Università di Firenze, Università di Perugia, INdAM consorziate nel CIAFM

DOTTORATO DI RICERCA
IN MATEMATICA, INFORMATICA, STATISTICA

CURRICULUM IN INFORMATICA
CICLO XXX

Sede amministrativa Università degli Studi di Firenze
Coordinatore Prof. Graziano Gentili

Towards Effective Anomaly Detection in Complex Dynamic Systems

Settore Scientifico Disciplinare INF/01

Dottorando:
Tommaso Zoppi

Tutor e Co-Tutor
Prof. Andrea Bondavalli
Dott. Andrea Ceccarelli

Coordinatori
Prof. Graziano Gentili
Prof. Cristina Pinotti

Anni 2014/2017

ABSTRACT

Anomaly detection can be used to infer the presence of errors or intrusions without observing the target service or application, but detecting variations in the observable parts of the system on which the service or the application resides. This is a promising technique in complex software-intensive systems, where either instrumenting the services' internals is exceedingly time-consuming, or encapsulation makes them not accessible. Unfortunately, in such systems anomaly detection is often made ineffective due to their dynamicity, which implies changes in the services or their expected workload.

The main target of this Thesis is to present our approach to enhance the efficacy of anomaly detection in complex dynamic systems. Evolving and Dynamic systems may often change their behavior, adapting it to the current context, making the characterization of the expected behavior, and consequently the identification of anomalies, a hard challenge. As a result, there are no clear state-of-the-art answers on applying error or anomaly detection in highly dynamic and complex systems, while some frameworks for performing anomaly detection in complex - not highly dynamic - systems have been described in the literature.

To contribute filling this gap, we put a promising state-of-the-art solution to work on data flows related to the *Secure!* system, a *Crisis Management System* which is structured as a *Service Oriented Architecture* (SOA). At first, we observed that applying such strategy as it was described for non-dynamic systems does not provide comparable detection scores, therefore we tried to adapt it by i) expanding the data collecting strategy, ii) considering additional information on the system, and iii) performing dedicated tuning of parameters of such strategy. This process led us to a customized version of the basic solution which has comparable scores with respect to other works targeting non-dynamic complex systems. At this point, we conducted an extensive experimental campaign targeting both the *Secure!* and the *jSeduite* SOAs based on the injection of specific types of anomalies to substantiate and confirm the progresses we obtained during our process.

However, the main result we obtained through these experiments was a precise definition of design guidelines that are mainly related to the necessity of frequently reconfiguring both the *monitoring strategy* and the *detection algorithms* to suit an *adaptive notion of expected and anomalous behavior*, avoiding *interferences* and minimizing *detection overheads*. After reporting and presenting these guidelines according to specific *viewpoints*, we present *MADneSs*, a framework which implements our approach to anomaly detection that is tailored for such systems.

The framework includes an adaptive *multi-layer* monitoring module. Monitored data is then processed by the anomaly detector, which adapts its parameters depending on the current behavior of the system, providing an anomaly alert. Lastly, we explore possible future implications explicitly targeting *Systems-of-Systems*, an architectural paradigm which in the recent years has started being adopted when building dynamic complex systems.

ACKNOWLEDGEMENTS

Surprisingly, also this step come to its end. A bit unexpected, since I never throught that I would have been a student for 21 years since primary school. Therefore, I learned that I am not really reliable when dealing with my future expectations, since my predictions are usually wrong.

On the other side, during the PhD years I saw places, people and ideas that were completely unexpected. Such experiences are something that contributed to build today's Tommaso, and I am really grateful to the people that gave me those opportunities. Hence, the first sincere thanks are directed to the people in the RCL group, who shared with me a massive amount of things. Important mentions go to the Brazilian professors who helped me with the accommodation, the logistics, and tried to create connections with their students. Thanks also to the people who shared the phd room in these years. And for the others co-workers, apologies if you are not mentioned, you can hate me in any way you want.

Just after mentioning university-related people, the most important thanks go to who - day by day - tolerated me during these years. I am not perfect, you know, but for sure you contributed (and, hopefully, you will continue) to limit the negative aspects of my nature, and this is extremely valuable: I cannot explain how much I appreciate that. Regarding the thesis, this is very specific and almost obscure (cit.) for non-expert people. If you are not expert and you want to get something from this document, pay attention to this page: easter eggs can be everywhere!

LIST OF RELEVANT PUBLICATIONS

The following publications have been produced in the context of the research work described in this Thesis.

SORT 2014 Ceccarelli, Andrea, Tommaso Zoppi, Andrea Bondavalli, Fabio Duchi, and Giuseppe Vella. "*A testbed for evaluating anomaly detection monitors through fault injection.*" In Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on, pp. 358-365. IEEE, 2014. [21]

SAFECOMP 2015 Ceccarelli, Andrea, Tommaso Zoppi, Massimiliano Itria, and Andrea Bondavalli. "*A multi-layer anomaly detector for dynamic service-based systems.*" In International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2015), pp. 166-180. Springer International Publishing, 2015. [22]

SOSE 2016 Marco Mori, Andrea Ceccarelli, Tommaso Zoppi, Andrea Bondavalli, *On the impact of emergent properties on SoS security*, in Proceedings of IEEE System of System Engineering Conference (SoSE 2016), Kongsberg, Norway. [69]

SAFECOMP 2016 Zoppi, Tommaso, Andrea Ceccarelli, and Andrea Bondavalli. "*Context-Awareness to Improve Anomaly Detection in Dynamic Service Oriented Architectures.*" In International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2016), pp. 145-158. Springer International Publishing, 2016. [108]

STUDENT FORUM - SRDS 2016 Zoppi, Tommaso, Andrea Ceccarelli, and Andrea Bondavalli. "*Challenging Anomaly Detection in Complex Dynamic Systems.*" In Reliable Distributed Systems (SRDS 2016), IEEE 35th Symposium on, pp. 213-214. IEEE, 2016. [107]

SAC 2017 Zoppi, Tommaso, Andrea Ceccarelli, and Andrea Bondavalli. "*Exploring Anomaly Detection in Systems of Systems.*" In Symposium on Applied Computing (SAC 2017) - Software Architecture: Theory, Technology, and Applications Track, ACM-SIGAPP 32th Symposium on, pp.1139-1146, 2017. [111]

Other related works were produced, altrough they are not published in official proceedings, since they are related to special sessions or conferences.

STUDENT FORUM - DSN 2015 Tommaso Zoppi, "*Multi-layer anomaly detection in complex dynamic critical systems*" In Dependable Systems and Networks (DSN 2015) - Student Forum Session, 2015

MINISY 2017 Tommaso Zoppi, "*Executing Online Anomaly Detection in Complex Dynamic Systems*" In 24th PhD MiniSymposium (MiniSy 2017), 2017. Available at `https://www.mit.bme.hu/eng/system/files/oktatas/9860/24Minisymp_proceedings.pdf`, pp. 86-89

During the PhD corse, the student also investigated other topics related to dependable and secure systems, mainly targeting Smart Grids as reference critical system within the IRENE project. The list of publications can be found below.

HASE 2016 Andrea Ceccarelli, Tommaso Zoppi, Paolo Lollini, Andrea Bondavalli, Francesco Lo Piccolo, Gabriele Giunta, Vito Morreale, *Presenting the Proper Data to the Crisis Management Operator: A Relevance Labelling Strategy*, In proceedings at IEEE High Assurance Systems Engineering Symposium (pp. 228-235), HASE 2016, Orlando, Florida (USA). [109]

ENERGYCON 2016 *Towards a Collaborative Framework to Improve Urban Grid Resilience* Jung Oliver, Vasenev Alexandr, Ceccarelli Andrea, Clarke Tony, Bessler Sandford, Montoya Lorena, Zoppi Tommaso, Chappell Keith, in Proceedings of IEEE Energy Conference (ENERGYCON 2016), Leuven, Belgium. [56]

JSEP 2017 Zoppi, Tommaso, Andrea Ceccarelli, Francesco Lo Piccolo, Paolo Lollini, Gabriele Giunta, Vito Morreale, and Andrea Bondavalli. "Labelling relevant events to support the crisis management operator." Journal of Software: Evolution and Process - HASE Special Issue (Wiley, 2017). [112]

RESACS 2017 Alexandr Vasenev, Dan Ionita, Tommaso Zoppi, Andrea Ceccarelli, and Roel Wieringa (2017) *Towards security requirements: Iconicity as a feature of an informal modeling language*. In 22nd International Conference on Requirements Engineering: Foundation for Software Quality - RESCS Workshop, REFSQ 2017, 27 February 2017, Essen, Germany (pp. 1-15). [96]

IRENE WORKSHOP - SMARTGIFT 2017 *A Tool for Evolutionary Threat Analysis of Smart Grids* Tommaso Zoppi, Andrea Ceccarelli, Marco Mori, (2017, March). In Smart Grid Inspired Future Technologies (pp. 205-211). Springer, Cham. [113]

IRENE WORKSHOP - SMARTGIFT 2017 *A Modeling Framework to Support Resilient Evolution Planning of Smart Grids* Tommaso Zoppi, Sandford Bessler, Andrea Ceccarelli, Edward Lambert, Eng Tseng Lau, Alexandr Vasenev,

(2017, March). In Smart Grid Inspired Future Technologies (pp. 233-242). Springer, Cham. [110]

Moreover, the following works contain submitted but not yet revised material, and therefore are reported separately.

TDSC 2017 Zoppi, Tommaso, Andrea Ceccarelli, and Andrea Bondavalli. "*MADneSs: a Multi-layer Anomaly Detection Framework for Complex Dynamic Systems*" Submitted to IEEE Transactions of Dependable and Secure Computing, 2017

TCPS 2017 Andrea Ceccarelli, Tommaso Zoppi, Alexandr Vasenev, Marco Mori, Dan ionita, Lorena Montoya, Andrea Bondavalli "*Threat Analysis in Systems-of-Systems: an Emergence-oriented Approach*" Submitted to the ACM Transactions on Cyber-Physical Systems, 2017

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Software-intensive systems such as cyber-physical infrastructures [80] or *Systems of Systems* [62] are composed of several different software layers and a multitude of services. Services implementation details are often not accessible, as they may be proprietary or legacy software. Additionally, offered services are often characterized by a dynamic behavior, making the services themselves and their interactions with other entities being updated, reacting to short-term updates of the environment. Further, these services can evolve through time, leading to changes in their requirements and consequently in their behavior [23].

As a result, instrumenting each individual service to monitor dependability-related properties in such complex systems is generally difficult if not unfeasible [26], [29], [104]. These difficulties are mainly related to the multitude of relations, interconnections and interdependencies that propagate potentially wrong decisions through the system: for example an update, a configuration change, or a malfunction in a single module or service can affect the whole system.

ANOMALY DETECTION To tackle this problem, several works are focusing on anomaly detection, which refers to the problem of finding patterns in data that do not conform to the expected behavior [25]. Such patterns are changes in the indicators characterizing the behavior of the system caused by specific and non-random factors. For example, pattern changes can be due to a system overload, the activation of software faults or malicious activities. However, dynamicity makes the characterization of the expected behavior, and consequently the identification of the anomalous one, a complex challenge. As an example, the set of (web)services or applications provided by the target system may change: services may be updated, added or removed.

Consequently, the definition of the *expected behavior* needs to be repeatedly updated, because its validity is going to be deprecated through time. The collection of data in such dynamic systems needs frequent reconfigurations of the algorithms and of the monitoring strategy. As a negative drawback, most of the state-of-the-art techniques start suffering of issues related mainly to the performance, i.e., training phases can occur very often, reducing the efficacy of the strategy. Frequent training phases can ovecome the normal activity of a system, that may spend more time in training their anomaly-based error detectors or failure predictors than in executing the usual tasks.

OUR CONTRIBUTION In the literature, there are no clear answers on applying error or anomaly detection in highly dynamic complex systems. To contribute filling this gap, in this Thesis we discuss our approach to anomaly detection

in complex dynamic systems. Our starting point was the study in [18], where authors showed that their solution was very effective in detecting anomalies due to software faults in an *Air Traffic Management* (ATM) system, which has clear functionalities which are not intended to change along time. Therefore, we applied such approach to a prototype of the *Secure!* system, a *Crisis Management System* which is structured as a *Service Oriented Architecture* (SOA), or rather one of the most common implementations of a complex system.

At first, we observed that applying such strategy as it was described for non-dynamic systems does not provide comparable detection scores. Anomalies were not detected as it was described in [18], meaning that some changes needed to be implemented to improve the performance of the strategy for dynamic systems. Since in the original proposal the authors monitored data related to 10 *Operating System* (OS) indicators, we first tried to expand the pool of monitored system indicators by observing data related to different system modules instead of observing only OS data. More precisely, when services exposed from such systems change, the notion of expected and anomalous behavior may change consequently requiring a new configuration of the anomaly detector: since layers underlying the services' layer are not modified when services change, the monitoring system is unaltered. It follows that the adoption of this *multi-layer* monitoring strategy does not negatively affect the applicability of our solution for dynamic systems.

In addition, to have a better characterization of the expected behavior of the system, we gathered some information related to the current context of the system which may be obtained at runtime e.g., which are the services being called at a given time instant. *Context-awareness* is then used to train the parameters of the implemented anomaly detection algorithm, tailoring them on the current context and ultimately maximizing their ability in detecting anomalies. As in [18], the algorithm we selected in our implementation is SPS (*Statistical Predictor and Safety Margin*, [14]), which is able to predict an acceptability interval for the next observed value based on a sliding window of past observations. SPS is more suitable for dynamic systems than other algorithms as clustering or neural networks [25] since it just requires short periods of training and consequently it is faster in recomputing the best values of its parameters.

Lastly, we performed dedicated tuning of parameters of the whole strategy, to understand which indicators provided actionable information for anomaly detection, and how to combine all the different alerts that the instances of the SPS algorithm - one for each data series related to the same indicator - may raise. This process led us to a customized version of the initial strategy in [18] which has comparable scores with respect to other works targeting non-dynamic complex systems. This is substantiated by an extensive experimental assessment conducted by conducting anomaly detection alternatively on the Secure! [4] *Crisis Management System* (CMS) and on *jSeduite* [30]. Both systems are structured

as a *Service Oriented Architecture* (SOA), where services are managed by different entities and are deployed on different nodes. Such services may incur in frequent updates, or even new services may be introduced, together with modification to their orchestration. Consequently, while instrumenting each service with monitoring probes is unfeasible, the opportunity to observe the underlying layers, i.e., *middleware*, *Operating System* (OS) and *network*, is offered in both systems.

THE FRAMEWORK Our experimental process lead devising some design patterns to be followed in order to define a monitoring and anomaly detection framework for a dynamic complex system. Design solutions are presented according to viewpoints, or rather dimension of analysis of a generic monitoring and anomaly detection system: *Purpose*, *type of Anomalies*, *Monitoring Approach*, *Selection of the Indicators*, and *Detection Algorithm*. For each viewpoint, design solutions are proposed to build an anomaly detection framework limiting the adverse effects that dynamicity - and consequent reconfigurations and context changes - has on our ability of identifying anomalies.

Such design patterns are then implemented in *MADneSs*, a *Multi-layer Anomaly DetectioN framEwork for complex Dynamic SystemS* tackling the challenges above. The monitoring approach we adopt in *MADneSs* consists in shifting the observation perspective from the application layer, where services operate, to the underlying layers, namely *operating system* (OS), *middleware*, and *network*. This allows detecting anomalies due to errors or failures that manifest in services that are not directly observed. Further, a more accurate definition of the context i.e., context-awareness, improves the detection accuracy. When monitoring a wide set of indicators, the most relevant for anomaly detection purposes should be identified depending on the current context, which is reconstructed by dedicated mechanisms.

SUMMARY AND THESIS STRUCTURE Overall, our main contributions to the state of the art on anomaly detection in complex dynamic systems consist in:

- depicting the typical challenges for performing anomaly detection in complex dynamic systems, togheter with the proposed solutions;
- structuring a multi-layer monitoring module observing different system layers;
- describing how the knowledge of the context i.e., context-awareness, can be used to improve anomaly detection;
- assessing the whole solution on more case studies;
- defining a methodology and the associated *MADneSs* framework for anomaly detection in dynamic complex systems;

- comparing *MADneSs* with related state-of-the-art frameworks.
- exploring possible future applications of our approach in *Systems-of-Systems*, a recent architectural paradigm for complex systems.

The Thesis is organized as follows. Section 2 presents the state of the art on dependability, monitoring and anomaly detection; the other sections build on such basics. Section 3 describes our process of building an anomaly detection strategy that is suitable for dynamic complex systems, built as subsequent incremental improvements of an initial solution. The section concludes discussing the typical challenges we face when performing anomaly detection in complex systems. The framework obtained at the end of this section is targeted by an extensive experimental campaign, which is presented in Section 4 along with the discussion of the results obtained. In such experimental campaign we applied our resulting framework to both the *Secure!* and the *jSeduite* Service Oriented Architectures. Section 5 presents the design guidelines which we devised during our process: these are dimensions of analysis of each monitoring and anomaly detection system for complex and dynamic systems. In the same section, such design guidelines are implemented in the resulting *MADneSs* framework, along with the associated methodology. Possible future implications of our findings are deepened in Section 6, with a particular emphasis that is put on applying anomaly detection in *Systems-of-Systems*. Section 7 concludes the Thesis.

2

BASICS AND RELATED WORKS

This section reports on the basics on dynamic and evolving systems, together with some basics on dependability and anomaly detection. Related works and state-of-the-art contributions are reported within the descriptions above.

2.1 DEPENDABILITY: THREATS, ATTRIBUTES, MEANS

According to [10], *Dependability is the ability to deliver service that can justifiably be trusted*. This definition stresses the need for justification of trust. Looking more in detail to services, an alternate definition is:

The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.

It is usual to say that the dependability of a system should suffice for the dependence being placed on that system. Therefore, the dependence of system A on system B represents the extent to which system A's dependability is (or would be) affected by that of system B. The concept of dependence leads to trust, which can very conveniently be defined as accepted dependence.

This section presents a basic set of definitions that will be used in the rest of the thesis. When dealing with dependability and security of computing and communication systems, the reference taxonomy and definitions are those given in [10]: this work is the result of a work originated in 1980, when a joint committee on *Fundamental Concepts and Terminology* was formed by the technical committee on *Fault-Tolerant Computing* of the IEEE CS.

2.1.1 *Basic Definitions*

In [10], a system is defined as an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena; the environment of the given system is composed by these entities. The boundaries of a system are the common frontiers between the system and its environment.

What the system is intended to do represents the function that a system realizes and is described by the functional specification for the subsequent validation. **What the system really does to implement its function**, is defined, instead, as the behavior of the system and is described by a sequence of states

that are targeted by verification processes. These two notions were formally defined in [33] as

Validation. The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with verification.

Verification. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation.

The system can be recursively seen as the integration of a set of components interacting together, where each component recursively is another system. When a component is considered to be atomic (or any further internal structure is not of interest) the recursion stops. Consequently, the total state of a system is a combination of the set of the (external) states of its atomic components.

The behaviour of a system (provider) as it is perceived by another system (user) is referred as *service*. The structure of the service can be considered an ensemble of three different parts:

- *interface*, or rather the provider's system boundary where service delivery takes place;
- *external state*, the part of the provider's total state that is perceivable at the service interface, while
- the remaining part is its *internal state*.

2.1.2 *Threats to Dependability: Faults, Errors, Failures*

When a given service implements the system function, the delivered service is correct. On the other side, when a service does not comply with the functional specification, one or more service *failures* occur. In other terms, a service failure can be seen as a transition from a correct provision of a service to an incorrect one, or rather to a state that does not implement the system function anymore. The period of delivery of incorrect service defines the *service outage*. The transition from incorrect service to the correct service is the *service recovery*. The deviation from a correct service may assume different forms that are called service *failure modes* and are ranked according to different severities.

Since a service is a sequence of the system's external states, a service failure means that **at least one (or more) external states of the system deviate from the correct service state**. The deviation is called an **error**, while the adjudged or hypothesized cause of an error is a **fault**. For this reason, the definition of an error is the part of the total state of the system that may lead to its subsequent service failure. It is important to note that errors may not reach the system's

external state and cause a failure. A fault is *active* when it causes an error, otherwise it is *dormant*.

Faults, errors and failures are considered threats to dependability as they can induce a system to deliver an incorrect service, or to not deliver the service at all. More in detail, the computation process can cascadingly induce an error that can propagate within a given component (internal propagation): an error is successively transformed into other errors. Error propagation from one component to another component occurs when an error reaches the service interface of the first component. When the functional specification of a system includes a set of several functions, the failure of one or more of the services implementing the functions may leave the system in a degraded mode that still offers a subset of needed services to the user. The specification may identify several such modes, e.g., slow service, limited service, emergency service.

2.1.3 *Dependability Attributes*

As developed over the past three decades, dependability is an integrating concept that encompasses the following attributes:

- *Availability*: readiness for correct service;
- *Reliability*: continuity of correct service;
- *Safety*: absence of catastrophic consequences on the user(s) and the environment;
- *Integrity*: absence of improper system alterations;
- *Maintainability*: ability to implement modifications and repairs;
- *Confidentiality*: the absence of unauthorized disclosure of information.

Based on the above definitions, *Security* is defined as the composition of the following attributes: confidentiality, integrity, and availability.

2.1.4 *Means to Attain Dependability*

Over the past 50 years many means have been developed to attain the various attributes of dependability and security. Those means can be grouped into four major categories:

- *Fault Prevention*: the techniques belonging to this class aim at preventing the occurrence or the introduction of faults in the system. Examples are design review, component screening, testing, quality of control methods, formal methods and software engineering methods in general.

- *Fault Tolerance*: the fault tolerance techniques allows a system providing a correct service even in presence of faults. Fault tolerance is carried out by error processing and fault treatment: the first aims at removing errors from the computational state, possibly before the occurrence of a failure, while the second aims at preventing faults from being activated again. This Thesis will address methodologies that belong to this group.

- *Fault Removal*: these techniques aim at reducing the presence (amount, likelihood) of faults, and they are obtained by means of a set of techniques used after that the system has been built. They are verification (checking whether the system fulfills its specifications), diagnosis (diagnosis the fault which prevented the verification conditions from being fulfilled), and correction.

- *Fault Forecasting*: the purpose of fault forecasting techniques is to estimate the number, the future incidence and consequences of faults. Indeed, no existing fault tolerant technique is capable to avoid a failure scenario. Therefore, fault forecasting represents a suitable mean to verify the adequacy of a system design with respect to the requirements given in its specification.

Fault and Error Classification

All faults that may affect a system during its life are classified according to eight basic viewpoints, leading to the elementary fault classes, as shown in Figure 1. If all combinations of the eight elementary fault classes were possible, there would be 256 different combined fault classes. However, not all criteria are applicable to all fault classes; for example, natural faults cannot be classified by objective, intent, and capability. Possible combined fault classes can be classified in three major partially overlapping groupings. *Development faults* include all faults occurring during development; if the faults is originated by hardware malfunction, we speak about *physical faults*. Lastly, all the *external faults* are grouped as interaction faults. Knowledge of all possible fault classes allows the user to decide which classes should be included in a dependability and security specification.

Failure Modes

The different ways in which the deviation is manifested are a systems service failure modes [77]. Each mode can have more than one service failure severity. Ranking the consequences of the failures upon the system environment enables failure severities to be defined.

The CRASH scale is an important and very useful failure classification [59] where the authors define a failure as an incorrect error/success return code,

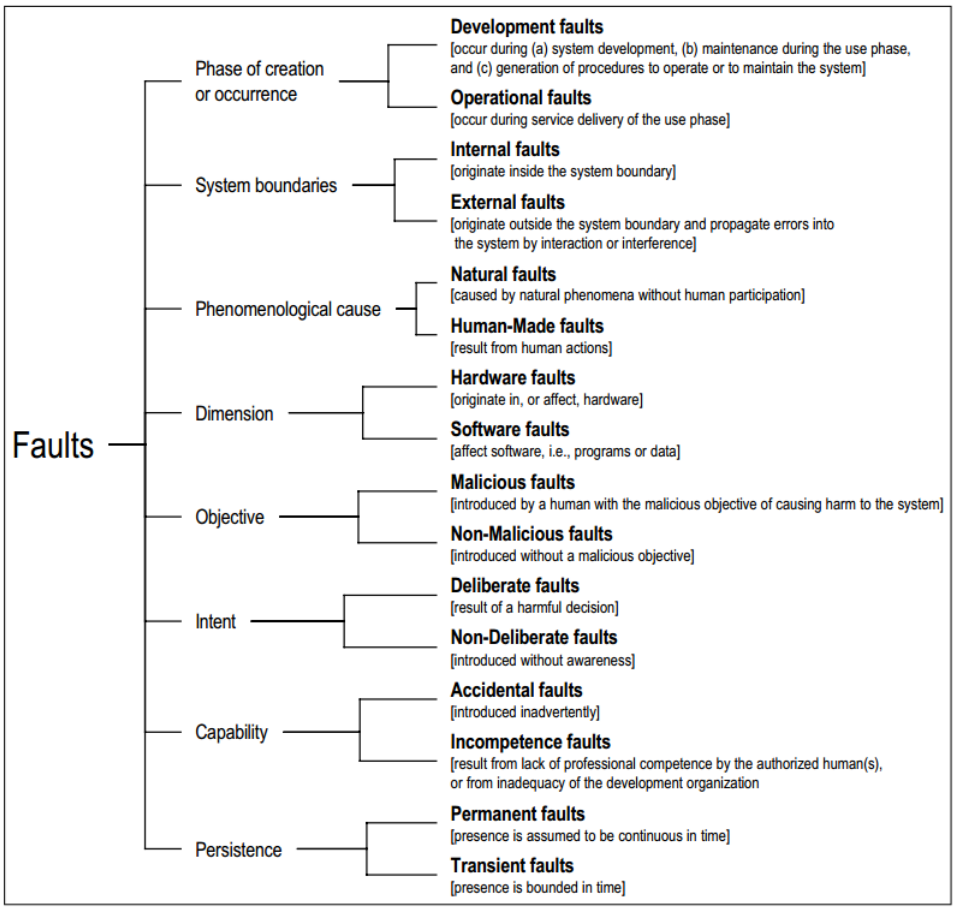


Figure 1: Fault Categories from [10]

abnormal termination, or loss of program control, stating that each failure revealed a robustness gap, in that the system failed to respond correctly to a set of inputs. As authors stated in such paper, it is possible to group failures according to the severity of the effect on an end-use system through the 5-points CRASH scale:

C Catastrophic (OS crashes/multiple tasks affected)

R Restart (task/process hangs, requiring restart)

A Abort (task/process aborts, e.g., segmentation violation)

S Silent (no error code returned when one should be)

H Hindering (incorrect error code returned)

The actual severity of each class on the 5-point scale depends on the specific application area. In a typical workstation environment, the CRASH ordering represents decreasing order of severity in terms of operational impact (i.e., *catastrophic* is the most severe, and *hindering* the least severe). However, relative severity could vary depending on the specific point of view. For example, software developers might perceive *silent* failures as a tricky concern because they indicate that an opportunity for software bug detection has been missed. A missed software bug may lead to an indirect error manifestation that is more difficult to track down, or may leave a latent bug to crop up after the software is fielded. For fault-tolerant systems, the *restart*, *silent*, and some *catastrophic* failures may be of the most concern. This is because fault-tolerant systems are typically constructed with a fail-fast assumption. Indeed, a *restart* failure or *hanging* types of *catastrophic* failure would create an undetected failure that persists for a period of time, rather than a fast system failure.

2.2 DYNAMIC AND EVOLVING COMPLEX SYSTEMS

A formal discussion on dynamic and evolutionary properties of complex systems comes from [23], where authors specifically target *Systems-of-Systems* (SoSes), an architectural paradigm which defines a system as an ensemble of subsystems with specific peculiarities.

Amongst all the peculiarities of SoSes, we remark here that most of the basic characteristics of SoSs are common to all the complex systems that are usually employed for different purposes e.g., *Service Oriented Architectures* (SOAs) or *Cloud* environments. These systems are intended to provide specific (web)services to the final user, and may change over time to suit different needs of the company or the individuals that are requesting such services. It follows that these systems are often not meant to change their behavior during all their operational life. They are characterized by specific properties such as

dynamicity and evolution that determine their current and future behavior. We define:

Evolution: Process of gradual and progressive change or development, resulting from changes in its environment or in the system itself.

Dynamicity: The capability of a system to react promptly to changes in the environment. A system that is not intended to change during its life is called static or semi-static.

Changes usually regard the services the system exposes to the final user. As example, a company which has *Business Intelligence* processes deployed on a SOA may want to update some of the key features to stay up-to-date with the market, or new customers may want some internal services act remotely. Moreover, new functionalities can be added to the system, as well as its internal organization, that may evolve by adding more hardware resources or more machines to the network, to the grid or to the cloud. Systems that are constituted by the aggregation of several subsystems are often called *complex systems*.

"**Complex Systems** is an approach to science studying how relationships between parts give rise to the collective behaviors of a system, and how the system interacts and forms relationships with its environment. There are three interrelated approaches to the modern study of complex systems: i) how interactions give rise to patterns of behavior, ii) the space of possibilities, and iii) the formation of complex systems through pattern formation and evolution". [12]

In a nutshell, complex systems that provide services are forced to evolve to suit the novel necessities of the user. However, checking the behavior of a complex system is not trivial. Some of its parts may be owned by third-party components, making the observation of the internals not possible. Moreover, due to dynamic and evolutionary characteristics of such systems, the usual behavior of some services may vary frequently, leading to reconfigure the whole behavior-checking structure. As a result, checking each individual service to monitor dependability-related properties is generally unfeasible [26] if not possible at all. Several works in the last decade [66], [29], [104] confirm this aspect, reporting on the difficulties of detecting service errors in complex systems before they escalate into a more severe failure. These difficulties are mainly related to the multitude of relations, interconnections and interdependencies that propagate potentially wrong decisions through the system: an update, a configuration change, or a malfunction in a single module or service can affect the whole system.

2.3 MONITORING EVOLVING SYSTEMS

To ensure that the system behaviour complies with its functional specification in their final environment, extensive and costly efforts are put during the system development cycle. Despite the evolution of fault prevention practices - that are used to increase the correctness of the system design and the software implementation -, *no existing technique is capable of avoiding a failure scenario*. When observing the behavior of computing systems, first it is required to monitor its actual behavior, and then to evaluate it. In both of these phases, the more accurate we are in the observation of the system, the more reliable are the results we collect, and the more accurate are the decisions we are able to take:

- in *on-line monitoring*, we collect data related to the system behavior;
- in the evaluation, we are able to take *off-line* decisions using collected information.

Depending on the system details, monitoring can present particular challenges [65]. System internals are not always easily observable, or are not observable at all. Many systems have limited resources or have real-time software requirements that must adhere to strict execution deadlines. It follows that we need to minimise the overhead of monitoring, since high overhead could compromise core system resources or affect scheduling, causing interference to the target application. The activity of checking monitored data can be performed:

- at runtime, if the monitoring outputs are checked at runtime,
- off-line, if the monitoring outputs are first stored in dedicated data structures and then analyzed in a post-processing phase.

This approach of monitoring and checking can be used for various purposes, as testing, debugging, verification, logging of errors, fault recovery of the system to a safe state and maintenance/diagnosis procedures in the field. Examples of runtime checking systems are detailed in [43], [68] [32]. Published reviews of monitoring approaches have been presented in the past [75]. In particoular, this review considers various monitoring approaches, including: the use of internal system signals with hardware probes (*hardware monitoring*), the addition of code to the target system's software in order to perform operations related to monitoring at certain points in the application's execution (*software monitoring*), the combination of both approaches (*hybrid monitoring*). Selected references [49], [40], [100] highlight the problems and advantages that have been considered in the past relating to these approaches. Emphasis is placed on achieving non-intrusive (or minimally intrusive) monitoring in embedded systems. The importance of non-intrusive monitors and the problems with intrusive software

monitoring systems are discussed in [49], where the authors present a less intrusive software monitoring system. Moreover, [40] examines non-intrusive and low intrusive monitoring in the context of more complex systems, and in particular embedded systems [100].

Some of the recent works adopt *Complex Event Processing* (CEP) systems to manage streams of data, especially for complex, large scale systems where large amounts of information is generated. We report here two relevant examples that have been considered as references to structure our work. In [98] and [37] the high flexibility allowed by CEP queries is used for anomaly detection in critical infrastructures. In particular, [98] proposes a Security Information and Event Management (SIEM) framework where the event correlation engine is able to parallelize arbitrary rules expressed as CEP queries. In [37] authors propose a CEP approach for on-line detection and diagnosis, which allows to detect intrusions by collecting diverse information at several architectural layers as well as perform complex event correlation.

2.3.1 *Basic Definitions*

In this thesis, the system in which monitoring activities are performed is referred to as the **target system** or **system under test**, and the software application whose execution is being monitored is referred to as the **target application** or **target service** [65].

Although the term monitoring can be used synonymously with observing, this Thesis uses the term monitoring to mean the combined task of observing and making use of the observations to track the target's behaviour. The specific elements of a monitor that practically allows to observe a target system are called **probes**. Probes are attached to the system (or placed within it) in order to provide information about the system's internal operation, and provide an intermediate output. Such probes can be classified [93] as

- *hardware probes*: pieces of additional hardware that monitor internal system signals;
- *software probes*: code instructions added to the target software in order to output information about the internal operation of the program.

The installation of probes in a given target system is called instrumentation. Probes must be capable of observing enough information about the internal operations of the system to fulfill the purpose of the monitoring. Moreover, the behaviour of the target system should not be affected when such probes are added.

2.3.2 *Classification of Monitoring Systems*

The classification of monitoring systems according to the probing approaches above has been fully reported in [93].

Hardware Monitors

A typical hardware monitoring arrangement or "bus monitor" uses dedicated monitoring hardware on the target system, e.g., to observe information sent over the internal system bus. The observed data is sent by the monitoring hardware for verification. The non-intrusivity of the hardware monitors for the target system using additional hardware for the monitoring function is a key advantage. The difficulties of monitoring the hardware are expanded in [43], where the authors noted the problem of newer systems offering less physical probe points, and suggested at the time that hardware monitors were becoming obsolete. Moreover, in [68] the authors also note the inapplicability of hardware monitors with respect to complex systems.

Software Monitors

There are different ways to modify the target system through the addition of software for monitoring. One solution is to perform observations directly on the target application, adding the software probes to the target application code. Alternatively, software probes can be inserted within the operating system of the target system. Another possible solution is to implement the probes as separate processes. The advantage of monitoring a system using a software monitor within the target is acknowledged in [31], where such monitors have access to extensive information about the operation of a complex system, in contrast to the limited information available externally to hardware probes.

HYBRID MONITORS Hybrid monitoring refers to approaches that use a combination of additional software and hardware to monitor a target system, relying on the advantages of each approach and at the same time attempting to mitigate their disadvantages. For example, the source code of the target process is instrumented; an additional code is added to it to emit events when process features (e.g. variable values) being monitored are updated. These events are sent to the dedicated monitoring hardware, which analyses the events and checks them against the provided monitoring rules. Various hybrid monitoring systems have been developed for distributed systems [100], real-time systems [49] and embedded systems [20].

2.3.3 *On-line Monitoring*

On-line monitoring aims at observing system events from several perspectives (faults, attacks, vulnerabilities) while the system is running. Several proposals exist where monitoring tools and facilities are proposed with slightly different aims and capabilities. Some examples of automatic on-line monitoring tools are the following [27]:

SWATCH (Simple WATCHer) [48]: it generates alerts based of the recognition of specific patterns while on-line scanning **log files**; its main drawback is that it does not support means to correlate recognized events. Swatch constitutes the basis of *LogSurfer* [78]

SEC (Simple Event Correlator) [82]: it recognizes specific patterns based on predefined rules; SEC is able to **correlate** observed events and trigger specific alarms.

Monitoring mechanisms in critical infrastructures are required to filter the events observed to understand the nature of errors occurring in the system, with respect to multiple classes of faults, ultimately feeding the diagnostic facilities. As it can be noted, most of these tools are using log files as the primary source of information. However, data can also be obtained by dedicated probes which read specific system indicators such as the memory usage.

In any case, the monitored information is extracted at runtime, filtering all the available information in order to identify only information relevant for monitoring purposes (the so called *events*); the diagnostic engine then processes the flow of the observed events, identifying unexpected trends both from the observation of suspicious events and from the correlation of specific simple not-alarming events [32]. The goal is in fact to recognize not only malfunctions on top of self-evident error events, but also alarming situations on top of observed (possibly not alarming) events. Such approach is general enough to be used for the identification of different kinds of alarming situations such as i) hardware faults (e.g. a *stuck* at error message), ii) software faults (e.g. an *exception* message), or iii) malicious attacks (e.g. several messages about a failed attempt to *login as root*).

2.4 ANOMALY DETECTION

The analysis of monitored data is a crucial stage to recognize if something unexpected has occurred in the system and to trigger error and fault handling procedures. Faults and errors in the *internal state* can be addressed at different levels, from the hardware (e.g., machine check, division by zero, etc.), to the services. In any case, the aim is to detect errors before they can propagate to

errors or failures for other different component(s) or to failure for the overall system.

Most of these techniques belong to direct detection strategies: they try to infer the system health status by directly querying it, or by analyzing events it is able to produce (e.g., logs, exceptions). However, problems can arise because of the nature of software faults. According to [6], it is impossible to anticipate all the faults and their generated errors in a system: *"If one had a list of anticipated faults, it makes much more sense to eliminate those faults during design reviews than to add features to the system to tolerate those faults after deployment. It is unanticipated faults that one would really like to tolerate"*. Additionally, in [36] authors describe experiments performed on a *Wide Area Network* to assess and fairly compare the *quality of service* (QoS) provided by a large family of failure detectors. Authors introduced choices for estimators and safety margins used to build several failure detectors.

For instance, the activation of a fault manifests in an incorrect internal state, i.e., an error; however, this does not necessarily mean that this error is detected. Consequently, some errors generate an exception or a message, while others are present but not yet detected. Besides causing a failure, both latent and detected errors may cause the system or component to have an anomalous behavior as a side effect. This non-normal behavior is called **anomaly**.

2.4.1 *Definitions*

According to [25], anomaly detection refers to the problem of *finding patterns in data that do not conform to the expected behavior*. Such patterns are changes in the system indicators characterizing an unstable behavior of the system caused by specific and non-random factors. For example, pattern changes can be due to a system overload, the activation of software faults or malicious activities. In the same work, authors classify anomalies as:

- *point*: a single data instance that is out of scope or not compliant with the usual trend of a variable. It is sometimes called outlier;
- *contextual*: a data instance that is unexpected in a context;
- *collective*: a collection of related data instances that is anomalous with respect to the entire data set.

Depending on the system, data can be stored in different ways. The way the data is stored strongly influences the data analysis strategy that turns out to be suitable for such case study. More in detail, some datasets provide one or more labels associated with a data instance, denoting if that instance of data corresponds to an unstable state of the system, e.g., an attack was performed at the time instant in which the data instance was collected. It should be noted that

obtaining labeled data which is accurate as well as representative of all types of behaviors is often prohibitively expensive. Labeling is often done manually by a human expert and hence requires a substantial effort to obtain the labeled training data set. Based on the extent to which the labels are available, anomaly detection techniques can operate in one of the following three modes [25].

Supervised Anomaly Detection

These techniques assume the availability of a training data set which has labeled instances for both normal and anomaly classes. A typical approach in such cases is to build a predictive model to classify *normal* and *anomaly* classes. Any unseen data instance is compared against the model to determine which class it belongs to. There are two major issues that arise in supervised anomaly detection. First, the anomalous instances are far fewer compared to the normal instances in the training data. Issues that arise due to imbalanced class distributions have been addressed in the data mining and machine learning literature [54]. Second, obtaining accurate and representative labels, especially for the anomaly class is usually challenging. A number of techniques that inject artificial anomalies in a normal data set to obtain a labeled training data set have been proposed [94], [7]. Other than these two issues, the supervised anomaly detection problem is similar to building predictive models.

Semi-Supervised Anomaly Detection

Techniques that operate in a semisupervised mode assume that the training data has labeled instances either for the normal or the anomaly class. Other instances are not labeled, meaning that they can belong to the same class of the labeled instances or that they can build a new class. Since they require less labels, their range of applicability is wider than supervised techniques. For example, in space craft fault detection [42], an anomaly scenario would represent an accident which is not easy to model. The typical approach used in such techniques is to build a model for the class corresponding to normal behavior, and use the model to identify anomalies in the test data by difference. A limited set of anomaly detection techniques exist that assume availability of *only* the anomaly instances for training [28], [38]. Such techniques are not commonly used, mainly because it is difficult to obtain a training data set which covers every possible anomalous behavior that can occur in the data.

Unsupervised Anomaly Detection

Techniques that operate in unsupervised mode do not require labeled training data, and thus are most widely applicable. The techniques in this category make the implicit assumption that *normal instances are far more frequent than anomalies in the data*; breaking this assumption would lead these techniques to generate a

huge number of false alarms. As a remark, many semi-supervised techniques can be adapted to operate in an unsupervised mode by using a sample of the unlabeled data set as training data.

2.4.2 *Algorithms for Anomaly Detection*

Different types of algorithms can be implemented depending on the structure of the system or the dataset under investigation. We report here some examples of unsupervised algorithms (see Figure 2), since they do not require labelled data and consequently are suitable in most of the application domains.

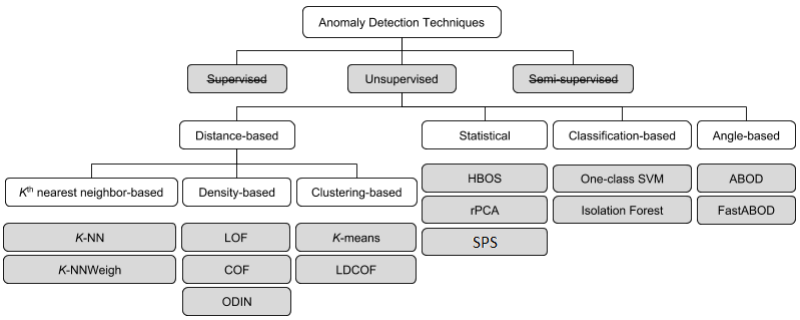


Figure 2: Families of Unsupervised Algorithms

Clustering-Based: K-means

K-means [85] is a popular clustering algorithm that consists of grouping data points into k clusters by their feature values. Objects are classified in the same cluster when they have similar feature values. First, the k cluster *centroids* are randomly initialized. Then, each data record is assigned to the cluster with the nearest centroid and the centroids of the modified clusters are recalculated. This process stops when the centroids are not changing anymore. A distance function should be specified in order to compute the distances between a data record and the centroids. The most popular distance function is the *Euclidean* distance, defined as:

$$d(x, y) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

where x and y are two input vectors with m quantitative features. Finally, scores of each data point inside a cluster are calculated as the distance to its centroid. Data points which are far from the centroid of their clusters are labeled as anomalies.

Distance-Based: Kth-Nearest Neighbor (kNN)

kNN [81] is a *distance-based* method which was primarily designed to identify global anomalies. For each data point, the k nearest neighbors are computed. Then, an anomaly score is given as the distance to the kth-nearest neighbor. Consequently, the data points that have the largest kNN distance are classified as anomalies.

Density-Based: Local Outlier Factor (LOF)

LOF [19] is a *density-based* method designed to find local anomalies. For each data point, the k nearest neighbors are computed. Then, using the computed neighborhood, the local density of a record is computed as the *Local Reachability Density (LRD)*:

$$\mathrm{LRD}(x) = \frac{|\mathrm{kNN}(x)|}{\sum_{o \in \mathrm{kNN}(x)} d_k(x, o)}$$

where $d_k(x, o)$ is the *Reachability Distance*, or rather the Euclidean Distance if the cluster in not highly dense [44]. Finally, the LOF score is computed by comparing the LRD of a record with the LRD of the previously computed k-nearest neighbors:

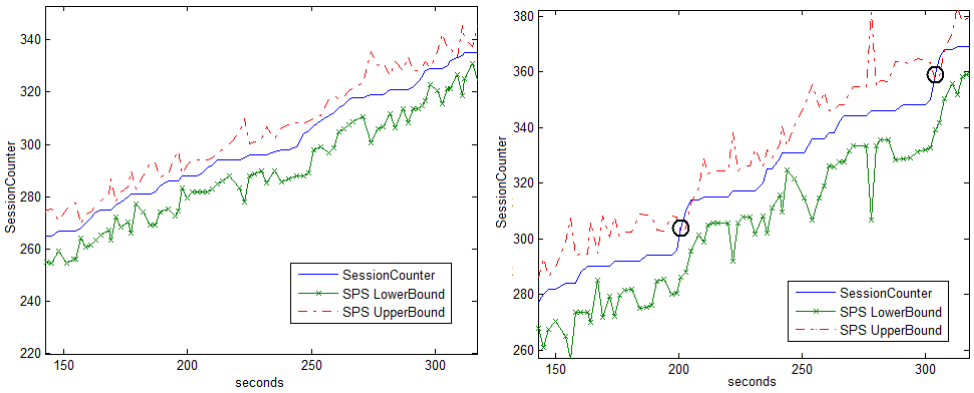
$$\mathrm{LOF}(x) = \frac{\sum_{o \in \mathrm{kNN}(x)} \frac{\mathrm{LRD}_k(o)}{\mathrm{LRD}_k(k)}}{|\mathrm{kNN}(x)|}$$

As LDCOF, data points classified as normal by LOF will have smaller scores, close to 1.0, and data points classified as anomalies will have larger scores, since both algorithms employ the idea of local densities.

Statistical: Statistical Predictor and Safety Margin (SPS)

The *Statistical Predictor and Safety Margin* (**SPS**) algorithm was initially designed to compute the uncertainty interval at a time within a given coverage, namely the probability that the next value of the time series will be inside the uncertainty interval. This algorithm can be adapted to compute adaptive bounds for anomaly detection activities with minor changes. As shown in [17], the uncertainty interval computed by SPS algorithm consists in a combination of i) the last value of the series, ii) the output of a predictor function (P), and iii) the output of a safety margin (SM) function, namely:

$$T_l(t) = x(t_0) - P(t) - SM(t_0), T_u(t) = x(t_0) + P(t) + SM(t_0)$$



(a) Normal Execution (b) Faulty Execution

Figure 3: SPS applied to normal (a) and faulty (b) experimental traces logging the *Http Sessions Counter*. Black circles identify anomalies detected by SPS.

Where T_l and T_u are the upper and the lower bounds at time t , and t_0 is the last value of the series. The *predictor* function provides an estimation of the behavior of the time series. The *safety margin* function aims at compensating possible errors in the prediction and/or in the measurement of the actual value of the time series. The safety margin is computed at and it is updated only when new measurements arrive. We refer to [14] for technical details about the predictor and the safety margin functions.

In a nutshell, SPS works as follows. It computes a prediction of the behavior of an indicator, based on a statistical analysis of the past observations. The past elements are managed through a sliding window mechanism, which keeps track of the last elements of such data series. The prediction produces the interval $[T_l, T_u]$ in which the value of the indicator is expected to fall. If the value of the indicator is outside the interval, SPS signals that an anomaly is suspected for such indicator. An example can be seen in Figure 3, where SPS was applied to traces related to the observation of the number of *HTTP sessions* during a normal (Figure 3a) and to a faulty (Figure 3b) experiment.

Angle-Based: Fast Angle-Based Outlier Detection (FastABOD)

This angle-based approach is a variant of the ABOD [47] algorithm. For each point, ABOD computes the angle to all the other pairs of points, while the weighted variance of these computed angles is used as anomaly score. **FastABOD**, instead, computes these angles only to the pair of points among the kNNs.

Classification-Based: One-class Support Vector Machine (one-class SVM)

Differently from the supervised support vector machines (SVMs), this algorithm is a commonly used method for semi-supervised anomaly detection [25] that aims to learn a decision boundary to group the data points [84]. However, **one-class SVMs** *can be used for unsupervised anomaly detection* when applying a soft-margin. In this scenario, the one-class SVM is trained with the dataset and then each data point is scored, considering as score the normalized distance of the data point from the determined decision boundary [8].

3

ANOMALY DETECTION IN COMPLEX DYNAMIC SYSTEMS

We present here the process we followed to identify the key challenges and obstacles behind the definition of an anomaly detection framework for dynamic complex system. All the steps we made are presented sequentially, highlighting the technical advancements we achieved. However, details about each step are expanded in the following chapters, along with a more complete experimental evaluation.

3.1 MOTIVATION AND OPEN CHALLENGES

Currently, **there are no clear state-of-the-art answers on applying error or anomaly detection in highly dynamic and complex systems**. High dynamicity, along with evolutionary characteristics of such systems, often make the characterization of the expected behavior, and consequently the identification of anomalies, a challenging task. In addition, complex relationships among modules lead to unpredictable behavioral fluctuations.

Problems are mainly related to the necessity of frequently reconfiguring the detection algorithms to match changes of the observed system. When the current context of the system changes very often, the definition of anomaly changes accordingly. While *point* anomalies are not usually dependent on the context, *contextual* and *collective* anomalies cannot be defined without characterizing the expected behavior of the system. Therefore, performing anomaly detection strongly depends on our ability to define a normal - expected - behavior as long as the system evolves.

Dealing with systems that may often change, most of the techniques presented in the literature start suffering of issues related mainly to the performance, since training phases can occur very often, reducing the efficiency of the strategy. Frequent training phases can ovecome the normal activity of a system, that may spend more time in training their anomaly-based error detectors or failure predictors than in executing the usual tasks.

3.2 ANOMALY DETECTION IN COMPLEX DYNAMIC SYSTEMS

As remarked in the previous section, anomaly detectors have been proposed to detect errors and intrusions [66] or to predict failures [11], based on the hypothesis that the activation of a fault or the manifestation of an error generates increasingly unstable performance-related behavior before escalating into a

failure. An anomaly detector analyzes such behavior, eventually triggering alerts.

3.2.1 *Detecting Specific Anomalies*

Point anomalies can be detected using algorithms that identify outliers [35] in a flow of observations, such as pattern recognition [46] or statistical-based methods which are able to reconstruct the statistical inertia of the data flow under investigation [18], [76].

Contextual anomalies can be detected by techniques that are able to tune their behaviour depending on the current state of the system. Concisely, they define the expected behaviour of the current context in which the system is running; successively, they use historical [35], user/operator-provided [106], or runtime [102] data to compare the expectations with the data provided by the monitoring modules. In this way, a single value that is not classified as a point anomaly can be labeled as anomalous because it is not expected in the current context.

Collective anomalies are usually harder to detect [25], and require more sophisticated detection techniques, either looking for specific patterns [106] or using wider training sets to better define them.

In the following, we give a report on the frameworks that perform anomaly detection in complex systems. A summary is reported in Table 1.

Framework	System Under Test			Anomaly Detection	
	Category	Case Study	Dynamicity	Target	Anomalies
ALERT [92]	Cluster Environment	*IBM* System S Stream Processing, PlanetLab	Medium	Anomaly Prediction	Point, Contextual
CASPER [11]	Air Traffic Control	*Selex-ES* System	Very Low	Failure Prediction	Point
[9]	Distributed Application	Web Banking	Low	Error Detection	Point, Collective
SEAD [73]	Cloud	*RUBiS* Distributed Online Benchmark, MapReduce	Low	Failure Prediction	Point, Contextual
TIRESIAS [101]	Distributed Environment	*EmuLab*	Low	Failure Prediction	Point
[41]	Distributed Environment	*Hadoop, SILK*	Low	Workflow Error Detection	Point
SSC [87]	Web Services	*DayTrader* Benchmark Application	Low	Intrusion Detection	Point, Contextual
McPAD [74]	-	*DARPA, GATECH, ATTACK* datasets	Very Low	Intrusion Detection	Point

Table 1: Anomaly Detection Frameworks for Complex Systems in the Literature

3.2.2 *Frameworks for Anomaly Detection in Complex Systems*

Different parts of a system have been deemed suitable for anomaly detection purposes. For example, indicator values obtained from observing the network layer are generally suitable for intrusion detectors [46], [76], while the operating system is usually monitored when detecting performance issues [9] or malware activities [99]. Some anomaly detection algorithms dynamically adapt their behavior to suit the current state of the system [73], [76], but in general they are not supported by a framework that manages a dedicated data collection without requiring manual intervention at services change.

Instead, several studies describe frameworks tackling anomaly detection in complex systems that are not expected to frequently change along time i.e., semi-static systems. In general, these works address either error detection or failure prediction, gathering data about indicators related to multiple system layers. In particular, in CASPER [11] the authors adopt different detection strategies based on symptoms aggregated through *Complex Event Processing* (CEP) techniques using data gathered by the non-intrusive observation of network traffic parameters. TIRESIAS [101] predicts crash failures through the observation of network, OS and application-specific metrics by applying an anomaly detection strategy that is instantiated on each different monitored parameter. Differently, SEAD [73] aims at detecting configuration or performance anomalies in cluster and cloud environments by observing data coming from the middleware or the cloud hypervisor. In [9], the authors describe a process for invariant building, including advanced filtering and scoring techniques aimed at selecting the most relevant ones. Moreover, in [18] the authors applied SPS to detect the activation of software faults in an *Air Traffic Management* (ATM) system that has a defined set of services and predictable workloads. Observing only OS indicators, SPS allowed performing anomaly-based error detection with the highest scores among the surveyed works.

Noticeably, while the works mentioned above target semi-static systems, ALERT [92] aims at triggering anomaly alerts to achieve just-in-time anomaly prevention in dynamic hosting infrastructures. The authors propose a novel context-aware anomaly prediction scheme to improve prediction accuracy.

3.3 Our approach to anomaly detection in complex dynamic systems

This Thesis stems from studies by the same authors [14], [17] who devised anomaly detection strategies to perform error detection using the Statistical Predictor and Safety Margin (SPS, see Section 2.4.2) algorithm. SPS is able to detect anomalies without requiring offline training; this was proved to be very

effective in less dynamic contexts [18], where the authors applied SPS to detect the activation of software faults in an *Air Traffic Management* (ATM) system.

Due to the widespread use of complex architectures and systems such as Service Oriented Architectures (SOAs, [34]) and Cloud [66] systems, we tried to adopt such successful strategy to a different category of systems, which we will call *dynamic complex systems* in our work. As highlighted in the previous section, in such systems the traditional anomaly detection techniques may not be applicable due both to frequent changes and to a very high number of connections and relations among modules of such systems. In the rest of this section we will describe step by step all the technical advancements we made during our work to define a suitable strategy for performing anomaly detection in complex dynamic systems, using [14], [17], [18] as core basic works.

3.3.1 *Case Study for Evaluation*

We executed our experiments to evaluate the effectiveness of our approach on a prototype of the *Secure!* Crisis Management System (CMS) [4]. *Secure!* exploits information retrieved from a large quantity and several types of data sources available in a target geographical area, in order to detect critical situations and command the corresponding reactions including guiding rescue teams or delivering emergency information to the population via the *Secure!* app. Input data to Secure! may come from the following sources: i) social media as for example *Twitter*, ii) web sites, iii) mobile devices and their embedded sensors as camera, microphone and GPS, iv) sensor networks available in the infrastructures (e.g., surveillance cameras, proximity sensors). Data is received, collected, homogenized, correlated and aggregated in order to produce a situation for the CMS system that is ultimately shown to operators in a control room which take the appropriate decisions. The final *Secure!* prototype is composed of different virtual machines in charge of the different services. While all the virtual machines will have the same configuration in terms of operating system, application server and security solutions, virtual machines and services are owned by different partners and may change due to updates, reconfigurations, or modification of their orchestration. Togheter, such machines build a distributed SOA, which exposes services for managing, processing and storing critical information on crises which are used by dedicated mobile apps and accessed by authorites when needed.

3.3.2 *Metrics for Evaluation*

As authors remark in [17], metrics coming from diagnosis literature are usually used to compare the performance of detectors: *coverage* measures the detector ability to reveal an anomaly, given that an anomalous trend really occurs;

accuracy is related to mistakes that an anomaly detector can make. Coverage can be measured as the number of detected failures divided by the overall number of failures, while for accuracy there are different metrics. Moreover, in [13] consider the *mean delay for detection* (MDD) and the *mean time between false alarms* (MTBFA) as the two key criteria for on-line detection algorithms. Analysis are based on finding algorithms that minimize the mean delay for a given mean time between false alarms and on other indexes derived from these criteria.

Another group of metrics [88] comes from pattern recognition and information retrieval with binary classification. All of these metrics are based on indexes (see Figure 4) representing the correct predictions (*true positives*, TP, or *true negatives*, TN) and the wrong ones, due to missed detections (*false negatives*, FN) or wrong anomaly recognitions (*false positives*, FP). More complex measures based on the abovementioned ones are: *precision* (also called positive predictive value), the fraction of retrieved instances that are relevant, *recall* (also known as *coverage*), the fraction of relevant instances that are retrieved, and *F-Score(β)*. Especially in the F-Score(β), varying the parameter β makes it possible to weight the precision with respect to the recall (note that F-Score(1) is referred as *F-Measure*).

However, since anomalies and related errors are rare events, using only precision and recall is not a good choice because they do not account for evaluating the detector mistake rate when no failure occurs. Hence, in combination with precision and recall, we use the *False Positive Rate* (FPR), which is defined as the ratio of incorrectly detected anomalies to the number of all the correctly labeled normal instances (TNs). Fixing precision and recall, the smaller the false positive rate, the better. Lastly, *Accuracy* [88] is the ratio of all correct decisions to the total number of decisions that have been taken (see Figure 4).

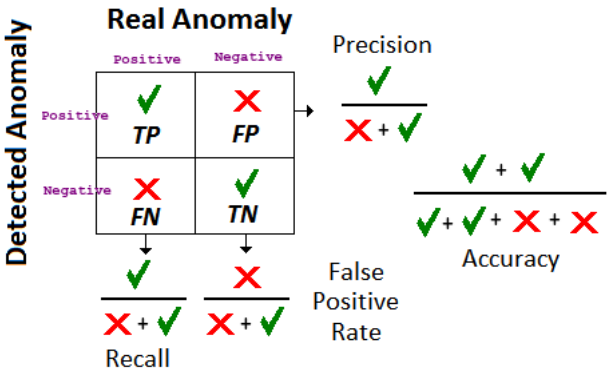


Figure 4: Scoring Metrics

As we noted in our literature analysis, not all the studies report FPR values, while almost all of them report their precision and recall scores. Despite not being optimal according to the conclusions above, we chose to adopt as main scoring metrics precision and recall, analyzing FPR where available. We aggregate

precision and recall in the **F-Score(2)** metric, which considers the recall twice more relevant than the precision. This is motivated considering that we are targeting critical systems, and consequently we prefer to reduce the occurrence of missed detections (FN), even at the cost of a higher rate of FP or rather low precision scores.

3.4 EXECUTING SPS ON A SERVICE ORIENTED ARCHITECTURE

3.4.1 *A First Experimental Setup*

First, we took the setup presented in [18] and we applied it to work with data collected from the *Secure!* system. This required to:

- **Creating a Workload** We identified 11 different safety/security-related web services, including authentication mechanisms, file storage, and event management that can be invoked by the *Secure!* users. Therefore, we created the *All Services* workload that invokes all them multiple times in different orders, with a time interval of 1 second between successive invocations. Overall, a single execution of the workload lasts approximately 65 seconds.

- **Anomalies Model**. In our experiments we inject faults that generate - among others - anomalies in the usage of the resources, namely the central memory (*MEMORY* anomaly), and the network, labeled as *NETWORK* anomaly.

- **Defining Injections** The injection points were chosen out of a list of functions used by the *Secure!* web services (package *com.liferay.portlet*). We targeted i) the function *documentlibrary.store.FileSystemStore* in charge of managing the addition of a directory in the Liferay filesystem, ii) the encoding strategy involved in the exchange of data between the database and the UI showing the calendar (*calendar.service.impl.CalEventLocalUtil._encodeKey* function), and iii) the creation of a SOAP model describing the request of adding a bookmark (*bookmarks.model.BookmarksFolderSoap.toSoapModel* function) in the user data. The injection is triggered by a chronometer: the code is mutated only when the timer expires. The chronometer is set to expire in three different instants, approximately at 70%, 80%, or 90% of the workload. This leads to 9 possible ways of injecting anomalies in our experiments: 3 injection points per 3 injection instants.

- **Instrumenting the OS** Probes that observe the OS indicators are located in kernel modules. 10 Indicators from OS are retrieved in our Linux implementation using *System Tap*, according to the setup in [18]. System Tap exploits the modularity of the Linux kernel design to produce a module that once loaded has visibility on kernel structures. The System

Tap compiler produces a kernel module which directly accesses kernel internal data. OS indicators generate signals for the next upper layers, and provide to a monitoring module with information about the current status of the system.

Overall, the whole experimental setup is composed by two machines. The first machine, or *Target Machine*, is one of the four virtual nodes composing the prototype of the *Secure!* system. This machine has been modified with the introduction of the OS probe, which is activated when each experiment starts and then shoutdown at the end of the experiment. The *Detector Machine*, instead, runs the data analysis logic (SPS algorithm) on the data that the OS probes installed in the Target Machine send through dedicated TCP socket. As soon as this data arrives, it is processed by the instances of the SPS algorithm, one for each of the 10 different indicators, which run in parallel. Also, to reduce the impact on the observed system, OS probes send their data without any rearrangement, which is left to the Detector Machine. Target and Detector Machines are virtual machines that run on a rack server with 3 Intel Xeon E5-2620@ 2.00 GHz processors.

Our experimental campaign was organized as follows: we executed 60 preliminary runs in which we inject faults generating the MEMORY anomaly and other 60 in which we inject faults generating the NETWORK anomaly. The faults generating the anomalies were i) *MEMORY* - an anomalous memory consumption (filling a Java LinkedList), and ii) *NETWORK* - a wrong network usage (fetching HTML text data from an external web page). The validation experiments are organized as follows: in 40 runs we inject the MEMORY anomaly, while in the other 40 runs the NETWORK anomaly is injected, always alternating the injection points as stated before.

3.4.2 *Analysis of the Results*

#	Layers	Indicators	P	R	F2	FPR
[18]	OS	10	97.0	100.0	99.3	1.9
[18] in SOA	OS	10	26.3	41.1	36.9	14.7

Table 2: Adapting SPS for SOA Systems

The first step in our process of building an anomaly detector for complex dynamic systems consists of adapting the work in [18] on the data gathered from the *Secure!* SOA. In such work authors applied SPS on data traces related to the trend of OS indicators on a virtual machine during the execution of an *Air Traffic Management* (ATM) system. The ATM case study, namely the SWIM-BOX

middleware for the interoperability of future ATM systems, consists of a system with complex functionalities that is intended to provide such functionalities or services for its entire life, without long-term planned evolutions or short-term dynamic changes. Differently from [18], we aim at applying SPS on the same OS indicators retrieved from a different target system. Nevertheless, our model of anomalies is different from the faults from *Orthogonal Defect Classification* (ODC [91]) authors injected in such paper. However, we tried to model generic anomalies in the memory and network usage, trying to replicate some of the effects that the manifestation of the ODC faults may have.

In our experiments we observed that switching to a more dynamic system had severe effects on the metric scores regarding anomaly detection. As it can be noted in Table 2, the same detection algoritm executed on data related to the same indicators on a more dynamic system lead to very poor metric scores. This makes the setup (which performed almost perfectly in the ATM system) useless in the *Secure!* system: it is worth mentioning the recall value of 41.1, which demonstrates how in our experiments SPS was able to detect only the 41% of the injected anomalies, while the remaining 59% were not detected at all. On the other side, applying SPS on OS indicators in the ATM system showed a perfect recall score of 100%.

It is important to remark that such results cannot be precisely compared: despite the fact we tried to replicate the experimental setup, some aspects - such as the model of anomalies - cannot be copied as they are. However, these results give us a rough estimation of how the approach in [18] is going to work in a more dynamic context. As it can be noted, *such approach is not well suited for dynamic systems*, and the scores in the last row of Table 2 highlight that changes need to be made to make this strategy suitable for a different category of systems.

Lesson Learned 1 (LL1). *Techniques that were proven useful for anomaly detection in static or semi-static systems could incur in a degradation of metric scores - an higher number of both FPs and FNs - when applied to data traces collected from more dynamic systems.*

3.5 ADAPTING THE UNDERLYING MONITORING STRATEGY

Since different faults in the system may cause errors which manifest as many different kinds of anomalies, a partial explanation of the poor results above could be that in complex systems observing only such a specific subset of indicators related to the same layer does not provide a comprehensive view of the target system. In fact, some of the surveyed frameworks such as [101], [92] analyze data coming from multiple system layers, namely OS and middleware, which in our case is represented by the *Java Virtual Machine*, since *Secure!* is written in *Java*.

3.5.1 *A Multi-Layer Monitoring Solution*

The instrumentation of the Java middleware requires implementing AS probes, or *Java* probes. Data about the middleware can be collected relying on the *Java Management Extensions* (JMX): JMX gets the values of the observed indicators by means of some Java objects called *Managed Beans* (MBeans). For instance, MBeans allows the runtime extraction of information on execution time, number of activations, risen exceptions, application errors, amount of data exchanged between services. Each MBean maps onto a Java indicator to be extracted, creating a significant set available to external modules. More in detail, we selected the following MBeans and Java indicators:

- **Java Threadings** (2 indicators). Provides information about Java thread management. These AS indicators report the amount of current running threads and the amount of all threads started since the Java Virtual Machine (JVM) started, including daemons, non-daemons and ended threads.
- **Managers** (4 indicators). Provides information about HTTP sessions: rate of session creation and expiration during the execution of a certain workload, or the sessions that have been rejected because the maximum limit has been reached.
- **Memory Pools** (6 indicators, 2 each). Provides information about memory allocation. The JVM sees different kinds of memory for different usages. We consider *Code Cache*, *Eden Space* and *Perm Gen* memory space as the most significant for our monitor. Code cache is non-heap memory used to compile and store native code. Eden space is heap memory used to allocate the major part of Java objects. Perm Gen (Permanent Generation) is non-heap memory used to manage class and method reflection.
- **Memories** (3 indicators). Provides information about heap/non heap memory management performed by the JVM.
- **Requests** (5 indicators). Provides information about management of HTTP requests.
- **Operating Systems** (5 indicators). In spite of its name, those are still middleware probes, because they provide information about the OS tools required by the JVM at such layer.

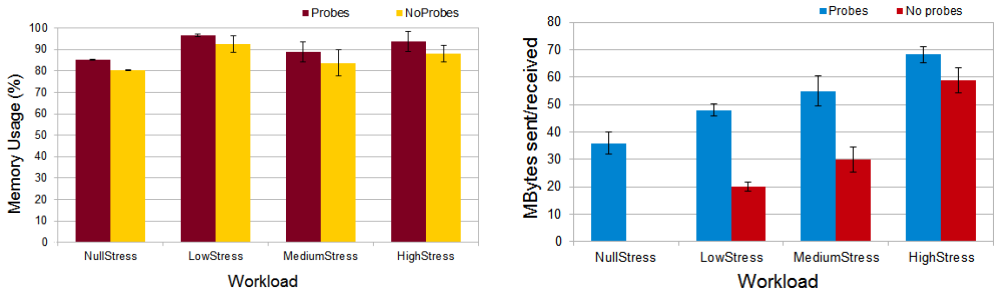
Overall, this update adds 25 indicators which are continuously monitored togheter with the 10 OS indicators read using *SystemTap*. This novel setup makes 35 indicators available for anomaly detection.

3.5.2 *Intrusiveness of Probes*

Since the probing system has changed, the analyses in [18] may not be valid anymore. To such extent, here we estimate the intrusiveness [15] of the probes on a Target Machine, both in terms of system load and network utilization. To reach this purpose we performed experiments using *Secure!* to observe the status of the Target Machine with and without probes. This requires developing the following measuring instruments.

- The *Target Machine* is equipped with a module that every second gets some information about system parameters using the *top* UNIX function; system parameters that we collect are CPU usage, RAM / swap usage, buffer size, system usage per task.
- The *Detector Machine* is used as in the other experiments without running the anomaly detector module. *Secure!* services are invoked and probes data is collected. On the Detector Machine, the packet sniffer *Wireshark* executes, collecting packet traffic data coming from the Target Machine. Note that the introduction of *top* and *Wireshark* modules does not affect the quality of the analysis: i) the packet sniffer *Wireshark* is not located on the Target Machine, and ii) the *top* command is launched in every test, including those without activating probes, ultimately *introducing a systematic error* which is considered equal in all experiments.
- To generate different workloads, we observed that the *AllServices* workload (see Table 15) is a cyclic sequence of invocations of *Secure!* SOAP services. This allowed us to create four different workloads *NullStress*, *LowStress*, *MediumStress*, *HighStress* that differ in the delays between subsequent invocations of the services. Noticeably, the workload *NullStress* simulate the execution of the Target Machine without calling any of the *Secure!* services during the time it is exercised. Each of the other three workloads generates different amount of loads on the Machine, ranging from a very low load (1 invocation every 3.2 seconds in *LowStress*), to higher loads (1 invocation per second in average in *HighStress*). The duration of the four workloads is approximately between 60 and 80 seconds. Note that the values above are comprehensive of the waiting time for the completion of the execution of the call, which is a rather long time for some services as these devoted to file transmission. Lastly, we remark that the workload *AllServices* that we used for the experimental campaign corresponds to the *LowStress* workload.

The following experiments are planned and executed: for each of the 4 workloads above, 5 runs with probes active and 5 runs with deactivated probes are executed for a total of 40 runs. Results related to the memory utilization



(a) Usage of RAM Memory

(b) Data sent through the Network

Figure 5: Experimental evaluation of intrusiveness of probes

are visible in Figure 5a, which shows the average results for each workload including standard deviation, matching the execution in presence and absence of active probes.

Data related to system utilization allows understanding how the probe processes affect the memory utilization (about 5% RAM on the Machine, that corresponds to about 150 MB, see the difference between the red and yellow bars in Figure 5a), while no significant alteration are identified for the CPU consumption and consequently not shown for brevity. The number of active tasks differs by 15: these are the tasks required to execute the *JVM* and *OS* for the available set of indicators.

Relevant differences for the execution with and without active probes can be observed in Figure 5b regarding the network usage, which is affected by data generated by the probes. For example, for the *NullStress* workload, the amount of bytes exchanged without probes is significantly less than those exchanged with active probes. The other three workloads produce network traffic, like remote calls or file transfer, that pile up on the traffic generated by the probes as it is evident from the figures. The additional invocations do not alter the amount of data transmitted by the probes; this is as expected, although it cannot be verified graphically in the figures mostly due to data aggregation at lower stack layers.

3.5.3 *Analysis of the Results*

Differently from the previous step, we instantiated a multi-layer anomaly detection strategy on our prototype of the *Secure!* [4] SOA. Briefly, observing indicators related to different layers will provide a more complete and comprehensive view of the system, and will provide us more data related to different modules to analyze. It is not straightforward that this will improve our detection scores, since more available data means more information to extract, and we may not be able to extract this information correctly. However, the works in

#	Layers	Indicators	P	R	F2	FPR
[18]	OS	10	97.0	100.0	99.3	1.9
[18] in SOA	OS	10	26.3	41.1	36.9	13.7
[21]	OS, Java	35	25.6	50.4	42.2	11.1
Extensive Multi-Layer	OS, Java, Network	42	31.4	60.3	50.9	9.3

Table 3: Adoption of a Multi-Layer Monitoring Strategy

the literature [101], [92] suggest adopting such strategy, and therefore we will implement and evaluate if the adoption of multi-layer monitors will help us improve metric scores. According to such studies, as an additional step, we instrumented also the */proc* filesystem of *Linux*, retrieving 7 network indicators.

The experimental results achieved (see Table 3) showed that metric scores were overall better, still highlighting very low precision scores, capping at 31.4. The bigger pool of indicators with respect to the previous setup reduced the number of missed detections i.e., the FNs are lower, and consequently the recall is higher. Indeed, some indicators have very discontinuous trends, meaning that they still raise a big amount of false positives, as confirmed by the precision and false positive rate scores.

We explain these outcomes as follows. SPS detects changes in a stream of observations identifying variations with respect to a predicted trend: when an observation does not comply with the predicted trend, an alert is raised. If the system has high dynamicity due to frequent changes or updates of the system components, or due to variations of user behavior or workload, such trend may be difficult to identify and thus predict. Consequently, our ability in identifying anomalies is affected because boundaries between normal and anomalous behavior cannot be defined properly. Further discussions are expanded in [22].

Lesson Learned 2 (LL2). *Montoring only the underlying layers of a system without knowing anything about the services does not help in tracing the boundaries between normal and anomalous behavior.*

3.6 CONTEXT-AWARENESS

We previously highlighted the need of acquiring more information on the target system, still maintaining the main benefits of the abovementioned approach. Consequently, we investigate which information on SOA services we can obtain in absence of details on the services internals and without requiring user context (i.e., user profile, user location). In SOAs, the different services share common

information through an *Enterprise Service Bus* (ESB, [51]) that is in charge of i) integrating and standardizing common functionalities, and ii) collecting data about the services. This means that static (e.g., services description available in *Service Level Agreements* - SLAs) or runtime (e.g., the time instant a service is requested or replies, or the expected resources usage) information can be retrieved using knowledge given by ESB. Consequently, having access to the ESB provides knowledge on the set of generic services running at any time t. We refer to this information as context-awareness of the considered SOA; note that we do not require information on the user context, contrary to what is typically done in the state-of-the-art on context-awareness [95].

Noticeably, we can exploit this **contextual information** to define more precisely the boundaries between normal and anomalous behavior of the SOA. For example, let us consider a user that invokes a *store file* service at time t. We can combine contextual information with information on the current behaviour of the service, which here is about data transfer. Therefore, if the *store file* service is invoked at time t, we expect an exchange of data during almost the entire execution of the service. If we observe no data exchange, we can reveal that something anomalous is happening.

3.6.1 *Collect Services Information*

Let us start from the example of the store file service. Our target is to characterize the normal behavior of the service, building a **fingerprint** of its usage. More in details, we need a description of the expected behavior of the service, meaning that we need to describe the usual trend of the observed indicators while the service is invoked. In such a way, we can understand if the current observation complies or not with the expectations. This information can be retrieved in a SOA by observing the ESB and producing a new service fingerprint when the addition, update or removal of a service is detected. In several cases it is also possible to obtain a static characterization of the services looking at their SLA, where each service is defined from its owner or developer for the final user. We remark that we do not consider any assumption about the services except their connection with the ESB: consequently, we can obtain services information from any kind of service running in the SOA platform.

3.6.2 *Integrate Information in the Anomaly Detector*

Summarizing, information about the services can be obtained i) statically, looking at SLAs, ii) at runtime, invoking services for testing purposes or iii) combining both approaches: in this Thesis we explore the second approach. This *contextual information* needs to be aggregated and maintained (e.g., in a database) together

with the calculated statistical indexes (e.g., mean, median), whenever applicable, to support the anomaly detection solutions.

3.6.3 *The Resulting Architecture*

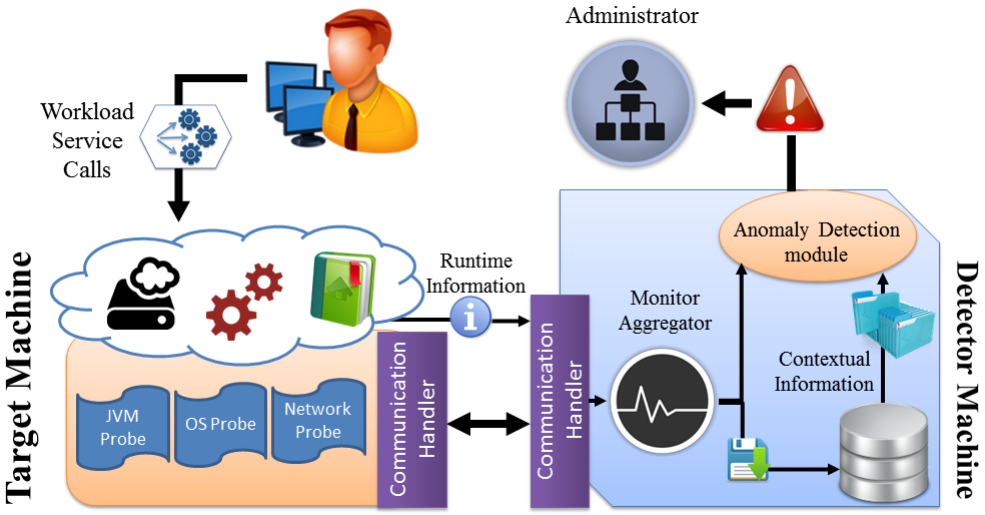


Figure 6: The resulting architecture of the Anomaly Detector

In Figure 6 we depict a high level view of the resulting framework which now includes contextual information. Starting from the upper left part of the figure, the framework can be described as follows. The user executes a workload, such as the *AllServices* workload in Section 3.4.1. In this machine probes are running, observing the indicators coming from 3 different system layers: i) OS, ii) middleware and iii) network. These probes collect data, at defined time instants. The probes forward the data to the communication handler. Indicators data related to the same time instant is aggregated in a **snapshot**, and sent to the communication handler of the Detector Machine.

Data is analyzed on a separate machine, the Detector Machine (which includes a Complex Event Processor - CEP). This allows i) not being intrusive on the Target Machine, and ii) connecting more Target Machines to the same Detector Machine (obviously the number of Target Machines is limited by the computational resources of the Detector Machine). The communication handler of the Detector Machine collects and sends these data to the *monitor aggregator*, which merges them with runtime information (e.g., list of service calls) obtained from the ESB. This allows storing contextual information in the database. Looking at runtime information, the monitor aggregator can detect changes in the SOA and notify the administrator that up-to-date services information is needed to appropriately tune the anomaly detector. The tuning includes the choice of the 10 indicators

out of the total 55 which are more performing in identified anomalies with the current setup. Some training runs are executed with injection of anomalies, and the indicators which gave better *F2* scores after applying SPS were selected as the 10 indicators to monitor for validation. When applying SPS, if at least 5 out of the 10 selected indicators raise an anomaly simoultaneously, an anomaly alert for the system is delivered to the administrator.

The snapshots collected when SOA is opened to users are sent to the *anomaly detection module*, which can query the database for services information and analyzes each observed snapshot to detect anomalies. If an anomaly is detected, the system administrator, which takes countermeasures and applies reaction strategies (which are outside from the scope of this work and will not be elaborated further), is notified.

3.6.4 *Analysis of the Results*

#	Layers	Indicators	Context	P	R	F2	FPR
[18]	OS	10	NO	97.0	100.0	99.3	1.9
[18] in SOA	OS	10	NO	26.3	41.1	36.9	13.7
[22]	OS, Java	35	NO	25.6	50.4	42.2	11.1
Extensive Multi-Layer	OS, Java, Network	42	NO	31.4	60.3	50.9	9.3
[108]	OS, Java, Network	42	YES	41.8	91.3	72.7	4.9

Table 4: Results with Context-Awareness and Selection of Indicators

At the present stage we use contextual information to characterize what the system is intended to do in a specific context, i.e., while providing a specific service. In other words, context-awareness allows defining more precise expectations about the behavior of the target system in a defined time window, or rather *its normal behavior*. This has key advantages for anomaly detection, since the boundaries between normal and anomalous data instances are traced more carefully. Results in Table 4 certify that having a better definition of the normal behavior positively impacts on our ability of detecting anomalies, with higher metric scores in the table. Noteworthy, the FPR dropped to approximately 5, while recall is over 90%. Altrough not perfect, we can state that assuming knowledge of the services that are running at time t on the observed machine gave us the opportunity to consider additional contextual information that resulted fundamental to improve our anomaly detection capabilities.

Indeed, there is still room for improving. More in detail, relations among indicators cannot be catched with the current setup: a simoultaneous light increase or decrease of more than one indicators' value may be related to the manifestation of an error, but it cannot be detected at the current stage. Also, depending on the anomaly, an optimal setup of the parameters needs to be found to maximize metric scores. Choosing the best 10 indicator to apply SPS may not be the better choice in some cases, or may be the worst choice at all.

Lesson Learned 3 (LL3). *Introducting Context Awareness helped defining the characteristics of the anomalies and tuning our anomaly detection stategies.*

3.7 REFINING THE FRAMEWORK

After introducing i) a multi-layer monitor, and ii) context-awareness, our anomaly detection framework needed to include a more robust training phase, directed to identify the best parameters setup e.g., defining which and how many indicators are useful for detecting a specific anomaly on a given target system. In the previous section we extracted the best 10 indicators that gave higher F2 scores in the training runs, raising an anomaly if and only if at least 5 out of 10 raised an anomaly simoultaneously. However, this basic setup may not be the optimal one, and further combinations need to be exploited. Moreover, the data analyzed by the SPS algorithm is related to single indicators separately, preventing us to correlate different potentially anomalous trends in related indicators. To such extent, we refined the framework by formalizing *data series* and *anomaly checkers*, and specifying as main parameters to train the *selected anomaly checkers* and the *anomaly treshold*.

3.7.1 *Data Series*

We define a data series as a triple

$$\texttt{indicator}, \texttt{data\_category}, \texttt{series\_layer}$$

Indicator represents the indicator responsible for the set of data we are analyzing. For example, this can be the usage of the memory, the number of accesses to the hard disk or the number of active threads managed by the OS. *Data_category* specifies if the data series refers either to PLAIN e.g., 234 threads are currently active, or to the linear difference DIFF among subsequent elements e.g., 2 threads were created since the previous observation. Lastly, *series_layer* describes the system layer from which the data are collected. By default, this is the system layer that offers the indicator values, and the *series_layer* value is the name of the layer. Noteworthy, a data series may also result from the

linear combination of two existing data series e.g., *cache_hit_rate = cache_hits / cache_misses*, originating a **composed data series**. For the sake of simplicity, in our study composed data series are obtained only through linear relations that use one single matemathical operator (+, -, *, /).

3.7.2 *Anomaly Checkers*

The detection of anomalies is demanded to a set of **anomaly checkers**, selected according to a given metric. An anomaly checker is assigned to a given data series, and evaluates if the current value of the selected data series is anomalous or expected following a set of given rules or algorithms. More anomaly checkers can be created for the same data series. Given a snapshot as input, each anomaly checker produces an anomaly score; the individual outcomes of the set of anomaly checkers is then combined to decide if an anomaly is suspected for the current snapshot. Consequently, an anomaly is raised only if this combined score meets or exceeds a given treshold. For each data series, we build two anomaly checkers:

- *Historical* (HIST): this anomaly checker implements a contextual check by comparing the values of a given data series with the expectations defined in the fingerprint. If this quantity is outside of the interval defined by *average* $\pm$ *standard deviation* in the fingerprint, an anomaly is raised.
- *SPS*: for a given data series, this anomaly checker applies the SPS algorithm described in [14], [17]. Such algorithm was preferred among others due to i) the knowledge we had of the implementation our research group has, and ii) promising past results as reported in [18], especially regarding *precision*, *recall* and *false positive rate* metrics).

3.7.3 *Selected Anomaly Checkers and Anomaly Threshold*

Once the metric i.e., *FScore(2)*, is defined, it is used to automatically detect the best configuration of each anomaly checker. Moreover, we detect anomalies depending on a set of anomaly checkers that are selected from the pool of available ones according to specific rules. The **selected anomaly checkers** are extracted by choosing:

- **BEST x**: the x anomaly checkers that have the best scores according to the metric e.g., *BEST 5* (B_5) represents the 5 anomaly checkers which show higher individual *FScore(2)* on the training set;
- **FILTERED y**: the y anomaly checkers that have the best metric scores, filtered to avoid having two anomaly checkers exercised on the same data

series. For example, it avoids selecting SPS and HIST anomaly checkers on the same *HeapMemoryUsage* PLAIN data series.

Once the selected anomaly checkers are defined, the appropriate **anomaly treshold** is selected as follows. A snapshot is evaluated as anomalous if at least a given number of selected anomaly checkers raise an anomaly reaching or exceeding the anomaly threshold. We evaluate different approaches to set this threshold, namely:

- **ALL**: all the selected anomaly checkers must evaluate their data series instance of the snapshot as anomalous;
- **QUARTER / THIRD / HALF**: a quarter / third / half of the selected anomaly checkers must evaluate their data series instance related to the snapshot as anomalous;
- **ONE**: the snapshot is evaluated as anomalous if at least one of the anomaly checkers raises an anomaly.

It is important to remark that the choice of the anomaly threshold heavily impacts on the overall detection performances. The usage of a single anomaly checker (i.e., ONE threshold) reduces the amount of false negatives; instead a consensus among several anomaly checkers, e.g., ALL threshold, reduces the number of (false) alarms raised, but may negatively affect false negatives.

3.7.4 *Setup of the Probes and Data Series*

Differently from the previous version of the probes, we looked at the data provided by *SystemTap* in our current implementation, which we inherited from [18]. 10 OS indicators were retrieved, but specific areas of the OS were not covered e.g., cache accesses, size of buffers. Then, we looked at the UNIX */proc* filesystem, which makes available a huge set of indicators related to the network - which we already instrumented - but also on the hysical and virtual operating system. Therefore we replaced the OS probe with a module which reads data from */proc* as well as it was done with the *network* probe. Intrusiveness is not negatively affected, since the *SystemTap* module is no longer running in the system, and calls to */proc* have minimum delays.

The updated probing system running the target machine is composed of three probes: OS and Network probes are shell modules reading data from the */proc* virtual filesystem, while the JVM probe consists in a Java module accessing performance data through *Java Management Beans* (MBeans). These three probes monitor a total of 55 different indicators: 23 from the OS, 25 from the *Java*, and 7 related to the network. These probes are coordinated by a Java-based communication handler that manages the collection of data, encapsulates them

in JSON format and sends the data through a TCP socket. The version of Java required on the target machine(s) to be instrumented with our probing system is 7 or higher. Here we remark that despite several enterprise solutions providing monitoring facilities exist, we chose to develop our own probes to limit the intrusiveness of such enterprise monitoring tools.

Regarding the observed data series, let us consider our set of 55 indicators and 49 couples of related indicators, which are combined using the +, -, *, / matemathical operations obtaining 196 novel composed data series. Note that the 49 couples of indicators were selected by choosing the couples which scored a Pearson correlating index [64] above 0.9. Considering both PLAIN and DIFF data categories for each indicator, we obtained 110 separate single data series and 392 composed data series. On each data series we can instantiate either the SPS or the HIST algorithm, totalizing 1004 possible anomaly checkers. The set of checkers to be used is reduced and then ranked during the training phase.

3.7.5 *Analysis of the Results*

#	Layers	Indicators	Context	P	R	F2	FPR
[18]	OS	10	NO	97.0	100.0	99.3	1.9
[18] in SOA	OS	10	NO	26.3	41.1	36.9	13.7
[22]	OS, Java	35	NO	25.6	50.4	42.2	11.1
Extensive Multi-Layer	OS, Java, Network	42	NO	31.4	60.3	50.9	9.3
[108]	OS, Java, Network	42	YES	41.8	91.3	72.7	4.9
Current (avg)	OS, Java, Network	55	YES	64.4	96.1	87.5	2.6

Table 5: Combining Context-Awareness and Composed Data Series

The results of the experiments in terms of precision, recall, FScore(2) and False Positive Rate are reported in Table 5. The last row reports on the average scores obtained regarding the detection of MEMORY and NETWORK anomalies. The improvement at this stage is consistent and it is mainly due to:

- the *inclusion of composed data series* as data source for creating SPS and HIST anomaly checkers. This allows shaping linear relations among different indicators, ultimately providing more knowledge of the monitored system.
- the definition of a training process directed at identifying the most performing anomaly checkers (*selected anomaly checkers*) togheter with the

best voting strategy to use with them. This strongly contributed to reduce the number of false positives raised by our framework when choosing an anomaly threshold that was too tight, labeling normal fluctuations of the trends of the indicators as anomalies. Relaxing this treshold lowers the number of false positives, but consequently exposes false negatives. The best tradeoff among these two is the chosen *anomaly treshold*.

With respect to the first application of SPS to the data of *Secure!*, the enhancements are surprising. The number of false negatives dropped consistently i.e., recall is higher than 95% and the false positive rate went down to 2.6 being roughly 14 at the beginning. It is important to remark that altrough higher than at the beginning, these scores are still significantly lower than the ones obtained in the study we took as starting point [18]. In fact, it is easy to observe that in several cases a false positive rate of 2.6 may not be accepted, expecially when an alarm triggers complex and expensive recovery strategies which in case of FP will result in a waste of time, money and resources. Nevertheless, depending on the system, this can be either a strong limitation for the application of our technique or a reasonable price to pay considering the difficulties of performing anomaly detection in such complex dynamic systems.

3.8 COMPARISON WITH RESPECT TO SURVEYED STUDIES

At this stage of the process of enhancing the capabilities of the anomaly detector, we compared our partial results with the anomaly detection scores from the surveyed studies (see Table 6). We consider results of the studies we already used in Table 1, and we also include our partial result. We show only results for anomalies related to the resource usage, because they are the only anomalies (MEMORY, NETWORK), we injected during our experimental process.

Overall detection performances (we show *Precision*, *Recall*, *F-Score(2)* and *False Positive Rate* where available) are strongly influenced by the characteristics of the target system. In fact, when the dynamicity of the system is low, it is easier to define the expected behaviour. This results in a significantly lower number of false positives and false negatives as in [18], [11] and [101] with respect to [9] (see Table 6). High precision scores are obtained also in [92], where ALERT is exercised in a cluster environment. The system shows good dynamicity, because hosts are added or removed; however, the cluster runs a fixed pool of tasks during its operational life. Moreover, SEAD [73] obtains high recall scores by analysing data gathered from an hypervisor, but no quantitative information to compute precision or FScore are reported.

Finally, our partial results present recall and false positive rate values that are competitive with the other solutions, especially considering that we are exercising the anomaly detector on a highly dynamic system. Our precision is low, meaning that many false positives are generated. We already mentioned that

Framework	Metric Scores			
	Precision	**Recall**	**FScore(2)**	**FPR**
CASPER [11]	88.5	76.5	78.6	11.26
[9]	76.0	99.0	93.3	n.a.
SEAD [73]	n.a.	92.1	n.a.	n.a.
TIRESIAS [101]	97.5	n.a.	n.a.	2.5
[18] (best setup)	97.0	100.0	99.3	1.9
ALERT [92]	~100.0	> 90.0	> 90.0	< 10.0
Our Results	64.4	96.1	87.5	2.6

Table 6: Comparing metric scores with the surveyed frameworks. Scores have been extracted from the original papers.

in our setup we favour recall because our aim is to minimize missed detections, even at the cost of a higher number of false positives.

3.9 SUMMARIZING: LESSONS LEARNED AND OPEN CHALLENGES

As highlighted in the beginning of this chapter, detecting anomalies in complex and dynamic systems is not trivial. There are intrinsic challenges due to the nature of such systems that need to be tackled in order to make the classic anomaly detection techniques suitable. To such extent, the preliminary analyses presented in the previous sections provided useful lessons and directions in identifying the main challenges that need to be tackled in order to tailor anomaly detection on such systems. Such lessons learned are combined togheter with some conclusions found in the literature [25], resulting in the following challenges.

CH 1 **Adaptive notion of expected behavior**. The intrinsic dynamicity of the target system leads to frequent changes in the expected - and consequently anomalous - behavior. This means that the model of the expected behavior needs to be repeatedly updated [25], because its validity is going to be deprecated through time (see *LL1* in Section 3.4 and *LL3* in Section 3.6). In [73] and [92], the authors propose self-adaptive anomaly detection strategies to deal with such evolving notion of expected behavior.

CH 2 **Avoiding Interferences and Minimizing Overhead**. The anomaly detection logic must not interfere with the target system. More in detail, the anomaly detection framework must not steal computational resources or introduce relevant overheads e.g., during training of the anomaly detection

algorithms. Intrusiveness of anomaly detection frameworks are evaluated in [11], [18], mainly analyzing CPU and RAM usage.

CH 3 **Applicable monitoring strategy**. Data is collected from different sources that compose the target system (see *LL2* in Section 3.5. However, insights of the services or components may not be observable, for example in case of third-party components or encapsulated components [58]. Moreover, the set of services may change; for example, services may be updated, added or removed. This calls for an applicable monitoring strategy that identifies viable monitoring targets and does not require manual reconfiguration when the services evolve.

CH 4 **Suitable anomaly detection algorithms**. The anomaly detection logic needs to rapidly cope to frequent changes of the expected behavior (*LL1*). Algorithms that require a massive training effort - such as clustering [73] or Markov-based models [11] - are not adequate when the system changes frequently.

CH 5 **Selection of the indicators**. To reduce their impact on the system and/or the network, monitors should observe only the minimum set of features (indicators), which are required by the anomaly detector to run its logic. For example, indicators values obtained observing the network layer are generally suitable for intrusion detectors [46], [76], while the operating system is usually monitored when detecting performance issues or malware activities [99]. Other studies on the selection of the indicators are in [9] regarding the filtering of invariants, and in [73] where authors describe how they select 14 indicators out of 653 from the *Xen* hypervisor.

4

EXPERIMENTAL EVALUATION

In this section, we present an extensive experimental evaluation of the resulting anomaly detection framework. In the previous section we described the process which led us to build an anomaly detector for complex dynamic systems (i.e., SOAs) with metric scores comparable with the works in the literature. However, the experiments we executed in such steps were not extensive, and a more complete experimental campaign is needed to evaluate all the components of such framework. More in detail, experiments will include a wider model of anomalies, an additional case study, and an higher number of experimental runs executed.

To the purpose of the evaluation, we run an automatic controller that checks input data and manages the communications among the different modules of a Target Machine and the Detector Machine. This facilitates the automatic execution of the experimental campaign without requiring user intervention except for the setup. All data are available at [1].

4.1 SOA CASE STUDIES

Here we describe the case studies we considered to evaluate our framework for anomaly detection in complex dynamic systems. It is worth mentioning that both case studies are based on a SOA structure, which nowadays is one the most common instantiation of complex dynamic system. The first is *Secure!*, which we already presented and used in the previous section, while the second one is the *jSeduite* SOA [30]. *jSeduite* is an open source SOA dealing with information broadcast inside academic institutions, and is composed of atomic web services representing information sources and *BPEL* orchestrations expressing business processes. Despite the fact this system is free from safety implications in case of malfunctions, some of the available services cover basic operations such as file uploads, error logs and data gathering that represents the pillars on which safety and/or security critical services are built upon. We distributed jSeduite on a *Glassfish Java*-based server, setting its services to use a *MySql* database.

We identified a subset of *jSeduite* services that are both characterizing this system and matching some of the services that are implemented in the *Secure!* CMS. A description of the whole set of services is available at [24]. In our setup, we also considered to use a subset of the services that were already used in this previous study, to help the discussion and some comparisons among the obtained results. This resulted in the definition of a workload composed by 8

different service calls, namely *ErrorLogger*, *DataCache*, *FileUploader*, *ApalWrapper*, *TvHelper*, *FeedRegistry*, *TwitterWrapper*, and *InternalNews*. The services are called multiple times without following a specific order.

4.2 MODEL OF ANOMALIES

We identify the model of anomalies that we aim to detect. To such extent, we review here well-known anomalies model from the literature identifying which of them are in scope for our work. Most of the works were already described in Table 1 regarding their characteristics and suitability for anomaly detection in complex dynamic systems.

Framework	**Categories of Anomalies**			
	Reconfiguration	*Misconfiguration*	*Interaction*	*Resource Usage*
ALERT [92]				✓
CASPER [11]				✓
[9]	✓	✓		✓
SEAD [73]				✓
TIRESIAS [101]				✓
[41]				✓
[67]	✓		✓	
Our Framework		✓	✓	✓

Table 7: Model of Anomalies targeted by the surveyed studies and by our framework. For each framework, we reported the categories of anomalies covered by their model of anomalies.

Most of these frameworks consider anomalies in the resource usage, while in [9] authors also consider anomalies related to reconfiguration and erroneous configuration (misconfiguration) of parameters of the targeted system or applications. Concerning reconfiguration, we assume to re-train the framework every time a reconfiguration is detected, making the detection of such anomalies out of scope. Instead, we do include anomalies concerning misconfiguration in our anomalies model. Moreover, other studies consider also anomalies due to interaction among modules and components of the complex system. In particular, [67] defines a set of behaviors that can emerge in complex systems, such as: i) deadlock/livelock, ii) trashing, iii) phase change, iv) synchronization and oscillation, and v) chaotic.

Similarly to reconfigurations, phase changes are considered main variations of our system, thus calling for a new training step and not considered in our anomalies model. Furthermore, since the Anomaly Detection module treats each Target Machine individually, synchronization, oscillation and chaotic behaviors have minor impact or likelihood. Additionally, our selected middleware (JVM) automatically controls thrashing: the JVM manages the garbage collection and context switches with the objective of avoiding performance degradation. Instead, deadlock and livelock should be considered in our model of anomalies as they are possible source of anomalies in complex systems.

Consequently, the resulting anomalies model is composed by i) performance anomalies, and in particular we identify four anomalies that we name *MEMORY*, *CPU*, *DISK*, *NET_USAGE*, ii) anomalies due to deadlock/livelock, that we label as *DEADLOCK*, and iii) anomalies due to misconfigurations; more in detail, we will focus on misconfiguration of the network permissions, labeling them as *NET_PERMISSION*.

4.3 INJECTION APPROACH

Similarly to what we expanded in Section 3.4.1 for Secure!, with *jSeduite* the injections were performed in 10 different functions related to 4 of the selected 8 web services (all belonging to the *fr.unice.i3s.modalis.jSeduite.technical* package). More in detail, we instrumented the *TvHelper* service (*tv.extract*), the *TwitterWrapper* service (*messaging.twitter.getIntendedTweets*, *messaging.twitter.getChannel*, and *messaging.twitter.getFreeTokens*), the *ApalWrapper* service (*apal.getTopWithTreshold*, *apal.getTop10*, *apal.getLoosers*, and *apal.getPromos*), and *FeedRegistry* (*registry.getURL*, *registry.getCategories*, and *registry.getNicknames*). As in Secure!, the injection is triggered by a chronometer which allows the execution of the mutated code only when at least 50% of the workload is already executed. Note that all the considered services are invoked through the whole workload.

In our experiments, we inject a single error in each individual run. Noteworthy, we are interested in *detecting the first anomaly that is generated after the activation of the error* and that can be explained by the error itself. In fact, the manifestation of a single fault can lead to several cascading effects on the system, with consequent variations from the expected behavior i.e., multiple anomalies. The complexity of our target system does not allow studying the propagation effects of the injected faults. Therefore, we are not able to distinguish if and especially how different anomalies are related. It follows that only the first anomaly that is detected after the activation of the injected fault is considered a true positive (TP) in our analysis. Further, this anomaly must be detected within a limited temporal distance from the injection time instant; this way, we are more confident that the anomaly is a consequence of the injected error and not a false positive (FP). We set this temporal distance to 1 second after some preliminary experiments.

Successive anomalies, which can be due to i) brand new manifestations of errors, ii) cascading effects of the injected anomalies, and iii) false positives, are ignored. We are aware that with this approach we are ignoring several anomalies that are most likely true positives. Consequently, another strategy is to include all the anomalies the framework raises after the activation of the fault, until the termination of the workload. As explained before, it is difficult to understand if such anomalies are related to the injected fault (TP) or are FP. Consequently, we disregard this approach despite it would probably improve our recall and precision.

4.4 EXPERIMENTAL CAMPAIGN

We present here how we organized our experimental campaign. We first conducted 100 golden runs in which we executed the chosen workload without any injection. We then performed runs with injections for the 6 anomalies that are present in our model of anomalies. For each anomaly, we executed 10 runs for each injection (9 in *Secure!*, 10 in *jSeduite*) as discussed before.

Overall, respectively 640 and 700 experiments for each case study were conducted. This number was defined observing the standard deviation for the results, which was acceptable i.e., smaller than the mean. To guarantee the correctness of the experiments, we restarted our target application - either *Secure!* or *jSeduite* - before the execution of each experiment. This allows completely resetting the status of the system running on the Target Machine before executing each experiment, providing independency among subsequent experiments. These experiments served both as preliminary runs for the training phase and to validate the efficacy of the our framework itself. In particular, all the 100 golden runs and 70% of the other runs were used for the training phase. The remaining runs were instead used for the validation of the framework.

4.5 RESULTS: SECURE!

4.5.1 *Detection Efficiency*

The results of the experiments regarding *Secure!* in terms of precision, recall, and FScore(2) are depicted in Figure 7. Each set of bars reports on the results with a given injection: as an example, the first set is related to injections of a MEMORY anomaly. Moreover, each set of bars reports results related to two possible choices of selected anomaly checkers and anomaly threshold, namely i) BEST 3 (ALL), labeled as B_3 and represented with solid fill in Figure 7, and FILTERED 10 (HALF), which is labeled as F_{10} with a striped pattern. In the figure, precision, recall and FScore(2) of B_3 strategies are reported using bars with solid fill, while the ones regarding F_{10} are filled with a vertical-striped

pattern. Overall, we can observe how recall scores, or rather *B3_Recall* and *F10_Recall* in the figure, are significantly higher than their precision counterparts. This is due to i) the intrinsic characteristics of the system, and ii) the setup we adopted during training. In fact, targeting FScore(2) as reference metric for the whole process favors anomaly checkers that give higher recall scores. Regarding MEMORY and DEADLOCK experiments in Figure 7, both scoring strategies were able to detect all the anomalies we injected, paying a price in terms of a relatively high number of false alarms. In fact, it is easy to observe that in several cases, precision is lower than 50%. Depending on the system, this can be either a strong limitation for the application of our technique or a reasonable price to pay considering the challenges identified for such complex dynamic systems.

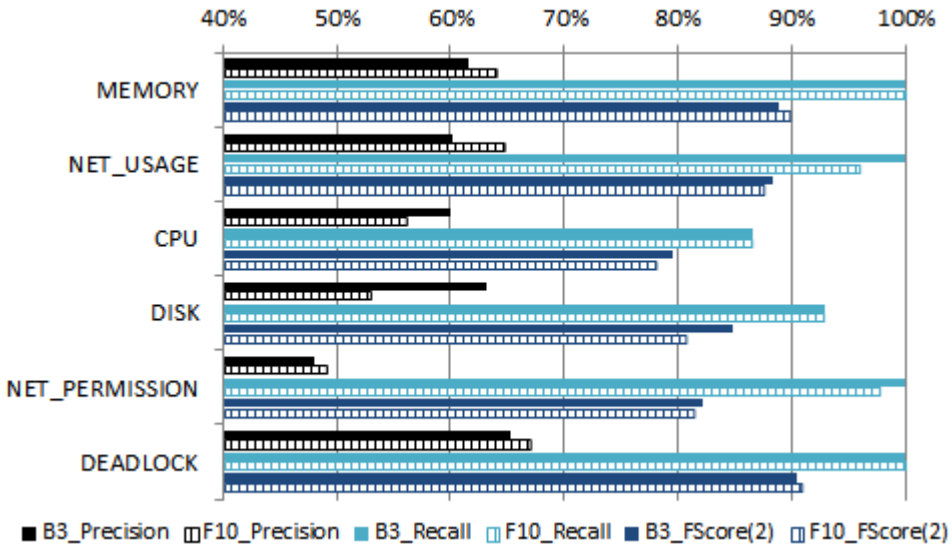


Figure 7: Precision, Recall and FScore(2) for experiments obtained using Secure!

4.5.2 *Choice of Anomaly Checkers*

Table 8 and Table 9 respectively report on: i) the data series used by the *selected anomaly checkers*, and ii) the 10 *selected anomaly checkers* that show the best FScore(2) when NET_USAGE is injected. The detailed list of the anomaly checkers for each category of anomalies is not reported here for brevity and can be found in [1]. In Table 8, we can observe how the data series related to the *selected anomaly checkers* are balanced among layers, meaning that all the three instrumented layers are fundamental in providing actionable information for anomaly detection. In Table 9 we notice that the checkers AC1 and AC2 use composed data series that are respectively built on data series DS1, DS2 and DS3, DS4. In Table 8, we can notice that 3 out of these 4 data series belong to

data from the network layer. Consequently, it can be observed that the composite data series that are used by the two best anomaly checkers are from the network layer.

#	Indicator	Data Category	Series Layer
DS1	Tcp_Close	PLAIN	NETWORK
DS2	Tcp_Established	PLAIN	NETWORK
DS3	Tcp_TimeWait	DIFF	NETWORK
DS4	SystemCpuLoad	DIFF	JVM
DS5	CPU User Processes	PLAIN	CENTOS
DS6	CurrentThreadUserTime	DIFF	JVM
DS7	HeapMemoryUsage.committed	DIFF	JVM
DS8	CPU Kernel Processes	PLAIN	CENTOS
DS9	Tcp_Syn	PLAIN	NETWORK
DS10	Buffers	PLAIN	CENTOS
DS11	Net_Received	PLAIN	NETWORK
DS12	CurrentThreadCpuTime	PLAIN	JVM
DS13	FreePhysicalMemorySize	PLAIN	JVM
DS14	Free Virtual Pages	PLAIN	CENTOS

Table 8: Simple Data Series (DSx) used by the *selected anomaly checkers*, in descending order

Although not shown in the tables for brevity, we further observed that, for the CPU anomalies, an anomaly checker executing SPS on the composed data series that is built upon $DS_6 + DS_{12}$ enters the top 5 of best checkers. This complies with our expectations since the activation of the CPU anomaly causes the execution of additional instructions for one second in the usual flow of service calls, ultimately altering CPU-related data series. Another interesting finding regards the experiments with the injection of *DEADLOCK* anomalies. Here the anomaly causes a new thread temporarily competing for an object that is requested by the main thread to be instantiated, resulting in the known concurrency issue. In this case AC5 becomes one of the best 3 checkers. More in detail, the instantiation of a new *Java* object (the thread) and the busy wait loop trying to acquire the token for the critical section cause the increase of heap memory usage.

Summarizing, the list of anomaly checkers in Table 9 is not intended to be a selection of data series and anomaly checkers valid for any complex system.

#	Data Series			Algorithm
	Indicator	Data Category	Series Layer	
AC1	DS1 / DS2	PLAIN	CROSS	HIST
AC2	DS3 / DS4	PLAIN	CROSS	HIST
AC3	CPU User Processes	PLAIN	CENTOS	SPS
AC4	CurrentThreadUserTime	DIFF	JVM	SPS
AC5	HeapMemoryUsage.committed	DIFF	JVM	SPS
AC6	CPU Kernel Processes	PLAIN	CENTOS	SPS
AC7	DS9 + DS10	PLAIN	CROSS	SPS
AC8	Net_Received	PLAIN	NETWORK	HIST
AC9	DS12 + DS13	DIFF	CROSS	SPS
AC10	Free Virtual Pages	PLAIN	CENTOS	SPS

Table 9: *Selected Anomaly Checkers* for runs with the injection of NET_USAGE anomaly, in descending order

Instead, this is a list of both the data series and the anomaly checkers that performed better in our experiments and consequently in the *Secure!* system. The list may vary depending on the characteristics of the target system. A guide for the selection of the appropriate data series when using our framework is further expanded in Section 4.7.2.

4.5.3 *Sensitivity Analysis*

We apply all the possible combinations among the selected anomaly checkers to identify the setup that ultimately gives a higher FScore(2) score. In Figure 8 we report the graphical result of the sensitivity analysis on the runs we used as validation of our framework to detect MEMORY anomalies. On the horizontal axis, we can observe the different strategies for the choice of the selected anomaly checkers. On the depth axis, instead, we report the anomaly thresholds we consider in our setup. For example, a snapshot can be labeled as anomalous if at least half (HALF in the figure) of the anomaly checkers raise an alert simultaneously.

Regarding the detection of the anomaly MEMORY, an optimal setup is represented by F_{10} (HALF), as can be observed in Figure 8. F_{10} (HALF) labels a snapshot as anomalous if at least five anomaly checkers trigger an anomaly for that snapshot. Other optimal setups can be obtained by considering the B_3, that raises an anomaly either if i) at least one out of three checkers detects an

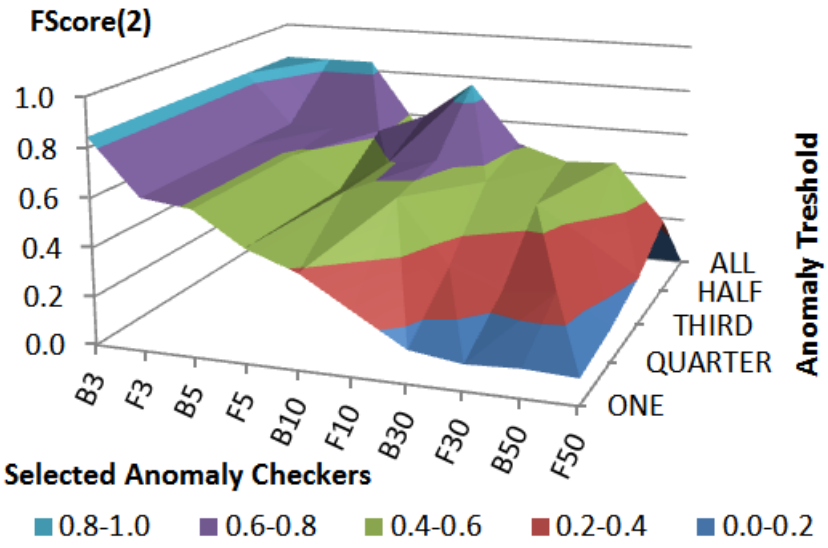


Figure 8: Sensitivity analysis of MEMORY experiments in Secure!

anomaly (ONE, QUARTER, THIRD on the z-axis of Figure 8), or ii) all three anomaly checkers raise anomalies (ALL in the z-axis).

Results of the sensitivity analyses for the other injected anomalies are not reported here for brevity, but lead to similar observations and can be found in [1].

4.6 RESULTS: JSEDUITE

4.6.1 *Detection Efficiency*

The results of the experiments regarding *jSeduite* in terms of precision, recall, and FScore(2) are depicted in Figure 9. As in Figure 7, each set of three bars reports on the results with the injection of a given anomaly. The scores we reported for each anomaly were obtained using the best possible setup of parameters (i.e., *selected anomaly checkers* and *anomaly treshold*) for each specific anomaly. Noticeably, the optimal setups vary when injecting different anomalies. This is not surprising, although it can be noted that for all the 6 anomalies the optimal *selected anomaly checkers* are always composed by at most 10 anomaly checkers, confirming the trend we already observed in the experiments regarding *Secure!*. Further discussions will be expanded in Section 4.7.

Overall, it turns out that recall scores are significantly higher than their precision counterparts. The number of false negatives is low, leading *recall* scores to be always higher than 65%. More in detail, regarding *DISK* and *NET_PERM* experiments in Figure 9, we were able to detect all the anomalies, paying a price in terms of a higher amount of false alarms.

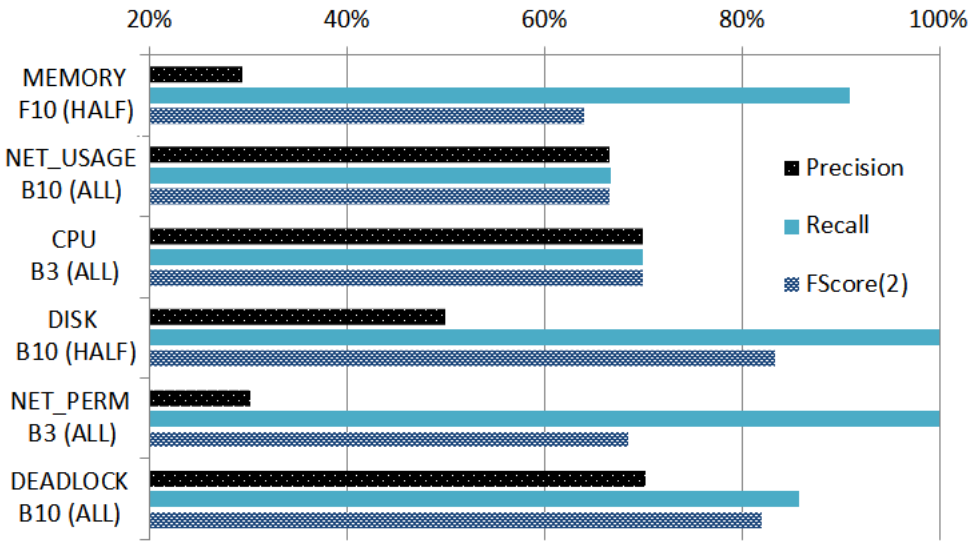


Figure 9: Precision, Recall and FScore(2) for experiments obtained using jSeduite

4.6.2 *Choice of Anomaly Checkers*

As counterparts of Table 8 and Table 9, Table 10 and Table 11 respectively report on: i) the data series used by the *selected anomaly checkers*, and ii) the 10 *selected anomaly checkers* that show the best FScore(2) when *NETWORK* anomaly is injected in the *jSeduite* experiments. The detailed list of the anomaly checkers for each category of anomalies is not reported here for brevity and can be found in [1]. In Table 8, we can observe how the data series related to the *selected anomaly checkers* are not balanced among layers. As expected, indicators related to the network played the most relevant role in identifying *NETWORK* anomalies. Other useful indicators were related to the OS, involving the number of files and highlighting an anomalous number of *CPU Idle / User* processes. This can be explained considering that the activation of the anomaly starts a process which fetches http data from a remote web site, storing read data in a dedicated file. The indicators gathered from the middleware (*JVM*), instead, do not take part to the *selected anomaly checker* set, meaning that this specific type of anomaly can be detected without looking at such data.

This trend is emphasized by the *selected anomaly checkers* (see Table 10): the 6 checkers with higher scores are built only on data series extracted from the network layer. In particular, *Tcp_Established* and *Tcp_Close* data series are widely used by most of these checkers, which apply either the HIST or SPS algorithm on complex data series created by combining such network-related simple data series. Further, we can observe how the selected checkers are mainly built on composed data series. To detect such *NETWORK* anomaly, different indicators were combined creating composed data series that provided

#	Indicator	Data Category	Series Layer
DS1	Tcp_Close	PLAIN	NETWORK
DS2	Tcp_Established	PLAIN	NETWORK
DS3	Tcp_Close	DIFF	NETWORK
DS4	Tcp_Established	DIFF	NETWORK
DS5	Active Files	PLAIN	CENTOS
DS6	Inactive Files	PLAIN	CENTOS
DS7	CPU Idle Processes	DIFF	CENTOS
DS8	CPU User Processes	DIFF	CENTOS
DS9	Net_Received	DIFF	NETWORK
DS10	Tcp_Listen	PLAIN	NETWORK
DS11	Major Page Faults	PLAIN	CENTOS

Table 10: Simple Data Series (DSx) used by the *selected anomaly checkers*

actionable information, turning out to be more useful than single simple data series.

4.6.3 *Sensitivity Analysis*

As in Section 4.5.3, we apply all the possible combinations among the selected anomaly checkers to identify the setup that ultimately gives a higher FScore(2) score. In Figure 10 we report the graphical result of the sensitivity analysis on the runs we used as validation of our framework to detect *MEMORY* anomalies. On the horizontal axis, we can observe the different strategies for the choice of the *selected anomaly checkers*, while on the depth axis we report the anomaly thresholds we consider in our setup.

Figure 10 highlights that the optimal setup is represented by F_{10} (HALF), which labels a snapshot as anomalous if at least five anomaly checkers trigger an anomaly for that snapshot. When considering checkers set composed by more than 10 elements the scores gradually degrade, meaning that the set of anomaly checkers that really determine our detection capabilities regarding *MEMORY* anomalies is limited.

Moreover, the usage of *anomaly tresholds* such as *THIRD, QUARTER, ONE* lowers the global *FScore(2)* since it raises the number of false positives, consequently lowering the precision. Results of the sensitivity analyses for the other injected anomalies are not reported here for brevity, but lead to similar observations and can be found in [1].

#	Data Series			Algorithm
	Indicator	Data Category	Series Layer	
AC1	DS1/DS2	DIFF	CROSS	HIST
AC2	DS3/DS4	DIFF	CROSS	HIST
AC3	DS3/DS4	PLAIN	CROSS	HIST
AC4	DS1-DS2	DIFF	CROSS	SPS
AC5	DS3-DS4	PLAIN	CROSS	SPS
AC6	Tcp_Close	DIFF	NETWORK	SPS
AC7	DS5-DS6	PLAIN	CROSS	SPS
AC8	DS7+DS8	DIFF	CROSS	SPS
AC9	DS9+DS4	PLAIN	CROSS	SPS
AC10	DS10-DS11	DIFF	CROSS	SPS

Table 11: *Selected Anomaly Checkers* for runs with the injection of NET_USAGE anomaly, in descending order

4.7 DISCUSSION OF THE RESULTS

We discuss here the scores obtained by applying our framework on both *Secure!* and *jSeduite* SOAs. Despite the intrinsic differences due to the application of our framework on two different case studies, we highlight the trends that turned out to be more relevant aggregating the results presented in Section 4.5 and Section 4.6. Data underlying the information presented here can be entirely found in [1].

4.7.1 *Detection Scores*

Overall, detection scores are evaluated taking *FScore(2)* as reference metric, since it combines *precision* and *recall*, weighting recall as more relevant than precision, as explained in Section 5.1. Experiments using the *Secure!* SOA scored higher values than the corresponding ones obtained using *jSeduite*.

Looking at the scores reported in Figure 7 and Figure 9, we can observe how both precision and recall scores are higher when analyzing traces related to the *Secure!* system. It is worth mentioning the results of the *NET_PERMISSION* anomaly: in both systems recall scores are very high, while precision is the lowest with respect to the other 5 anomalies. This can be explained as follows: several *selected anomaly checkers* are able to detect fluctuations related to the injected anomaly, but indeed these checkers are affected by a huge variability of the underlying data series, that leads them to raise a very high number of false

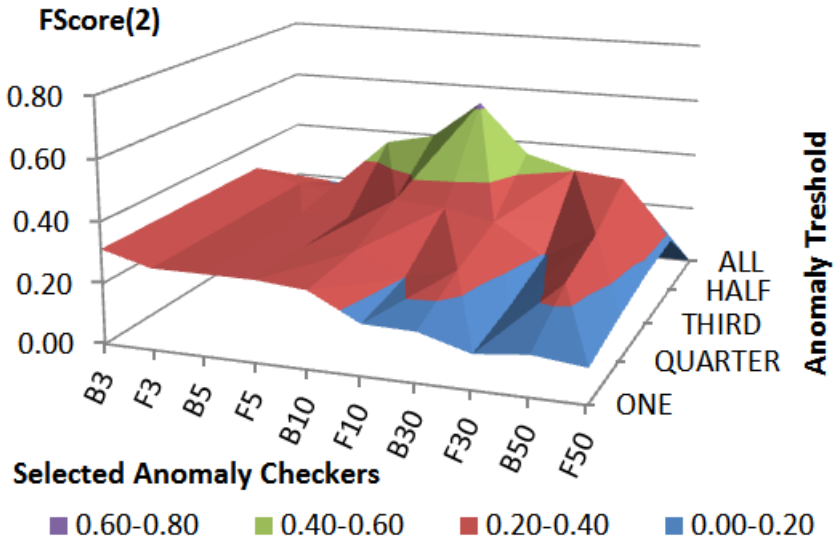


Figure 10: Sensitivity analysis of MEMORY experiments in jSeduite

alarms, thus generating low precision scores. Recall, instead, is generally low when trying to detect *CPU* anomalies. In the architecture of a system, processing units are always stimulated by a plethora of different tasks, including OS-level operations that cannot be controlled, since they may refer to less relevant context switches which may alter the CPU trend unpredictably. To limit the number of false positives, anomaly checkers are forced to set their internal parameters (e.g., the *pdv*, *pdf* paramaters of SPS) to looser values, consequently leading to miss some anomalies, which turns out to be FNs rather than TPs.

4.7.2 *Choice of the Indicators*

The large availability of indicators - and, consequently, anomaly checkers - forced us to aggregate the *FScores* obtained during the training phase for each anomaly checker, emphatizing the role of the indicators employed by the checkers. Results are shown in Table 12. Here we report a list of the most relevant indicators according to the strategy we used to aggregate the single scores of the anomaly checkers, the *TOP_10 Avg FScore(2)*. This calculates the average of the *FScores* obtained during training by the best 10 anomaly checkers that use a data series - either simple or composed - which regards a specific indicator. Looking at the first row of Table 12, the best 10 anomaly checkers that involve the *Tcp_Established* indicator as a part of the investigated data series score on average 0.21 as *FScore(2)* considering all the experiments with injection of anomalies in both *Secure!* and *jSeduite* systems.

Indicator		**TOP_10 Avg FScore(2))**			
Name	Layer	Rank	Avg	*Secure!*	*jSeduite*
Tcp_Established	NETWORK	1	0.21	0.25	0.17
Tcp_Close	NETWORK	2	0.21	0.23	0.19
Tcp_TimeWait	NETWORK	3	0.18	0.26	0.10
Net_Received	NETWORK	4	0.15	0.18	0.14
CPU Idle Processes	CENTOS	5	0.14	0.11	0.17
Buffers	CENTOS	6	0.13	0.18	0.07
CurrentThreadUserTime	JVM	6	0.13	0.18	0.08
CPU User Processes	CENTOS	6	0.13	0.12	0.13
Minor Page Faults	CENTOS	9	0.12	0.12	0.12
CurrentThreadCpuTime	JVM	9	0.12	0.13	0.10
FreePhysicalMemorySize	JVM	9	0.12	0.13	0.11
Tcp_Syn	NETWORK	9	0.12	0.17	0.06
HeapMemUsage.committed	JVM	9	0.12	0.15	0.09

Table 12: Most relevant Indicators for experiments of both *Secure!* and *jSeduite*

Further, we can observe that the 4 most relevant indicators belong to the *NETWORK* layer. This is remarked in Table 13, where the *TOP_10 Avg FScore(2)* was calculated aggregating all the indicators belonging to a specific layer. In particular, *NETWORK* layer turned out to be the most relevant, while *JVM* and OS (*CENTOS*) layers are a bit below on average. It is important to remark that such tables report on average scores: having the *NETWORK* layer as the best on average does not mean that the most performing anomaly checker is built on a data series that use indicators coming from the *NETWORK*. Therefore, our result is that the *NETWORK* layer provided on average good indicators for our analysis such as *Tcp_Established*, *Tcp_Close*, *Tcp_TimeWait*, and *Net_Received* in Table 12, but it is not straightforward that the best checkers for identifying - as example - *MEMORY* anomalies use indicators coming from such layer. This information is really anomaly and system-dependent and must be analyzed case by case.

4.7.3 *Contribution of the Algorithms*

Along with the analysis on the most relevant layers to instrument that was expanded before, we also analyzed the impact on the choice of the algorithms that are implemented by the anomaly checkers. Unfortunately, in our implemen-

Layer	TOP_10 Avg FScore(2))			
	Rank	Avg	*Secure!*	*jSeduite*
NETWORK	1	0.20	0.23	0.16
JVM	2	0.18	0.18	0.17
CENTOS	3	0.17	0.16	0.17

Table 13: Most relevant Layers to monitor for experiments regarding both *Secure!* and *jSeduite*

tation we considered only two algorithms, namely *SPS* and *HIST* (see Section 5.3.1), which limits the number of statistics we are going to present. In fact, we can only compare SPS and HIST algorithms to find out which one was more in the *selected anomaly checker* sets selected by our framework in all the experiments we executed. Table 14 summarizes our analysis on algorithms: we reported average and standard deviation scores regarding three metrics. These report on the number of checkers that are built on a given algorithm considering the best 10 (*OCC_TOP_10*), best 50 (*OCC_TOP_50*), and best 100 (*OCC_TOP_100*) checkers in each experiment building our experimental campaign.

Algorithm	OCC_TOP_10		OCC_TOP_50		OCC_TOP_100	
	Avg	Std	Avg	Std	Avg	Std
SPS	5.50	1.76	37.90	1.01	81.60	0.65
HIST	4.50	2.56	12.10	3.18	18.40	3.00

Table 14: Algorithms contribution for experiments regarding both *Secure!* and *jSeduite*

Overall, we can notice that the number of SPS checkers, regardless the size of the considered best checkers set, occur in a higher number. Especially when increasing the size of the considered set, the difference between the amount of SPS and HIST checkers becomes more evident. In particular, looking at the *OCC_TOP_100* results in Table 14, we observe that approximately 84 anomaly checkers over the best 100 use SPS as detection algorithm. This trend is confirmed also by the *OCC_TOP_50* metric, while the *OCC_TOP_10* results highlights that the best 10 checkers are usually balanced among the two algorithms. Summarizing, on average there are 4-5 HIST-based checkers that are ranked in the top 10 in our experimental campaign, while following postitions of the rank are mainly occupied by SPS-based checkers.

4.7.4 *Sensitivity Analysis*

Also regarding the choice of the *selected anomaly checkers* and the *anomaly threshold* parameters, differences can be observed among the two case studies. The optimal setups for both parameters were the same i.e., $F_{10}(HALF)$, despite that in the *Secure!* case study we observed that also the B_3 setup for the *selected anomaly checkers*, was sub-optimal. Overall, as shown in Figure 8 and Figure 10, scores are generally lower in *jSeduite* than in the *Secure!* case study. In particular, precision results are low in *jSeduite*, leading also to lower *FScore(2)* values thn in *Secure!*. Recall itself is usually high - more than 80% -, representing a big achievement keeping in mind the challenges related to perform anomaly detection in dynamic systems.

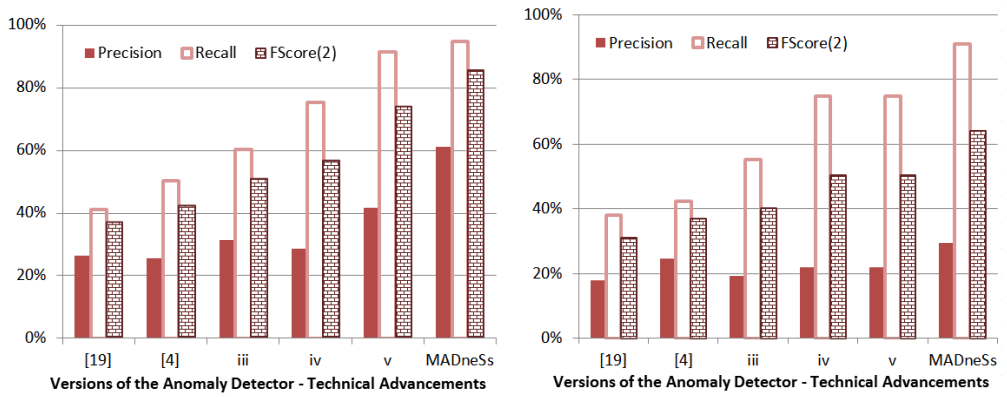
We observe that, when possible, the framework automatically tries to balance the average low *precision* score by suggesting an *anomaly threshold* that requires several anomaly checkers to raise an anomaly simoultaneously, e.g., *HALF, ALL*. This lowers the number of false alarms, since a consensus among different anomaly checkers is requested, therefore limiting the fluctuations due to single indicators that may raise anomalies that are not linked with the occurrence of faults.

4.7.5 *Summary of the Incremental Improvements*

To conclude the analysis of the results, we present our resulting setup of the framework by enphatizing the subsequent introductions of technical advancements, starting from the anomaly detector running SPS in static systems which is described in [18]. Such technical advancements have already been summarized in Table 5. We present here the scores obtained step-by-step by considering: i) the model of anomalies we built in this chapter, and ii) the metric scores regarding both *Secure!* and *jSeduite* SOAs.

In Figure 11 we depicted *Precision*, *Recall* and *F-Score(2)* that we obtained when we applied the above anomaly detectors to our SOAs. The figure allows noticing that the introduction of a new technical advancement always increased both recall and *FScore(2)*. In particular, *F-Score(2)* and recall are significantly improved when network probes are added. This can be easily explained, because the observation of the network layer introduces new relevant data series (see Table 13) for anomaly detection.

Instead, precision often decreases when including new data series, either due to the instrumentation of a new layer or to the addition of the DIFF data type (technical advancement *iv* in Figure 11). We explain this as follows. The introduction of novel data series increases the number of anomaly checkers, without the ability to distinguish which are the most effective (this is instead done in the successive technical advancements, number *v* on the x-axis in Figure



(a) Secure!

(b) jSeduite

Figure 11: Detection capabilities for the six versions of anomaly detectors regarding the MEMORY anomaly

11). Consequently, the anomaly checkers may be able to improve detection capability (increase of recall), at the cost of an increased number of false alarms. This will be solved in the successive technical advancement, where information on the context is included and it will allow improving the definition of the expected behavior.

Lastly, the introduction of a composed data series and the inclusion of a sensitivity analysis lead to the final version of the framework, and consequently to the best scores in Figure 11.

4.8 PERFORMANCE

4.8.1 *Training Time*

According to our methodology, we need to build the fingerprints of the (web) services. Here we investigate the time required to i) exercise the expected workload by conducting preliminary runs, and ii) analyse these data to characterize services and to obtain the parameters that each anomaly checker requires.

In Table 15 we report the time required to execute preliminary runs: we compute the time needed to test i) a single web service, and ii) all the web services in a row (*All Tests*). Once preliminary runs are conducted - and services information is stored in the database - data is analysed to select the best combination of parameters for each considered anomaly checker. The performance of this operation is strictly dependent on the characteristics of the anomaly checker, and on the amount of training data that is used to select the best configuration.

Regarding anomaly checkers, we analysed the time needed to select the best configuration using the data related to experiments conducted on *Secure!*. We

Service		Single Test	
Name	Type	Average	St.Dev
Authentication_getCredentials	Authentication	8.88	0.60
WT_addBookmark	Serv. Test	13.25	1.22
WT_addBookmarkFolder	Serv. Test	9.35	0.65
WT_addEventInCalendar	Serv. Test	11.05	0.86
WT_addFiles	Serv. Test	11.25	0.43
WT_createFolder	Serv. Test	12.10	0.94
WT_deleteFolder	Serv. Test	10.97	0.22
WT_getBookmarkFolder	Serv. Test	16.36	4.16
WT_getFolder	Serv. Test	11.06	0.22
WT_newPoll	Serv. Test	9.55	1.24
WT_removeEventInCalendar	Serv. Test	10.39	0.50
All Tests	Test All	72.98	7.65
All Services	Workload	64.04	5.87

Table 15: Execution Time of Tests and Workload

analysed all the 478 training runs, that corresponds to 100 golden runs and 63 runs for each of the 6 anomalies of our experimental evaluation. For (average, median, standard deviation), the SPS anomaly checkers measured (42.68, 42.21, 1.15) ms, while the HIST measured (0.45, 0.33, 0.19) ms. While these times are short, we remark that they are related to the analysis of a single anomaly checker. In the worst case, when all the possible 1004 anomaly checkers are considered, we respectively need 1348.9 and 14.2 seconds to select the best configurations of anomaly checkers for SPS and HIST for a single set of 63 training runs.

4.8.2 *Notification Time*

After discussing the time needed for training, it is important to evaluate how fast the response of the framework is. An anomaly detector that is not able to provide a quick anomaly evaluation could be useless although it performs a very efficient detection of anomalies.

To evaluate the notification time, we analyze the *observation time* (ot), *probe-monitor transmission time* (pmtt), *data aggregation time* (dat), and *detection time* (dt) quantities related to the experiments. To understand the impact that the setup of the probing system has on the notification time, we exercised the *AllServices* workload on the *Secure!* system with different setups of the probes. We used the

same setup described in Section 4.4, without injecting any anomaly. We collected performance data of 100 golden runs for each of the following setups of the probes:

- *Single Indicator*: data related to a single indicator per layer is read i.e., *JVM SessionCounter*, *CentOS Buffers*, *Network Tcp_Syn*;
- *Half Indicators*: a few indicators are monitored during each run i.e., 15 *CentOS*, 4 *Network*, 7 *JVM*;
- *All Indicators*: all the indicators are observed for each layer i.e., 23 *CentOS*, 7 *Network*, 25 *JVM*. This is the setup that was adopted in the experimental campaigns we presented previously;
- *No CentOS*: the probes monitoring the *CentOS* layer are disabled;
- *No Network*: the probes monitoring the *Network* layer are disabled;
- *No JVM*: the probes monitoring the *JVM* layer are disabled;

As scoring metric, we considered *precision*, while the *selected anomaly checkers* and *anomaly treshold* parameters were respectively set to *BEST 10* and *ALL*, as it results from the sensitivity analysis related to such data.

The results of such experiments are reported in Table 16. It can be noted that the task that needs more time to be completed is the transmission of the information from the Target Machine to the Detector Machine. In fact, the time quantity *pmtt* represents on average the 53% of the whole notification time (see last row of the table. The detection time is short, scoring on average the 8% of *nt*: these performances are strictly related both to the number and to the type of the anomaly detectors that need to be checked on each step. While *HIST* checkers need on average 10 µs to execute their tasks, the *SPS* ones need on average 320 µs each with the current hardware setup of the machines in which we ran the experiments. This means that if the *selected anomaly checkers* set is mainly built of *SPS* chechers, the time needed for detection is supposed to grow.

Moreover, we observe that the time spent from the probes in retrieving data (*ot*) is not too dependent on the amount of indicators from the same layer we consider. We can also observe that despite the *ot* seems strongly dependent on the number of observed indicators (the value in the experiment "many indicators" is doubled compared to the "less indicators" or "single indicator" one in Table 16), from the last experiment we can observe how the ot value is significantly lower if we do not consider the *JVM* probe. For each indicator, this probe activates the *JMX* primitives, asking to JVM to look at different *MBeans*. CentOS and Network probes, instead, use UNIX primitives (e.g., *top* command, reading */proc/vmstat* file) which retrieve the whole block of possible indicators, so selecting all or only a small subset of them will not heavily affect the *ot* value.

Probing Setup	ot			pmtt			dat		
	avg	std	% nt	avg	std	% nt	avg	std	% nt
Single Indicator	6.32	0.50	30.6	12.01	41.89	58.2	0.38	0.01	1.8
Half Indicators	6.06	0.95	27.5	12.25	39.76	55.7	0.52	0.07	2.3
All Indicators	11.72	1.35	37.5	15.49	49.63	49.5	1.46	3.05	4.7
No CentOS	12.66	0.52	41.5	15.31	32.33	50.1	1.26	2.23	4.1
No Network	12.75	0.39	39.7	15.95	35.07	49.7	0.95	0.08	3.0
No JVM	6.83	0.92	31.4	12.70	20.68	58.3	0.50	0.10	2.3
Average %			34.7			53.6			3.0

Probing Setup	dt			nt	
	avg	std	% nt	avg	std
Single Indicator	1.93	0.32	9.4	20.64	42.72
Half Indicators	3.18	0.96	14.4	22.01	41.74
All Indicators	2.61	0.49	8.4	31.28	54.52
No CentOS	1.31	0.17	4.3	30.54	35.25
No Network	2.44	0.41	7.6	32.10	35.96
No JVM	1.75	0.21	8.0	21.79	21.91
Average %			8.7		

Table 16: Estimation of the notification time with different setups of the probing system. Results are reported in milliseconds

Summarizing, also with the setup of the machines that could be improved both in terms of hardware and network speed, the notification time *nt* is very challenging for non-real time systems. However, it might be needed to look at a bigger set of indicators of to use more predictors for anomaly detection purposes. This would lead to a wider time window between the observation of a system snapshot and the possible anomaly notification, slowdowning the whole process.

5

MADNESS: A MULTI-LAYER ANOMALY DETECTION FRAMEWORK FOR DYNAMIC COMPLEX SYSTEMS

This section presents the main results of this Thesis. More in detail, we first expand the design guidelines we extracted from the experiments we run and discussed previously; then, such guidelines are implemented in the *MADneSs* framework.

5.1 DESIGNING A FRAMEWORK FOR ANOMALY DETECTION

Here we explore the main design principles - summarized as *viewpoints* (VP) - behind a monitoring framework for anomaly detection, highlighting: i) the purpose of the framework, ii) the investigated anomalies, iii) the monitoring approach, iv) the indicators to be monitored, and v) the anomaly detection technique. In Table 1 we reported several frameworks obtained through a non-systematic literature review in which authors adopted different approaches to solve the design challenges discussed below. In Table 17, instead, we grouped all the relevant characteristics of such frameworks according to the viewpoints that are discussed in this section.

5.1.1 *Viewpoints*

Here we summarize different ways to design, develop and maintain dependable and/or secure complex systems as *viewpoints*. More in detail, these viewpoints are dimensions of analysis for designing a monitoring and anomaly detection framework for complex systems. In particular, we will expand and focus on the viewpoints *purpose*, *anomalies*, *monitoring approach*, *selected indicators*, *detection algorithm*.

VP: Purpose

As discussed in Section 2.4, anomaly detection was proven effective to the purpose of security and dependability. Depending on the specific needs of the administrator or the owner of the system, a monitoring framework can be designed to prioritize security (e.g., intrusion detection) or dependability (e.g., error detection, failure prediction) through the identification of anomalous behaviors. This choice influences the whole planning of the framework, defining the threats we want to detect.

APPROACHES IN EXISTING FRAMEWORKS The frameworks in Table 17 are meant to conduct anomaly detection for different purposes. Frameworks for error detection [9] investigate anomalies to interrupt the fault-error-failure chain. Failure predictors [11], [101], [73] assume that errors already manifested in the system, and try to avoid their escalation in failures or the propagation to unsafe states. In the security domain, we can classify i) intrusion detectors [74], [87], which represent a security layer preventing or blocking possible malicious attacks, and ii) malware detectors, which analyse the system to identify anomalous behaviours due to malicious modules that are already infecting the system. Frameworks aiming at detecting malware through anomaly detection are not reported in the table since they are mainly directed to standalone mobile devices.

VP: Anomalies

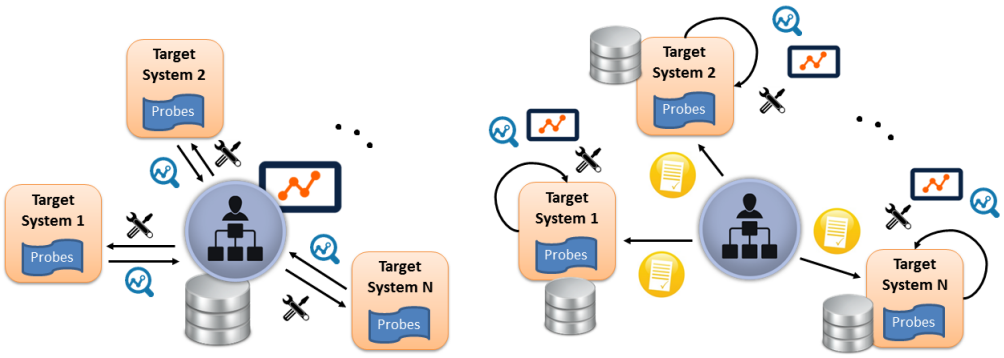
Right after the choice of the purpose of the framework, a deep analysis of the system is needed to clarify which can be the sources of errors that may damage the system while executing. Once the major sources of possible errors will be identified, we can proceed to set the anomaly detector to look at specific parts of the system that are related to such root causes. In fact, anomalies are consequences of the manifestation of errors, and their definition is a key activity while organizing a monitoring framework, since it *helps tracing the boundaries between normal and anomalous behaviors* for the part of the system being intrumented with probes. Beyond that, a key advantage of anomaly detection with respect to - as example - pattern or fingerprint based detectors is that in some cases the manifestation of **unknown errors** or zero-day attacks may generate anomalies that are detected by the framework also if it is not specifically targeting them. However, this will happen only when the expectations - or rather what the system is intended to do during its normal behavior - are clearly defined. To such extent, anomalies can either be defined as unexpected usage of resources, or suspicious requests to modules, or new actions taken by users without previous acknowledgment.

APPROACHES IN EXISTING FRAMEWORKS Almost all the anomaly detectors investigate anomalies related to resource usage. This has key drawbacks when monitoring for intrusion detection [74], [87], because common attacks such as *Denial of Service* or *Ping Flood* conduct an abnormal number of requests to the targeted system through the network to damage the normal operations. However, when monitoring for dependability, an anomalous usage of system resources such as i) memory and CPU [101], [9], ii) disk accesses [92], [73], iii) network-only detectors [11], or iv) logs [41], may be due to the activation of some errors in one of the service or module composing the system. Instead, only a few studies focus the attention on anomalies due to misconfiguration or

wrong interactions [9], [67]; these are usually harder to define and also to detect, making the implementation of a framework targeting such specific aspects more complex.

Framework	Purpose	Anomalies	Monitoring Approach	Selected Indicators	Detection Algorithm
ALERT [92]	Dependability	Resource Usage	Distributed	From IBM SystemS, PlanetLab layers	Decision Tree Classifier
CASPER [11]	Dependability	Resource Usage	Centralized	Network layer	Hidden Markov Models
[9]	Dependability	Reconfiguration, Misconfiguration, Resource Usage	Distributed	*CPU*, *Memory* usage, Network layer	Invariants
SEAD [73]	Dependability	Resource Usage	Centralized	Domo and Xen Hypervisor layers	Support Vector Machines
TIRESIAS [101]	Dependability	Resource Usage	Distributed	*CPU*, *Memory Usage*, *Context Switches*	Dispersion Frame Technique
[41]	Dependability	Resource Usage	Distributed	log files	Finite State Automation
SSC [87]	Security	Resource Usage	Centralized	*proc*, *sysinfo* UNIX data, *JVM*	Most Appropriate Collaborative Component Selection
McPAD [74]	Security	Payload	Centralized	*HTTP* data exchange	Support Vector Machines

Table 17: Characteristics of the Framework according to Viewpoints



(a) Centralized Approach (b) Distributed Approach

Figure 12: Centralized and Distributed Approaches

VP: Monitoring Approach

Different monitoring approaches [55], [65], [103] can be adopted depending on where the data analysis engine e.g., the anomaly detector, is executed. Moreover, data sets storing historical or generic support data that are used for analyses can be i) put on an external machine coordinating the monitoring and data analysis activities, or ii) distributed on the target nodes of the complex system. This results in the following two monitoring and data analysis approaches, as it can be seen in Figure 12.

CENTRALIZED APPROACH An external coordinator (Detector Machine) manages both monitoring and data analyses activities. It sends basic rules to setup the probes on the Target Machine(s) and waits for monitored data, that is gathered by the probes on the Target Machine(s) and is directly sent to the Detector Machine without performing any additional operation. The Detector Machine also keeps track of historical or support data to assist anomaly detection. Once received, monitored data is analyzed for anomaly detection purpose, and eventually an alert is raised to the administrator if anomalies are detected (see Figure 12a).

DISTRIBUTED APPROACH In this setup, the coordinator only provides to Target Machines policies or rules for data collection and analysis e.g., set of indicators to monitor, thresholds or parameters of the anomaly detector (see Figure 12b). This allows the Target Machines to share a common core of parameters for data analysis. With this approach, the coordinator does not represent a bottleneck; instead, each Target Machine must allow running custom tasks that may drain system resources, negatively impacting on system performances.

APPROACHES IN EXISTING FRAMEWORKS Depending on the context, frameworks for anomaly detection can be designed to centralize or decentralize the heaviest computing operations. Distributing operations [92], [41] reduces the bottleneck around the coordinator, but requires well-developed distribution of loads and tasks among the target machines. Nevertheless, the surveyed anomaly detection frameworks [74], [87] targeting security do not usually consider a distributed data analysis approach. Instead, the preferred method is to send collected data to a central elaboration unit. This does not allow sharing parameters of the anomaly detection strategy with all the target machines, blocking adversaries that want to intercept such communications in order to read, corrupt or modify such critical parameters.

VP: Selection of Indicators

Nowadays software is becoming more complex and consequently a large number of performance indicators e.g., memory usage, cache hits, packets shared through the network, can be captured by specific probes at defined time instants. Observing indicators related to different layers of the system e.g., *OS*, *network*, *database* can provide a more accurate view of the system. The observed data needs to be transmitted and analyzed continuously, potentially slowdowning the monitored system. Thus, it becomes fundamental to select only those indicators that are most useful to detect anomalies. [52]

In fact, previous research shows that even in a complex system the set of relevant variables is typically quite small [50]. Moreover, depending on the specific analyses that will be conducted using the monitored data, indicators can be classified extracting a minimum set that allows reaching defined performance scores. For example, sets of indicators were extracted targeting failure prediction [52], anomaly detection through invariants [9] and errors due to software faults [18].

An important remark should be done to consider the requirement of having all target machines synchronized to a global time. Otherwise, it is not possible to build a reasonable global time base. This affects our ability of merging information coming from different target machines ultimately providing polluted data to the data analysis modules. For example, consider the final report about a major power blackout occurred in parts of the US and Canada in 2003. Here the authors declare that i) the Task Force's investigators labored over thousands of items to determine the sequence of events, and that ii) the process would have been faster and easier if there had been wider use of synchronized data recording devices [60].

APPROACHES IN EXISTING FRAMEWORKS Most of the anomaly detectors observe performance indicators targeting OS [101], [9] and network [11], [101], [92], [74] layers. We explain this results as follows: i) these layers are always

present in a complex system, and ii) enterprise monitoring tools [3], [2], [5] offer probes to observe these two layers. Moreover, several indicators regarding the memory and cache management can be retrieved only at OS-level, because middleware e.g., *JVM*, application servers such as *Apache Tomcat*, act at a higher stack level.

VP: Detection Algorithm

As highlighted in [25], a key aspect of any anomaly detection technique is the nature of the input data. Each data instance might consist of only one attribute (univariate) or multiple attributes (multivariate). In the case of multivariate data instances, all attributes might be of same type or might be a mixture of different data types. The nature of attributes determines the applicability of anomaly detection techniques. For example, for statistical techniques [89] specific statistical models have to be used for continuous and discrete data. Similarly, for nearest-neighbour-based techniques [79], the nature of attributes would determine the distance measure to be used. Moreover, when aggregated measures instead of raw data are provided e.g., distance or similarity matrix, techniques that require original data instances such as classification-based techniques [35] are not applicable. Most of the techniques mentioned above need training data to learn the characteristics of both normal and anomalous instances to classify probes' data. Focusing on complex systems, we observe that these systems can be characterized by intrinsic dynamicity, often changing their behaviour and, consequently, the characteristics of both normal and anomalous behaviours. This calls for a new training phase, requiring i) to collect train data and ii) to find the optimal parameters related to the chosen techniques.

When dynamicity is very high, training can overcome the normal activity of the system, resulting in large periods of unavailability of the anomaly detector and slowdowning the usual tasks that run on the Target Machine(s). This means that anomaly detection techniques that do not need exhaustive sets of training data i.e., *unsupervised detection algorithms*, are more suitable because they do not require periods of unavailability for training [18], [90].

APPROACHES IN EXISTING FRAMEWORKS Different studies adopt different data analysis approaches: as explored in [25], specific anomaly detection approaches call for a suitable anomaly detection algorithm or technique. This results in a widespread use of statistical (3 out of 9 in Table 17) and machine learning (4 out of 9) algorithms, while [9] and [87] respectively score anomalies using invariants and components selection. As already expanded, despite the fact that statistical and machine learning worked very well in the studies reported in Table 17, from the complex-systems perspective the usage of these algorithms raises important concerns that cannot be ignored.

5.1.2 *Performance*

To guarantee the best support either for dependability or security purposes, anomaly detectors are intended to analyze monitored data and provide their results rapidly and with a low number of wrong interpretations. Consequently, the *notification time*, or rather the time between the observation of system data through probes and the evaluation of its anomaly degree, should be minimized. Moreover, an inaccurate evaluation can result in i) *false positives*, which can cause the execution of non-required reaction strategies by the administrator, or ii) missed detections (*false negatives*), with possible severe consequences.

Taking into account the following performance targets is mandatory and it has to be part of the development phase of anomaly detection frameworks.

Detection Performance

Depending on the purposes of the targeted complex system, the reference metric may change: for example, in systems where false negatives (i.e., missed detection of an anomaly) can heavily impact the system, recall is more relevant than precision. Instead, when detection of anomalies (both TP and FP) calls for expensive reaction strategies, FP must be minimized, thus emphasizing precision more than recall.

Notification Time

Another performance index that needs to be addressed is the notification time, that is the time between the observation of a data instance and its evaluation. This quantity can be considered as a composition of (see Figure 13):

- observation time (ot): the time slot spent from the probing system to get system data by the probes;
- probe-monitor transmission time (pmtt): the time needed to transmit all the observed data to the monitor
- data aggregation time (dat): the time used by the monitor to aggregate and parse the received data
- storing time (st): time spent from the monitor to store the aggregated data in the chosen data container;
- monitor-detector transmission time (mdtt): the time needed to transmit the data aggregated from the monitor to the anomaly detector tool;
- detector time (dt): the time used for the anomaly detector to compute its calculations based also on previously collected historical data;

- alert time (at): the time needed to deliver the anomaly alert to the system administrator.

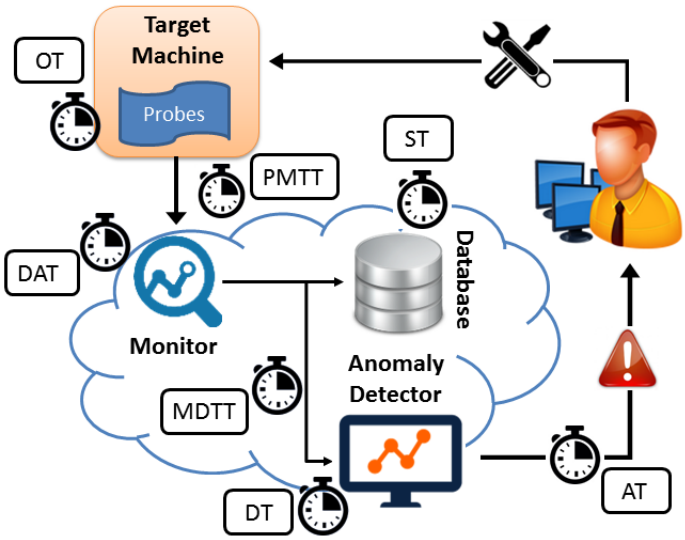


Figure 13: Time quantities through the workflow

Depending on the chosen monitoring approach, these quantities can be combined to obtain the notification time (nt) as follows.

CENTRALIZED In this approach, the coordinator i) runs the monitor and the anomaly detector and ii) hosts the database in which support data (if any) are stored. Considering the anomaly alert as a simple notification e.g., text message or email, the quantities *mdtt* and *at* may represent negligible instants of time. Assuming *nt* as the notification time, in such a context its value is expressed as the linear combination of the remaining quantities:

$$nt = ot + pmtt + dat + st + dt$$

When collected data do not need to be stored for further analyses, the quantity *st* can be considered null, thus resulting in a notification time that is calculated as

$$nt = ot + pmtt + dat + dt$$

DISTRIBUTED Monitoring and data analysis logic are placed on the Target System(s), while the coordinator supports these activities providing parameters

or rules e.g., set of indicators to monitor, rules for anomaly detection. Consequently, each target machine should run dedicated modules that can interfere with the tasks that are usually executed on its target machine resulting in a higher intrusiveness level that needs to be taken into account. Nevertheless, considering that i) data can be stored in the database simultaneously with the aggregations performed by the monitor, and ii) the possible alert needs to be forwarded to the coordinator, the *nt* can be estimated as:

$$nt = ot + max\{dat, st\} + dt + at$$

5.2 OUR FRAMEWORK FOR ANOMALY DETECTION IN DYNAMIC SYSTEMS

5.2.1 *Multi-Layer Monitoring*

Our approach consists in shifting the observation perspective from the application layer where the services reside to the underlying layers such as operating systems, application servers, network protocols [105] or databases [57]. Anomalous behaviors generated by services e.g., due to manifestation of errors or attacks, can be detected observing exclusively data gathered from the underlying system layers. Consequently, we apply a multi-layer monitoring strategy (CH.3). The system layers that we chose to instrument are the OS, the network, and the *Java*-based middleware. We choose to investigate indicators related to such layers since in a system OS and network are common layers, while *Java* constitutes the basic virtual machine to execute several applications or middleware e.g., *Apache Tomcat* or *JBoss*.

5.2.2 *Detection Algorithm*

Monitored data is processed by the selected anomaly detection techniques. To cope with the dynamicity of the system, adaptive anomaly detection techniques need to have their parameters adapted to the current context (CH.1). In particular, we propose to: i) check if the current data instance complies with the expectations we previously extracted digging historical data, and ii) run a suitable anomaly detection algorithm such as the *Statistical Predictor and Safety Margin* (SPS, [14]) algorithm.

In general, this self-adaptive algorithm allows detecting the observations that do not follow the statistical inertia of a data series with reduced computational or memory demands (CH.1, CH.4) [14], [18]. However, they are effective in identifying point anomalies, lacking in identifying groups of subsequent anomalies (collective anomalies).

5.2.3 *Context-awareness and Contextual Information*

Server-side context awareness and the resulting contextual information can facilitate the description of the expected behaviour of the services (CH.1, CH.3). In our approach, contextual information has a key role in defining the boundaries between expected and anomalous behaviour of the system.

We aim at taking advantage of contextual information to characterize the expected behaviour of the system during the execution of a service, ultimately building a fingerprint of its usage. More in detail, we need to describe the expected trend of monitored indicators while the service is invoked. Observing the ESB, the fingerprint is created when the addition, update or removal of a service is detected. Then, this information is aggregated and maintained in a database, together with statistical indexes e.g., mean or median, whenever applicable, to support anomaly detection.

5.2.4 *Composed data series*

Relations between different indicators are often difficult to catch. A possible approach based on invariant properties was proposed in [9] to detect anomalies. Briefly, invariants are stable relations among system indicators that are expected to hold during normal operating conditions: a broken invariant reflects an anomalous state of the system. Unfortunately, considering an invariant as a stable relation is not valid, therefore the usage of invariants calls for dedicated selection strategies that are difficult to define and update, and heavy to execute. Therefore, we choose to model relations between indicators by interpreting data series as sequences of data points obtained through the combination of two or more data series (CH.2). The result is a composed data series, which can be analysed as any other data series related to a single indicator. Consequently, in the rest of the paper we distinguish between simple data series and composed data series.

Further, initial training allows selecting the indicators that are relevant to detect anomalies; online training (performed when new fingerprints are defined) allows updating such list of indicators according to the current context (CH.5).

5.2.5 *Facing Point, Contextual and Collective Anomalies*

We explore how the considered techniques cooperate to detect *point*, *contextual*, and *collective* anomalies (CH.4). According to this classification, in Table 18 we reported the techniques that we adopted in our approach together with the anomalies they can detect. In particular, unsupervised anomaly detection algorithms such as SPS are meant to identify values that do not follow the

statistical inertia of the data series under study, and consequently is suitable for detecting point anomalies.

Anomaly Type	**Technique**	**Requirements**
Point	Unsupervised Detection Algorithm (SPS)	Training of SPS
Contextual	Contextual Check	Gathering data about expected behavior of services
Collective	Unsupervised Detection Algorithm (SPS), Contextual Check involving Complex Data Series	Training of SPS, Gathering data about expected behavior of services, Definition of relations among simple data series

Table 18: Techniques for detecting categories [25] of anomalies.

Contextual information makes us able to check if the observed behaviour is compliant with the expected behaviour i.e., perform a contextual check. Consequently, this makes it possible to identify several contextual anomalies.

Dealing with collective anomalies is generally more difficult. In some cases, the checks that label a single data instance as either point or contextual anomaly can also successfully identify collective anomalies, or rather multiple anomalous data instances. However, collective anomalies may not differ significantly from the expected trend in a given context, or they can be erroneously evaluated as a new trend resulting from system dynamics. To cope with this specific category of anomalies, we adopt *composed data series*. This allows describing complex relations between indicators, possibly leading to identify collective anomalies.

5.2.6 *Online Training*

When dealing with dynamic systems, frequent training phases are needed to keep the parameters and the anomaly checkers compliant with the current notion of expected and anomalous behavior. Several authors [63], [45], [86], [83] working on detector or predictors in the context of complex systems proposed an **online training** approach.

A strong support to the design of online training techniques comes from systems that continuously monitor trajectories. Often, these systems try to adapt detector's parameters as they evolve to understand if they are following expected or anomalous paths (see conformal anomaly detection [63]). In [45], authors

tackle online training for failure prediction purposes i) continuously increasing the training set during the system operation, and ii) dynamically modifying the rules of failure patterns by tracing prediction accuracy at runtime. A similar approach is also adopted to model up-to-date *Finite State Automata* tailored on sequences of system calls for anomaly-based intrusion detection purposes [86] or *Hidden Semi Markov Models* targeting online failure prediction [83].

When the target system is dynamic, it may change its behavior in different ways, triggering new training phases aimed at defining the "new" expected behavior. Moreover, according to [45], the training set is continuously enriched by the data collected during the executions of services, providing wide and updated datasets that can be used for training purposes. This training phase starts once one of the triggers is activated. We are currently using three triggers that can be detected looking at the SOA and system setups: i) update of the workload, ii) addition or update of a web service in the platform, iii) hardware update.

5.3 INSTANTIATION OF MADNESS

Here we describe the *MADneSs* framework, that implements the design choices described and motivated before.

5.3.1 *High-Level View*

In Figure 14 we depict a high level view of *MADneSs*; from left to right, the framework can be described as follows. The users execute a workload, which is a sequence of invocations of services hosted on several physical or virtual machines. One or more host machines can be monitored, thus becoming the *Target Machines*, as shown in the bottom left of the figure. In each target machine, probes observe the performance indicators related to 3 different system layers: i) OS, ii) middleware (*Java Virtual Machine*, JVM) and iii) network. These probes repeatedly collect data at specific time instants, which is then aggregated in a **snapshot** of the system hosted on the Target Machine, which therefore contains the observation of indicators retrieved at a defined time instant. The probes forward the snapshot to the communication handler, which encapsulates and sends the snapshot to the communication handler of the *Detector Machine*. Here, the anomaly detection module analyzes such monitored data.

Performing data analysis on a separate machine allows i) reducing intrusiveness on the Target Machine (CH.2), and ii) connecting more Target Machines to the same Detector Machine. The Target Machines are instrumented with custom probes, which have only basic functionalities to minimize the disturbance of target system (CH.3). The communication handler of the Detector Machine collects and sends the snapshots to the monitor aggregator, which merges them

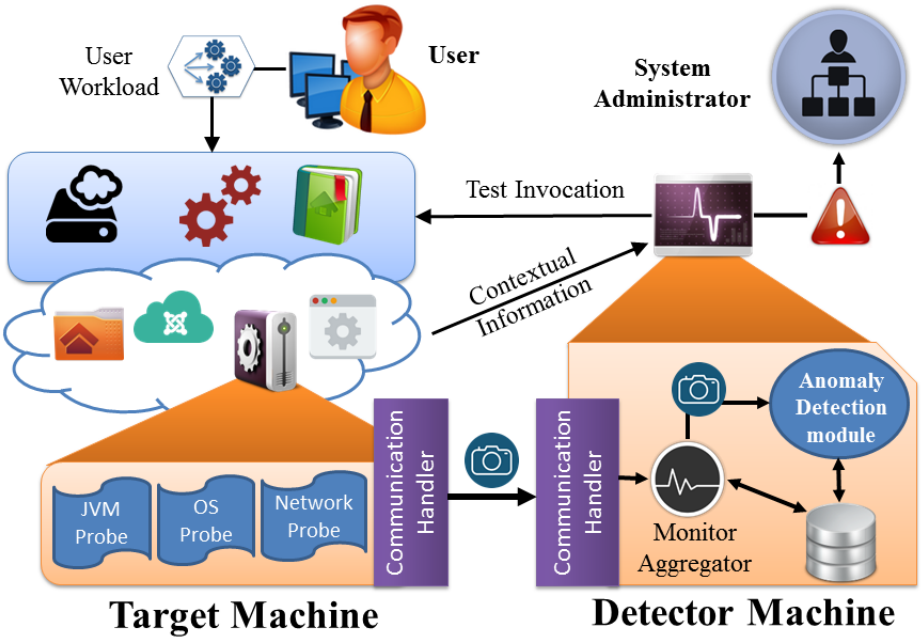


Figure 14: High-Level view of MADneSs

with a **fingerprint** of each service that is obtained through tests and stored in the database (see the bottom-right of Figure 14) for further comparisons. More in details, once changes in the services are detected, tests are run (test invocation) to gather a novel or updated fingerprint of the services. In our implementation, the fingerprint is composed from contextual information extracted from the integration layer of the complex system (i.e., the *Enterprise Service Bus*, ESB in SOAs). When a snapshot is sent to the Anomaly Detection module, it queries the database for services fingerprints, which are used to analyze the snapshot. If the snapshot is evaluated as anomalous, an alert is raised, e.g., the system administrator is notified. Possible countermeasures or reaction strategies are outside from the scope of this work and will not be elaborated further.

Insights on Monitor Aggregator

Periodically, e.g., once per second, the *Monitor Aggregator* of the Detector Machine in Figure 14 provides to the anomaly detection module a snapshot of the observed system, composed of the quantities retrieved from the probes installed on the Target Machine(s). For each indicator, two quantities are provided: i) **PLAIN**: the current observation read by the probes, and ii) **DIFF**: the difference among the current PLAIN value and the previous one.

The aggregation of data coming from probes is managed by a CEP. Such module has been developed specifically to process huge amounts of data in near real-time, ultimately facilitating collection and aggregation of data from

different Target Machines, managing the different data flows in parallel. Further, it facilitates the combination of snapshots and fingerprints performed by the Monitor Aggregator, and their usage by the Anomaly Detection module.

Insights on Anomaly Detector

The Anomaly Detection module includes a set of **anomaly checkers**. An anomaly checker is assigned to a given data series, and evaluates if the current value of the selected data series is anomalous or expected following a set of given rules. More anomaly checkers can be created for the same data series: given a snapshot as input, each anomaly checker produces an anomaly score. Some of them are selected according to a given metric e.g., *F-Score(2)*, building the *selected anomaly checker* set. The individual outcomes of each selected anomaly checker is then combined to decide if an anomaly is suspected for the current snapshot. Consequently, an anomaly is raised only if this combined score mets or exceeds a given *anomaly treshold*.

5.3.2 *Methodology to Execute the Framework*

The methodology to exercise *MADneSs* is composed of two steps that can be repeated when major reconfigurations occur: *Training Phase* and *Runtime Execution*.

Training Phase

This phase is organized in two steps. In the first step, *fingerprints* of the services are obtained through the test invocation in Figure 15. Then, preliminary runs exercising the expected workload are executed, storing the retrieved data in the database. Preliminary runs are conducted by either i) observing the behavior of the system through functional tests, or ii) injecting anomalies in one of the SOA services, and observing the effects they generate on the monitored indicators. The service in which anomalies are injected may be a custom service devoted exclusively to testing, allowing to modify its source code. This strategy results particularly useful when it is not feasible to perform injections into the services exposed by the target system.

During this step, the services are not opened to users, which consequently are waiting until the SOA is available again. If the SOA is deployed for its first time, the deploy is completed only after step i) and ii) are completed. Once the SOA is available to users, it is expected that only few services will be updated each time, requiring only specific tests and consequently only short periods of unavailability. Moreover, to avoid bothering the user, preliminary runs can be exercised in low peak load periods such as at night or on mirror systems. Scalability and possible

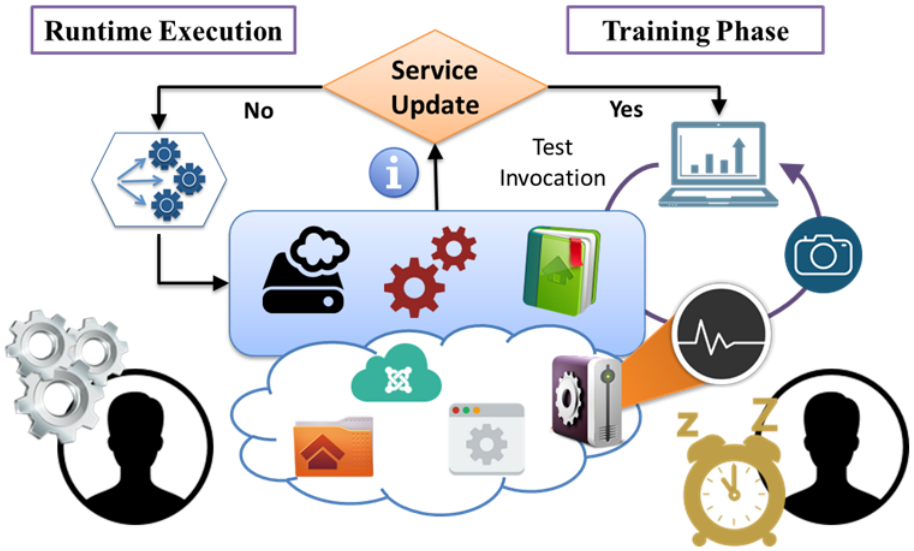


Figure 15: Methodology to exercise MADneSs

solutions to reduce training times are respectively explored in Section 5.5 and Section 5.2.6.

In the second step, services information and data extracted from preliminary runs are used by the anomaly detection module to train its parameters (CH.1), automatically choosing the more profitable selected anomaly checkers and the anomaly threshold.

Runtime Execution

The Monitor Aggregator merges each snapshot with the fingerprints representing contextual information, sending them to the anomaly detection module. Depending on the results of the analysis performed by the anomaly detection module, an anomaly alert is raised; for example, a mail notification is issued to an administrator. If a service update is detected during this phase, a new training phase is scheduled. The scheduling policy is strictly dependent on the characteristics of the system, and it is outside of the scope of the MADneSs engine.

5.3.3 *Implementation Details and Requirements*

The framework is meant to work with the aim of identifying any of the 6 anomalies we described in our model of anomalies in Section 4.2. However, different model of anomalies can be adopted, calling for novel studies on the effectiveness of *MADneSs* in identifying such specific kinds of anomalies.

To run the framework, it is required to work with two machines, that we labeled as Target and Detector Machines. *Target Machine(s)* identify virtual or physical machines which host a part of a complex system, or the complex system as a whole. We assume that the Target Machine(s) have a *Linux OS* and use a Java-based application server to deploy (web)services to allow our probing system to perform effectively. The architecture of *MADneSs* is meant to scale when monitoring more than a Target Machine, keeping the monitored data flows separate. The *Detector Machine* must be able to run a *Java* software based on the Complex Event Processor *Esper*, that is an open-source software based on Event Stream Processing techniques. In particular, the *Anomaly Detection module* is realized in a multithreading code that fetches the data from a *MySQL* database and instantiates the anomaly checkers on the data series provided by the *Monitor Aggregator*. The *test invocations* are generated through a sequence of *SOAP* services calls, which are executed through *Apache AXIS* primitives.

5.4 COMPLEXITY ANALYSIS

The time needed to select the best parameters' setup for each algorithm is linear to the number of runs used for training. Let us consider *te* as the number of training runs, while *O(SPS)* and *O(HIST)* represent the computational complexity to train a given checker respectively for SPS and HIST using data from a single run. It follows that the worst-case complexity to select the best configuration of a generic checker on a single experiment can be approximated as

$$O(te) * max\{O(SPS), O(HIST)\}$$

In other words, the complexity of the training phase is strictly related to the complexity of the heaviest anomaly detection algorithm implemented (SPS in our case).

5.5 SCALABILITY

Here we argument how our framework can scale when varying the resources of the Detector Machine, the number of connected Target Machines, and the monitored data.

5.5.1 *Impact of Training*

Once preliminary runs are executed, collected data is aggregated by the monitor and sent to the anomaly detector, who chooses the *selected anomaly checkers* and the most profitable *anomaly treshold*. This requires testing combinations of data series and detection algorithms. The cost of this operation has a linear

growth with respect to the number of data series and detection algorithms. In general, the higher is the number of data series to test, the higher is the time needed to complete the training. However, some algorithms may require short periods of training, since they either have a limited set of parameters to select or simpler strategies: this is the case of the historical checker (*HIST*). Instead, other algorithms, such as SPS, have more parameters that need to be instantiated.

5.5.2 *Impacts on Runtime Execution*

In Section 3.5.2 we analyzed the intrusiveness of system probes for OS and *JVM*, and the network load required to transfer monitored data to a remote Detector Machine. Although the analysis did not consider network probes and the data processing was much simpler than ours, main results are still valid. Moreover, in [21] we empirically demonstrated that we were able to aggregate, process and analyze data coming from 5 different Target Machines at the same time. This is adequate for the Secure! system as it is composed of four machines, while *jSeduite* can be distributed on a number of machines that can be defined depending on the specific installation.

Further, in Section 4.8.2 we computed the time elapsed from the observation of a snapshot on the Target Machine until its evaluation. This time is 32.10 ± 5.99 ms in our experiments. It is considered fully adequate for the *Secure!* and *jSeduite* systems, because it is significantly lower than the reaction time of the administrator that receives notifications.

6

BEYOND MADNESS

After presenting and discussing *MADneSs*, we expand here possible future implications of the strategies we listed in this Thesis.

6.1 MOVING TO SYSTEMS OF SYSTEMS

As remarked in [23], several definitions of *Systems-of-Systems* (SoS) have been proposed in the literature according to real-world applications in different areas, including dependability [73] and security [87]. According to [53], we consider that *an SoS is an integration of a finite number of Constituent Systems which are independent and operable, and which are networked together for a period of time to achieve a certain higher goal*. Constituent Systems (CSs) can either be existing legacy systems or newly developed components and they may include physical objects and humans: a CS is an autonomous subsystem of an SoS, consisting of computer systems and possibly of controlled objects and/or human role players that interact to provide a given service [70].

An SoS may have different degrees of control and coordination [39], identifying four categories, namely *directed*, *acknowledged*, *collaborative* and *virtual*. A *directed* SoS is managed by a central authority providing a clear objective to which each CS is subordinate; the CSs that form the SoS may operate independently, but they are subordinated to the central purpose. An *acknowledged* SoS has a clear objective but the CSs might be under their own control thus funding an authority in parallel with the SoS. In a *collaborative* SoS, the central management organization does not have coercive power and CSs act together to address shared common interests. Finally, a *virtual* SoS has no clear objective and its CSs do not even know one another. The degree of control and coordinated management of the CSs that form the SoS is relatively tight in a directed SoS, but it gets looser as we move to the acknowledged, collaborative and finally virtual category.

6.2 CHARACTERISTICS OF SOSS

The challenges posed to design, develop and maintain dependable and/or secure SoSs can be summarized as characteristics, or rather dimensions of analysis for such SoSs. In particular, we will expand here *architecture*, *dynamicity* and *evolution*, *emergence*, *governance*, *time*, *dependability* and *security*.

6.2.1 *Architecture*

The architecture of an SoS can be defined in terms of heterogeneous CSs interacting through cyber or physical channels. In [39], authors define an interface of a CS as the module where the services of a CS are offered to other CSs. This is called *Relied Upon Interface* (RUI). In particular, this interface among CSs is "relied upon" with respect to the SoS, since the service of the SoS as a whole (the *macro-level*) relies on the services provided by the respective CSs across the RUIs. Different instantiations of this RUI may exist [61], [39], depending i) on the kind of data exchanged, and ii) on the way the services are provided to other CSs. More in detail, *Relied Upon Message Interfaces* (RUMIs) establish the cyber data that are exchanged and the timing of message exchange, while *Relied Upon Physical Interfaces* (RUPIs) enable the physical exchange of things or energy among CSs. Lastly, *Relied upon Services* (RUSs) define how (part of) a CS is offered as a service through a RUI providing itself under a given Service Level Agreement (SLA).

Architectures of dependable and secure applications can be characterized as **mixed-criticality** architectures, where different parts of the system have different dependability and security requirements. To cope with this issue, in [16] authors propose architectural hybridization [97], where different subsets of requirements are satisfied in different parts of the target system.

6.2.2 *Evolution and Dynamicity*

Dynamicity and evolution are two important challenges of SoS and they have effects on security and dependability requirements. Dynamicity refers to short-term changes of the SoS e.g., in response to environmental variations or components failures. Evolution, instead, refers to long-term changes that are required to accomplish variations of the requirements in face of an ever-changing environment [53], [16].

6.2.3 *Emergence*

An emergent phenomenon manifests when CSs act together, and the emergent phenomenon is not observable by looking at single CSs separately. For instance, if a crowd enters a narrow alley then it alters its movements, individuals reduce their pace in order to avoid hitting or getting too close to others in front. This collaborative behavior does not emerge if we consider individuals separately: this means that an SoS is not just the sum of its CSs. Emergence can be expected or unexpected, detrimental or non-detrimental [62]. Non-detrimental are for example self-organization and evolution of biological systems, while detrimental are for example traffic jams due to the interaction of single cars. Moreover,

emergence can be expected or unexpected. In particular, detrimental unexpected emergent phenomena may expose vulnerabilities or lead to novel faults that are consequently difficult to tolerate [67].

6.2.4 *Governance*

Distributed ownership of individual components is a challenge for any complex system [71], which is usually an ensemble of existing systems, including third-party, OTS or more in general non-proprietary components. SoS governance is significantly more complicated and must change to accommodate the business requirements of an SoS.

6.2.5 *Time*

In a recent report from the GAO to the *US Congress* it is noted that a global notion of time is required in nearly all infrastructure SoSs, such as telecommunication, transportation, energy, etc. In large cyber-physical SoSs the availability of a global sparse time is fundamental to reduce the cognitive complexity to understand, design and implement SoS [62]. However, CSs typically use unreliable clocks. With respect to monitoring, this may result in inconsistent timestamps in observed data, leading to misunderstandings or wrong interpretations. It is thus relevant that CSs share a global view of time.

6.2.6 *Dependability and Security*

SoSs are composable systems, with a high degree of uncertainty on their boundaries. Since the environment may unpredictably change, or it may be so various becoming hard to model, the whole monitoring and assessment process can be negatively affected. Monitoring an SoS means to devise adaptive monitors that are able to cope with several environments and a variable number of interacting CSs.

6.3 A POSSIBLE CONTRIBUTION TO SOS LITERATURE

Summarizing, an SoS is not simply an ensemble of CSs: instead, CSs individually operating at a micro-level cooperate to provide new functionalities that emerge at a macro-level [62]. Critical SoS should avoid or mitigate **detrimental emerging phenomena** which can damage the whole system and the connected critical components. However, if unexpected, emerging phenomena cannot be easily avoided or mitigated through the rules that we set using our knowledge of the SoS. Considering the structure of the CSs, which includes physical objects and

humans, it appears that observing all the internals of CSs to check their behavior may be not possible. Thus, the monitoring effort should be directed to *Relied Upon Message Interfaces* (RUMIs) and *Relied Upon Physical Interfaces* (RUPIs).

All the issues above call for a monitoring solution that i) continuously observes the SoS to avoid or mitigate detrimental phenomena, ii) gathers data of RUMIs and RUPIs or internal data of CSs where possible, and iii) is able to infer the status of the properties of the macro-level looking only at data collected at micro-level. As the reader can notice, these aspects are mostly shared with the challenges and motivations behind *MADneSs* (see Section 3.9): consequently, we strongly believe that the main aspects discussed in this Thesis will be useful also to design anomaly detection solutions for SoSes.

6.4 BRINGING ANOMALY DETECTION INTO SOS DESIGN

According to the description of the viewpoints in Section 5.1.1, here we list potential design approaches that can bring SoS and anomaly detection together. In Table 19, for each viewpoint, we summarize the approaches for constructing an anomaly detection framework that can help adhering with the guidelines of a given viewpoint.

PURPOSE OF THE FRAMEWORK Building a framework that effectively uses anomaly detection for both dependability and security purposes can be a challenging goal. In fact, frameworks designed for intrusion detection are strongly dependent on the observation of network usage indicators. Further, malware oriented detection strategies should monitor OS attributes to understand if something is already damaging the system and maybe trying to steal or corrupt critical data from the hard drive. Regarding dependability monitoring, performance indicators observed in middleware e.g., thread number, cache usage and memory management, can reveal the manifestation of errors at application level that may escalate into failures in the near future. Regardless the chosen target, governance aspects play a decisive role in defining i) which CSs can be instrumented with probes, ii) the communication channels among them and iii) other general rules that could limit or support the effectiveness of the anomaly detection technique under consideration.

ANOMALIES Being at a high abstraction level when describing SoSs, it is difficult to define specific kinds of anomalies that could be targeted in such systems. Each CS acts indipendently, and may have its internal checks directed to either dependability or security properties. Possible internal errors in CSs may impact at the macro-level, resulting in more damaging failures at the higher abstraction level. However, interconnections among CSs except for RUMIs and RUPIs are not usually well-defined, and several relations may not be known unless they become evident after the manifestation of either positive or negative properties. It follows that anomaly detection turns out to be useful in checking

the so called *unknowns*, or the boundaries of the known system itself. Anomaly detection algorithms may analyze data related to interconnections and requests among CSs while they cooperate to reach the scope defined at the macro-level, potentially identifying anomalous behaviors that could impede to hit the target.

MONITORING AND DATA ANALYSIS APPROACHES Another key point is related to the architecture of the SoS, and mainly the characteristics, the roles and the ownerships of each CS and their interconnections. Monitored data must be labelled consistently in the whole SoS, since data acquisition through probes and monitors constitutes the basis for the anomaly detection process. This should include handling time issues that can lead to missed synchronizations or wrong timestamps assigned to each observation. As example, if the targeted SoS is under a (Distributed) Denial of Service attack, having an unsynchronized assignment of timestamps could lead to wrongly interpret anomalies in each threatened CS, without understanding the shared cause generating the anomalies.

In general, CSs can perform tasks with heterogeneous levels of criticality. It follows that depending on the criticality of each CS the monitoring and data analysis approach must change, adopting an architectural hybridization [97] that allows checking more carefully the CSs that are responsible for the most critical tasks. In particular, we can envision an hybrid monitoring approach which i) runs a centralized coordinator that collects and analyzes data coming from critical CSs, and ii) provides a set of parameters or rules for the anomaly detection algorithms that will be executed directly in the CSs that do not execute critical tasks. This allows monitoring critical CSs without burdening the centralized coordinator, since it does not need to analyze data observed on less critical CSs. This choice also impacts notification time (see Section 4.8.2).

We remark that this hybridization might be tailored depending on the category of the SoS. In directed and acknowledged SoSs, it is easier to identify common thresholds or trends because the objective is mostly shared among CSs. Instead, when CSs act together (collaborative SoS) and have limited knowledge of the other components of the SoS (virtual SoS), identifying shared rules for anomaly detection becomes very hard. In this context, the monitoring strategy must be distributed and customized as much as possible to suit the characteristics of CSs.

MONITORED INDICATORS As in *MADneSs*, the adoption of a multi-layer monitoring approach allows obtaining information about the state of the services (the macro-level from an SoS perspective) or the applications observing the underlying layers (SoS micro-level), without instrumenting the application or the service layer. The general idea is that when an application encounters a problem e.g., a crash in one of its functionalities, it generates an anomalous activity that can be observed looking at specific indicators of the underlying layers e.g., the number of active threads is abruptly decreasing. This solution is suitable even

when services changes frequently. The result is a monitoring solution coping with evolution and dynamicity of the targeted SoS, giving a widespread and adaptive support to the modules responsible for the dependability and security assessment.

ANOMALY DETECTION TECHNIQUE While a plethora of techniques for performing anomaly detection exist [25] in the literature, only a few of them can be considered suitable for anomaly detection in SoS. This is mainly due to i) evolution and dynamicity properties, which call for adaptive algorithms that can quickly reconfigure its parameters without needing of time-consuming testing phases, and ii) emergence, which can be unexpected, making techniques based on rules or on static pattern recognition less effective i.e., no rules or faulty patterns for unexpected phenomena are known. Consequently, the most suitable algorithms belong to the statistical and the online machine learning groups. In particular, statistical algorithms such as *SPS* work with a sliding window of past observations that are used to build a prediction. If the monitored value is not compliant with the predicted value, an anomaly is raised. Similarly, online machine learning techniques e.g., gradient-descend based [72], can build classifiers that change their behavior according to the evolution of the observed system, automatically tuning their main parameters. Emerging phenomena can be therefore detected because we assume that they cause the generation of values for specific parameters that are far from the nominal behavior.

SoS Viewpoint	Description of the Technique	Frameworks
Architecture	Consider Architectural Hybridization, i.e., link different CSs or blocks of CSs with a given level of safety that needs to be accomplished	CASPER [11] (Black Box)
Evolution and Dynamicity	Make Anomaly Detection able to tune its parameters when an evolution or a configuration change is detected. Algorithms and strategies for the detection of anomalies should work with poor knowledge of the history of the system e.g., online machine learning techniques, since this can change very often. Monitoring support needs to be adaptive as well.	[21], SEAD [73], SSC [87]
Emergence	Adopt models and libraries of anomalies targeting emerging behaviours, e.g., deadlock, livelock, unwanted synchronization	[22], ALERT [92], SSC [87]
Governance	Difficult to generalize. Communications must be fast enough to provide data observed by the probes to the monitor and to the anomaly detector, either if the approach is distributed or centralized.	-
Handling Time	Synchronize the clocks with an NTP server. The resulting clock precision is enough to label timestamps if real-time requirements are not intrinsic of the SoS.	CASPER [11] (Generic clock synchronization), [21] (NTP)
Dependability and Security	Build a Multi-Layer monitoring structure connected to adaptive Anomaly Detection modules	[21], SEAD [73], SSC [87]

Table 19: Tailoring Anomaly Detection on SoS Characteristics

7

CONCLUSIONS

Nowadays the society is characterized by an increasing dependency on *critical* computing systems whose failures can heavily impact the safety of human beings. Unfortunately, no existing technique is able to avoid a failure scenario: faults, which manifest as errors that may escalate into failures, are inevitable in larger and especially dynamic complex systems. The occurrence of such faults may stop or halt the execution of the whole system or even turn the system execution in the wrong direction. As a consequence, the past years have seen a growing interest in methods for designing, monitoring, controlling and assessing safety and security of such systems. These topics are becoming increasingly challenging as their size, complexity and interactions are steadily growing. For these reasons, the performance objectives are expanding from the traditional goal of achieving smooth network operation to having high levels of dependability, especially security, accuracy, quality, efficiency, reliability, and fault tolerance.

This thesis proposed design solutions to improve the dependability of highly dynamic complex systems and infrastructures. *Anomaly Detection* techniques are presented here as a way to ensure that the system behaviour complies with its functional specification in their final environment. Our approach to anomaly detection in complex dynamic systems was developed starting from the study in [18], where authors showed that their solution was very effective in detecting anomalies due to software faults in an *Air Traffic Management* (ATM) system. We then applied such promising approach to a prototype of the *Secure!* system, a *Crisis Management System* which is structured as a *Service Oriented Architecture* (SOA). We first observed that applying such strategy as it was described for non-dynamic systems does not provide comparable detection scores, meaning that some changes needed to be implemented to improve the performance of the strategy for dynamic systems. Changing the monitoring approach by adopting a *multi-layer* strategy did not help solving the problem, altrough it improved our partial scores. Then, we took advantage of *Context-awareness* to train the parameters of the *SPS* algorithm, tailoring them on the current context. Lastly, we performed dedicated tuning of parameters of the whole strategy, selecting i) adequate indicators sets, and ii) appropriate tresholds. Such experimental process ended up by listing some research challenges related to dynamic systems which negatively affect the efficacy of tradictional anomaly detection techniques. In a nutshell, we discussed i) an *adaptive notion of expected behavior*, ii) *avoiding interferences and minimizing detection overheads*, iii) the *monitoring strategy*, iv) the

suitablity of anomaly detection algorithms and, finally, v) the *selection of the indicators*.

After defining this stable version of our framework for experimental evaluations, we conducted an extensive experimental campaign on two SOAs: the *Secure!* Crisis Management System (CMS) and *jSeduite*, which are composed of multiple interacting services owned by different entities. The analysis included metric scores, performance, and sensitivity analyses, showing that our approach has potential for the considered target systems. Noticeably, such discussion escalated in some design guidelines to be followed in order to build an effective anomaly detection system for complex and dynamic systems.

The key aspects of our proposal were integrated in *MADneSs*, a novel framework for anomaly detection that copes with anomalies defined by a large anomaly model, includes an automatic tuning of its parameters, and manages complex relations among system indicators. MADneSs integrates a multilayer monitor, which allows shifting the observation perspective from the application layer - where services operate - to the underlying layers, and specifically we selected the operating system (*OS*), the middleware (*JVM*), and the network. In complex systems, the application or service layer is dynamic and may change frequently; instead, the underlying layers are subject to smaller updates, therefore a monitor can be instrumented there without requiring substantial maintenance. The detection algorithm was chosen accordingly, resulting in the adaptive *SPS* algorithm, which calculates an acceptability interval for a data instance depending on a sliding window od previous observations. Importantly, we showed how context-awareness helps clarify the boundaries between expected and anomalous behaviors, ultimately improving the detection accuracy.

In the last chapter of the Thesis, instead, we explored how our approach can contribute to create a mindset to error, malware or intrusion detection when dealing with *Systems-of-Systems*, a novel architectural paradigm which is gaining relevance in the recent years.With the support of a state of the art review, we first identify the design principles and the performance targets of a monitoring and anomaly detection framework. Then, we noticed that these aspects are mostly shared with the challenges and motivations behind *MADneSs*: consequently, we strongly believe that the main aspects discussed in this Thesis will be useful also to design anomaly detection solutions for SoSes.

BIBLIOGRAPHY

[1] Archive of support files. https://github.com/tommyippoz/Miscellaneous-Files/blob/master/20171026_Zoppi_Thesis_Archive.rar. Accessed: 2017-10-31. (Cited on pages 63, 67, 70, 71, 72, and 73.)

[2] Ganglia project. ganglia.sourceforge.net. Accessed: 2017-10-31. (Cited on page 89.)

[3] Nagios project. www.nagios.org. Accessed: 2017-10-31. (Cited on page 89.)

[4] Secure! project. http://secure.eng.it/. Accessed: 2017-10-31. (Cited on pages 18, 43, and 50.)

[5] Zenoss - own it. www.zenoss.com. Accessed: 2017-10-31. (Cited on page 89.)

[6] Russell J Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys (CSUR)*, 22(1):35–68, 1990. (Cited on page 32.)

[7] Naoki Abe, Bianca Zadrozny, and John Langford. Outlier detection by active learning. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 504–509. ACM, 2006. (Cited on page 33.)

[8] Mennatallah Amer, Markus Goldstein, and Slim Abdennadher. Enhancing one-class support vector machines for unsupervised anomaly detection. In *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*, pages 8–15. ACM, 2013. (Cited on page 37.)

[9] Leonardo Aniello, Claudio Ciccotelli, Marcello Cinque, Flavio Frattini, Leonardo Querzoni, and Stefano Russo. Automatic invariant selection for online anomaly detection. In *International Conference on Computer Safety, Reliability, and Security*, pages 172–183. Springer, 2016. (Cited on pages 41, 42, 59, 60, 61, 64, 84, 85, 86, 88, 89, and 93.)

[10] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004. (Cited on pages 15, 21, and 25.)

[11] Roberto Baldoni, Luca Montanari, and Marco Rizzuto. On-line failure prediction in safety-critical systems. *Future Generation Computer Systems*,

45:123–132, 2015. (Cited on pages 39, 41, 42, 59, 60, 61, 64, 84, 86, 88, and 107.)

[12] Yaneer Bar-Yam. General features of complex systems. (Cited on page 27.)

[13] Michèle Basseville, Igor V Nikiforov, et al. *Detection of abrupt changes: theory and application*, volume 104. Prentice Hall Englewood Cliffs, 1993. (Cited on page 44.)

[14] Andrea Bondavalli, Francesco Brancati, and Andrea Ceccarelli. Safe estimation of time uncertainty of local clocks. In *Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on*, pages 1–6. IEEE, 2009. (Cited on pages 18, 36, 42, 43, 56, and 92.)

[15] Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai, and Michele Vadursi. Foundations of measurement theory applied to the evaluation of dependability attributes. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 522–533. IEEE, 2007. (Cited on page 49.)

[16] Andrea Bondavalli, Andrea Ceccarelli, Paolo Lollini, Leonardo Montecchi, and Marco Mori. System-of-systems to support mobile safety critical applications: Open challenges and viable solutions. *IEEE Systems Journal*, 2016. (Cited on page 102.)

[17] Antonio Bovenzi, Francesco Brancati, Stefano Russo, and Andrea Bondavalli. A statistical anomaly-based algorithm for on-line fault detection in complex software critical systems. *Computer Safety, Reliability, and Security*, pages 128–142, 2011. (Cited on pages 35, 42, 43, and 56.)

[18] Antonio Bovenzi, Francesco Brancati, Stefano Russo, and Andrea Bondavalli. An os-level framework for anomaly detection in complex software systems. *IEEE Transactions on Dependable and Secure Computing*, 12(3):366–372, 2015. (Cited on pages 18, 40, 42, 43, 45, 46, 47, 49, 51, 54, 56, 57, 58, 59, 60, 61, 77, 88, 89, 92, and 109.)

[19] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000. (Cited on page 35.)

[20] Jean Paul Calvez and Olivier Pasquier. Performance assessment of embedded hw/sw systems. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*, pages 52–57. IEEE, 1995. (Cited on page 30.)

[21] A. Ceccarelli, T. Zoppi, A. Bondavalli, F. Duchi, and G. Vella. A testbed for evaluating anomaly detection monitors through fault injection. In *Proceedings - IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014*, pages 358–365, 2014. cited By 3. (Cited on pages 5, 51, 100, and 107.)

[22] A. Ceccarelli, T. Zoppi, M. Itria, and A. Bondavalli. A multi-layer anomaly detector for dynamic service-based systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9337:166–180, 2015. cited By 3. (Cited on pages 5, 51, 54, 58, and 107.)

[23] Andrea Ceccarelli, Andrea Bondavalli, Bernhard Froemel, Oliver Hoeftberger, and Hermann Kopetz. Basic concepts on systems of systems. In *Cyber-Physical Systems of Systems*, pages 1–39. Springer, 2016. (Cited on pages 17, 26, and 101.)

[24] Andrea Ceccarelli, Marco Vieira, and Andrea Bondavalli. A testing service for lifelong validation of dynamic soa. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 1–8. IEEE, 2011. (Cited on page 63.)

[25] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009. (Cited on pages 13, 17, 18, 32, 33, 37, 40, 60, 89, 94, and 106.)

[26] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 452–461. IEEE, 2008. (Cited on pages 17 and 27.)

[27] Alessandro Daidone. *Critical infrastructures: a conceptual framework for diagnosis, some applications and their quantitative analysis*. PhD thesis, PhD thesis, Università degli studi di Firenze (December 2009), 2010. (Cited on page 31.)

[28] Dipankar Dasgupta and Nivedita Sumi Majumdar. Anomaly detection in multidimensional data using negative selection algorithm. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1039–1044. IEEE, 2002. (Cited on page 33.)

[29] Sidney Dekker. *Drift into failure: From hunting broken components to understanding complex systems*. CRC Press, 2016. (Cited on pages 17 and 27.)

[30] C Delerce Mauris, L Palacin, S Martarello, S Mosser, and M Blay Fornarino. Plateforme seduite: une approche soa de la diffusion d'informations. *University of Nice, I3S CNRS, Sophia Antipolis, France*, 2009. (Cited on pages 18 and 63.)

[31] WR Deniston. Sipe: A tss/360 software measurement technique. In *Proceedings of the 1969 24th national conference*, pages 229–245. ACM, 1969. (Cited on page 30.)

[32] Elias P Duarte Jr, Roverli P Ziwich, and Luiz CP Albini. A survey of comparison-based system-level diagnosis. *ACM Computing Surveys (CSUR)*, 43(3):22, 2011. (Cited on pages 28 and 31.)

[33] William R Duncan. A guide to the project management body of knowledge. 1996. (Cited on page 22.)

[34] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005. (Cited on page 43.)

[35] Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. A geometric framework for unsupervised anomaly detection. In *Applications of data mining in computer security*, pages 77–101. Springer, 2002. (Cited on pages 40 and 89.)

[36] Lorenzo Falai and Andrea Bondavalli. Experimental evaluation of the qos of failure detectors on wide area network. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 624–633. IEEE, 2005. (Cited on page 32.)

[37] Massimo Ficco and Luigi Romano. A generic intrusion detection and diagnoser system based on complex event processing. In *Data Compression, Communications and Processing (CCP), 2011 First International Conference on*, pages 275–284. IEEE, 2011. (Cited on page 29.)

[38] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996. (Cited on page 33.)

[39] Bernhard Frömel and Hermann Kopetz. Interfaces in evolving cyber-physical systems-of-systems. In *Cyber-Physical Systems of Systems*, pages 40–72. Springer, 2016. (Cited on pages 101 and 102.)

[40] R Fryer. Low and non-intrusive software instrumentation: a survey of requirements and methods. In *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, volume 1, pages C22–1. IEEE, 1998. (Cited on pages 28 and 29.)

[41] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009. (Cited on pages 41, 64, 84, 86, and 88.)

[42] Ryohei Fujimaki, Takehisa Yairi, and Kazuo Machida. An approach to spacecraft anomaly detection problem using kernel feature space. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 401–410. ACM, 2005. (Cited on page 33.)

[43] Ann Q Gates, Steve Roach, Oscar Mondragon, and Nelly Delgado. Dynamics: Comprehensive support for run-time monitoring. *Electronic Notes in Theoretical Computer Science*, 55(2):164–180, 2001. (Cited on pages 28 and 30.)

[44] Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PloS one*, 11(4):e0152173, 2016. (Cited on page 35.)

[45] Jiexing Gu, Ziming Zheng, Zhiling Lan, John White, Eva Hocks, and Byung-Hoon Park. Dynamic meta-learning for failure prediction in large-scale systems: A case study. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 157–164. IEEE, 2008. (Cited on pages 94 and 95.)

[46] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 57–68. IEEE, 2015. (Cited on pages 40, 42, and 61.)

[47] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011. (Cited on page 36.)

[48] Stephen E Hansen and E Todd Atkins. Automated system monitoring and notification with swatch. In *LISA*, volume 93, pages 145–152, 1993. (Cited on page 31.)

[49] M Harelick and A Stoyen. Concepts from deadline non-intrusive monitoring. In *24th IFIP Workshop on Real-Time Programming, Saarland, Germany*, 1999. (Cited on pages 28, 29, and 30.)

[50] Guenther Hoffman and Miroslaw Malek. Call availability prediction in a telecommunication system: A data driven empirical approach. In *Reliable Distributed Systems, 2006. SRDS'06. 25th IEEE Symposium on*, pages 83–95. IEEE, 2006. (Cited on page 88.)

[51] Michael N Huhns and Munindar P Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet computing*, 9(1):75–81, 2005. (Cited on page 52.)

[52] Ivano Irrera, Joao Duraes, Marco Vieira, and Henrique Madeira. Towards identifying the best variables for failure prediction using injection of realistic software faults. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 3–10. IEEE, 2010. (Cited on page 88.)

[53] Mohammad Jamshidi. *System of systems engineering: innovations for the twenty-first century*, volume 58. John Wiley & Sons, 2011. (Cited on pages 101 and 102.)

[54] Mahesh V Joshi, Ramesh C Agarwal, and Vipin Kumar. Mining needle in a haystack: classifying rare classes via two-phase rule induction. *ACM SIGMOD Record*, 30(2):91–102, 2001. (Cited on page 33.)

[55] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 5(2):121–150, 1987. (Cited on page 87.)

[56] O. Jung, S. Bessler, A. Ceccarelli, T. Zoppi, A. Vasenev, L. Montoya, T. Clarke, and K. Chappell. Towards a collaborative framework to improve urban grid resilience. In *2016 IEEE International Energy Conference, ENERGYCON 2016*, 2016. cited By 3. (Cited on page 6.)

[57] Ashish Kamra, Evimaria Terzi, and Elisa Bertino. Detecting anomalous access patterns in relational databases. *The VLDB Journal The International Journal on Very Large Data Bases*, 17(5):1063–1077, 2008. (Cited on page 92.)

[58] Gunjan Khanna, Padma Varadharajan, and Saurabh Bagchi. Automated online monitoring of distributed applications through external monitors. *IEEE Transactions on Dependable and Secure Computing*, 3(2):115–129, 2006. (Cited on page 61.)

[59] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*, pages 72–79. IEEE, 1997. (Cited on page 24.)

[60] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011. (Cited on page 88.)

[61] Hermann Kopetz, Bernhard Frömel, and Oliver Höftberger. Direct versus stigmergic information flow in systems-of-systems. In *System of Systems*

Engineering Conference (SoSE), 2015 10th, pages 36–41. IEEE, 2015. (Cited on page 102.)

[62] Hermann Kopetz, Oliver Höftberger, Bernhard Frömel, Francesco Brancati, and Andrea Bondavalli. Towards an understanding of emergence in systems-of-systems. In *System of Systems Engineering Conference (SoSE), 2015 10th*, pages 214–219. IEEE, 2015. (Cited on pages 17, 102, and 103.)

[63] Rikard Laxhammar and Göran Falkman. Online learning and sequential anomaly detection in trajectories. *IEEE transactions on pattern analysis and machine intelligence*, 36(6):1158–1173, 2014. (Cited on page 94.)

[64] Joseph Lee Rodgers and W Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988. (Cited on page 58.)

[65] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004. (Cited on pages 28, 29, and 87.)

[66] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, 36(1):42–57, 2013. (Cited on pages 27, 39, and 43.)

[67] Jeffrey C Mogul. Emergent (mis) behavior vs. complex software systems. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 293–304. ACM, 2006. (Cited on pages 64, 85, and 103.)

[68] Aloysius K Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real Time Technology and Applications Symposium*, pages 252–262, 1997. (Cited on pages 28 and 30.)

[69] M. Mori, A. Ceccarelli, T. Zoppi, and A. Bondavalli. On the impact of emergent properties on sos security. In *2016 11th Systems of Systems Engineering Conference, SoSE 2016*, 2016. cited By 2. (Cited on page 5.)

[70] Marco Mori, Andrea Ceccarelli, Paolo Lollini, Bernhard Frömel, Francesco Brancati, and Andrea Bondavalli. Systems-of-systems modeling using a comprehensive viewpoint-based sysml profile. *Journal of Software: Evolution and Process*, 2017. (Cited on page 101.)

[71] Ed Morris, Pat Place, and Dennis Smith. System-of-systems governance: New patterns of thought. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2006. (Cited on page 103.)

[72] Srinivas Mukkamala, Guadalupe Janoski, and Andrew Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707. IEEE, 2002. (Cited on page 106.)

[73] Husanbir S Pannu, Jianguo Liu, and Song Fu. A self-evolving anomaly detection framework for developing highly dependable utility clouds. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 1605–1610. IEEE, 2012. (Cited on pages 41, 42, 59, 60, 61, 64, 84, 86, 101, and 107.)

[74] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Computer networks*, 53(6):864–881, 2009. (Cited on pages 41, 84, 86, and 88.)

[75] Bernhard Plattner and Juerg Nievergelt. Special feature: Monitoring program execution: A survey. *Computer*, 14(11):76–93, 1981. (Cited on page 28.)

[76] Stanislav Ponomarev and Travis Atkison. Industrial control system network intrusion detection by telemetry analysis. *IEEE Transactions on Dependable and Secure Computing*, 13(2):252–260, 2016. (Cited on pages 40, 42, and 61.)

[77] David Powell. Failure mode assumptions and assumption coverage. In *FTCS*, volume 92, pages 386–395, 1992. (Cited on page 24.)

[78] James E Prewett. Analyzing cluster log files using logsurfer. In *Proceedings of the 4th Annual Conference on Linux Clusters*. Citeseer, 2003. (Cited on page 31.)

[79] Sutharshan Rajasegarar, Christopher Leckie, Marimuthu Palaniswami, and James C Bezdek. Distributed anomaly detection in wireless sensor networks. In *Communication systems, 2006. ICCS 2006. 10th IEEE Singapore International Conference on*, pages 1–5. IEEE, 2006. (Cited on page 89.)

[80] Ragunathan Raj Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM, 2010. (Cited on page 17.)

[81] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *ACM Sigmod Record*, volume 29, pages 427–438. ACM, 2000. (Cited on page 35.)

[82] John P Rouillard. Real-time log file analysis using the simple event correlator (sec). In *LISA*, volume 4, pages 133–150, 2004. (Cited on page 31.)

[83] Felix Salfner and Miroslaw Malek. Using hidden semi-markov models for effective online failure prediction. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 161–174. IEEE, 2007. (Cited on pages 94 and 95.)

[84] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001. (Cited on page 37.)

[85] Erich Schubert, Alexander Koos, Tobias Emrich, Andreas Züfle, Klaus Arthur Schmid, and Arthur Zimek. A framework for clustering uncertain data. *Proceedings of the VLDB Endowment*, 8(12):1976–1979, 2015. (Cited on page 34.)

[86] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 144–155. IEEE, 2001. (Cited on pages 94 and 95.)

[87] Nathan Shone, Qi Shi, Madjid Merabti, and Kashif Kifayat. Misbehaviour monitoring on system-of-systems components. In *Risks and Security of Internet and Systems (CRiSIS), 2013 International Conference on*, pages 1–6. IEEE, 2013. (Cited on pages 41, 84, 86, 88, 89, 101, and 107.)

[88] Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In *Australasian Joint Conference on Artificial Intelligence*, pages 1015–1021. Springer, 2006. (Cited on page 44.)

[89] Vasilis A Sotiris, W Tse Peter, and Michael G Pecht. Anomaly detection through a bayesian support vector machine. *IEEE Transactions on Reliability*, 59(2):277–286, 2010. (Cited on page 89.)

[90] Hui Su and J David Neelin. Teleconnection mechanisms for tropical pacific descent anomalies during el nino. *Journal of the atmospheric sciences*, 59(18):2694–2712, 2002. (Cited on page 89.)

[91] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, volume 21, pages 2–9, 1991. (Cited on page 47.)

[92] Yongmin Tan, Xiaohui Gu, and Haixun Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 173–182. ACM, 2010. (Cited on pages 41, 42, 47, 51, 59, 60, 64, 84, 86, 88, and 107.)

[93] Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. Replay debugging of real-time systems using time machines. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003. (Cited on pages 29 and 30.)

[94] James P Theiler and D Michael Cai. Resampling approach for anomaly detection in multispectral images. In *AeroSense 2003*, pages 230–240. International Society for Optics and Photonics, 2003. (Cited on page 33.)

[95] Hong-Linh Truong and Schahram Dustdar. A survey on context-aware web service systems. *International Journal of Web Information Systems*, 5(1):5–31, 2009. (Cited on page 52.)

[96] A. Vasenev, D. Ionita, T. Zoppi, A. Ceccarelli, and R. Wieringa. Towards security requirements: Iconicity as a feature of an informal modeling language. In *CEUR Workshop Proceedings*, volume 1796, 2017. cited By 0. (Cited on page 6.)

[97] Paulo E Veríssimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006. (Cited on pages 102 and 105.)

[98] Valerio Vianello, Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patiño-Martínez, Rubén Torres, Rodrigo Díaz, and Elsa Prieto. A scalable siem correlation engine and its application to the olympic games it infrastructure. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 625–629. IEEE, 2013. (Cited on page 29.)

[99] Michael R Watson, Angelos K Marnerides, Andreas Mauthe, David Hutchison, et al. Malware detection in cloud computing infrastructures. *IEEE Transactions on Dependable and Secure Computing*, 13(2):192–205, 2016. (Cited on pages 42 and 61.)

[100] Conal Watterson and Donal Heffernan. Runtime verification and monitoring of embedded systems. *IET software*, 1(5):172–179, 2007. (Cited on pages 28, 29, and 30.)

[101] Andrew W Williams, Soila M Pertet, and Priya Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007. (Cited on pages 41, 42, 47, 51, 59, 60, 64, 84, 86, and 88.)

[102] Anthony D Wood, John A Stankovic, Gilles Virone, Leo Selavo, Zhimin He, Qiuhua Cao, Thao Doan, Yafeng Wu, Lei Fang, and Radu Stoleru. Context-aware wireless sensor networks for assisted living and residential monitoring. *IEEE network*, 22(4), 2008. (Cited on page 40.)

[103] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163–188, 2005. (Cited on page 87.)

[104] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. *ACM SIGPLAN Notices*, 49(4):687–700, 2014. (Cited on pages 17 and 27.)

[105] Yongguang Zhang and Wenke Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 275–283. ACM, 2000. (Cited on page 92.)

[106] Yu Zheng, Huichu Zhang, and Yong Yu. Detecting collective anomalies from multiple spatio-temporal datasets across different domains. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 2. ACM, 2015. (Cited on page 40.)

[107] T. Zoppi, A. Ceccarelli, and A. Bondavalli. Challenging anomaly detection in complex dynamic systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 213–214, 2016. cited By 0. (Cited on page 5.)

[108] T. Zoppi, A. Ceccarelli, and A. Bondavalli. Context-awareness to improve anomaly detection in dynamic service oriented architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9922 LNCS:145–158, 2016. cited By 0. (Cited on pages 5, 54, and 58.)

[109] T. Zoppi, A. Ceccarelli, P. Lollini, A. Bondavalli, F. Lo Piccolo, G. Giunta, and V. Morreale. Presenting the proper data to the crisis management operator: A relevance labelling strategy. In *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, volume 2016-March, pages 228–235, 2016. cited By 0. (Cited on page 6.)

[110] Tommaso Zoppi, Sandford Bessler, Andrea Ceccarelli, Edward Lambert, Eng Tseng Lau, and Alexandr Vasenev. A modeling framework to support resilient evolution planning of smart grids. In *Smart Grid Inspired Future Technologies*, pages 233–242. Springer, 2017. (Cited on page 7.)

[111] Tommaso Zoppi, Andrea Ceccarelli, and Andrea Bondavalli. Exploring anomaly detection in systems of systems. In *Proceedings of the Symposium on Applied Computing*, pages 1139–1146. ACM, 2017. (Cited on page 5.)

[112] Tommaso Zoppi, Andrea Ceccarelli, Francesco Lo Piccolo, Paolo Lollini, Gabriele Giunta, Vito Morreale, and Andrea Bondavalli. Labelling relevant

events to support the crisis management operator. *Journal of Software: Evolution and Process*, 2017. (Cited on page 6.)

[113] Tommaso Zoppi, Andrea Ceccarelli, and Marco Mori. A tool for evolutionary threat analysis of smart grids. In *Smart Grid Inspired Future Technologies*, pages 205–211. Springer, 2017. (Cited on page 6.)