<u>POLITECNICO DI TORINO</u>

DISTRIBUTED PROGRAMMING 2 - Year 2019 / 2020

SPECIAL PROJECT 2

OPTIMIZED FIREWALL ANOMALY RESOLUTION

Mohamed Mamdouh Tourab, s259371@studenti.polito.it

Referrals:
Fulvio Valenza,
Jaloliddin Yusupov,
Riccardo Sisto

Contents

1 Introduction

This report is related to *Distributed programming II* course. The purpose of the project is to:

- Design data representations (described by means of an XML schema) respectively of the firewall rules and anomalies.
- Design a REST API that can receive as input the rules using the previously defined data formats (XML or JSON) and that can perform the optimized analysis.

2 Background

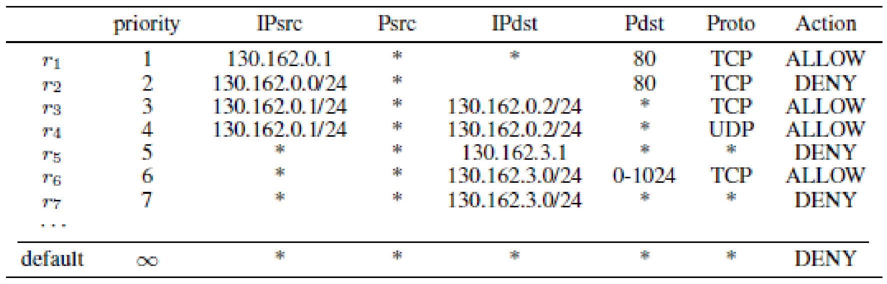
Firewalls have been widely used to protect not only small and local networks, but also large enterprise networks. The configuration of firewalls is mainly done manually by network administrators. Thus, it suffers from human errors. A typical example of error in firewall configurations is conflict among two firewalls rules, which arises when the effect of one rule is influenced or altered by another one, e.g. the actions of the two rules (that are both satisfied simultaneously) contradict each other. In literature, several solutions have been proposed for firewall conflict detection. However, a detected conflict has to be solved manually by administrators, and none of the proposed approaches even tries to minimize such necessary administration operations. The implementation of this service can be used after detecting the anomalies in the firewall by sending the list of rules and the set of detected anomalies and by following a certain algorithm in anomalies removal a new list of optimized rules i.e (does not contain any of the previously detected anomalies) is returned. A quick explanation for firewall rules and anomalies is in the following section.[2]

2.1 Firewall Rules

Firewalls operate by examining a data packet and performing a comparison with some predetermined logical rules. The logic is based on a set of guidelines programmed in by a firewall administrator, or created dynamically and based on outgoing requests for information. This logical set is most commonly referred to as firewall rules, rule base, or firewall logic. Most firewalls use packet header information to determine whether a specific packet should be allowed to pass through or should be dropped.An example for the firewall rules used in this project is shown in figure 1.[1]

2.2 Firewall Anomalies

Simply anomalies in firewall are any error in defining the rules may compromise the system security by letting unwanted traffic pass or blocking desired traffic. Manual definition of rules often results in a set that contains conflicting, redundant or overshadowed rules, resulting in anomalies in the policy. In this project anomalies are divided as shown in Figure 2. [1]



	priority	IPsrc	Psrc	IPdst	Pdst	Proto	Action
r_1	1	130.162.0.1	*	*	80	TCP	ALLOW
r_2	2	130.162.0.0/24	*		80	TCP	DENY
r_3	3	130.162.0.1/24	*	130.162.0.2/24	*	TCP	ALLOW
r_4	4	130.162.0.1/24	*	130.162.0.2/24	*	UDP	ALLOW
r_5	5	*	*	130.162.3.1	*	*	DENY
r_6	6	*	*	130.162.3.0/24	0-1024	TCP	ALLOW
r_7	7	*	*	130.162.3.0/24	*	*	DENY
...							
default	∞	*	*	*	*	*	DENY

Figure 1: Example of packet filter rule set .

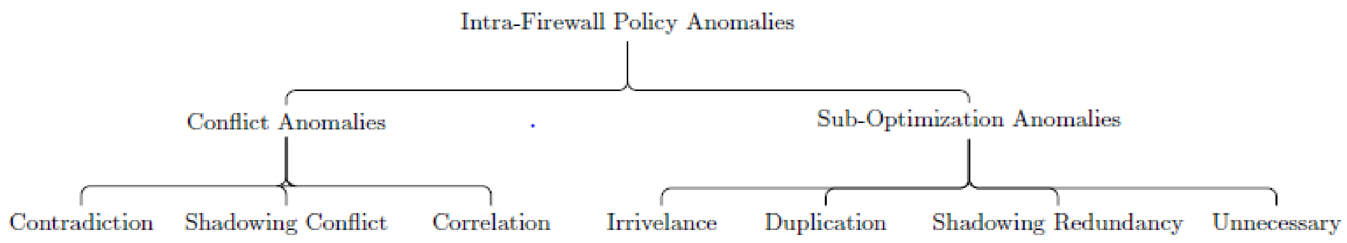


Figure 2: Classification of Intra-Firewall Policy Anomalies

A *sub-optimization* anomaly arises when redundant rules or other less efficient policy implementations are present. These types of anomalies can be solved automatically without any interference from the network administrator.[2]
A *conflict* anomaly arises when the effect of one rule is influenced or altered by another one. Typically a conflict occurs when a set of policies rules. This type of anomaly needs the (two or more) are simultaneously satisfied. This type of anomaly cannot be automatically resolved and an evaluation and action from the administrator is needed.[2]

3 Starting point

Firstly, I started understanding how firewall works, How does packet filtering takes place inside a firewall and how to compare two rules manually inside the rule set and understand whether these two rules are dependent, correlated or disjoint. Secondly, I started designing the XML schema for rule and anomaly.

4 Model Design

In this section I will explain all the schemes designed for this project. A Brief explanation for each schema file is provided below:

- firewall_rules.xsd: This schema file model the rules set inside firewall.
- conflict_schema.xsd: This schema file model the set of anomalies between the set of rules.
- webservice_input_schema.xsd: This schema file model the web service input and it contains set of rules and the detected anomalies inside this set of rules.
- contradict_solution.xsd: This schema file model how the network administrator could solve a contradiction anomaly.
- correlation_solution.xsd: This schema file model how the network administrator could solve a correlation anomaly.
- shadowing_conflict_solution.xsd: This schema file model how the network administrator could solve a shadowing conflict anomaly.
- solve_request.xsd: This schema file model collects the previous solution's schema into one xml file.

4.1 Rules Representation

Let's have a closer look on the firewall_rules.xsd schema. The following diagram shows the hierarchical representation of the rules inside firewalls.

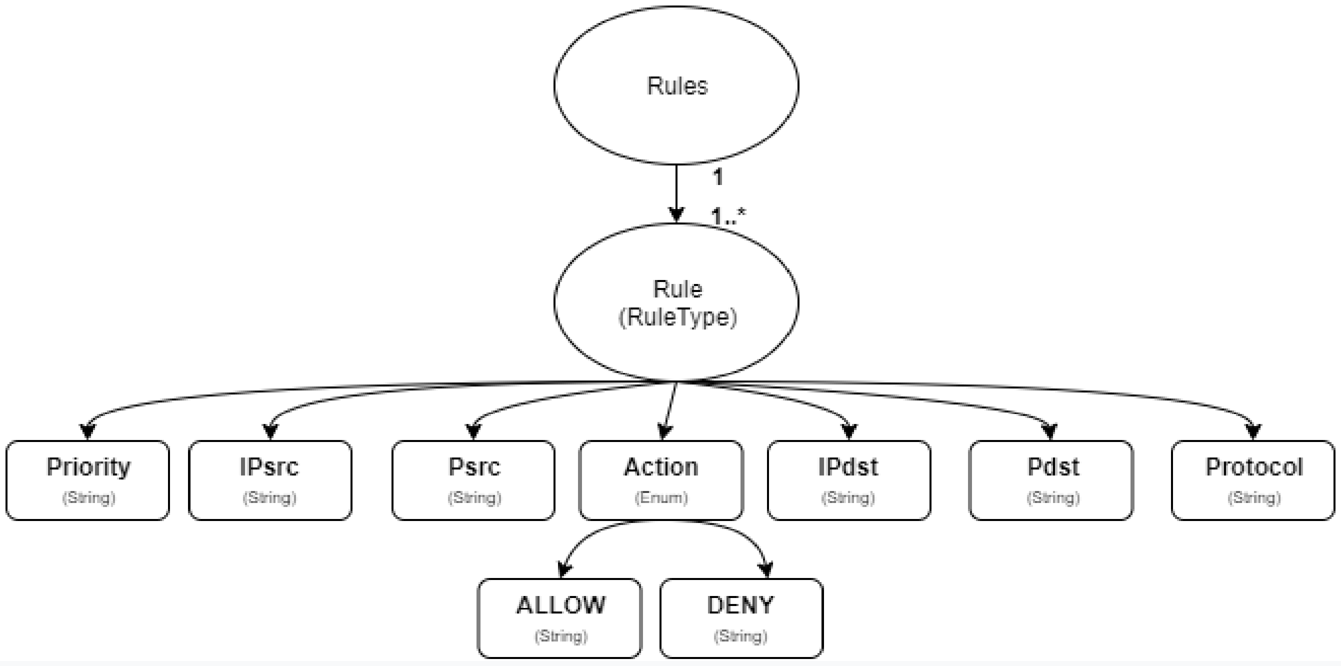


Figure 3: Rules Representation

4.2 Anomalies Representation

Let's have a closer look on the conflict_schema.xsd schema. The following diagram shows the hierarchical representation of the anomalies and how we can represent a single anomaly i.e (AnomalyType inside the implementation).

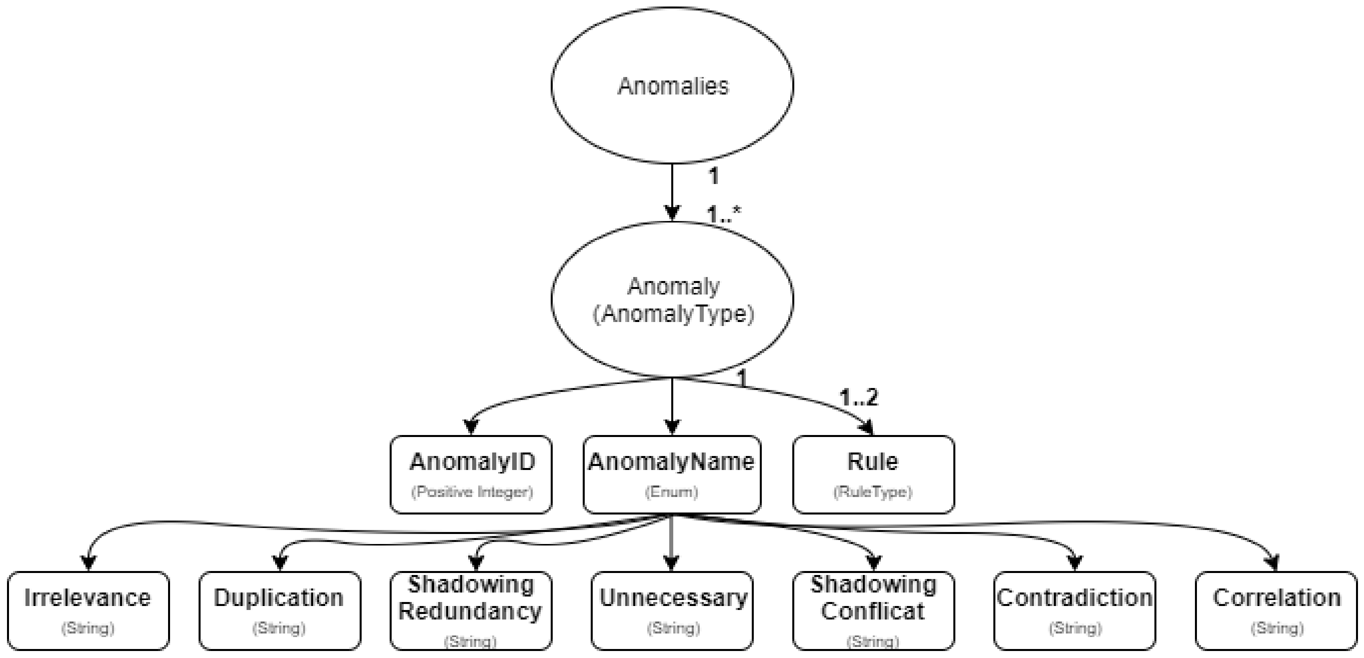


Figure 4: Anomalies Representation

4.3 Web Service Input Representation

Now it's time to see how we can contact our service using webservice_input_schema.xsd schema. The following diagram shows the hierarchical representation of the anomalies and how we can represent a single anomaly i.e (AnomalyType inside the implementation).This approach is used as input for the web service as a way of **reducing the interaction between the client and the web service as much as possible**, by creating such hierarchy all the input needed for our analysis is just provided in one POST request. I followed this approach based on the idea that a firewall rule set might contain up to 40.000 rules which is huge number if we used a finer granularity to create our input, e.g if we create each rule by using a separate POST request we will windup with a huge number of requests between the client and the service. Similarly for anomalies as the possibility of having bigger number of anomalies increase as the number of rules increase.

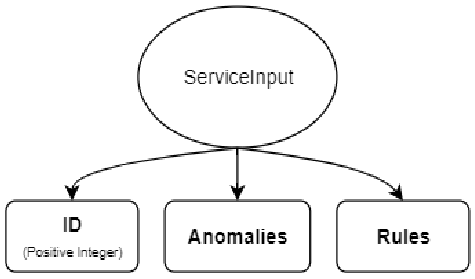


Figure 5: Web Service Input Representation

4.4 Contradiction Solution Representation

Now let's consider the possible solutions for the anomalies. This is just the overview of each solution more description of how we can use it will be discussed later. Firstly, we are going to start with the representation for *contradiction* anomaly solution.

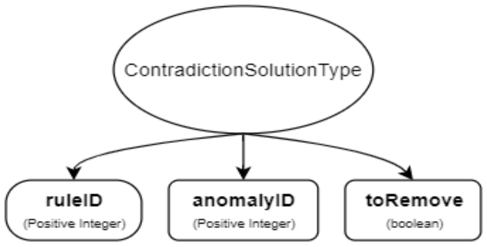


Figure 6: Contradiction Solution Representation

4.5 Correlation Solution Representation

Secondly, The *correlation* anomaly solution representation. Check Figure 7.

4.6 Shadowing Conflict Solution Representation

Lastly, The *Shadowing Conflict* anomaly solution representation. Check Figure 8.

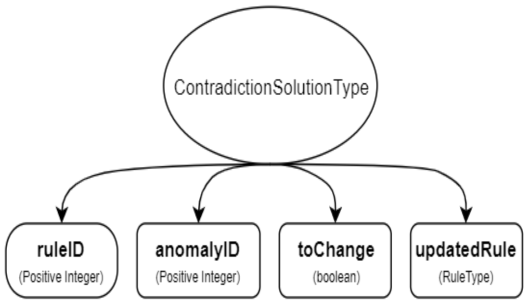


Figure 7: Correlation Solution Representation

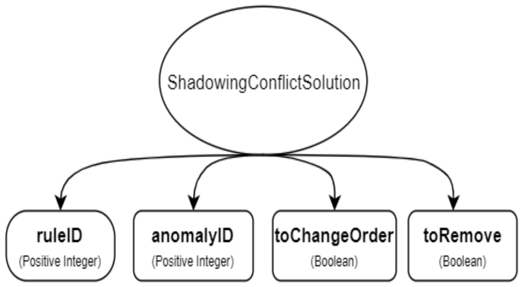


Figure 8: Shadowing Conflict Solution Representation

4.7 Solve Request Representation

Following the same approach as we did previously in the web service input. The same idea will be applied here, we will merge the previous solutions into one file, In that way we reduced enormous number of requests that could overwhelm the server. The three solutions i.e (contradiction, correlation,shadowing conflict) will be grouped together in a new schema called solveRequest which will be sent to the sever by the network administrator to solve these problems.

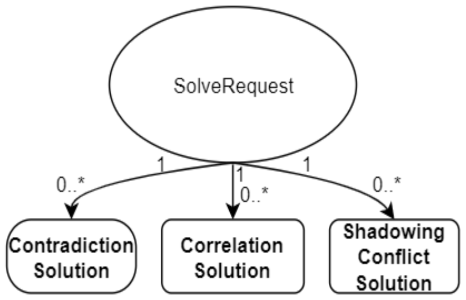


Figure 9: Solve Request Representation

5 RESTful web service

The developed **RESTful web service** allows you to store and retrieve information about the SeriveInput schema previously described. It permits to add, delete and modify its resources.

5.1 Schemes Details

Before we dive into the details of the API, we need to spend some more time on how to use the previous schemes.

5.1.1 ServiceInput Schema

As mentioned above this schema is specially designed to contain all the rule set and the accompanying anomalies. It is also worth mentioning for the second time that this schema will effectively reduce the number of POST request for our service. I will this XML file to demonstrate the correct way of using this schema. The main resource in our service is the ServiceInput, it's consists of three main elements inside (Rules,Anomalies) which are mandatory, each one of these element is simply a list of rules and anomalies, you can refer to the components of each rule and anomaly in Figure 3 & Figure 4. The third element is an ID for this input which is not filled by the network administrator but it's set by the database engine of the web service and it's returned to the requesting client so that it can access this resource later. In this way we achieve our goal to have a **HATEOAS** web service.

5.1.2 SolveRequest Schema

Now I will speak more about the solve request schema, This schema is used to edit the created ServiceInput resource by performing a PUT request on the resource.In this section I will also cover the other nested schemes i.e (Contradiction Solution,Correlation Solution,Shadowing Conflict solution). An example for how to use this schema is found here. By creating this file and performing a put request on a certain resource, you will execute the solution mentioned inside this file. After finishing this solution there shouldn't be any remaining anomalies and a final optimized list of the firewall rules is sent back to the network administrator. Let's have better look at the components of the SolveRequest:

- Contradiction Solution: This type of anomaly is resolved by deleting on of the two involved rules. So we need to specify which anomaly will be solved, set the toRemove flag to true and specify the rule ID that needed to be removed.
- Shadowing Conflict Solution: This type of anomaly has two possible solutions: In both cases we will set the anomalyID to the anomaly ID we are solving
  - Removing the second rule. In this case we will set the toChangeOrder to false and toRemove to true and specify the ruleID that will be removed
  - Put the second rule before the first rule. In this case we need to set toChangeOrder to true and toRemove to false and there's no need to set the rule ID.
- Correlation Solution: This type of solution also have two possible way to solve.

  - Leave Everything as it is. In this case you set the anomalyID to the ID of the desired anomaly and toChange to true.
  - Re-write the two rules. In this case you will need to create two correlationSolutions element. The first one you will set toChange to true and specify the anomaly ID and the rule ID of the first rule and insert the new rule. The same idea is also applied for the second element but enter the second rule ID and the new rule information.

5.2 Development

The RESTful has been developed both in Eclipse IDE for Enterprise Java Developers and IntelliJ IDEA, with the following features:

- Framework: Jersey 2.2 and JAXB
- Server: Tomcat v9.0.33
- Swagger API

The used library are available here.
In the README of the GitHub repository is described how to configure the environment for both *Eclipse* and *IntelliJ IDEA*. Source code is also available.

5.3 Methods

All method listed below are under the path:

- localhost:8080/**{project_name}_war_exploded/rest**. Incase the used IDE is Intellij
- localhost:8080/**{project_name}/rest**. Incase the used IDE is Eclipse

OFAR				
Method	Path	Description	Parameters	Response
GET	-	Read all the resources currently stored in database. However, not all the data will be returned. **Paging mechanism** is used, the user needs to specify the number of items per page and the required page	**QueryParam(”page”)** an integer that specifies the page number required, **QueryParam(”itemsPerPage”)** an integer that specifies the number of items per each page	200 OK and the ServiceInputs structures or 404 Not Found if the user specified a page that doesn’t exist.
GET	id	Read single resource.	**Query Param(”resolutionType”)** Possible values are (”solveIrrelevance” ,”solveDuplication” ,”solveShadowingRedundancy” ,”solveSub-optimization”) the default value is ”” which will return the data as it is without performing any action on the data	200 OK and the request ServiceInput resource or 404 not found if the request resource is not found.
POST	-	Create a new ServiceInput resource.	The body must contains the ServiceInput.XML file matching the schema.	201 Created and the created resource is returned or 400 Bad Request if the body of the request is not matching schema or there’s a validation error.
PUT	id	Apply solve request to a certain resource.	The body must contain SolveRequest.XML file that’s valid against the schema	200 OK if the PUT request executed successfully, 404 not found if the request resource is not found or 400 bad request if the Body doesn’t get validated against the schema .
DELETE	id	Delete a specific resource from the database.	-	200 OK if the request executed successfully and the deleted resources is returned 204 No Content if the item is not found.

6 Project Test

In order to have a robust web service various test have been carried out to ensure that all the units inside the algorithms is functioning properly. In addition, to ensure that the web service APIs are working as desired and robust enough for corner cases a test client is developed to test the service by performing random various operations to ensure that everything is working properly and final output is matching what's described inside the paper.

6.1 JUnit Test

6.1.1 Conflict Resolver Test Class

In this class (i.e *ConflictResolverTest.java*) a various functions that are implemented in the *ConflictResolver.java* class are being tested. The list of functions available inside the test class listed below:

- **testGetRules()** Check that getRules() correctly get the set of rules from the conflictResolver.
- **testGetAnomalies()** Check that getAnomalies() correctly get anomalies from the conflictResolver.
- **testResolveAnomalies()** Check that resolveAnomalies() correctly removes all sub-optimization anomalies and the rules involved without any interference from the network administrator.
- **testRemoveIrrelevanceAnomaly()** Check that removeIrrelevance() correctly remove irrelevance anomalies.
- **testRemoveUnnecessaryAnomaly()** Check that removeUnnecessaryAnomaly() correctly removed all the unnecessary anomalies and the involved rules. The data inside the paper contain 6 anomalies and 6 involved rules by applying the recommended solution the 6 rules should be removed and consequently the anomalies will be removed.
- **testUnnecessaryAnomalyChecker()** Check that the unnecessaryAnomalyChecker correctly capture the unnecessary anomalies between rules.
- **testRemoveDuplicationOrShadowingRedundancyAnomaly()** Check that removeDuplicationOrShadowingRedundancyAnomaly() correctly removes the desired type of anomalies based on the arguments sent to the function.
- **testGetConflictAnomalies()** Check that getConflictAnomalies() correctly return the list of conflict anomalies that needs to be solved by network admin.
- **testExecuteSolveRequest()** Check that executeSolveRequest is working properly and the solve request is applied correctly and all conflict anomalies are solved.
- **testAnomalyTypeToString()** Check that toString method inside the AnomalyType is working correctly.

6.1.2 Web Service Unit Test

Inside this class various operations are made to to unit test the functions available inside the Resource class (i.e *OptimizerResource.java*) by performing some operations e.g (POST,DELETE,PUT,GET). The functions that are implemented inside these class are:

- **wrong_post_request()** will add a wrong **ServiceInput** resource, the expected status code will be *400 Bad Request*.
- **correct_post_request()** will add a correct **ServiceInput** resource, the expected status code will be *201 Created*.
- **correct_put_request()** will edit a ServiceInput resource with a correct **SolveRequest**, the expected status code will be *200 OK*.
- **wrong_put_request()** will edit a ServiceInput resource with a wrong **SolveRequest**, the expected status code will be *400 Bad request*.
- **wrong_delete_request()** will perform delete on a wrong resource, the expected status code will be a *204 No Content*.
- **correct_delete_request()** will perform delete on an existing resource, the expected status code will be a *200 OK*.

6.2 Client Test

A client resource (i.e *ClientResource.java* is developed also to test the web service by performing various consecutive operations that are available inside *ServiceClient.java*. After performing these operations the client resource will return that the test is finished successfully, if an error occurred during one these tests an exception will be thrown and returned to the client. All of the above information related to the web service function and the client resource that's used for testing can be found inside the **swagger documentation** that is accessible after deploying the service and accessing the index page.

References

[1] Abedin M., Nessa S., Khan L., Thuraisingham B. (2006) Detection and Resolution of Anomalies in Firewall Policy Rules. In: Damiani E., Liu P. (eds) Data and Applications Security XX. DBSec 2006. Lecture Notes in Computer Science, vol 4127. Springer, Berlin, Heidelberg `https://doi.org/10.1007/11805588_2`

[2] FULVIO VALENZA, MANUEL CHEMINOD Dip. di Automatica e Informatica (DAUIN), Politecnico di Torino, Italy (e-mail:fulvio.valenza@polito.it). National Research Council of Italy (CNR–IEIIT), Corso Duca degli Abruzzi 24, I-10129 Torino, Italy(e-mail: manuel.cheminod@ieiit.cnr.it) Corresponding author: Fulvio Valenza (e-mail: fulvio.valenza@polito.it).