v, CK and CK^v are instantiated (i.e., deployed) to determine the same initial Execution State.

The fact that the controller and its replica are asynchronous complicates the achievement of conditions C2 and C3. Condition C2 requires that CL^v provides to function c the same value for current time r used by CL. In our proposal, C2 is achieved by attaching as metadata to a_i the value of current time r_i used by CL, which is then provisioned to CL^v. Condition C3 requires that CL^v provides to function c the same set of control events x_i used by CL, which in turn are selected by function q based on their timestamps. Complications are:

- a timestamp does not reflect the actual availability of control events in CK, i.e., after being produced an event must be transmitted from SW to CK, and rendered available by CK; and
- events x_i must be available in CK^v at the time in which CK^v performs the query (i.e., the function q); and
- CK^v must not provide to CL^v any extra-event from those appearing in x_i.

Condition C3 is achieved by attaching as metadata to a_i the timestamp of the most recent event found in x_i, which is then used for determining x_i from k_i^v on the replica side. In particular, stream *meta* (Figure 4.3) is sufficient to determine x_i and k_i^v. In what follows we detail how condition C3 is achieved utilizing *meta*.
Condition C3 is rewritten as follows. Defining $\tau(m)$ the function providing the timestamp of event m. Defining $f(t, x)$ the function providing all the events in set x having timestamp less or equal to t. Defining:

$$\tau_i^M \coloneqq \max_{m \in x_i} \tau(m)$$

we redefine x_i of Eq. 4.4 for the Replica as:

$$x_i^v \coloneqq f(\tau_i^M, q(k_i^v, e_i^v)) \tag{4.6}$$

Introducing the following conditions:

$$\text{Control event } m \text{ enters in } k \text{ and } k^v \text{ in the order of their timestamps} \tag{C4}$$

$$\forall i,\ \tau_i^M \text{ is unique} \tag{C5}$$

$$\forall i,\ \exists\, m \in k_i^v \mid \tau(m) = \tau_i^M \tag{C6}$$

we have that condition C3 holds if C4–C6 holds and Eq. 4.6 is used. In words, conditions C4–C6 implies that k_i^v contains at least all the events in x_i. Eq. 4.6 determines that events with timestamp more recent than τ_i^M are removed from x_i^v.

Stream *meta* (Figure 4.3) carries r_i and τ_i^M. Condition C4 holds assuming that SW uses a reliable order-preserving communication protocol for each stream

established with the IDS (i.e., m and $meta$ streams).

Condition C5 is achieved by designing the SW with a single FIFO queue for the timestamping and forwarding service, which, assuming a host's system clock sufficiently accurate, ensures a unique timestamp for each control event.

Condition C6 is implemented as follows. Function q is implemented both by datastore CK (query server) and CL (query user). CK takes as a parameter the time interval $\Delta_f = (t_s, t_e]$, and serves a selection of control events having timestamp in Δ_f, where Δ_f is a selection criteria determined by an Execution State. CK has different behavior depending on the value of t_e: in case $t_e = \infty$, CK serves the query as soon as possible (*unbounded* query) using the events currently available in the Knowledge. On the contrary, in case t_e is finite the query is served only when a event with timestamp t_e is available (*bounded* query). In an event-driven production of actuation, the Algorithm is supposed to produce an actuation payload as soon as possible after new control events enter the Knowledge. Therefore, in this case, CK produces unbounded queries since they guarantee maximum reactivity. The Algorithm typically requires to receive only control events it has not already seen. Therefore, to produce a_i, the Algorithm requests all events more recent than the most recent event seen in the previous iteration, i.e., the query selection criteria interval is:

$$\Delta_f = (\tau^M_{(i-1)}, \infty) \tag{4.7}$$

which determines an unbounded query.
Bounded queries are used to realize condition C6 on the replica side. In particular, LV intercepts the queries of CL^v (i.e., Eq. 4.7) and substitutes t_e to produce the selection interval:

$$\Delta_f = (\tau^M_{(i-1)}, \tau^M_i] \tag{4.8}$$

The newly obtained bounded query of Eq. 4.8 is provisioned to CK^v, which implements Equation 4.6.

Being the CA and the IDS executed in distinct hosts with possibly different computer architectures, differences in the result of arithmetic operations involving real numbers may lead to false positives in the detection. Assuming such differences estimable [47], Eq. 4.5 can be redefined as: $h(a_i, a^v_i) \leq \epsilon_v$, where h provides a distance (e.g., Euclidean distance) between the payload of actuation events, and ϵ_v is arbitrarily small.

4.3.2 Match task

This Section discusses the detection of unrestricted attacks to the host executing the controller, e.g., the SW may also be tampered. In this scenario, an intruder may send a tampered actuation to the System but a genuine actuation to LV. In order to detect such kinds of attacks, the IDS uses CK^w and CK^v for verifying the consistency of the streams involving SW and the network switch (lower part of Figure 4.3). In particular, the network switch is configured to push the payloads to both SW and

SW^w. Component SW^w produces control events for CK^w with different timestamps of those assigned by SW. However, the events and their timestamps preserve their relative order compared to those of SW. The LV verifies the consistency between CK^v (i.e., k^v) and CK^w (i.e., k^w).

Control Events in k^v determines a sequence for each kind of events (e.g., actuation) for a total of three sequences. Similarly, three sequences are determined by events in k^w. An attack is detected if at least one of the following conditions C7–C9 do not hold:

The payload of control events is consistent. (C7)

The timing of control events is consistent. (C8)

The presence of control events is consistent. (C9)

Condition C7 is verified as follows. Given a sequence of a certain kind in k^v (e.g., sequence of actuation), and the sequence of the same kind in k^w, the two sequences are identical despite the value of the timestamps (in case k^v and k^w have a different number of elements, the extra elements are not considered).

Condition C8 is verified as follows. Considering a sequence S of events of a certain kind in k^v. We have:

$$\sum_{m \in S} \|\tau(m) - \tau(m')\| < \delta \tag{4.9}$$

where m' is the element in k^w corresponding to m, and $\delta > 0$ is an arbitrary threshold for the cumulative difference of timestamps of corresponding events. The value of δ can be properly assigned by considering the average difference of system clocks of SW and SW^w, the time duration of S, and the delays and jitter of the network links employed by CA and IDS, we assume known.

Condition C9 consists in verifying the discrepancy between the number of elements in the sequences of a certain kind in k^v and k^w. In particular, the difference of sizes of two sequences should not exceed a certain threshold, which can be properly assigned by considering the jitter of the network links used by SW and SW^w.

4.4 Mitigation and Prevention of attacks

ASiMOV adopts an immutable deployment model for the microservices, which improves system security since malicious changes introduced by an intruder to a specific microservice instance are unlikely to persist past redeployment [113]. The usage of cloud technologies, e.g., Container as a Service platforms [54] and orchestration layers [18] (e.g., Kubernetes) enables to actuate a cyber-attack mitigation and prevention strategy based on a redeployment of a compromised CA. In [15], the authors describe a generic microservice architecture by the mean of a set of images of containers and network parameters. In particular, a specific application made of microservices (in our case, the CA) determines a set of declarative primitives

dynamically provided to an orchestrator, that is DMAN in Figure 5.2 (e.g., an application that includes a container cluster manager, such as Kubernetes). By adopting the above approach, the ASiMOV mitigation and prevention strategy are reconfiguration policies executed by DMAN, i.e., to stop, start and reconfigure images of the container(s) involved in a cyber-attack. The redeployment of a container can be done on its original host or in a different host in order to prevent a new attack in the short term. Typically, such redeployment takes a few seconds (2-5) to start its execution.

The model proposed by ASiMOV enables to reconstruct the State of a Controller $\mathcal{S} = [k, e]$ of a CA into a new instance of a CA (named CA-Heal, or "healed CA" as in Figure 3.4).
The *controller state migration* process is defined as the following mapping:

$$\bigcup_{t \in \Delta_c} m, meta \ \mapsto \ k^h, e^h \tag{4.10}$$

where the left argument of the mapping are the control events and metadata from a CA to the IDS during time interval Δ_c (see Figure 4.3) and k^h, e^h are respectively the Knowledge and Execution State of a healed CA at the end of the migration process. The migration operates at the application level and therefore does not propagate any tampering possibly occurring at the level of the execution environment (e.g., tampering of the OS). The proposed mechanism offers the opportunity not only to mitigate detected attacks but also to prevent attacks not yet detected. In particular, we argue that periodic state migration may prevent tampering to the execution environmment of CA (e.g., malware stealthily waiting to participate in a coordinated attack involving multiple CAs).
The state migration process recreates the State of a Controller of a CA into a fresh instance of a CA (named CA-heal). During the operation, the IDS transfers control events and metadata (as received by a CA) to an application called CA-Heal-temp, which, at the end of the process, is the CA-heal.

4.4.1 Controller state migration

This Section presents the state migration process, which is further detailed in Section 5.2.1 of Chapter 5 "Implementation". In the Controller state migration process, an application named CA-Heal-temp receives a history of control events (and associated metadata). CA-Heal-temp is made by the same components of the IDS. During the migration, the State of the Controller $\mathcal{S}^h = [k^h, e^h]$ of CA-Heal-temp updates upon receiving from IDS a Knowledge (that is a set of control events previously produced by CA) and metadata (that are associated to the actuation control events). At the end of the state migration process, CA-Heal-temp is reconfigured as a CA (we name CA-Heal to disambiguate from CA) and connected with the System. At the end of a controller state migration process, the execution environment of a healed CA is a sanitized version of that of CA.

The steps of the state migration process realizing Eq. 4.10) are:

1. a fresh instance of CA-Heal-temp is deployed, i.e., instances of components CL^h, CK^h, SW^h, and LV^h are started. The components are configured to

cooperate in a state migration process (i.e., additional tasks of those of Figure 5.1).

2. LV^h is provisioned with k^v and metadata from the IDS. LV^h forwards k^v to CK^h. Concurrently, components CL^h, CK^h, and LV^h interact as in the Compare task of IDS, until LV^h consumes k^v and metadata. The migration process discards the actuation produced by CL^h since the purpose is only to update the Execution State of CA-Heal-temp.

3. Once CL^h has recomputed each of the actuation in k^v, the state migration is completed. The CA-Heal-temp application is reconfigured as a CA i.e., components CL^h, CK^h, SW^h are reconfigured and connected as a CA, and LV^h is stopped. The obtained application, named CA-Heal, becomes the new CA for the System.

A prototype implementation was employed to validate the described state migration process.

Chapter 5

Implementation

This Chapter details the implementation of the ASiMOV architecture regarding its main functionalities, i.e., to realize direct control over a System while verifying the outputs of the Control Logic (Section 5.1). Besides, the deployment-based mitigation and prevention mechanism are described (Section 5.2).

5.1 Implementation of ASiMOV

The proposed implementation of ASiMOV adopts a microservice architecture [92]. The components CL, CK, SW, LV in Figure 4.3 are microservices. Hereafter, the terms "microservice" and "component" are used interchangeably. The interaction among microservices is driven by message events, where a message event occurs at the production or the consumption of a message, e.g., a message could carry a control event m, or the notion of current time r. A message broker employing a publish-subscribe model manages the transmission of messages. Each microservice is deployed into an application container [75]. Our choice is motivated by the fact that containers have a small footprint, are portable, and fast to deploy.

In what follows, firstly, we describe the tasks carried on by the CA and IDS (Section 5.1.1). Then, we describe the choreography's roles of the microservices to produce and verify actuation payloads (Section 5.1.2). Finally, we illustrate the design of the core routine for all the microservices, that defines how they react to message events (Section 5.1.3).

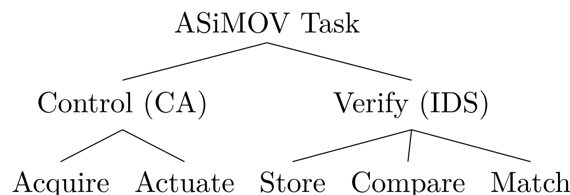


Figure 5.1. Classification of the tasks and sub-tasks performed by ASiMOV's applications CA and IDS.

5.1.1 Implementation of the CA and IDS models

An instance of the ASiMOV architecture executes two tasks: the *Control task* and the *Verify task*, as shown in Figure 5.1. For the sake of simplicity, the Figure does not show the tasks relative to the control state migration process.

The microservices of the CA carry out the *Control task*, which consists of the following concurrent sub-tasks:

- *Acquire* task:
  - CA creates control events m, i.e., CA timestamps control signals p (parameter), s (sensing) and a (actuation);
  - CA stores m into CK and forwards them to the IDS.
- *Actuate* task:
  - CA utilizes m from the CK to produce a for the System;
  - for each produced a, CA produces a metadata message for the IDS.

The Acquire task is active independently of the Actuate task, i.e., control signals are always timestamped, stored, and forwarded to the IDS. In an event-driven production of actuation, the Acquire task activates the Actuate task, i.e., an iteration of the Actuate task begins upon acquiring s or p. In the proposed implementation, the Algorithm performs a selection of events as in Eq. 4.7.

The microservices of the IDS carry out the *Verify task*, which consists of the following concurrent sub-tasks:

- *Store* task:
  - IDS stores any control event m and metadata, transmitted from CA to CK^v;
  - LV stores any received a (i.e., the payload of m that are actuation) and metadata from CA into local FIFO queues to be used in the Compare task.
- *Compare* task:
  - IDS uses the knowledge in CK^v, and the metadata in the local FIFO queues of LV, to reproduce the execution environment operations performed by the CA. The described process is realized using a trusted instance of the Control Logic (i.e., CL^v). The so produced i-th actuation is compared with the i-th actuation as received by the CA, as described in Section 4.3.1.
- *Match* task:
  - IDS attempts to match m from the CA with the control signals p, s, a forwarded by the network switch, as described in Section 4.3.2.

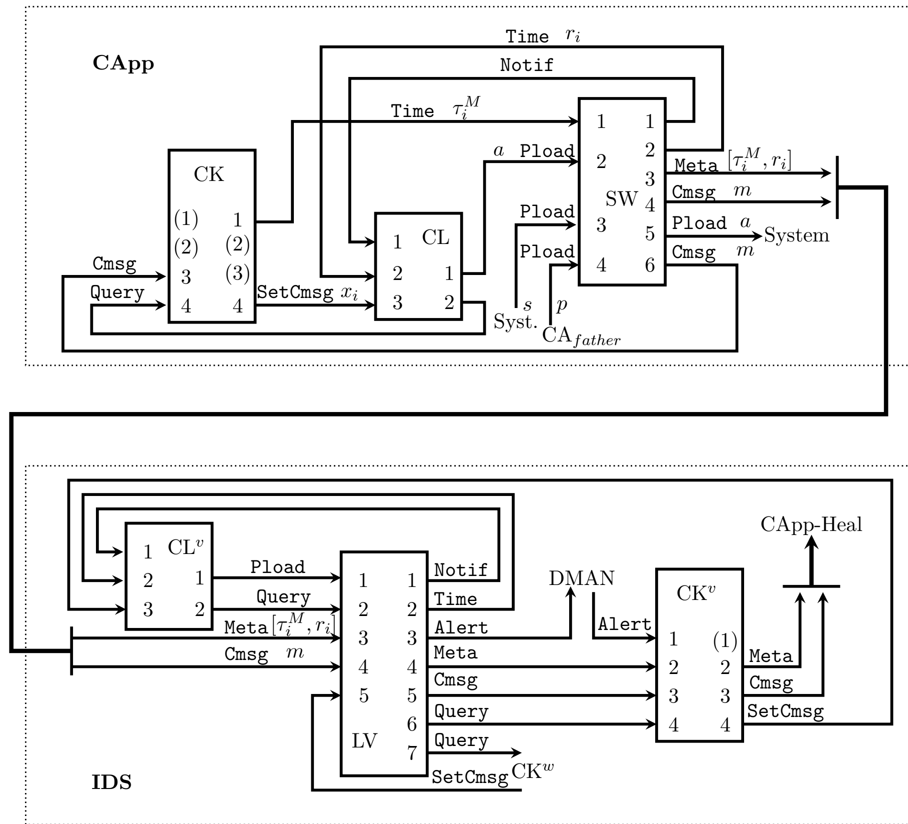


Figure 5.2. Detailed view of the Control Application and IDS.

Class	Attributes	Description
`Time`	`Double value`	implements a value of time (e.g., r_i, τ_i^M)
`Pload`	`Double value`	implements the payload of a control event s, a or p

`Cmsg`	`Pload pload`
	`Time tstamp`
	`String type`
		implements a control event m
		`type` is `'act'`, `'sen'`, or `'par'`

`Notif`	`Boolean value`	notification of an event
`SetCmsg`	`<Set>Cmsg set`	a set of `Cmsg` (i.e., x_i)
`Query`	`Time start` `Time end`	time interval Δ_f used to select x_i within an iteration
`Meta`	`Time rI` `Time tauM`	meta-data relative to the production of an actuation, i.e., $[r_i, \tau_i^M]$

Table 5.1. Classes that extend the `BusIO` abstract class and that implement the communication messages

5.1.2 Microservices Choreography

This Section details the Actuate and Compare tasks, i.e., to produce and verify actuation. Figure 5.2 represents the CA and IDS applications, where each of their

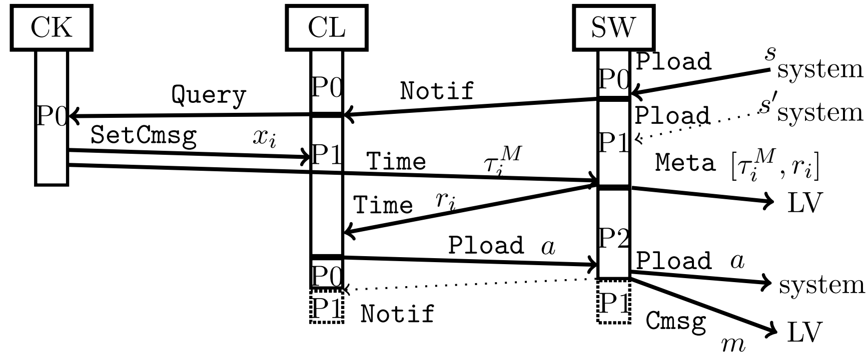


Figure 5.3. Interaction diagram for an iteration of the Actuate task.

components is a block with input and output ports (respectively on the left and right of a block). For simplicity, the CK^w is not shown, as it is required for the Match task only (i.e., LV uses port `IN` 5 and `OUT` 7 to verify conditions C7–C9 of Section 4.3.2). In the Figure, the DMAN on the IDS side is an application managing the deployment of the CA (e.g., a container cluster manager like Kubernetes), which is used for mitigating or preventing cyber-attacks, as detailed in Section 4.4.1.

Each component has a number of `inSize` and `outSize` ports. In Figure 5.2, a port is indicated inside the parenthesis when a component does not need to use such port, e.g., CK of CA does not use ports `IN 1` and `2`, which are instead utilized by CK^v in IDS.

Each connection between two ports is a communication channel consisting of a FIFO queue managed by a message broker. Condition C4 of Section Section 4.3 requires an order-preserving delivery of messages. In a message broker like RabbitMQ there are proper ways to guarantee the order of messages within subscriptions.

Messages are extension of the abstract class `BusIO`, and are described in Table 5.1. A control event m is transported by message `Cmsg` having three attributes: the *type* i.e., sensor, actuator or parameter; the *payload*, containing a value i.e., sensor reading, actuation command or parameter value; and the *timestamp*. As an example, a message produced by SW at time 1.25 transporting a sensor reading with value 3.4 is: $m = [\texttt{sen}, 1.25, [3.4]]$.

In what follows, we describe the microservices choreography during an iteration of the Actuate and the Compare tasks.

Actuate task

Figure 5.3 is the interaction diagram for an example of an iteration of the Actuate task. During a single iteration, each component passes through a certain number of internal synchronization phases, starting from phase P0. When all components are in phase P0 the CA is idle: CL waits for a `Notif` and SW waits for `Pload` at `IN` 3 or 4 . SW produces `Notif` upon consuming `Pload` (s in Figure 5.3), then SW enters in phase P1. The timestamping and forwarding of s is not shown in the Figure as it is part of the Acquire task. Upon consuming a `Notif` message, CL produce a

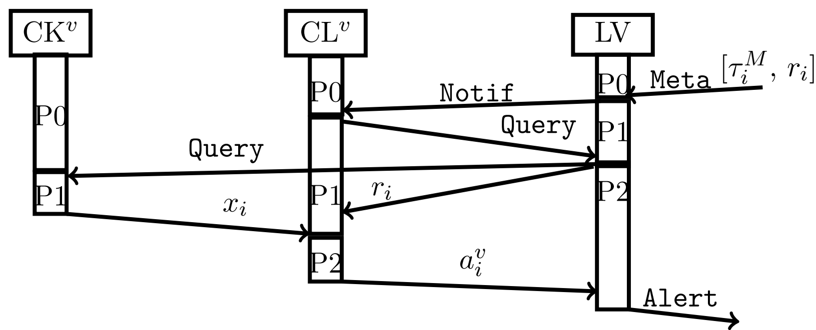


Figure 5.4. Interaction diagram for an iteration of the Compare task.

`Query` message for CK. Then, CL enters in phase P1 i.e., waits for x_i and r_i. When queried, CK produces a `SetCmsg` message containing x_i i.e., CK computes function q. We assume CL performs a query with selection criteria $\Delta_f = (\tau^M_{(i-1)}, \infty)$ (Eq. 4.7), which determines an unbounded query as defined in Section 4.3.1. Therefore, x_i is the set of control events entered in the Knowledge after $x_{(i-1)}$ was served, which in this example consists merely in s. Moreover, CK produce a `Time` message τ_i^M on `OUT` 1 for SW (i.e., maximum timestamp in x_i). This ends the role of CK for the current iteration.

Upon receiving τ_i^M SW accesses its host's system clock to produce the notion of current time for CL (i.e., `Time` message r_i). Then, SW produces `Meta` message for LV (i.e., $[\tau_i^M, r_i]$). Finally, SW enters in phase P2 i.e., it waits for an actuation payload from CL.

CL updates its Execution State (e.g., estimates the current state of the System) and compute actuation `Pload` a_i upon consuming x_i and r_i, then returns to phase P0.

During the Acquire task, SW timestamps a_i to produce m for both LV and CK (the latter transmission is not shown in Figure 5.3). We emphasize that the timestamp assigned to a_i is in general different (i.e., successive) from r_i. This may be an advantage in terms of performance of the control system since, in a successive iteration of the Actuate task, a state estimator will see a timestamp for a_i accounting for the time taken by CL to produce a_i.

In case that during the current iteration additional `Pload`(s) was consumed in the Acquire task (e.g., s' in Figure 5.3), SW immediately goes back to phase P1 and produces an additional `Notif`. This continues until CL have consumed any new payload i.e., until:

$$\max_{\forall i, x_i} \tau_i^M = \max_{\forall m \in k_i} \tau(m) \tag{5.1}$$

When Equation 5.1 holds the Actuate task is suspended (i.e., SW return to phase `P0`).

Compare task

This section describes how conditions C2 and C3 of Section 4.3.1 are achieved. We assume C1 true, e.g., the components CL, CL^v, and CK, CK^v are always deployed

Listing 5.1. Pseudo-code of the `microcycle`

```
1  this \\ a datastructure containing persistent state
2  List<BusIO>[] inFIFO \\ array of lists of messages
3  while True:
4
5     \\ COMPUTE JOB
6     BusIO[] in = new BusIO[inSize] \\ init as Null
7     FOR i = 1 TO inSize: \\ unconsumed first-out inputs
8        in[i] = inFIFO[i].read() \\ previously reiceved
9     BusIO[] out = new BusIO[outSize] \\ init as Null
10    Bool[] consumed = new Bool[inSize] \\ init as False
11    compFunct(this, in, consumed, out)
12    removeConsumed(consumed,inFIFO)
13
14    \\ COMMUNICATE JOB
15    transmit(out) \\ push to message broker
16    reiceve(inFIFO) \\ pull from message broker
```

using the same immutable container images.

Figure 5.4 provides the interaction diagram for an iteration of the Compare task. On the IDS side, instances of the components CK and CL (i.e., CK^v and CL^v) interface with LV. During the Store task (not shown in Figure 5.4) LV forwards the received `Meta` and `Cmsg` (i.e., at ports `IN` 3 ,4) to CK^v. Additionally, LV stores the same `Meta`, and the `Cmsg` that are actuation, into local FIFO queues to be consumed during the Compare task. Upon disposing of an actuation and the correspondent `Meta`, LV initiates an iteration of the Compare task by producing a `Notif` for CL^v. As described in Section 4.3.1 LV modifies the unbounded query of CL^v (Eq. 4.7) into the bounded query of Eq. 4.8. Finally, LV provides CL^v with the time r_i contained in the metadata.

Upon receiving the bounded query, CK^v may need to enter in phase `P1`, i.e., to wait for a message with timestamp τ_i^M, we assume eventually provisioned by the Store task. Being CL^v provisioned with the same r_i and x_i used by CL, conditions C2 and C3 of Section 4.3.1 holds.

LV compares the actuation `Pload` a_i^v produced by CL^v with the corresponding actuation `Cmsg` received by CL (Eq. 4.5). The result of the comparison may produce an `Alert` at `OUT 4`, which triggers the mitigation mechanism.

5.1.3 Microcyle routine

Microservices CL, CK, LV, and SW execute a routine, named `microcycle`, implementing the reaction to message events according to the specific component. The function `compFunct` is part of the `microcycle` and characterizes the behavior of a component. All the microservices execute the same `microcycle` and differ for the implementation of `compFunct`. Listing 5.1 is the pseudocode for the `microcycle`.

A microservice maintains the following persistent variables (lines 1,2 of Listing 5.1):

- `this`: data structure characterizing the state of an instance of a component (i.e., variables of Table 5.2);

C.	State Var	Description
SW CL CK LV	Int phase	value used for a choreography-based coordination of components
CK	SetCmsg k	history of Cmsg k (Knowledge)
CK	List<Meta> km	FIFO of Meta maintained only in CK^v and used only for state migration
CL	SetCmsg x	set of Cmsg for the current iteration of the Actuate task
CK	Query pendQ	query pending for current iteration of the Actuate or Compare task
SW CL	Time mRecC	greatest timestamp among Cmsg ever consumed by CL (Eq. 5.1)
SW	Time mRecP	greatest timestamp among Cmsg ever produced by SW
CL	params	data-structure containing run-time values of the control scheme e.g., desired state, estimated state of the System
LV	List<Meta> metas List<Cmsg> acts	FIFO queues used in the Compare task

Table 5.2. State variables and related components. Non-primitive datatypes are defined in Table 5.1.

- inFIFO: an array of lists of BusIO objects. The i-th array cell maintains the FIFO queue relative to the i-th input port of the component.

The microcycle perpetually executes two jobs: *Compute* and *Communicate*. The *Compute* job (lines 6-12) consists of the following steps:

1. initializes the following volatile arrays, i.e., variables valid within a single cycle: in, containing the "first-out" element of inFIFO; out, containing the messages (if any) produced in the cycle; and consumed, boolean flags signaling what are the inputs ports from which messages was consumed in the cycle (if any);

2. invokes function compFunct characterizing the computation of a particular micro-service (line 11). The function does side-effect on variables this, out and consumed, based on the content of this and inFIFO;

3. invokes function removeConsumed which, for each element True in consumed, remove the first-out messages from the correspondent inFIFO queue.

The *Communicate* job (lines 15-16) consists of the following steps:

1. transmit: push asynchronously all messages contained in out array to the message broker;

2. reiceve: pulls one or more messages (if any) from the message broker, which are appended to the corresponding inFIFO queues.

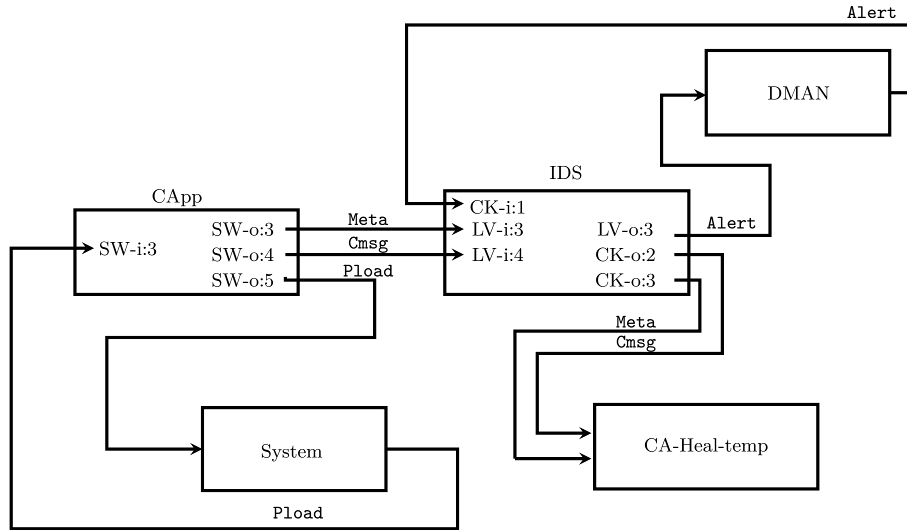


Figure 5.5. High-level view of the four applications CA, IDS, DMAN and CA-Heal-temp. The latter two applications have a role in the mitigation and prevention strategy.

As described in Listing 5.1 the `microcycle` is perpetually executed i.e., the `reiceve` function is part of a polling cycle toward a message broker. This is an oversimplification introduced for the sake of clarity. In the real implementation, the `microcycle` sleeps until there are new messages to process.

The pseudo-code of the `compFunct` for CL, CK, LV and SW is presented in the Annex, Listings 9.1, 9.2, 9.3 and 9.4 respectively. Table 9.5 (in the Annex) describes the functions used in the pseudo-code. Tables 9.1 to 9.4 provides I/O the ports definition and description for all the components.

5.2 Mitigation and prevention of cyber-attacks

Figure 5.5 provides a high level of abstraction representation of ASiMOV during normal operations. The Figure is a higher level of abstraction view of Figure 5.2, showing DMAN (Deployment Manager of Section §3.3.2) and CA-Heal-temp (Section 4.4.1). DMAN realizes a deployment-based mitigation and prevention mechanism. CA-Heal-temp is a temporary application, utilized for the state migration process (introduced in §4.4.1) At the end of the process, CA-Heal-temp is reconfigured as a CA i.e., it is a healed instance of CA (i.e., CA-Heal in Fig. 3.5).
When the DMAN receives an `Alert`, it makes sure that an instance of CA-heal-temp is ready to receive the State of the Controller. The state migration is triggered when DMAN provides back an `Alert` to the SW^v in the IDS, which in turn starts provisioning a history of `Meta` and `Cmsg` to CA-Heal-temp.

5.2.1 Controller state migration

The state migration introduces additional functionalities to the already described components CK, SW, LV, while CL remains unchanged. In particular, there are

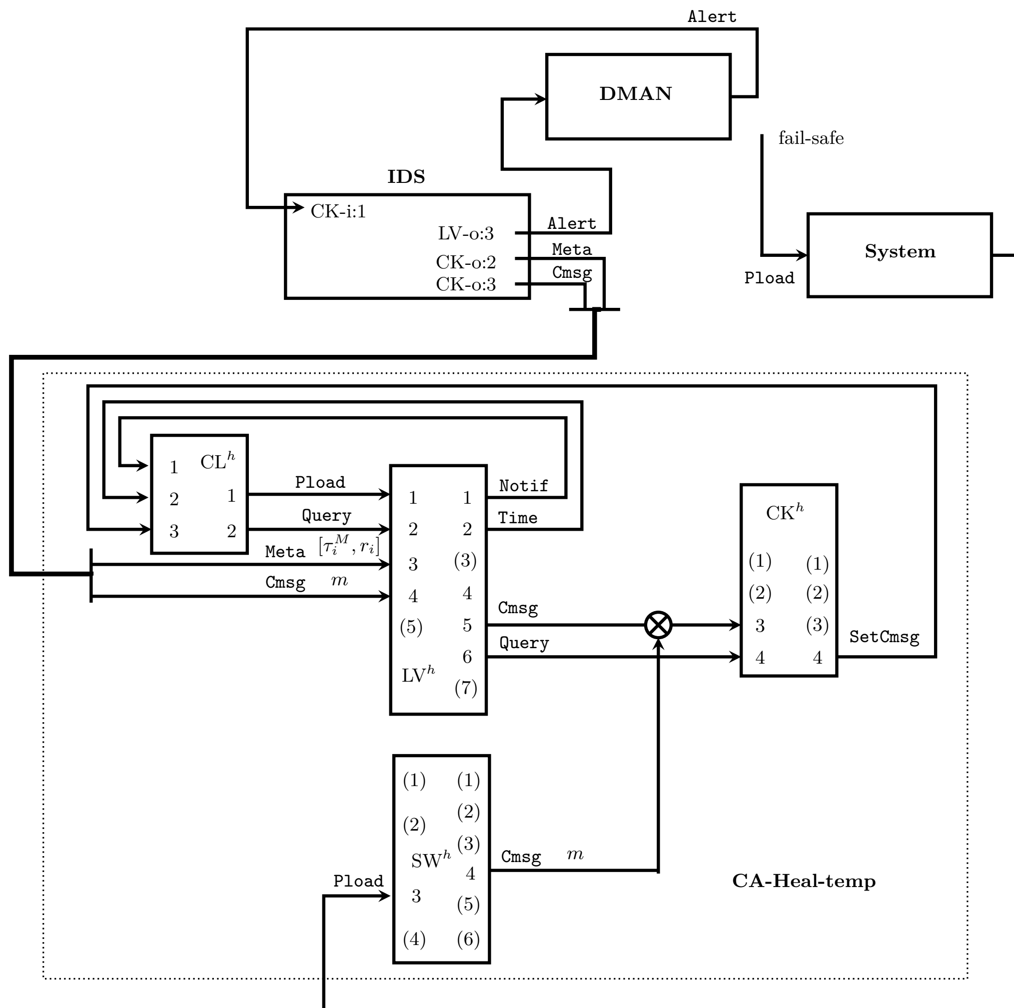


Figure 5.6. CA-Heal-temp is a different application then CA, and is used to update an execution environment (Step 2 of the state migration process) accordingly to a Knowledge with associated metadata, provided by the IDS. At the end of the process, CA-Heal-temp is reconfigured as a CA, which receives the so obtained State of the Controller.

additional phases in the microcycles of the components that work in a similar way to those described in Section 5.1.2. In what follows, we detail how ASiMOV realizes the state migration using a choreography-based interaction between components of CA-Heal-temp and IDS. In the description, we consider the case of a mitigation of a cyber-attack. The same process with minimal changes is adopted for a prevention process.

Figure 5.6 shows the communication channels established between components during the state migration process. When an attack is detected, the IDS provides an `Alert` message to DMAN. Therefore, the compromised CA is immediately disconnected from the System. We assume that after its disconnection from CA, the System is governed by an emergency control scheme, i.e., a (sequence of) pre-recorded

commands.
We elaborate in more detail the three steps for the migration of the state introduced in Section 4.4.1.

- Step 1: DMAN starts the orchestration operations to prepare a new instance of CA-Heal-temp, containing components CL^h, LV^h, SW^h and CK^h connected as in Figure 5.6. During the process, the sensing signals from the System could be stored in a proper component within the Security devices for later usage, or discarded. As soon as SW^h and CK^h instances in CA-Heal-temp are ready, the sensing signals begin to be routed to `IN` 3 of SW^h of CA-Heal-temp. For simplicity, we assume that there are no changes of parameters during the state migration, hence `IN` 4 of SW^h is disconnected. The SW^h and CK^h carry out the Acquire task with the fresh data from the System.

- Step 2: when all the components in CA-Heal-temp are ready, DMAN sends an acknowledge `Alert` to IDS, i.e., CK^h `IN` 1 . The `Alert` triggers the provisioning of the entire content of control events and metadata contained in the IDS (CK^v `OUT` 2 , 3) to CA-Heal-temp (LV^h `IN` 3 , 4). Therefore, the CK of CA-Heal is provisioned both with fresh sensing from the System and with historical sensing, actuation and parameters from the IDS (i.e., circled cross at `IN` 3 of Figure 5.6). The LV^h, CK^h, and CL^h carry out the Compare task. In particular, LV^h triggers CL^h to produces actuation while it modifies its query accordingly to metadata as done with a replica on the IDS side. The produced actuations are discarded. While CL^h updates its Execution State, the conditions C1–C3 of Section 4.3.1 must apply. For this purpose, the system clock used by SW^h of CA-Heal must be in advance of that used by CA, SW. Under this assumption, no fresh messages from the System can be included in the set x_i result of a query made by CL^h.

- Step 3: Once the entire historical Knowledge is processed by CL^h, DMAN sends reconfigures the components of CA-Heal-temp as in CA (i.e., the upper part of Figure 5.2). Finally, the LV^h is removed, and CA-Heal-temp become CA-Heal (i.e., it is connected with the System).

During Step 3 of the controller state migration, CA-Heal-temp is reconfigured as a CA, i.e., there are modification to the structure of the application during its execution. Dynamic changes may lead to an application's state inconsistency [53]. One way to avoid inconsistencies is to suspend each component in a consistent state, before proceeding with a reconfiguration. The remaining of this Section details how ASiMOV avoids inconsistencies during Step 3 of the controller state migration, employing a choreographic approach in which also DMAN participates.

The components of CA-Heal-temp communicate with DMAN trough dedicated channels transporting acknowledges messages (not shown in Figure 5.6). A dummy message is a message with an irrelevant but unambiguously distinguishable payload, which only serves the purpose of synchronizing two components.
Component CK^v (on the IDS side) appends to the history of control events transmitted to LV^h (`IN` 4 of LV^h) a dummy `Cmsg` denoted with m'. The message enables LV^h to distinguish the last `Meta` of the history, denoted as $[\tau_i^{M'}, r_i']$. Upon reiceving

m', LV^h waits for the query with selection interval $\Delta = (\tau_i^{M'}, \infty)$ from CL^h. Upon receiving the query, LV^h is informed that CL^h completed the update of its internal state (i.e., it consumed the whole history).

At this stage, CL^h waits for `SetCmsg` and has a consistent state (i.e., CL^h is suspended). LV^h procedes with the following operations:

- 1: LV^h produces a dummy `Query` for CK^h
- 2: LV^h produces an acknowledge for DMAN, telling that LV^h and CL^h are suspended, then LV^h suspends

Upon receiving the acknowledge from LV^h, DMAN produces an acknowledge for SW^h. Upon receiving the acknowledge from DMAN, SW^h proceeds with the following operations:

- 1: SW stops consuming `Pload` from System
- 2: SW produces a dummy `Cmsg` for CK^h, denoted m'', then SW suspends

During its execution, CK^h awaits for the following two messages: the dummy `Query` from LV (step 1 of operations of LV^h), and the dummy `Cmsg` m''.
At the time in which CK^h receives the two cited messages, there is the following situation:

- CL^h, LV^h, and SW^h are suspended in a consistent state
- all the message queues used for intra-host communication within the application are empty
- there may be `Pload` messages accumulated for the port `IN` 3 of SW^h, that are the messages produced by System and unconsumed since SW is suspended (i.e., step 1 of operations of SW^h)

Component CK^h procedes with the following operation:

- CK^h produces an acknowledge for DMAN, telling that CK^h is ready to receive a new configuration, then CK^h suspends

At this stage, all the components of CA-Heal-temp application are suspended in a consistent state, and their reciprocal message queues are empty. Upon receiving the acknowledge from CK^h, DMAN reconfigures each of the components as described in Step 3 of the controller state migration (i.e., LV^h is stopped). Upon being reconfigured, CL^h, CK^h, and SW^h produce a final acknowledge for DMAN. After producing the acknowledge for DMAN, SW^h still does not consume any `Pload` from the System. After receiving the acknowledge from all the components, DMAN sends a further acknowledge to SW^h, which finally starts consuming `Pload` messages from the System and hence producing `Notif` for CL^h.

Simulation Parameter	Value
Delay: CA to/from IDS hosts	600 ms.
Delay: CA host to/from System	200 ms.
Delay: IDS host to System	1000 ms.
Delay: message broker transmission delay (both CA and IDS)	2 ms.
Time required for any microcycle of any microservice	0.0001 ms.
System sensing messages produced every second	10
System sensing noise gaussian standard deviation	0.1
System actuator max. absolute value (i.e., saturation)	3
Controller proportional gain	2

Table 5.3. Simulation parameters used to produce Figure 5.7

5.3 Dynamics of an attacked System

In this Section, we show the results of a simulated successful Internal attack to the CL component. More specifically, we show the effects of the attack and the mitigation mechanism on the state of a physical System and on the Execution States of a CA and a CA-Heal-temp.

As an explicative example, we consider a System's model of a tank of liquid, i.e., a first-order system. In the considered model, the System is the ensemble of the physical liquid level, together with sensors (e.g., the measure of liquid level) and actuators (e.g., valves to enter or to eject liquid). The state of the System is described with a single real number (i.e., the level of liquid). The control system adopts a Proportional feedback control scheme. In particular, being s the current state of the System, $\hat{s}$ the desired state, then the produced actuation is $a = K(\hat{s} - s)$, where $K > 0$ is the proportional gain parameter.
The ASiMOV microservices are implemented as continuous-time Matlab Simulink blocks using the SimEvents library and connected as in Figure 5.2.

Table 5.3 provides the parameters utilized in the simulation. Figure 5.7 shows the time evolution of the state of the System (black line in Figure). The desired state for the System (i.e., *setpoint*) is 1, and the System's state value at the initial time is -1. The control action drives the state of the System trough discrete actuation commands. A red circle in the Figure represents an actuation received by the System at a certain time.

The sensor values from the System are subjected to Gaussian noise. Therefore CL implements a simple state estimator consisting of a weighted average of the past 3 sensor readings. The System's state estimation is an example of information contained in the Execution State (i.e., not in the Knowledge).

In the simulation, we model an attack to the CL component. At the time $t_M = 8$, an intruder starts producing a sequence of malicious actuation payloads. In this example of an attack, the actuator is saturated to its minimum possible value, i.e., -3. At the time t_D, the attack is detected by the IDS. Hence the compromised CA is disconnected from the System, and a fail-safe actuation command is provided to the System, i.e., an actuation command with payload value 0. A fail-safe command is an example of an additional mitigation strategy that can be employed while waiting for the preparation of CA-Heal.

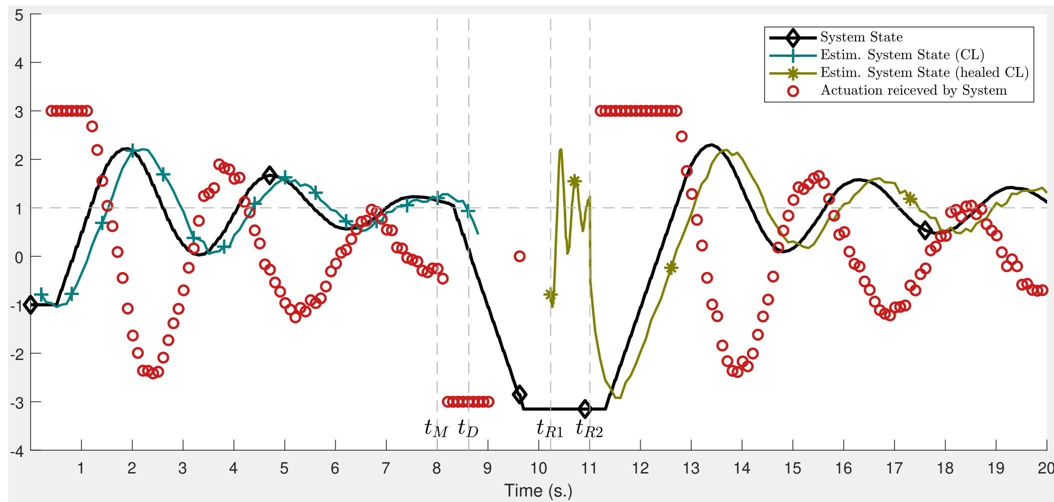


Figure 5.7. Simulated attack to the Control Logic controlling a first order System, and the mitigation. The attack moves the System's state away from the setpoint. After the attack is detected and mitigated, the setpoint is reached again by the System.

The state of the System is under the intruder's control during the time window spanning from the reception of the first malicious actuation command to the reception of the fail-safe actuation command. Several factors influence the length of this time window, like the delay between CA host and IDS, and the message broker communication delay - that we analyze in Chapter 6.

Interval $[t_D,t_{R1})$ models the time required for the mitigation mechanism to instantiate CA-Heal. During time interval $[t_{R1},t_{R2})$ the mitigation mechanism consists of a controller state migration from the IDS to CA-Heal, as described in Section 4.4.1. The Figure shows the System's state estimation as done by CA-Heal (yellow line) during the process. The Execution State of CA-Heal updates exactly as the attacked CL during interval $[0, t_M)$. The update process of CA-Heal takes less time then the CA since the only limitation is the computation capabilities of the host executing the CA-Heal. From time t_{R2} CA-Heal restores a correct control action over the System.

5.4 Deployment-based Mitigation and Prevention

An application (CA, IDS) is operative after its components are operative. In turn, a component (microservice) is immediately operative after its deployment. ASiMOV architecture adopts the principle of immutable deployment. Therefore, the information required for the deployment is immutable, and entirely contained in an instance of the *deployment model for a CPS* (see UML model of Figure 3.6, Section 3.3.1). The usage of cloud platforms may bring benefits to a CPS, in particular when the "cyber" processes require much computational capacity. The biggest obstacle to the adoption of virtualization technologies in a CPS is the possible conflict between the sharing of computational resources and the ability to meet time deadlines, which is a fundamental aspect in real-time control systems.

A Smart Factory is an example of a CPS where the adoption of virtualization

technologies is nowadays in an advanced stage. Considering the control scheme of a Smart Factory, at the highest hierarchical levels (compared to lower levels):

- controllers need more computational capacity, e.g., for estimating the state of a complex systems determined by a sub-tree of the hierarchical controllers;
- controllers produce actuation with lower frequencies;
- controllers are less adapt to control physical systems due to the latency.

In case the computational capabilities of the device(s) running a cloud platform are opportunely dimensioned (i.e., concerning the number of CAs, and the frequency with which the CAs must produce actuation), the interference between different CAs does not lead to the violation of their deadlines.
Relatively to direct control, the usage of cloud platforms is adapt to the highest levels of a hierarchy (i.e., not for the control of physical systems), while at lower levels each of the CA may need to use a dedicated device, such as a microcomputer. Relatively to the verification of ASiMOV, a cloud platform could contain the IDS(s) of all the controllers of a CPS (i.e., also the replica of the physical system's controllers). In ASiMOV the verification does not influence real-time control, because the former is decoupled with the latter (requirement R4 of Section 3.1).

5.4.1 Deployment Manager

This Section describes the internal architecture of the Deployment Manager (DMAN), introduced in Section 3.3.2. The DMAN is an application responsible for the deployment of CAs on the field devices and of IDSs on the security devices. Depending on the network infrastructure, the DMAN may also be responsible for (part of) the configuration of the networking devices, e.g., to establish site-to-site VPNs, to configure software-defined switches, or routers enabling the required communication channels between applications. For the sake of simplicity, we omit considering the configuration of the network.

We distinguish between two different kinds of computing devices in a CPS:

- *Pool of resources*: realized by a cloud platform (e.g., OpenStack) using powerful computing devices. A pool of resources consists of multiple physical devices to be shared between multiple CAs (or multiple IDSs).
- *Microcomputer device*: a computing device that is not part of a Pool of resources, to be assigned to the control of few, or only one System(s).

Depending on the particular virtualization technology chosen for the deployment of applications, appropriate interfaces, configuration, and orchestration tools have to be employed. Based on work [15], the rest of this Section describes how the DMAN manages the deployment of a containerized implementation of the ASiMOV architecture.

The DMAN can be seen as a controller (i.e., autonomic computing) having the goal of maintaining deployed the instance of a deployment model for a CPS. Sensing

of DMAN is probing signals defined at different layers of computer architecture (from the Hardware Abstraction Layer to the Application layer) from the devices of an infrastructure. An alert from an IDS is also sensing. Actuation of the DMAN is commands and controls aiming to maintain available the control scheme of a CPS also in the case of cyber-attacks. To accomplish this task, the DMAN includes the following service components:

- **IaaS Interface**: an IaaS interface for cloud platforms i.e., Apache LibCloud
- **Configurator**: a service integrating logic with the Master of a Configuration Management Engine (CME) operating at the OS level, i.e., SaltStack
- **Orchestrator**: a service integrating logic with the Master of a Container Cluster Manager (CCM), i.e., Kubernetes
- **Deployment Controller**: a service integrating logic with the previously defined IaaS Interface, Configurator, and Orchestrator services.

Figure 5.8 shows the relation between the components of the DMAN. The logic in the Configurator traduces a high-level representation of the desired state defined at the OS level (entering arrow "setMinions") into the desired state for a CME Master. Practically speaking, the Configurator customizes templates configuration files (i.e., YAML files) and provision them to a SaltStack Master, which is internal to the Configurator itself. Similarly, the Orchestrator traduces a high-level representation of the desired state, defined at the level of Kubernetes objects, into sequences of invocations to the Kubernetes Master API.

The following conditions render the DMAN operative:

- there is one pool of resources made of field devices (field pool) , and a pool of resources made of security devices (security pool).
- DMAN can instantiate VMs in each of the two pool of resources;
- each of the microcomputer devices is a Minion of the CME Master.

The two introduced pools of resources should be highly isolated e.g., they should belong to two different cloud tenants or, ideally, are realized by two different cloud platforms.

At the time of the deployment, the DMAN performs the following operations, which are preliminary to the actual deployment process.

- On each pool of resources:
  - DMAN starts, trough the IaaS interface, a certain number of VMs in the field and security pools. The provisioned VM *image* consists of a lightweight OS, which is pre-configured to realize a Minion of the CME cluster, having its Master in the Configurator component;
  - DMAN configures (trough the Configurator) each of the newly instantiated VMs as a Node of the CCM cluster, having its Master in the Orchestrator component.

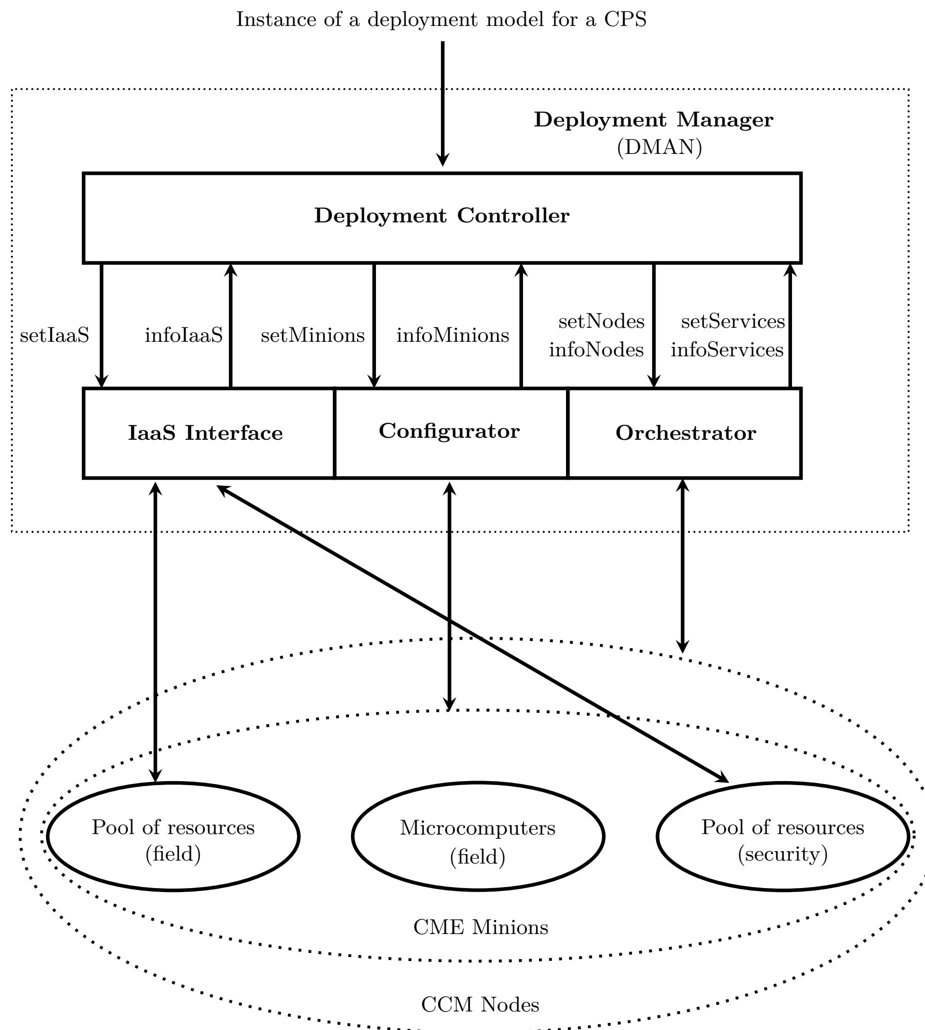


Figure 5.8. Components of the Deployment manager, their internal API invocation, and the relation with computational resources of a CPS.

- On each Microcomputer device:
  - similarly to what done on the field pool, the DMAN configures the microcomputer devices as Nodes of the CCM cluster.
- For each newly created Node of a CCM cluster (both on pools of resources and Microcomputer devices):
  - DMAN assigns (trough the Orchestrator) appropriate Kubernetes Labels to the Nodes. The labels allow distinguishing if a field or a security device hosts is hosting the Node. The Nodes receives one or more Kubernetes labels.

The preliminary operations described above are executed in steps, where at each step the Deployment Controller waits for a desired state (e.g., IaaS state, OS

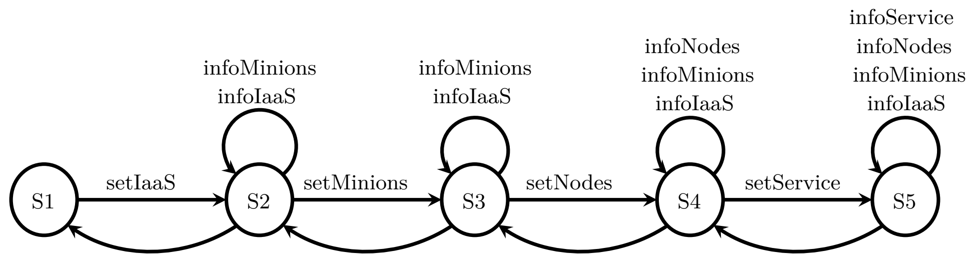


Figure 5.9. Finite State Automata representing the state of the Deployment Controller during the deployment operations, starting from an empty CME cluster.

state) to be reached. Figure 5.9 represents the internal state of the Deployment Controller as a Finite State Automata. Table 5.4 describes the APIs offered by the internal components and utilized by DMAN in the described process. The preliminary deployment operations start in state S1, and the transition between S1 and S2 represents a request for the IaaS interface to instantiate VMs (i.e., the invocation of setIaaS API). In S2, the Deployment Controller waits until the desired IaaS state is reached (i.e., repeatedly invokes infoIaaS API). Also, all the Minions (each one in VMs) have to join the CME cluster (i.e., repeatedly invokes infoMinions API). During the waiting time, the state of the Configuration Controller may retreat to S1, e.g., previously started VM ceases to run unexpectedly.

Once all the Minions are ready, the Deployment Controller transits to S3 by providing a configuration for each of the Minion. The configuration aims to realize a Node of a CCM cluster in each of the Minion. In S3, the Nodes are waited to join the CCM Master while continuing checking that the desired state at IaaS level remains reached.

Once the Nodes are ready, they are configured (transition between S3 and S4). In particular, Labels are assigned to the Nodes. Once the Nodes are configured, the preliminary operations for the deployment are completed (state S4).

At state S4, there is a unique CME cluster made of Minions, which are both system-level VMs (deployed in the pools of resources) and Microcomputer devices. Moreover, there is a unique CCM cluster made of Nodes spanning in both the pools of resources and the Microcomputers (see the lower part of Figure 5.8). The Nodes are labeled accordingly to their localization (field or security). Nodes on Microcomputers has a label with the ID of the device.

We highlight that for increased isolation, and therefore security, multiple clusters should be employed instead of one. The clusters could be made to communicate through a federation mechanism.

The deployment of CA(s) and IDS(s) consists of publishing a set of Kubernetes Services, represented by transition S4–S5 of Figure 5.9. The invocation of setService API is done accordingly to the instance of a deployment model provided as an input to the DMAN. For each Service, the correspondent Kubernetes *spec* parameter enforces the deployment constraints arising both from the ASiMOV architecture and from user-level constraints. The deployment constraints from the ASiMOV architecture define a Pod for each CA and IDS. The Labels previously assigned to the

Component	**Offered API Functions**	**Description**
IaaS Interface	infoIaaS(`pId`)	Provides the state of IaaS Pool of resources identified by `pId`
	setIaaS(`img`,`pId`,`vmId`)	Starts VM image `img` into pool of resources `pId`, provisioning the file `vmID` which acts as an ID for the VM
Configurator	infoMinions	Provides the set of available SaltStack minions, their `vmId`, and if they met their desired OS state or not
	setMinion(`c`, `minId`)	Assigns to minion `minId` the configuration `c` (desired OS state)
Orchestrator	infoNodes	Provides information about all the Nodes of the Kubernetes cluster. This includes diagnostic information from the Kubernetes *node status*
	setNode(`nId`,`lab`)	Assigns a `spec` to the Kubernetes Node `nId`. In particular, specifies the set of labels `lab` for the Node
	infoServices	Provides information about all the Services running on a Kubernetes Cluster. This include the Kubernetes *pod status* information for each Pod exposed as a Service
	setService(`dMod`,`p`)	Publish a Service consisting of a Pod accordingly to the instance of deployment model `dMod`. The parameter `p` enforces the deployment constraints (both deriving from the ASiMOV architecture, and user-level)

Table 5.4. Components of the Deployment Manager

Nodes, united to the value of the *spec* parameters provided by the Orchestrator, force the CAs to be deployed (and orchestrated) on field devices, while the IDSs stay on security devices. User-level constraints allow, for example, to force the deployment of a specific CA on a particular subset of microcomputer devices, having a low latency with the assigned physical System, which enables to reduce the control loop delay to acceptable levels. For the same need, a CA can be forced to be deployed on a Node that reserves a certain number of CPU-Cores. The attribute `dConstraints` of the CA-Class in a deployment model contains the user-level constraints defined for each CA.

Once the deployment is completed, the Pods realizes the CAs and IDSs of a CPS. Therefore, the Deployment Controller remains in state S5 of Figure 5.9 and repeatedly checks the Kubernetes *status* (i.e., infoService API) to check that all the Pods are "healthy". While in S5, each of the desired state defined for the layers of

computer architecture (Hardware Abstraction Layer, OS, Application) are reached. The state remains in S5 until at least one of the desired states is not reached anymore. As already mentioned, if a VM stops unexpectedly, the state retreat from S5 to S1. For simplicity, we do not describe the possibility of random faults or other unintentional malfunctions.

As envisioned by ASiMOV, for mitigation and prevention purposes, the DMAN receives an `Alert` determining a specific CA to heal. In this case, the DMAN appropriately changes the desired state of one or more layers of computer architecture. Consequently, the Deployment Controller exits the state S5. How the desired state changes according to an attack is part of a choice that depends on the available resources, on level of security required, and on time limits in play. For example, the new desired state could involve creating a new instance of a CA in the same Node. This process is quick, but it leaves the new instance in the same device, where the attack occurred. For increased security, a new Node should be created, possibly in a different Pool of resources, or on a different Microcomputer.

When a CA has to be destroyed, the state migration described in Section 5.2.1 requires that the components of the CA-Heal are opportunely configured to receive a verified Knowledge. For this purpose, the components of CA-Heal receive an opportune configuration through the Kubernetes *Init Container* option, which realizes the provisioning of a configuration for a container VM.

Chapter 6

Performances analysis

This Chapter deals with the problem of delay in control systems, and in particular to the delays introduced by ASiMOV. Any implementation of a Controller makes the *control frequency* and the *control loop delay* not ideal, which may compromise the performances and stability of a control system. It is appropriate to quantify the effects of implementing a control scheme using ASiMOV, as opposed to a conventional monolithic controller. Section 6.1 study trough simulation what are the effects on the performances of a simple control system when the control loop delay increases. Section 6.2 provides a model estimating the effects of the adoption of ASiMOV, i.e., *i*) the increase in control loop delay, *ii*) the decrease of the maximum control frequency, and *iii*) the detection delay.

Since ASiMOV utilizes a verification process that is asynchronous to the control loop, a successful attack may leave a physical System under the control of the attacker for a specific time interval. As introduced in Section 1.2.4, the estimation of the detection delay enables: *i*) to determine the possible damage coming from a successful attack, and *ii*) the usage of optimization-based techniques aiming in preventing damage. For these reasons, the model introduced in Section 6.2 estimates the ASiMOV detection delay as a function of the networking and computation capabilities of an infrastructure.

6.1 Effects of delay on control systems

In this Section, we assess the impact of a **control loop delay** T on the performances of a control system. In particular, it is considered the *responsiveness* of a control system, which is the ability of the Controller to promptly bring the state of the System close enough to the desired state. This Section aims to clarify the effects of T on a generic control system. The presented results are a contribution to the understanding of control systems to the readers not familiar with Control Theory.

The presence of a control loop delay may reduce the performances or compromise the stability of a control system. In some cases, these side effects can be alleviated by merely tuning the Controller's parameters to optimal values for the introduced delay. If this is not enough, it may be necessary to change the adopted control scheme. There are already many control-theoretical solutions to the problem of a time-invariant delay T. When the delay is time-varying, the analysis of its effects is

considerably complicated. From a practical point of view, a time-varying delay can be made time-invariant through the introduction of a buffering mechanism. However, in case the delay has high variability, it is not easy to choose an output timing for the buffer, which guarantees the constant presence of output data.

To measure the responsiveness of a control system, we consider the *settling time*, i.e., the time taken for the controller to get within a tolerance $\pm U$ of a new equilibrium value, without subsequently deviating from it by that amount. We consider the control system introduced in Section 5.3 i.e., a tank of liquid governed by a Proportional controller. In terms of a dynamic model, such a tank is a first-order system, having transfer function $1/s$ in the Laplacian domain. The controller provides actuation $a = K(\hat{x} - x)$ where $\hat{x}$ is the desired state, x the actual state, and $K > 0$ the proportional gain (i.e., a parameter for the controller).

The provided simulations are conducted using an ideal mathematical model. Specifically, we model in Matlab Simulink the transfer function of the closed loop system:

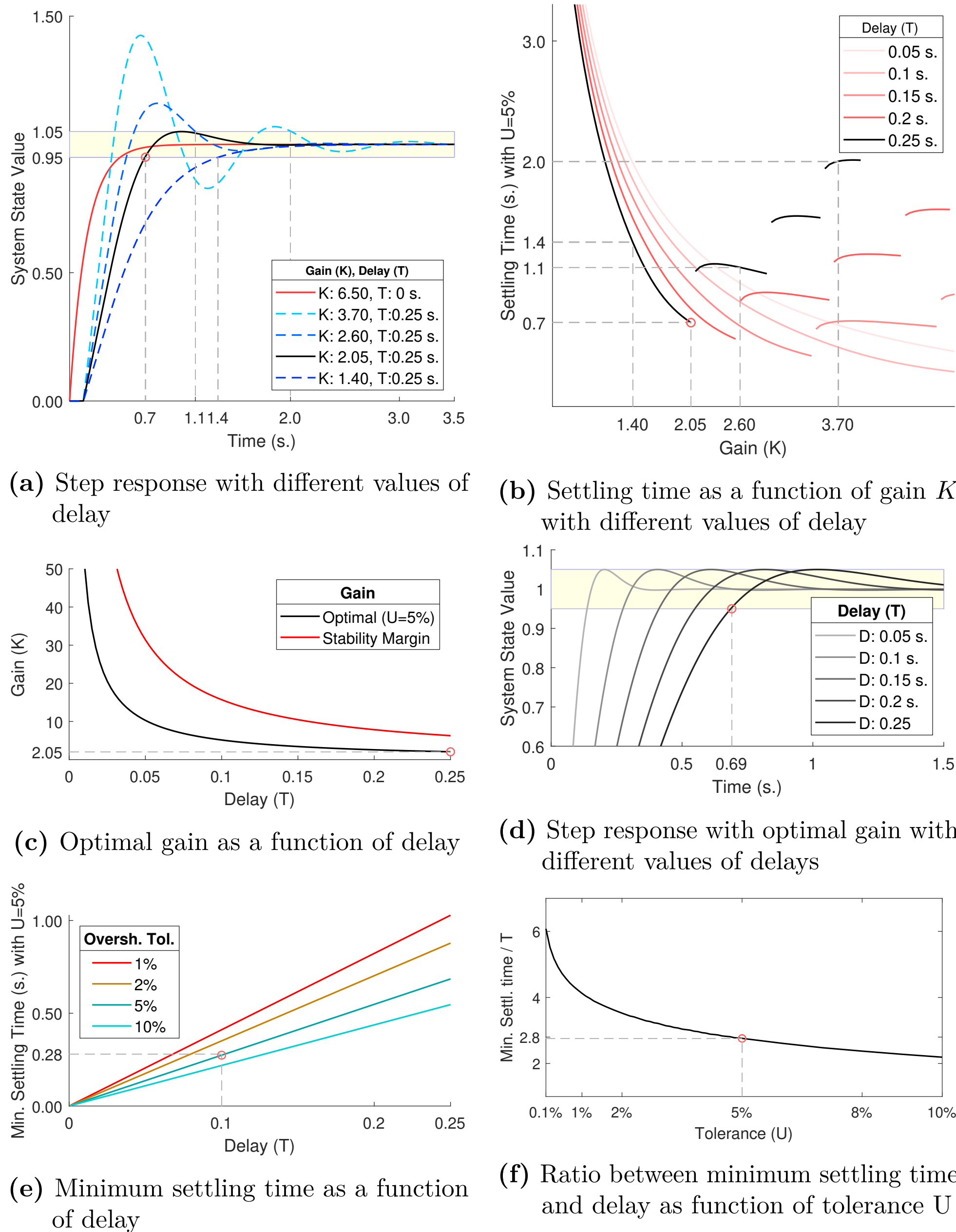
$$H(s) = \frac{P(s)}{1 + P(s)}$$

where

$$P(s) = K \exp(-Ts)\frac{1}{s}$$

and $T > 0$ is the value of a constant control loop delay.

The simulations describe the performance of the control system relative to the transient response to a *step input*, i.e., we study the settling time when the desired system state (liquid level) is brought from the value of $\hat{s} = 0$ to $\hat{s} = 1$. Figure 6.1a shows in red the step response in absence of delay with $K = 6.5$. Delay introduces oscillations in the dynamics of the System. Indeed, there are introduced in the characteristic equation of the control system an infinity of *poles*, and stability is guaranteed if $K < K^* = 2/(\pi T)$ [97] where K^* takes the name of *stability margin.* Among the set of $K < K^*$, there exist an optimal gain, which minimizes the settling time for a given tolerance of U.
Figure 6.1a shows the step response with a delay of $T = 0.25s$ and different values for K, among which $K = 2.05$ is the optimal for a tolerance of $U = 5\%$. Figure 6.1b shows the settling time as a function of the gain K for different value of the delay T and $U = 5\%$, in which the optimal gain is a minimum. Increasing the delay increases the minimum settling time, which practically means the controller response is slowed. This phenomenon can be observed in Figures 6.1c and 6.1d. Figure 6.1c shows in black the optimal gain, and in red the stability margin as functions of the delay. Figure 6.1d shows the step response using optimal gains for different values of the delay. Equally spaced delays result in equally spaced settling time. As Figure 6.1e shows, the minimum settling time is linear with the delay and decrease when the tolerance U increase. As reported by Figure 6.1f, the settling time decrease exponentially with the increase of tolerance, despite the value of the delay. Indeed the plot is valid for any value of T. Shortly speaking, given a specific delay, we can reduce the settling time by increasing tolerance. Hence low precision control systems can still work when the CA is deployed on a cloud platform. Conversely, high precision control system requires a controller with smaller delay, e.g., a fog-edge deployment model.



(a) Step response with different values of delay

(b) Settling time as a function of gain K with different values of delay

(c) Optimal gain as a function of delay

(d) Step response with optimal gain with different values of delays

(e) Minimum settling time as a function of delay

(f) Ratio between minimum settling time and delay as function of tolerance U

Figure 6.1. Settling time as a function of gain K with different values of delay

6.2 Delays introduced by ASiMOV

This Section analyzes the performances of ASiMOV compared to a monolithic implementation of a controller that does not include any intrusion detection and response mechanisms. The performance metrics used for assessing the performance of ASiMOV are:

- the *actuation delay* T^{Act}, that is the time required for the production of an actuation payload upon consuming sensing;

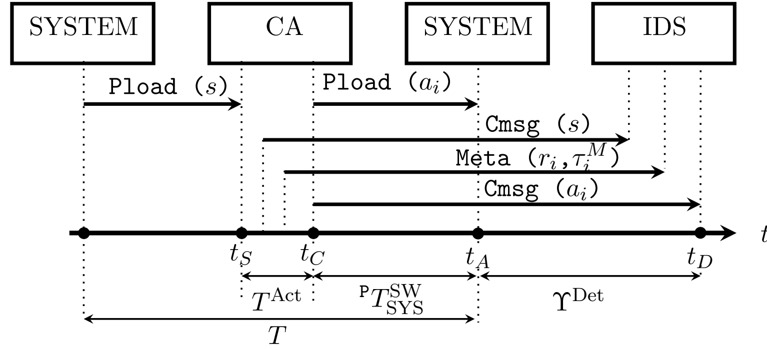


Figure 6.2. Communication occurring between different hosts. Before starting the production of a_i, CA transmits other messages to the IDS. Therefore, the computations of a_i (controller) and a_i^v (replica) may be concurrent.

- the *detection delay* Υ^{Det}, also referred as detection latency [101], that is the amount of time required to detect an attack.

The analysis of the actuation delay allows quantifying how ASiMOV increases the time required to produce actuation and hence how it decreases the maximum achievable rate of actuation produced per second (max *control frequency*). The cited side effect may lead to a decrease in performance or stability compared to a monolithic controller.

The analysis of the detection delay is part of the answer to the research question **RQ3** (Section 1.3). The amount of time required to detect a successful attack is information usable at run-time by optimization-based control techniques to keep an appropriate distance from damaging states (Section 1.2).

6.2.1 Delay model

Let us define:

- the *control loop delay* T is the time between the production of sensing and the receipt of a corresponding actuation, as seen by the System in an event-driven production of actuation;
- the *maximum control frequency* f_M is the number of sensing per second which can be processed to produce the same number of actuation;
- the *intra-host communication delay* T^*_{CA} (T^*_{IDS}) is the time required for a message broker to transmit a message between any microservices within the CA (the IDS);
- the *inter-host communication delay* $T^*_{\text{CA-IDS}}$ ($T^*_{\text{CA-SYS}}$) is the time required to transmit a message between any microservice of the CA host and any microservice of the IDS host (the System).

As specified in the remaining of this Section, the intra-host and inter-host communication delays can be considered independent of a specific message type, i.e., the `BusIO` class (§5.1.1).

We denote with ${}^{\texttt{mes}}T^{\text{send}}_{\text{rec}}$ a communication delay, where "send", "rec" and "`mes`"indicate respectively the sender, the recipient and the initial letter of a `BusIO` class e.g., ${}^{\texttt{P}}T^{\text{SW}}_{\text{SYS}}$ is the communication delay for the SW to transmit a `Payload` to the System. There are introduced the following time instants:

- t_S: sensing `Pload` s initiating the i-th iteration of the Actuate task is received by SW;
- t_C: actuation a_i is produced by SW for the System;
- $t_A = t_C + {}^{\texttt{P}}T^{\text{SW}}_{\text{SYS}}$: the actuation a_i reaches the System;
- t_D: assuming a_i tampered, the time in which the attack is detected;

where $0 \le t_S \le t_C \le t_A$, and $t_D \ge 0$.

Figure 6.2 shows the defined time instants, intervals, and inter-host exchanges of messages.

The *actuation delay* is defined as:

$$T^{\text{Act}} = t_C - t_S \tag{6.1}$$

that is the reaction time of a controller i.e., the time required for the production of an actuation payload upon consuming sensing. T^{Act} contributes to the control loop delay T.

The *additional actuation delay* is denoted as $\hat{T}^{\text{Act}}$, that is the increase in T^{Act} (and therefore in T) introduced by ASiMOV, compared to a monolithic implementation.

We denote with symbol "Υ" the difference between two time instants, i.e., possibly a negative value. The *detection delay* Υ^{Det} is the time required by ASiMOV to detect the first tampered actuation received by the System due a cyber-attack, and is defined as:

$$\Upsilon^{\text{Det}} = t_D - t_A \tag{6.2}$$

where t_D and t_A are relative to a tampered actuation. Considering Figure 5.7, Υ^{Det} is the time interval starting when the System receives the first malicious actuation (i.e., payload value -3), ending in t_D. A positive Υ^{Det} is the time interval left to the intruder for controlling the System. Aspects influencing the value of Υ^{Det} are related to the communication delays between the CA, the System, and IDS. Moreover, the differences in the computation capabilities of the hosts executing CA and IDS plays a role in the value of Υ^{Det}, as detailed in Section 6.2.3. For simplicity, we do not consider the additional time interval required to disconnect the compromised CA after an `Alert` is issued.

We distinguish between two kinds of operations carried out during the workflow of the ASiMOV's tasks:

- *logic operations*: all and only the operations performed by microservices during their `microcycle` (i.e., function in Listing 5.1);

- *communication operations*: all operations outside the `microcycle` function i.e., performed by message broker(s) and networking device(s) for transporting messages between microservices.

The study of $\hat{T}^{\text{Act}}$ and Υ^{Det} are based on a model for the choreography realized by the ASiMOV components. There are introduced the following definitions and notation.

- *Stage*: the `microcycle` of a microservice waits, consumes and produces specific types of messages during a so-called stage of the choreography of a task. During a stage, a component may transit between two different component's phases (see Section 5.1.2).

- *Logic delay*: is the time required by a microservice to complete the logic operations of a stage, i.e., to produce message(s) upon consumption of incoming message(s) accordingly to the choreography. Assuming the message(s) to be consumed in a stage are available for a microservice, the logic delay is the time required to execute all the operations in one or more executions of the `microcyle`, which consume (produce) all the inputs (outputs) message(s) required by the stage.

- *Communication delay*: is the time required by message broker and networking devices to complete the communication operations transmitting a single message from microservice "A" to microservice "B", assuming that "B" is continuously ready to receive a message.

A communication delay begins when microservice "A" completed the communicate job of its `microcyle`, which enters a message in its local array `out`, and ends when "B" has completed the communicate job of the `microcyle` entering the message in its array of queues `inFIFO`. When "A" and "B" belongs to two different applications (e.g., SW of CA, LV of IDS), the communication delay also accounts for the network delay between the two different application hosts.

Figure 6.3 shows workflow of ASiMOV for the Acquire, Actuate, Store and Compare tasks. The workflow is composed by logic and communication delays, which are associated with stages of the considered tasks. In the Figure, we use the following graphical formalism:

- a microservice in a particular stage is represented as a block with two vertical lines with consumed (incoming) and produced (outgoing) messages. The time interval between the two vertical lines is the logic delay associated to the stage. We denote with $T^{\text{service}}_{\text{stage}}$ the duration of a logic delay, where "service" is a microservice and "stage" is the number of the stage of the microservice in the workflow e.g., T_1^{CK} is the time required to CK to serve an unbounded query, and T_2^{CL} is the time required by CL to update its Execution State (i.e., System's state estimation and production of an actuation), given that `Time` and `SetCmsg` are available in its `inFIFO` queues;

- when multiple messages enter a microservice in a stage (i.e., left vertical line), the correspondent time instant is the time in which the last of the incoming

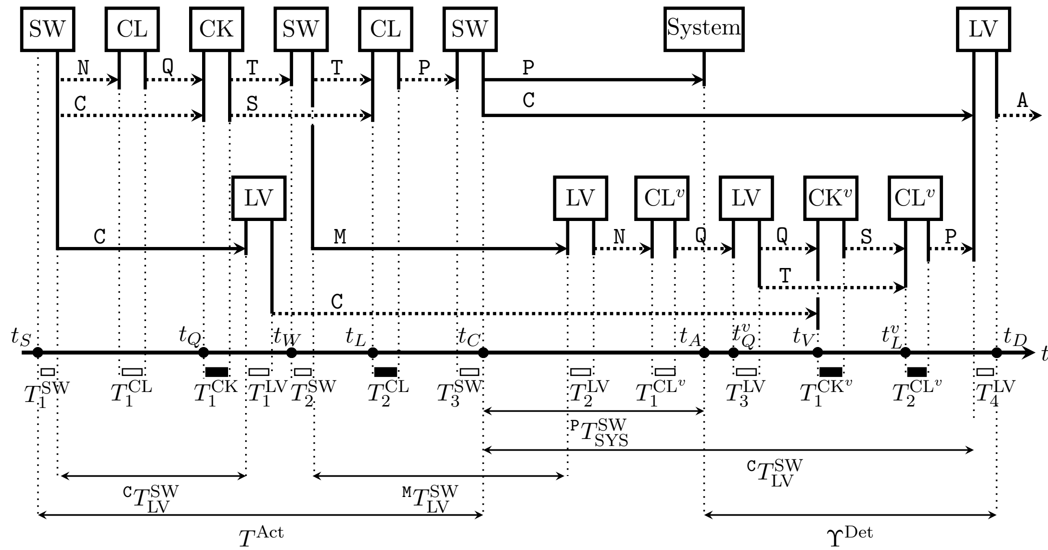


Figure 6.3. Representation of the logic delays (rectangles on the time axis) and communication delays (arrows) introduced by the microservices in the Acquire, Actuate, Store, and Compare tasks. The negligible logic delay (under approximation AP1) are empty rectangles. The letter over an arrow is the initial letter of a `BusIO` object carried by a message (see Table 5.1 e.g., `P` is `Payload`, `S` is `SetCmsg` x, `T` is `Time` r, or τ_i^M). A solid arrow is an inter-host transmission of a message (e.g., CA to IDS), while a dotted arrow is an intra-host transmission (e.g., within the CA).

messages is consumed. Similarly, the time instant relative to the right vertical line is the time in which the last of outgoing messages is produced.

6.2.2 Approximations

In order to evaluate the performances of ASiMOV we introduce the following approximation in the delay model, that are motivated and discussed in the rest of this chapter:

ASiMOV introduces a negligible additional amount of logic operations (i.e., within `microcycle` function) compared to a monolithic implementation. (AP1)

Given a choreographic stage (i.e., a block in Figure 6.3), there is a maximum of one non-negligible Compute job (i.e., a sub-function of the `microcycle`) among those occurring during the stage. (AP2)

The communication delay between two microservices is constant among different transmissions. (AP3)

The communication delay of microservices within the same application (i.e., CA and IDS) is independent of the type of message (e.g., `Cmsg`, `SetCmsg`) and the source and destination components (e.g., CK to SW, or SW^v to CL^v). (AP4)

The logic delay for all microservices is negligible. (AP5)

AP1 is reasonable since ASiMOV maintains the same complexity of monolithic control logic, and introduces few additional logic operations. As an example, an additional logic operation consists in comparing the current microservice phase with a value (e.g., Listing 9.1 of the Annex, line 7), or creating set of messages `SetCmsg` (line 12) which in normal operating conditions contains one or a few elements (e.g., most recent sensing). During an iteration of the Control task there are approximately 100 additional logic operations. Usually, 1000 actuation per second is a sufficient control frequency for a System, then ASiMOV introduces approximately 10^5 additional logic operations per second, we assume incapable of saturating the capabilities of a host. We highlight that AP1 implies that the delay introduced by the Communicate job, that is the time to read from `inFIFO` and write to `out`, is negligible. As a consequence of AP1, any non-negligible logic delay is determined by the same operations performed by a monolithic implementation, i.e., we assume the control logic must necessarily query a database of control events, update its execution environment (e.g., system's state estimation), and compute an actuation.

AP2 holds since we intentionally defined each of the stages such that they contain only one non-negligible Compute job. In particular, the logic a monolithic controller is entirely executed in one of the executions of the CK and CL's `microcycle`, i.e., during T_1^{CK} (select a subset of control events) and during T_2^{CL} (state estimation and computation of actuation). Therefore, using approximations AP1, AP2 the only non-negligible logic delays are T_1^{CK}, $T_1^{\text{CK}^v}$, T_2^{CL}, $T_2^{\text{CL}^v}$, which are represented as black rectangles in Figure 6.3. Under AP1 and AP2, all the messages to be consumed (produced) in a choreography's stage appear in Figure 6.3 to be consumed (produced) at the same time instant, and the logic delay introduced by a stage is the time required to complete the non-negligible Compute job of a stage (if any).

AP3 holds assuming a constant computational and networking load. AP4 is reasonable in case the difference in the size of messages transporting `BusIO` objects is negligible compared to the time required to a message broker to provide access to its message queues. Under AP3 and AP4 it is valid the definition of the intra-host (inter-host) communication delays T^*_{CA}, T^*_{IDS} ($T^*_{\text{CA-IDS}}$, $T^*_{\text{CA-SYS}}$). AP5 allows isolating the effects of the communication delay over the actuation delay $\hat{T}^{\text{Act}}$.

6.2.3 Performances when CA is at rest

The analysis that follows is valid when the following condition holds:

CA and IDS are *at rest* when sensing events arrives from the System i.e., the message broker queues, and the `inFIFO` queues of all the microservices are empty when sensing arrives. (C10)

Condition C10 holds when the inter-arrival time of sensing Γ^S is constant and:

$$T^{\text{Act}} < \Gamma^S \tag{C11}$$

Using approximation AP1, AP2, and assuming that C10 holds, the following equations describe the values of time instants of Figure 6.3:

$$
\begin{aligned}
t_Q &= t_S + T_1^{\mathrm{SW}} + \max\left\{({}^{\mathtt{N}}T_{\mathrm{CL}}^{\mathrm{SW}} + T_1^{\mathrm{CL}} + {}^{\mathtt{Q}}T_{\mathrm{CK}}^{\mathrm{CL}}), {}^{\mathtt{C}}T_{\mathrm{CK}}^{\mathrm{SW}}\right\} \\
t_W &= t_Q + T_1^{\mathrm{CK}} + {}^{\mathtt{T}}T_{\mathrm{SW}}^{\mathrm{CK}} \\
t_L &= \max\left\{(t_W + T_2^{\mathrm{SW}} + {}^{\mathtt{T}}T_{\mathrm{CL}}^{\mathrm{SW}}), (t_Q + T_1^{\mathrm{CK}} + {}^{\mathtt{S}}T_{\mathrm{CL}}^{\mathrm{CK}})\right\} \\
t_C &= t_L + T_2^{\mathrm{CL}} + {}^{\mathtt{P}}T_{\mathrm{SW}}^{\mathrm{CL}} + T_3^{\mathrm{SW}} \\
t_A &= t_C + {}^{\mathtt{P}}T_{\mathrm{SYS}}^{\mathrm{SW}} \\
t_Q^v &= t_W + T_2^{\mathrm{SW}} + {}^{\mathtt{M}}T_{\mathrm{LV}}^{\mathrm{SW}} + T_2^{\mathrm{LV}} + {}^{\mathtt{N}}T_{\mathrm{CL}^v}^{\mathrm{LV}} + T_1^{\mathrm{CL}^v} + {}^{\mathtt{Q}}T_{\mathrm{LV}}^{\mathrm{CL}^v} \\
t_V &= \max\left\{(t_Q^v + T_3^{\mathrm{LV}} + {}^{\mathtt{M}}T_{\mathrm{CK}^v}^{\mathrm{LV}}), (t_S + T_1^{\mathrm{SW}} + T_1^{\mathrm{LV}} + {}^{\mathtt{C}}T_{\mathrm{LV}}^{\mathrm{SW}} + {}^{\mathtt{C}}T_{\mathrm{CK}^v}^{\mathrm{LV}})\right\} \\
t_L^v &= \max\left\{(t_V + T_1^{\mathrm{CK}^v} + {}^{\mathtt{S}}T_{\mathrm{CL}^v}^{\mathrm{CK}^v}), (t_Q^v + T_3^{\mathrm{LV}} + {}^{\mathtt{T}}T_{\mathrm{CL}^v}^{\mathrm{LV}})\right\} \\
t_D &= \max\left\{(t_C + {}^{\mathtt{C}}T_{\mathrm{LV}}^{\mathrm{SW}}), (t_L^v + T_2^{\mathrm{CL}^v} + {}^{\mathtt{P}}T_{\mathrm{LV}}^{\mathrm{CL}^v})\right\} + T_4^{\mathrm{LV}}
\end{aligned}
\tag{6.3}
$$

Introducing approximations AP3, AP4 and considering, without the loss of generality, $t_S = 0$, then from Eqs. 6.3 we obtain:

$$
\begin{aligned}
t_A &= 5 \cdot T_{\mathrm{CA}}^* + T_1^{\mathrm{CK}} + T_2^{\mathrm{CL}} + T_{\mathrm{CA\text{-}SYS}}^* \\
t_D &= 3 \cdot T_{\mathrm{CA}}^* + T_{\mathrm{CA\text{-}IDS}}^* + \max\left\{T_{CA}^{\mathrm{BNeck}}, T_{IDS}^{\mathrm{BNeck}}\right\}
\end{aligned}
\tag{6.4}
$$

where:

$$
\begin{aligned}
T_{CA}^{\mathrm{BNeck}} &= 2 \cdot T_{\mathrm{CA}}^* + T_1^{\mathrm{CK}} + T_2^{\mathrm{CL}} \\
T_{IDS}^{\mathrm{BNeck}} &= 5 \cdot T_{\mathrm{IDS}}^* + T_1^{\mathrm{CK}^v} + T_2^{\mathrm{CL}^v}
\end{aligned}
$$

From Eq. 6.4 and Eq. 6.1 the actuation delay T^{Act} is:

$$
T^{\mathrm{Act}} = 5 \cdot T_{\mathrm{CA}}^* + T_1^{\mathrm{CK}} + T_2^{\mathrm{CL}} \tag{6.5}
$$

Under AP1 and AP2, the logic operations of a monolithic controller coincides with those occurring during T_1^{CK} and T_2^{CL}. Therefore, under AP1-AP4 and C10, the additional control loop delay is: $\hat{T}^{\mathrm{Act}} = 5 \cdot T_{\mathrm{CA}}^*$. Practically speaking, the increase of control loop delay T introduced by ASiMOV over a monolithic implementation has the same order of magnitude of the communication delay of a message broker, which is typical of few milliseconds.

The time intervals T_{CA}^{BNeck} and T_{IDS}^{BNeck} are potential bottlenecks for the completion of the Verify task, i.e., the greater time interval among the two is the bottleneck for the comparison of a trusted and non-trusted actuation, which is realized during T_4^{LV}. From Eq. 6.2 and Eq. 6.4, the detection delay is:

$$
\Upsilon^{\mathrm{Det}} = -T_{CA}^{\mathrm{BNeck}} + \max\left\{T_{CA}^{\mathrm{BNeck}}, T_{IDS}^{\mathrm{BNeck}}\right\} + T_{\mathrm{CA\text{-}IDS}}^* - T_{\mathrm{CA\text{-}SYS}}^*
$$

defining:

$$
\begin{aligned}
\Upsilon^{\mathrm{Network}} &= T_{\mathrm{CA\text{-}IDS}}^* - T_{\mathrm{CA\text{-}SYS}}^* \\
\Upsilon^{\mathrm{Hosts}} &= T_{IDS}^{\mathrm{BNeck}} - T_{CA}^{\mathrm{BNeck}}
\end{aligned}
\tag{6.6}
$$

the detection delay is rewritten as:

$$\Upsilon^{\text{Det}} = \max\left\{0, \Upsilon^{\text{Hosts}}\right\} + \Upsilon^{\text{Network}} \tag{6.7}$$

As already introduced, ASiMOV leaves the System under attack for a duration of Υ^{Det}, which in the best cases converges to $\Upsilon^{\text{Network}}$. The first of the two components of Eq 6.7 can be rendered 0 by disposing of an opportune computation power on the IDS side i.e., such that $\Upsilon^{\text{Hosts}} \leq 0$. The component $\Upsilon^{\text{Network}}$ is the difference of latency between the hosts of CA – IDS, and the hosts CA – System. As an example, an industrial network setup having $T^*_{\text{CA-IDS}} = 100$ ms, $T^*_{\text{CA-SYS}} = 10$ ms and an IDS host with sufficient computational power, leaves an attack to the control logic undetected for $\Upsilon^{\text{Det}} \approx 90$ ms.

6.2.4 Performances when CA is not at rest

We are particularly interested in the changes in the behavior of CA introduced by a critical T^*_{CA}, i.e., when condition C10 does not hold.

Using approximation AP1-AP5, T^{Act} is entirely determined by the communication delay T^*_{CA} of the message broker on the CA side. Assuming the communication delay of a monolithic implementation negligible compared to that of a message broker (i.e., the former uses shared memory which is much faster), then the additional actuation delay $\hat{T}^{\text{Act}}$ coincides with T^{Act}. Precisely, the components T_1^{CK}, T_2^{CL} of Eq 6.5 are set to 0 to obtain:

$$\hat{T}^{\text{Act}} = T^{\text{Act}} = 5 \cdot T^*_{\text{CA}} \tag{6.8}$$

The communication delay T^*_{CA} is said *critical* with respect to a certain Γ^S when C10 does not hold, in which case Eq. 6.3 may lose descriptiveness. From Eq. 6.8 and assumption C10 we have:

$$T^*_{\text{CA}} > \frac{1}{5}\Gamma^S \quad \implies T^*_{\text{CA}} \quad \text{is critical} \tag{6.9}$$

6.2.5 Simulation results

In what follows, we present simulation results showing the behavior of CA as a function of the intra-host communication delay T^*_{CA}, which is the amount of time required to deliver a message using a message broker. Typically, such an amount of time has value in the interval $(1, 20)$ milliseconds [26]. In the simulations we vary T^*_{CA} in the interval $(0, 20)$ milliseconds, and we consider three different values for Γ^S i.e., 50, 10 and 5 milliseconds.

We implement an event-driven production of actuation for the CA (i.e., Eq. 4.7 of Sec. 4.3.1), meaning that an actuation is produced as soon as new sensing arrives. Under these conditions, CA is supposed to realize a constant interdeparture time of actuation Γ^A that is equal to Γ^S, i.e., with value respectively 50, 10 and 5 milliseconds (which corresponds respectively to a control frequency of 20, 100 and 200 Hertz). To implement AP1–AP5, we simulate a negligible logic delay i.e., 10^{-6} seconds for any `microcycle` execution.

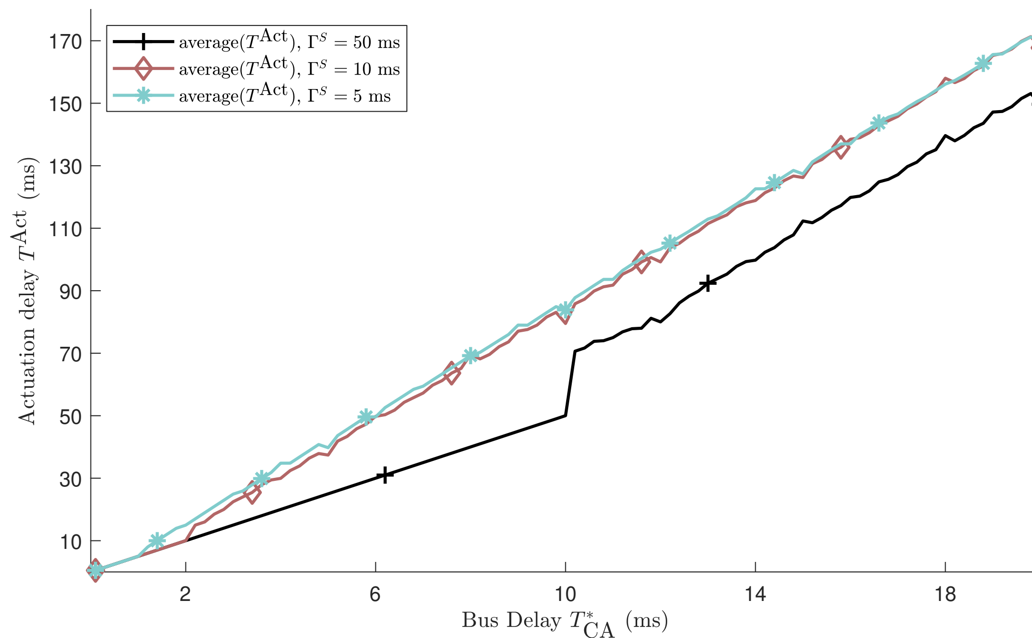


Figure 6.4. Actuation delay T^{Act} as a function of the intra-host communication delay T^*_{CA} when AP5 holds.

Each point of curves in Figures 6.4, 6.5, 6.6, 6.7 is obtained running a simulation where 1000 sensing messages arrive to the CA. The x axis in the Figures is the intra-host communication delay T^*_{CA}. In the simulations, the only non-negligible delay considered is the intra-host communication delay. When the intra-host communication delay is non-critical, the observed behaviour coincides with the model of Eq. 6.5. Otherwise, there are possibly undesired behaviours which are detailed by the Figures.

Figure 6.4 shows actuation delay T^{Act} as a function of T^*_{CA}. From approximation AP5 the measured value coincides with the additional actuation delay $\hat{T}^{\text{Act}}$. When T^*_{CA} is non-critical (e.g., $T^*_{\text{CA}} < 10$ ms when $\Gamma^S = 50$ ms) the measured actuation delay has value of Eq. 6.8. When T^*_{CA} becomes critical, there is an increase in T^{Act}, which grows linearly with T^*_{CA}.

Figure 6.5 is relative to the number of sensing messages used to produce an actuation, i.e., size(x^A) is the number of sensing messages in set x^A obtained by CL from CK (i.e., an unbounded query from selection criteria of Eq. 4.7). When T^*_{CA} is non-critical, CL obtains one sensing message for each of the queries. In this situation, ASiMOV can behave as a traditional controller producing an actuation upon receiving sensing. When T^*_{CA} becomes critical, the average number of messages used for an actuation increases with a linear fashion, while the maximum number of sensing ever employed to produce an actuation increases with discontinuities. The observed behavior is not easily explainable in terms of a model since it depends on the internal dynamics of the messages accumulating into the message broker queues.

Figure 6.6 is relative to the capability of CA to stay abreast of a particular frequency of sensing messages. Being Γ^A the interdeparture time achieved by the CA, the Figure shows that when T^*_{CA} is non-critical, the ratio Γ^A/Γ^S is 1, i.e., CA can

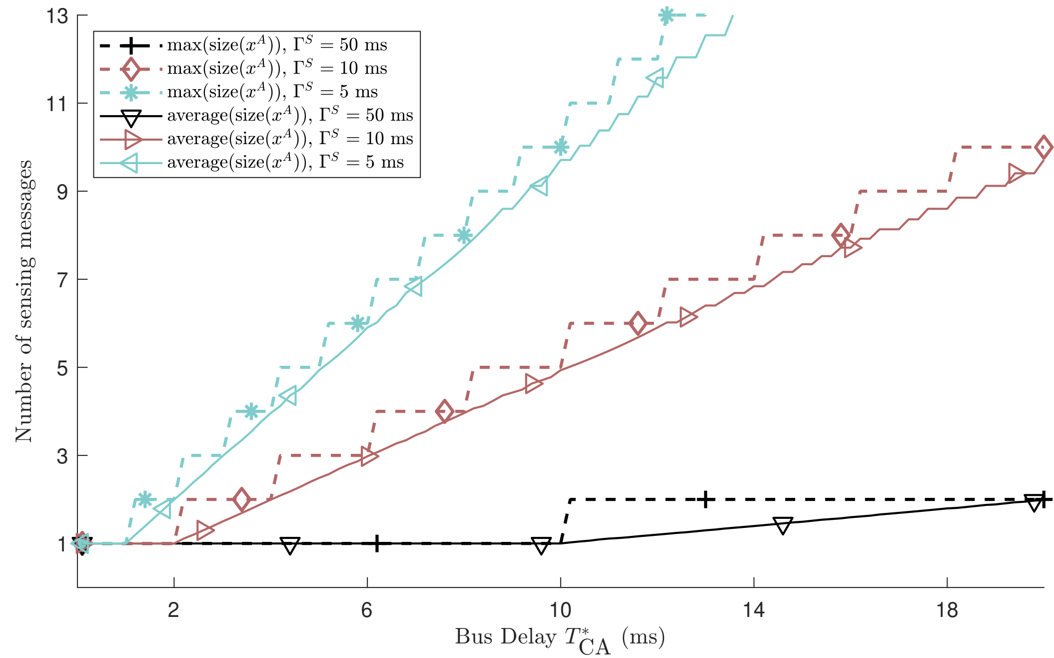


Figure 6.5. The maximum and the average number of sensing messages x^A used to produce an actuation.

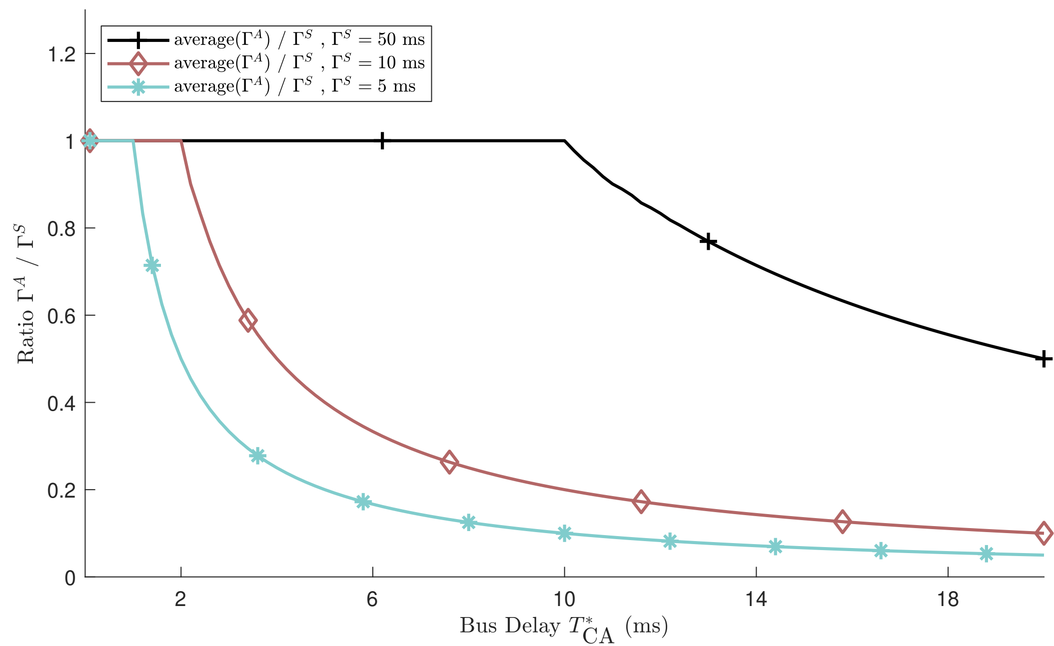


Figure 6.6. The capability of CA to produce an actuation for each received sensing message, in a event-driven communication process.

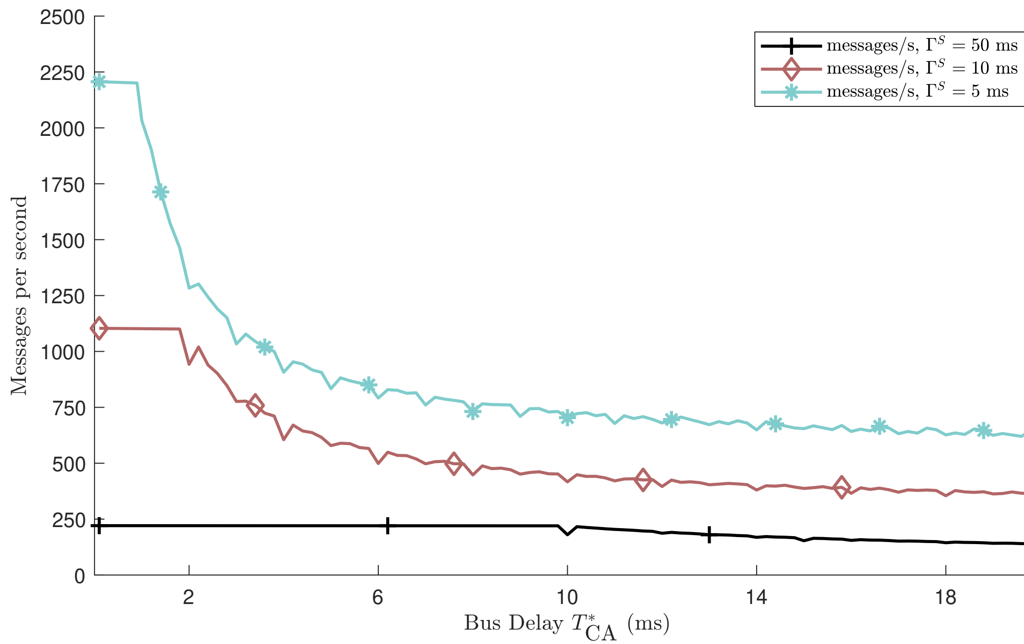


Figure 6.7. The average number of bus messages traversing the message broker bus per second.

produce an actuation for each received sensing. Practically speaking, if $T^*_{\text{CA}} < \frac{1}{5} \cdot \Gamma^S$, then $\Gamma^A = \Gamma^S$. Therefore, when T^*_{CA} is $1 - 20$ ms, the maximum control frequency f_M that ASiMOV can achieve is $200 - 10$ Hertz. When T^*_{CA} is critical, there are fewer actuation payloads produced then sensing received. The implications of such behavior on the performances of a control system depend on different factors (e.g., the characteristics of the System under control). An insufficient number of actuation per second may compromise the performances or even stability of the control system.

Figure 6.7 shows the number of messages traversing the message broker bus per seconds, i.e., each point in the Figure is the sum of the number of messages of any kind generated by the arrival of the 1000 sensing messages, divided by the time in which there is activity on the message broker bus during a simulation. The cited sum accounts for the actual size of set x used by CL, i.e., a message of class `SetCmsg` counts as the number of payloads contained in the set. The observed values seem compatible with a typical implementation of a message broker. As an example, RabbitMQ is capable of delivering tens of thousands of messages per second in a typical server setup[26].

Chapter 7

Related works

In this thesis, we consider the Industrial Control Systems (ICS) as a reference scenario to describe state of the art in the protection of a CPS. The most advanced instance of an ICS is called a **smart factory**. Among all the real-world applications of a CPS, the smart factory is one of the scenarios in which the adoption of ASiMOV is possible with minimal changes. First of all, ICS typically provides for the replication of control signals at the network level (e.g., TAP devices), as required by ASiMOV. Moreover, as discussed in more detail in this chapter, the smart factory envisages the use of virtualization technologies, as required by the ASiMOV's mitigation and prevention mechanism.

7.1 A reference ICS scenario: the smart factory

The smart factory is a digitalized, intelligent, sustainable IoT-based manufacturing system capable of improving industrial production performance and quality [70, 116, 3, 49, 59, 2]. Smart factories are the 4th generation of the industry [116, 60], and have the following characteristics [114]: the components of the manufacturing system are networked, and it is possible to collect useful data from them in real-time (included the component's state); autonomous and automatic processes can be executed based on optimized manufacturing plans; advanced manufacturing services can be provided on the shop floor (cf. Figure 7.1) and to external systems. Those characteristics are realized by integrating data, artificial intelligence, and distributed computing technologies with physical components and industrial processes [116, 49, 60].

The operations of a smart factory rely on a complex control plane consisting of a set of interconnected control layers (cf. Figure 7.1) that range from the direct control of physical Systems (level 1) to the production scheduling (level 4). The control scheme is distributed and hierarchical by definition: each layer controls the lower one, and each layer could contain a hierarchy of controllers.

The adoption of virtualization technologies is pervasive where there is a need for large computational capacities and an automated deployment of applications. The smart factory transformation is one of the examples of CPS in which virtualization technologies have become a requirement [34, 66, 52, 89]. Virtualization technologies enable the mitigation and prevention mechanisms of ASiMOV.

Making a factory smart brings many benefits [116, 60, 112] but also increase

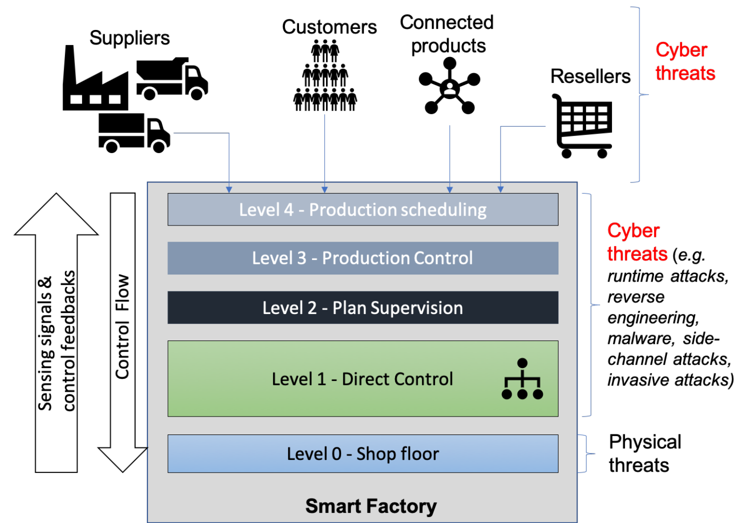


Figure 7.1. The Smart Factory: actors, control levels and cyber risks

cyber risks [108, 93], e.g., financial loss due to production downtime, impairment of products' quality, damage to the reputation, harm to humans, and physical/environmental damage. Because of the interconnection of heterogeneous control systems, from production planning and scheduling to shop floor control, the smart factory attack surface is wider than in the 3rd generation industry [93]. Malwares like Mirai, Stuxnet, Triton, Dugu, Havex/Dragonfly could be used take over the control of industrial IoT devices and control systems [108].

Although there are many research results aiming to improve the security of CPS (e.g. [115, 95, 24, 17, 77, 89, 11, 21, 77, 98]), none of them propose solutions that goes beyond the traditional cyber-security approach, i.e., semi-automatic human-assisted detection and response [41], based on complex security policies. The traditional approach is impractical in smart factories because: an attack can target thousand of controllers; the human in the loop can compromise the high availability requirements (i.e., unforeseen disruption of production services); IoT devices used in the control process have limited resource capability and cannot run complex security mechanisms, like traditional Intrusion Detection Systems (IDS). Therefore, it is crucial to investigate innovative self-protection mechanisms to increase resiliency to cyber-attacks targeting the control logic of industrial Control Applications (i.e., the software implementation of an industrial controller) running on resource-limited devices [90].

7.2 Enabling technologies and security of the smart factory

In this Section, we analyze the literature of two research areas related to ASiMOV. The first is *enabling technologies for the smart factory*, that includes research work addressing the challenge of moving the control plane of a factory from dedicated and

on-premise systems into cloud computing or edge computing platforms. The second domain of investigation is *intrusion detection and prevention* of cyber-risks that includes research works presenting new techniques for the detection of cyber-attack and the prevention or mitigation of such attacks.

7.2.1 Enabling technologies for the smart factory

In [110] the authors propose a framework that incorporates industrial wireless networks, cloud, and fixed or mobile terminals with smart artifacts. The role of the cloud layer is to process and analyze the massive amount of information generated from sensors to support system management and optimization, including supervision and control.

In [96] the authors introduce the concept of Cloud-integrated Cyber-Physical Systems and propose an architecture aiming at solving challenges like virtual resource management, the scheduling of cloud resources, and life cycle management in the context of Digital Factory.

Work [67] proposes a prototype Cloud Manufacturing, where physical systems are connected to the cloud and controlled using Raspberry Pi devices. Evolution of Cloud Manufacturing is the Software-defined Cloud Manufacturing proposed in [104]. In that work, the authors propose to use the software-defined systems approach to solve the complexity issue and the reconfigurability issues in cloud manufacturing.

Research works that investigate how to migrate the direct control level into the cloud are [32, 34, 56, 69, 106]. None of those work concretely address the cyber-security issues.

The concept of PLC as a service implemented within a cloud-based infrastructure is proposed in [32]. The performances of the cloud-based PLC and a legacy PLC are compared, showing that latency is an issue that needs to be fixed to adopt cloud-based PLC in hard real-time scenarios. In [34] the authors address the question of how a Soft-PLC (PLC realized as software components) can be augmented with cloud-required aspects such as elastic scalability and multi-tenancy. The authors present an architecture that realizes these properties and allows for the implementation of a multi-tenant, horizontally scalable Soft-PLC. As in [32] the authors conclude that the cloud-based solution can be applied only to soft real-time control scenarios. For the above reasons, we propose a solution aiming at reducing the latency in the control loop, and then to make possible the use of cloud-based PLC also in hard real-time contexts.

The concept and implementation for Cloud-based Industrial Control Services (CICS) as a next-generation PLC is presented in [56]. The proposed solution aims to replace the traditional PLC for applications with uncritical timing in a Smart Factory context. The authors investigate possible deployment in the cloud of a PLC compliant with the IEC 61131-3 standard.

In [69] the authors propose a modeling framework for describing, developing, and composing cyber-physical manufacturing services (CPMS) for service-oriented smart manufacturing systems [102]. Such kind of framework could be the core for the design and implementation of fog computing-based industrial control systems.

In [106] the authors propose to use microservices to encapsulate: sensors, actuators, and the low-level coordination logic required to offer more advanced functionality

compared to the one offered by the mechanical unit. Microservices are deployed on an edge/fog computing platform that provides computational services to the different layers of a smart factory.

ASiMOV is designed considering that it is possible to virtualize the direct control layer of a smart factory and that the Control Applications could run on general-purpose edge devices or cloud platforms.

7.2.2 Intrusion detection, prevention and mitigation

In the last decade, many efforts have been devoted to study the problem of intrusion detection in cyber-physical systems and industrial control systems (ICS) [41, 77].

As proposed in [41], IDS for ICS can be classified as: protocol analysis-based, traffic mining-based, and control process analysis-based. Protocol analysis–based IDS (e.g. [61, 80, 115]) detect malicious attacks by checking whether the transmission packets in an industrial control network violate the industrial protocol specifications, e.g., Modbus, Modbus/TCP, ICCP/TASE.2, DNP3. Protocol analysis–based IDSs have two limitations: the detection accuracy is influenced by the protocol model and the defined rules. A discrepancy with a real physical system can introduce a high false-positive rate. Moreover, the detection ability against unknown attacks is poor, and the time to parse data packets is long. Traffic mining-based IDS (e.g. [109, 64, 63, 95]) try to solve the inefficiency of the above technique by building nonlinear and complex relationships between the network traffics and the normal/abnormal system behaviors. Traffic mining-based IDS use machine learning and artificial intelligence techniques (e.g., PCA, fuzzy logic, deep learning) to identify anomalous behaviors in industrial networks. Such an IDS technique could leverage the static and well-known network topology of ICS and the stability of the network traffic. Control process analysis-based IDS make full use of the semantic information and peculiarity of ICS to detect intrusions and include techniques like process data analysis, control command analysis, and ICS physical model analysis. Process data analysis-based IDS (e.g. [37, 19, 24]) analyze the value of physical variables (like pressure and temperature) to understand if the security status of the physical process is violated. Control command analysis-based IDS (e.g., [20, 62, 17]) detect attacks by analyzing the sequence of control commands sent to the ICS to find to predict misbehavior. ICS physical model analysis-based IDS (e.g. [21, 1]) are based on an accurate model of the ICS and predict attacks comparing the output of the real system with the output predicted by the model.

One possible way to detect the compromise of a device in a CPS is to employ remote attestation [99], in which a verifier entity submit challenges to a device. The device can solve a challenge only if its state is not compromised. The challenges submitted to a device can be onerous, for example, to hash a portion of the RAM, which is a problem for real-time control. Remote attestation methods assume that the verifier entity knows the internal state of the device to be verified. Therefore, attestation is highly effective in case the state of a device does not need to have substantial changes. Conversely, if the state of the device is subjected to changes (e.g., due input values), the verifier must be informed. Work [31] proposes to combine a physics-based statistical method with remote attestation, in order to abstract from the possible changes in the state of a controller for a physical system. The

method is based on a learning phase utilizing the inputs of the controller (i.e., sensor readings) and is, therefore, is suitable for controllers that perform cyclic tasks, e.g., tasks occurring at the leaves of a hierarchical tree of a smart factory. Vice versa, the method proposed in this thesis is Behavior-Specification-Based, i.e., does not require a training phase, and therefore it is adapt complex hierarchical control schemes, such Smart Factory, where the applicability of methods using training is difficulty estimable.

ASiMOV is inspired by the control process analysis-based detection technique, but there are two main differences: for each controller of the ICS, there is an IDS, and instead of using a model of the control system, it uses a replica of the controller to detect misbehavior dues to cyber-attacks.

An approach similar to ASiMOV is Cymbiote [90]. Cymbiote is a device designed to be connected to a controller to monitor different sensing, actuation, and communication with other controllers, and with the sensors/actuators, environmental parameters. Cymbiote processes all this information to detect cyber-attacks or failures. Cymbiote integrates IDS techniques belonging to the three categories mentioned above and is capable of detecting attacks to the control logic, control messages, and sensors/actuators. ASiMOV and Cymbiote are based on the same idea to connect an IDS to each controller on the shop floor, however, there are two main differences. Firstly, Cymbiote requires a specific configuration for each family of a control device and a training phase on historical data, while ASiMOV does not. Secondly, Cymbiote is not capable to automatically recover from attacks, except in case of hardware or software failure, when the software configuration or the hardware is reset.

Concerning the prevention of cyber-attacks, a variety of security architectures have been proposed: software-based isolation and virtualization [73]; Trusted Computing based on secure hardware (e.g., Trusted Platform Module); and processor architectures providing secure execution (e.g., ARM TrustZone [111], AEGIS [100], OASIS [83], and Intel Software Guard Extensions [74]). ASiMOV on one side is proposed as an alternative to those approaches because they are too complex for low-end embedded systems, which are typically designed for specific tasks and optimized for low power consumption and minimal costs. On the other hand, ASiMOV assumes that the IDS component runs on a trusted/isolated platform that could be realized utilizing a secure execution environment, like vTPM [14] or SGX enclaves [8].

Chapter 8

Conclusions and future works

This thesis proposes ASiMOV, an innovative approach to realizing self-protecting control applications that are resilient to cyber-attacks targeting their logic. Specifically, we propose a model and the architecture for a self-protecting Control Application (CA) and the related cyber-attack detection and mitigation mechanisms. The fundamental characteristic of ASiMOV, compared to existing solutions, is that the output of the control logic is verifiable without any training phase, i.e., ASiMOV contains a Behavior-Specification-Based Intrusion Detection System, having ideal accuracy performances (no false positive/negative). Detection is based on redundancy without voting, i.e., a replica is assumed safe from cyber-attacks and is the reference for the correct control logic. Mitigation and prevention is based on the orchestration of a microservice-based architecture with an immutable deployment. By adopting an event-sourcing pattern, the state of a CA can be migrated between different hosts at run-time.

ASiMOV is a novel architecture for a CA to be employed in the automatic control of physical systems. Therefore, this thesis provides a model and experimental results that allow establishing whether the delay introduced by this architecture can be adequate for the control of a specific System (in normal conditions and under cyber-attacks). The results are summarized as follows.

The delay analysis shows that ASiMOV introduces a control loop delay overhead of the same order of magnitude of the time T^*_{CA} required by a message broker to deliver a message within the microservices of the CA, i.e., milliseconds or tens of milliseconds. Consequently, the class of existing control systems compatible with the same control loop delay could adopt ASiMOV without any change or just with a re-tuning of parameters to compensate for the additional delay. However, the previous statement holds as long as T^*_{CA} does not exceed a certain threshold, which depends on the interarrival time of received sensing. With a T^*_{CA} of $1 - 20$ ms ASiMOV is capable of sustaining a sensing interarrival time higher than $5 - 100$ ms, corresponding to a maximum sampling rate of $200 - 10$ Hz. If T^*_{CA} exceeds the just mentioned threshold, ASiMOV delivers a reduced number of actuation per second, which could be a problem for hard real-time control. The detection delay Υ^{Det} is the time an intruder can control a System without being detected. Assuming that the IDS's host has sufficient computational capacity (i.e., the IDS is not a computational bottleneck), Υ^{Det} is the difference of the delay between CA–IDS hosts

and the delay between CA–System hosts. Being able to estimate the detection delay of IDS for Cyber-Physical-Systems accurately is a crucial feature, since it can be used by optimization-based control logic to mitigate or prevent damage through control action.

The proposed solution has value under the assumption that it is possible to protect a centralized subset of computational resources. This work does not investigate in detail how this condition can be achieved. The proposed architecture decouples the verification process from direct control, i.e., the two activities are asynchronous. This enables to investigate to use multiple replicas, instead of a single, trusted replica, as proposed in this thesis. Possibly, the replicas could employ software diversification or consensus algorithms, which become feasible given the immense computing power of cloud platforms and the fact that the verification is asynchronous to the control loop. We considered unnecessary the use of a framework for the formal verification of the absence of bugs in the provided implementation of ASiMOV. This choice comes from the fact that the interaction between microservices consists of a relatively simple choreography. In particular, there is a single datastore microservice, and we do not employ distributed transactions. Our confidence in the absence of bugs in the proposed implementation is based on an extensive series of simulations of a prototype implementation, where we introduced a randomly variable communication delay between microservices without obtaining deadlock or race problems. However, we have not considered the case of faults e.g., the loss of a message. In this sense, the proposed architecture is not still ready for production. This thesis proposes a model for estimating the detection delay against attacks of control logic. The estimate concerns a single component of a distributed architecture, but in future works could be extended to any topology of interconnected controllers. The usage of diversification could be considered at the level of cloud platforms, to protect against attacks on the deeper layers of computer architecture.

Currently, ASiMOV suffers from the following minor limitations not yet resolved:

- The Knowledge in CK and CK^v (i.e., control events) grows indefinitely, so: *i*) data store CK may run out of space; *ii*) the time required for the controller state migration process may be impractical. The solution we are investigating envisages: *i*) on the CA side older control events are eliminated; *ii*) on the IDS side, the CL^v periodically serializes its Execution State into a single parameter control message, to be used as a checkpoint by CL^h (healed CA) during the state migration process.
- Sensing or parameter `Pload` arriving at the network switch during a state migration process can accumulate for a while before receiving a timestamp. Depending on the case, this could lead to degradation of the performances of the control logic (e.g., state estimation process). The solution we are investigating envisages to use a temporary `Pload` cache on the IDS side, and letting SW^h (healed CA) assign timestamps at a later time. However, the provision of deferred `Pload` may lead to timestamp values which are very different from the real ones. Therefore, the state estimation performed by CL^h may degrade. A more refined solution is to use the same timestamp values assigned by SW^w

(IDS side). However, this solution raises the problem of synchronizing the system clock of the hosts running SW^w and SW^h.

- A problem mentioned in this thesis is that different computer architectures working with a representation of real numbers could give different results for the same operation. This problem should be thoroughly investigated for a real-world implementation of ASiMOV. Slightly different values in the state of an application may lead to divergence with the state of a replica. In this sense, it is necessary to investigate appropriate methods that guarantees divergence of state, e.g., by truncation of numbers.

Chapter 9

Annex: Pseudo-code

<table>
<tr><td rowspan="10">SW</td><td rowspan="4">IN</td><td>1</td><td>`Time`</td><td>Most recent timestamp τ_i^M of the query relative to the production of the i-th actuation</td></tr>
<tr><td>2</td><td>`Pload`</td><td>Actuation payload as produced by CL</td></tr>
<tr><td>3</td><td>`Pload`</td><td>Sensor payload as reiceved from System</td></tr>
<tr><td>4</td><td>`Pload`</td><td>Parameter payload as reiceved by an external entity e.g., father CA</td></tr>
<tr><td rowspan="6">OUT</td><td>1</td><td>`Notif`</td><td>Notifies CL that it is time to produce an actuation</td></tr>
<tr><td>2</td><td>`Time`</td><td>Provides to CL the notion of current time r_i</td></tr>
<tr><td>3</td><td>`Meta`</td><td>[τ_i^M,r_i] forwarded to IDS</td></tr>
<tr><td>4</td><td>`Cmsg`</td><td>Timestamped control message for IDS. It is a copy of `OUT` 6</td></tr>
<tr><td>5</td><td>`Pload`</td><td>Forwarded actuation payload as received from `IN` 2 to the System</td></tr>
<tr><td>6</td><td>`Cmsg`</td><td>Timestamped control message for CK. It is a copy of `OUT` 4</td></tr>
</table>

Table 9.1. Ports definition and description for the SW component.

CL	IN	1	`Notif`	Triggers the production of an actuation. See `OUT 1`
		2	`Time`	Notion of current time to be used for the production of an actuation
		3	`SetCmsg`	Subset x_i of the knowledge for the production of the current actuation. See `OUT 2`
	OUT	1	`Pload`	Payload of the current actuation
		2	`Query`	Query to obtain subset x_i of the knowledge. See `IN 3`

Table 9.2. Ports definition and description for the CL component.

LV	IN	1	`Pload`	Payload of an actuation produced by local CL, to be verified (in IDS) or ignored (in CA-Heal)
		2	`Query`	Query as produced by CL, to be modified accordingly to previously reiceved `Meta`
		3	`Meta`	In IDS: reiceved by CA and used for the verification. In CA-Heal: used for state migration
		4	`Cmsg`	Similar to `IN` 3 , but contains a timestamped message
		5	`SetCmsg`	Set of messages utilized only for the match task to protect against Unrestricted attack
	OUT	1	`Notif`	Same functionality of SW, `OUT1`
		2	`Time`	Same functionality of SW, `OUT2`
		3	`Alert`	Triggers the mitigation (or prevention) mechanism, which initiates a state migration
		4	`Meta`	Same functionality of SW, `OUT3`
		5	`Cmsg`	Same functionality of SW, `OUT6`
		6	`Query`	Query modified from what received from `IN` 2
		7	`Query`	Query utilized to obtain set of messages at `IN` 5

Table 9.3. Ports definition and description for the LV component.

Component	Direction	#	Port	Description
CK	IN	1	`Alert`	Initiate the procedure for state migration. See `OUT 2` and `3`
		2	`Meta`	Store the Meta for later use, i.e., for when state migration is required
		3	`Cmsg`	Store the Cmsg in the knowledge
		4	`Query`	Reiceve a query. See `OUT` 4
	OUT	1	`Time`	Provide the maximum timestamp of the result of a query τ_i^M
		2	`Meta`	Used only for state migration. Transfers all the stored `Meta`
		3	`Cmsg`	Used only for state migration. Transfers all the stored `Cmsg`
		4	`SetCmsg`	Serves a query for the i-th actuation by providing a subset of the knowledge x_i

Table 9.4. Ports definition and description for the CK component.

`Int chooseIn(Pload[] arrPload,Int[] inI)` Used by SW during the Acquire task to choose a single input port when multiple types of payload are available. This avoids the monopolization of SW by a certain type of control signal (e.g., sensing). `arrPload` are three `Pload` objects (possibly `Null`), `inI` are the three correspondent input indexes. `chooseIn` returns an input index (e.g., randomly, round robin) among those having its correspondent `arrPload` not `null`, or returns `Null` when all `arrPload` are `Null`

`SetCmsg select(SetCmsg k, Query i)`
Returns a selection of `Cmsg` from Knowledge k based on their timestamp. ∀`m` in output:
`(m.tstamp > i.start)AND(m.tstamp <= i.end)`

`Bool contains(SetCmsg k, Time t)` Returns `True` if there is a message `m` in `k` : `m.tstamp == t`
`Time maxTimeStamp(SetCmsg s)` Returns the maximum timestamp among elements in the set `s`
`updParams(params, SetCmsg x, Time t)` updates the Execution State of CL (i.e., state var `this.params` of CL)
`Pload calcAct(params, Time t)` computes an actuation given the Execution State and current time
`Bool compare(Pload a1, Pload a2)` returns True if the two actuation payloads `a1,a2` are identical, or similar enough to not raise the detection of an attack
`Bool notNull(Object obj)` returns True only if `obj` (e.g, a `BusIO`) is not Null. In case `obj` is a `List` or a `Set`, returns True only if `obj` is empty
`Int countType(SetCmsg x, String type)` counts the number of `Cmsg` of a certain type (e.g., `'sen'`) in set `x`
`Time systemTime()` returns the system time as seen by the execution environment
`add/read/rem(List/SetCmsg s, BusIO element)` Respectively add, read and remove `element` from List or Set `s`. In case `s` is a `List` the functions implement a FIFO operation, i.e., `add` appends to the tail, `read` and `remove` operates on the head.

Table 9.5. Functions used by microservices.

Listing 9.1. Pseudo-code of the compFunct CL

```
1  \\ Actuate task
2  IF this.phase == 0 AND notNull(in[1]): \\ Notif in
3     \\ Produces query for CK
4     consumed[1] = true
5     out[2] = new Query(mRecC,Double.Infinity)
6     this.phase = 1
7  ELSEIF (this.phase == 1):
8     IF notNull(in[2]) AND notNull(in[3]):
9        consumed[2] = True
10       consumed[3] = True
11       Time r = in[2]
12       SetCmsg x = in[3]
13       updParams(this.params, x, r) \\ update the Execution State
14       out[1] = calcAct(this.params, r) \\ produces a
15       this.mRecP = maxTimeStamp(x) \\ used for successive query
16       this.phase = 0
```

Listing 9.2. Pseudo-code of the compFunct SW

```
1  Time r = systemTime()
2
3  \\ Acquire task
4  Cmsg[] arrPload = [in[2],in[3],in[4]] \\ possibly Null
5  Int sIn = chooseIn(arrPload,[2,3,4]) \\ choose one input port
6  IF notNull(sIn):
7     consumed[sIn] = True
8     String type = ''
9     IF sIn == 2: \\ Reiceved actuation
10       type = 'act'
11       IF this.phase == 2 \\Iteration ends
12          this.phase = 0
13          out[5] = new Pload(in[sIn].pload) \\to System
14    IF sIn == 3: \\ Reiceved sensing
15       type = 'sen'
16    IF sIn == 4: \\ Reiceved parameter
17       type = 'par'
18    Cmsg newMes = new Cmsg(in[sIn].pload, r, type)
19    out[4] = newMes \\ to LV
20    out[6] = newMes \\ to CK
21    if type == "sen" OR type == "par":
22       this.mRecP = r
23
24 \\ Actuate task
25 IF this.phase == 0:
26    IF this.mRecC < this.mRecP
27       out[1] = new Notif()
28       this.phase = 1
29 ELSEIF (this.phase == 1) AND notNull(in[1]):
30    consumed[1] = True
31    out[2] = r \\ current Time to CL
32    Time tauM = in[1].value \\max timestamp in set 'x'
33    out[3] = new Meta(r,tauM) \\ [r, tauM] to LV of IDS
34    IF tauM > this.mRecC: \\update tauM if needed
35       this.mRecC = tauM
36    this.phase = 2
```

Listing 9.3. Pseudo-code of the `compFunct` CK

```
1  \\ Acquire task
2  IF notNull(in[3]):\\Reiceved Cmsg
3     consume[3] = True
4     add(this.k,in[3]) \\ Store in the knowledge
5
6  \\ Used only by IDS to store metadata for state migration
7  IF notNull(in[2]): \\ Reiceved Meta
8     consume[2] = True
9     add(this.km,in[2]) \\ Store in the knowledge
10
11 \\ Used only in IDS
12 IF notNull(in[1]):\\Reiceved Alert
13    \\ Begins state migration to mitigate an attack
14    \\ using out 2, 3
15    \\ ...
16
17 \\ Actuate task
18 SetMes x = Null
19 IF this.phase == 0: \\ no pending queries
20    IF notNull(4): \\ reiceved query
21       consumed[4] = True
22       this.pendQ = new Query(in[4])
23       IF this.pendQ == Double.Infinite: \\ unbounded query (CL)
24          this.phase == 2
25       ELSE: \\ bounded query (LV)
26          this.phase = 1
27 IF this.phase == 1: \\ waiting for most recent Cmsg (bounded)
28    IF contains(x,this.pendQ.end) == True:
29       x = select(this.k,this.pendQ)
30
31 \\ An unbounded query must deliver at least one 'sen' or 'par'
32 ELSEIF this.phase == 2: \\attempting to serve an unbounded query
33    x = select(pendQ)
34    Int ns = countType(x,'sen')
35    Int np = countType(x,'par')
36    if (ns+np) <= 0:\\x is empty, or with 'act' only
37       x = Null \\ will retry the query later
38
39 IF this.phase == 1 OR this.phase == 2:
40    IF notNull(x): \\ Query can be served
41       out[1] = maxTimeStamp(x) \\ send max timestamp to SW
42       out[4] = x \\ send query result to CL
43       this.phase = 0
```

Listing 9.4. Pseudo-code of the compFunct LV

```
1  \\ Store Task
2  IF notNull(in[3]): \\ meta in
3     consumed[3] = True
4     add(this.metas, new Meta(in[3]))
5  IF notNull(in[4]): \\ cmsg in
6     consumed[4] = True
7     out[5] = new Cmsg(4) \\ to CK of IDS
8     IF in[4].pload == 'act': \\Store act
9        add(this.acts, new Cmsg(in[4]))
10 IF this.phase == 0: \\ LV is not currently verifying
11     IF notNull(this.metas):
12         IF notNull(this.acts):
13          \\There is a meta-cmsg couple in internal FIFOs
14          \\Therefore, verification of an actuation can begin
15          out[1] = new Notif()
16            this.phase = 1
17
18 \\Verify Task
19 IF this.phase == 1 AND notNull(in[2]) \\ unbounded query in
20    \\ Modifies query interval to produce bounded query
21    consumed[2] == True
22    Time ts = in[2].start
23    Time te = (get(this.metas)).tauM \\ get max timestamp
24    out[6] = new Query(ts, te)
25    this.phase = 2
26 IF this.phase == 2 AND notNull([1])
27    \\ a trusted actuation was reiceved
28    if notNull(this.acts)
29       \\ compares untrusted (from CA) and trusted actuations
30       Pload trusted = in[1]
31       consumed[1] = True
32       Pload untrusted = (get(this.acts)).pload
33       Bool valid = compare(trusted,untrusted)
34       if NOT valid: \\ Alert to DMAN
35          out[3] = new Alert()
36          \\ Begin state mitigation operations
37          \\ ...
38       else: \\ no attack was detected
39          rem(this.metas) \\ remove last
40          rem(this.acts) \\ remove last
41          this.phase == 0
```

Bibliography

[1] Robust detection filter design in the presence of time-varying system perturbations. *Automatica 33*, 3 (1997), 471 – 475.

[2] Smartfactory – towards a factory-of-things. *Annual Reviews in Control 34*, 1 (2010), 129 – 138.

[3] Development of innovative strategies for the korean manufacturing industry by use of the connected smart factory (csf). *Procedia Computer Science 91* (2016), 744 – 750. Promoting Business Analytics and Quantitative Management of Technology: 4th International Conference on Information Technology and Quantitative Management (ITQM 2016).

[4] Alshuqayran, N., Ali, N., and Evans, R. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)* (2016), IEEE, pp. 44–51.

[5] Alur, R., and Madhusudan, P. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems* (2004), Springer, pp. 1–24.

[6] Antsaklis, P. J., Passino, K. M., and Wang, S. An introduction to autonomous control systems. *IEEE Control Systems Magazine 11*, 4 (1991), 5–13.

[7] Arcangeli, J.-P., Boujbel, R., and Leriche, S. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software 103* (2015), 198–218.

[8] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. Scone: Secure linux containers with intel sgx. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.

[9] Arsanjani, A. Service-oriented modeling and architecture. *IBM developer works 1* (2004), 15.

[10] Axelsson, S. Intrusion detection systems: A survey and taxonomy. Tech. rep., Technical report, 2000.

[11] Banerjee, A., Venkatasubramanian, K. K., Mukherjee, T., and Gupta, S. K. S. Ensuring safety, security, and sustainability of mission-critical cyber–physical systems. *Proceedings of the IEEE 100*, 1 (Jan 2012), 283–299.

[12] Baset, S. A., Tang, C., Tak, B. C., and Wang, L. Dissecting open source cloud evolution: An openstack case study. In *Presented as part of the 5th {USENIX} Workshop on Hot Topics in Cloud Computing* (2013).

[13] Bass, L., Merson, P., and O'Brien, L. Quality attributes and service-oriented architectures. *Department of Defense, Technical Report September* (2005).

[14] Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. Vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (USA, 2006), USENIX-SS'06, USENIX Association.

[15] Bottoni, P., Gabrielli, E., Gualandi, G., Mancini, L. V., and Stolfi, F. Fedup! cloud federation as a service. In *European Conference on Service-Oriented and Cloud Computing* (2016), Springer, pp. 168–182.

[16] Bratterud, A., Happe, A., and Duncan, R. A. K. Enhancing cloud security and privacy: the unikernel solution. In *Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017-23 February 2017, Athens, Greece* (2017), Curran Associates.

[17] Brugman, J., Khan, M., Kasera, S., and Parvania, M. Cloud based intrusion detection and prevention system for industrial control systems using software defined networking. In *2019 Resilience Week (RWS)* (Nov 2019), vol. 1, pp. 98–104.

[18] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *ACM Queue 14* (2016), 70–93.

[19] Carcano, A., Coletta, A., Guglielmi, M., Masera, M., Nai Fovino, I., and Trombetta, A. A multidimensional critical state analysis for detecting intrusions in scada systems. *IEEE Transactions on Industrial Informatics 7*, 2 (May 2011), 179–186.

[20] Carcano, A., Fovino, I. N., Masera, M., and Trombetta, A. State-based network intrusion detection systems for scada protocols: A proof of concept. In *Critical Information Infrastructures Security* (Berlin, Heidelberg, 2010), E. Rome and R. Bloomfield, Eds., Springer Berlin Heidelberg, pp. 138–150.

[21] Cárdenas, A. A., Amin, S., Lin, Z.-S., Huang, Y.-L., Huang, C.-Y., and Sastry, S. Attacks against process control systems: Risk assessment, detection, and response. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 355–366.

[22] CHATURVEDI, A. Securing microservice architectures, a holistic approach, https://www.ibm.com/downloads/cas/jy6lnawx.

[23] COBLENZ, M., SUNSHINE, J., ALDRICH, J., MYERS, B., WEBER, S., AND SHULL, F. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering* (2016), ACM, pp. 736–747.

[24] COLBERT, E., SULLIVAN, D., HUTCHINSON, S., RENARD, K., AND SMITH, S. A process-oriented intrusion detection method for industrial control systems, 2016.

[25] COLOMBO, A. W., BANGEMANN, T., KARNOUSKOS, S., DELSING, J., STLUKA, P., HARRISON, R., JAMMES, F., LASTRA, J. L., ET AL. Industrial cloud-based cyber-physical systems. *The IMC-AESOP Approach 22* (2014).

[26] DOBBELAERE, P., AND ESMAILI, K. S. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* (2017), ACM, pp. 227–238.

[27] DRAGONI, N., GIALLORENZO, S., LAFUENTE, A. L., MAZZARA, M., MONTESI, F., MUSTAFIN, R., AND SAFINA, L. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[28] ESFAHANI, P. M., VRAKOPOULOU, M., MARGELLOS, K., LYGEROS, J., AND ANDERSSON, G. Cyber attack in a two-area power system: Impact identification using reachability. In *Proceedings of the 2010 American control conference* (2010), IEEE, pp. 962–967.

[29] FARWELL, J. P., AND ROHOZINSKI, R. Stuxnet and the future of cyber war. *Survival 53*, 1 (2011), 23–40.

[30] GASPARETTO, A., BOSCARIOL, P., LANZUTTI, A., AND VIDONI, R. Trajectory planning in robotics. *Mathematics in Computer Science 6*, 3 (2012), 269–279.

[31] GHAEINI, H. R., CHAN, M., BAHMANI, R., BRASSER, F., GARCIA, L., ZHOU, J., SADEGHI, A.-R., TIPPENHAUER, N. O., AND ZONOUZ, S. Patt: Physics-based attestation of control systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)* (2019), pp. 165–180.

[32] GIVEHCHI, O., IMTIAZ, J., TRSEK, H., AND JASPERNEITE, J. Control-as-a-service from the cloud: A case study for using virtualized plcs. In *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)* (May 2014), pp. 1–4.

[33] Goethals, T., Sebrechts, M., Atrey, A., Volckaert, B., and De Turck, F. Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)* (2018), IEEE, pp. 1–8.

[34] Goldschmidt, T., Murugaiah, M. K., Sonntag, C., Schlich, B., Biallas, S., and Weber, P. Cloud-based control: A multi-tenant, horizontally scalable soft-plc. 909–916.

[35] Gualandi, G., and Casalicchio, E. A self-protecting control application for iiot. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)* (2019), IEEE, pp. 152–157.

[36] Gualandi, G., and Casalicchio, E. Use of redundancy in the design of a secure software defined industrial control application. In *2019 Sixth International Conference on Software Defined Systems (SDS)* (2019), IEEE, pp. 102–109.

[37] Hadžiosmanovic, D., Sommer, R., Zambon, E., and Hartel, P. H. Through the eye of the plc: Semantic security monitoring for industrial processes. Association for Computing Machinery.

[38] Hemsley, K. E., Fisher, E., et al. History of industrial control system cyber incidents. Tech. rep., Idaho National Lab.(INL), Idaho Falls, ID (United States), 2018.

[39] Hightower, K., Burns, B., and Beda, J. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.", 2017.

[40] Hindy, H., Brosset, D., Bayne, E., Seeam, A., Tachtatzis, C., Atkinson, R., and Bellekens, X. A taxonomy and survey of intrusion detection system design techniques, network threats and datasets. *arXiv preprint arXiv:1806.03517* (2018).

[41] Hu, Y., Yang, A., Li, H., Sun, Y., and Sun, L. A survey of intrusion detection on industrial control systems. *International Journal of Distributed Sensor Networks 14*, 8 (2018), 1550147718794615.

[42] Huebscher, M. C., and McCann, J. A. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR) 40*, 3 (2008), 7.

[43] Humayed, A., Lin, J., Li, F., and Luo, B. Cyber-physical systems security—a survey. *IEEE Internet of Things Journal 4*, 6 (2017), 1802–1831.

[44] Iannucci, S., and Abdelwahed, S. Model-based response planning strategies for autonomic intrusion protection. *ACM Trans. Auton. Adapt. Syst.*, 1, 4:1–4:23.

[45] Iannucci, S., Abdelwahed, S., Montemaggio, A., Hannis, M., Leonard, L., King, J., and Hamilton, J. A model-integrated approach to

designing self-protecting systems. *IEEE Transactions on Software Engineering* (2018), 1–1.

[46] James Lewis, M. F. Microservices, http://martinfowler.com/articles/microservices.html.

[47] Jézéquel, F., Lamotte, J.-L., and Saïd, I. Estimation of numerical reproducibility on cpu and gpu. In *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on* (2015), IEEE, pp. 675–680.

[48] Kang, H., Le, M., and Tao, S. Container and microservice driven design for cloud infrastructure devops. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on* (2016), IEEE, pp. 202–211.

[49] Kang, H. S., Lee, J. Y., Choi, S., Kim, H., Park, J. H., Son, J. Y., Kim, B. H., and Do Noh, S. Smart manufacturing: Past research, present findings, and future directions. *International Journal of Precision Engineering and Manufacturing-Green Technology 3*, 1 (2016), 111–128.

[50] Kephart, J. O., and Chess, D. M. The vision of autonomic computing. *Computer*, 1, 41–50.

[51] Khraisat, A., Gondal, I., Vamplew, P., and Kamruzzaman, J. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity 2*, 1 (2019), 20.

[52] Kirsch, J., Goose, S., Amir, Y., and Skare, P. Toward survivable scada. In *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research* (New York, NY, USA, 2011), CSIIRW '11, ACM, pp. 21:1–21:1.

[53] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering 16*, 11 (1990), 1293–1306.

[54] Kratzke, N., and Peinl, R. Clouns-a cloud-native application reference model for enterprise architects. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2016 IEEE 20th International* (2016), IEEE, pp. 1–10.

[55] La, H. J., and Kim, S. D. A service-based approach to designing cyber physical systems. In *2010 IEEE/ACIS 9th International Conference on Computer and Information Science* (2010), IEEE, pp. 895–900.

[56] Langmann, R., and Stiller, M. Cloud-based industrial control services. In *Online Engineering & Internet of Things* (Cham, 2018), M. E. Auer and D. G. Zutin, Eds., Springer International Publishing, pp. 3–18.

[57] Langone, J., Key, S., Alder, U. S., et al. Vmware vsphere virtual machine encryption performance.

[58] Lascu, T. A., Mauro, J., and Zavattaro, G. Automatic deployment of component-based applications. *Science of Computer Programming 113* (2015), 261–284.

[59] Lasi, H., Fettke, P., Kemper, H.-G., Feld, T., and Hoffmann, M. Industry 4.0. *Business & Information Systems Engineering*, 4, 239–242.

[60] Lee, J., Bagheri, B., and Kao, H.-A. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters 3* (2015), 18 – 23.

[61] Lin, H., Slagell, A., Di Martino, C., Kalbarczyk, Z., and Iyer, R. K. Adapting bro into scada: Building a specification-based intrusion detection system for the dnp3 protocol. Association for Computing Machinery.

[62] Lin, H., Slagell, A., Kalbarczyk, Z., and Iyer, R. K. Semantic security analysis of scada networks to detect malicious control commands in power grids (poster). Association for Computing Machinery.

[63] Linda, O., Manic, M., Vollmer, T., and Wright, J. Fuzzy logic based anomaly detection for embedded network security cyber sensor. In *2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)* (April 2011), pp. 202–209.

[64] Linda, O., Vollmer, T., and Manic, M. Neural network based intrusion detection system for critical infrastructures. In *2009 International Joint Conference on Neural Networks* (June 2009), pp. 1827–1834.

[65] Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. Nist cloud computing reference architecture. *NIST special publication 500*, 2011 (2011), 1–28.

[66] Liu, M., Guo, C., and Yuan, M. The framework of scada system based on cloud computing. In *Cloud Computing* (Cham, 2014), V. C. Leung and M. Chen, Eds., Springer International Publishing, pp. 155–163.

[67] Liu, X. F., Shahriar, M. R., Sunny, S. N. A., Leu, M. C., and Hu, L. Cyber-physical manufacturing cloud: Architecture, virtualization, communication, and testbed. *Journal of Manufacturing Systems 43* (2017), 352 – 364. High Performance Computing and Data Analytics for Cyber Manufacturing.

[68] Lu, D., Huang, D., Walenstein, A., and Medhi, D. A secure microservice framework for iot. In *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)* (2017), IEEE, pp. 9–18.

[69] Lu, Y., and Ju, F. Smart manufacturing systems based on cyber-physical manufacturing services (cpms). *IFAC-PapersOnLine 50*, 1 (2017), 15883 – 15889. 20th IFAC World Congress.

[70] Lucke, D., Constantinescu, C., and Westkämper, E. Smart factory - a step towards the next generation of manufacturing. In *Manufacturing Systems and Technologies for the New Frontier* (London, 2008), M. Mitsuishi, K. Ueda, and F. Kimura, Eds., Springer London, pp. 115–118.

[71] Lun, Y. Z., D'Innocenzo, A., Malavolta, I., and Di Benedetto, M. D. Cyber-physical systems security: a systematic mapping study. *arXiv preprint arXiv:1605.09641* (2016).

[72] Lyons, R. E., and Vanderkul, W. The use of triple-modular redundancy to improve computer reliability. *IBM journal* (1962).

[73] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy* (May 2010), pp. 143–158.

[74] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. Association for Computing Machinery.

[75] Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[76] Mitchell, R., and Chen, I.-R. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys (CSUR) 46*, 4 (2014), 55.

[77] Mitchell, R., and Chen, I.-R. A survey of intrusion detection techniques for cyber-physical systems. *ACM Comput. Surv. 46*, 4 (Mar. 2014), 55:1–55:29.

[78] Mo, Y., and Sinopoli, B. Integrity attacks on cyber-physical systems. In *Proceedings of the 1st international conference on High Confidence Networked Systems* (2012), ACM, pp. 47–54.

[79] Montesi, F. *Choreographic programming.* IT-Universitetet i København, 2014.

[80] Morris, T., Vaughn, R., and Dandass, Y. A retrofit network intrusion detection system for modbus rtu and ascii industrial control systems. In *2012 45th Hawaii International Conference on System Sciences* (Jan 2012), pp. 2338–2345.

[81] Nagy, L., Ford, R., and Allen, W. N-version programming for the detection of zero-day exploits. Tech. rep., 2006.

[82] Ogata, K. Modern control engineering. *Instructor 201709* (2017).

[83] Owusu, E., Guajardo, J., McCune, J., Newsome, J., Perrig, A., and Vasudevan, A. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. Association for Computing Machinery.

[84] Pasqualetti, F., Dörfler, F., and Bullo, F. Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design. In *2011 50th IEEE Conference on Decision and Control and European Control Conference* (2011), IEEE, pp. 2195–2201.

[85] Pasqualetti, F., Dorfler, F., and Bullo, F. Control-theoretic methods for cyberphysical security: Geometric principles for optimal cross-layer resilient control systems. *IEEE Control Systems Magazine 35*, 1 (Feb 2015), 110–127.

[86] Pasqualetti, F., Dorfler, F., and Bullo, F. Control-theoretic methods for cyberphysical security: Geometric principles for optimal cross-layer resilient control systems. *IEEE Control Systems Magazine 35*, 1 (2015), 110–127.

[87] Preda, M. D., Gabbrielli, M., Giallorenzo, S., Lanese, I., and Mauro, J. Dynamic choreographies: Theory and implementation. *arXiv preprint arXiv:1611.09067* (2016).

[88] Qin, Z., Li, Q., and Chuah, M.-C. Defending against unidentifiable attacks in electric power grids. *IEEE Transactions on Parallel and Distributed Systems 24*, 10 (2012), 1961–1971.

[89] Ren, L., Zhang, Y., Luo, Y., and Zhang, L. A virtualization approach for distributed resources security in network manufacturing. In *2010 IEEE International Conference on Industrial Engineering and Engineering Management* (Dec 2010), pp. 1524–1528.

[90] Rice, T. R., Seppala, G., Edgar, T., Choi, E., Cain, D., and Mahserejian, S. Development of a host-based intrusion detection and control device for industrial field control devices. In *2019 Resilience Week (RWS)* (Nov 2019), vol. 1, pp. 105–111.

[91] Richardson, C. *Microservices patterns.* Manning Publications Shelter Island, 2018.

[92] Richardson, C. *Microservices Patterns.* MEAP, 2018.

[93] Sadeghi, A.-R., Wachsmann, C., and Waidner, M. Security and privacy challenges in industrial internet of things. Association for Computing Machinery.

[94] Sebenik, C., and Hatch, T. *Salt Essentials: Getting Started with Automation at Scale.* " O'Reilly Media, Inc.", 2015.

[95] Shone, N., Ngoc, T. N., Phai, V. D., and Shi, Q. A deep learning approach to network intrusion detection. *IEEE Transactions on Emerging Topics in Computational Intelligence 2*, 1 (Feb 2018), 41–50.

[96] Shu, Z., Wan, J., Zhang, D., and Li, D. Cloud-integrated cyber-physical systems for complex industrial applications. *Mob. Netw. Appl. 21*, 5 (Oct. 2016), 865–878.

[97] SIPAHI, R., NICULESCU, S., ABDALLAH, C. T., MICHIELS, W., AND GU, K. Stability and stabilization of systems with time delay. *IEEE Control Systems Magazine 31*, 1 (Feb 2011), 38–65.

[98] SRIDHAR, S., AND GOVINDARASU, M. Model-based attack detection and mitigation for automatic generation control. *IEEE Transactions on Smart Grid 5*, 2 (March 2014), 580–591.

[99] STEINER, R. V., AND LUPU, E. Attestation in wireless sensor networks: A survey. *ACM Computing Surveys (CSUR) 49*, 3 (2016), 51.

[100] STRACKX, R., PIESSENS, F., AND PRENEEL, B. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks* (Berlin, Heidelberg, 2010), S. Jajodia and J. Zhou, Eds., Springer Berlin Heidelberg, pp. 344–361.

[101] STRIKI, M., MANOUSAKIS, K., KINDRED, D., STERNE, D., LAWLER, G., IVANIC, N., AND TRAN, G. Quantifying resiliency and detection latency of intrusion detection structures. In *MILCOM 2009-2009 IEEE Military Communications Conference* (2009), IEEE, pp. 1–8.

[102] TAO, F., LAILI, Y., XU, L., AND ZHANG, L. Fc-paco-rm: A parallel method for service composition optimal-selection in cloud manufacturing system. *IEEE Transactions on Industrial Informatics 9*, 4 (Nov 2013), 2023–2033.

[103] TEIXEIRA, A., SHAMES, I., SANDBERG, H., AND JOHANSSON, K. H. Revealing stealthy attacks in control systems. In *2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton)* (2012), IEEE, pp. 1806–1813.

[104] THAMES, L., AND SCHAEFER, D. Software-defined cloud manufacturing for industry 4.0. *Procedia CIRP 52* (2016), 12 – 17. The Sixth International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2016).

[105] THÖNES, J. Microservices. *IEEE software 32*, 1 (2015), 116–116.

[106] THRAMBOULIDIS, K., VACHTSEVANOU, D. C., AND KONTOU, I. Cyber-physical microservices and iot-based framework: The case of evolvable assembly systems. *arXiv:1807.07363 [cs.SE]* (2018).

[107] TSAI, W.-T., SUN, X., AND BALASOORIYA, J. Service-oriented cloud computing architecture. In *2010 seventh international conference on information technology: new generations* (2010), IEEE, pp. 684–689.

[108] TUPTUK, N., AND HAILES, S. Security of smart manufacturing systems. *Journal of Manufacturing Systems 47* (2018), 93 – 106.

[109] VOLLMER, T., AND MANIC, M. Computationally efficient neural network intrusion security awareness. In *2009 2nd International Symposium on Resilient Control Systems* (Aug 2009), pp. 25–30.

[110] Wang, S., Wan, J., Li, D., and Zhang, C. Implementing smart factory of industrie 4.0: An outlook. *Int. J. Distrib. Sen. Netw. 2016* (Jan. 2016), 7:7–7:7.

[111] Winter, J. Trusted computing building blocks for embedded linux-based arm trustzone platforms. Association for Computing Machinery.

[112] Xu, X. From cloud computing to cloud manufacturing. *Robotics and Computer-Integrated Manufacturing 28*, 1 (2012), 75 – 86.

[113] Yarygina, T., and Bagge, A. H. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)* (2018), IEEE, pp. 11–20.

[114] Yoon, S., Um, J., Suh, S.-H., Stroud, I., and Yoon, J.-S. Smart factory information service bus (sibus) for manufacturing application: requirement, architecture and implementation. *Journal of Intelligent Manufacturing 30*, 1 (Jan 2019), 363–382.

[115] Yusheng, W., Kefeng, F., Yingxu, L., Zenghui, L., Ruikang, Z., Xiangzhen, Y., and Lin, L. Intrusion detection of industrial control system based on modbus tcp protocol. In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)* (March 2017), pp. 156–162.

[116] Zhong, R. Y., Xu, X., Klotz, E., and Newman, S. T. Intelligent manufacturing in the context of industry 4.0: A review. *Engineering 3*, 5 (2017), 616 – 630.