A new approach to deploy a self-adaptive distributed firewall

DA COSTA JÚNIOR, Edmilson P, DA SILVA, Carlos Eduardo <http://orcid.org/0000-0001-9608-439X>, PINHEIRO, Marcos and SAMPAIO, Silvio

Published version

DA COSTA JÚNIOR, Edmilson P, DA SILVA, Carlos Eduardo, PINHEIRO, Marcos and SAMPAIO, Silvio (2018). A new approach to deploy a self-adaptive distributed firewall. Journal of Internet Services and Applications, 9 (12).

Journal of Internet Services and Applications

RESEARCH Open Access

A new approach to deploy a self-adaptive distributed firewall

Edmilson P. da Costa Júnior[1], Carlos Eduardo da Silva[1*], Marcos Pinheiro[2] and Silvio Sampaio[1]

Abstract

Distributed firewall systems emerged with the proposal of protecting individual hosts against attacks originating from inside the network. In these systems, firewall rules are centrally created, then distributed and enforced on all servers that compose the firewall, restricting which services will be available. However, this approach lacks protection against software vulnerabilities that can make network services vulnerable to attacks, since firewalls usually do not scan application protocols. In this sense, from the discovery of any vulnerability until the publication and application of patches there is an exposure window that should be reduced. In this context, this article presents Self-Adaptive Distributed Firewall (SADF). Our approach is based on monitoring hosts and using a vulnerability assessment system to detect vulnerable services, integrated with components capable of deciding and applying firewall rules on affected hosts. In this way, SADF can respond to vulnerabilities discovered in these hosts, helping to mitigate the risk of exploiting the vulnerability. Our system was evaluated in the context of a simulated network environment, where the results achieved demonstrate its viability.

Keywords: Distributed firewall, Self-adaptive software, Network security, Software vulnerability assessment

1 Introduction

Several institutions all over the world deal with complex network infrastructure, involving an increasing number of equipment (e.g., switches, routers) and servers, usually providing different services. These environments may contain several types of vulnerabilities that could be exploited by an attacker. In this way, it is extremely important to maintain software systems up to date with versions that fix known vulnerabilities. Considering the diversity of activities and variety of research topics conducted throughout an university, it is common to find situations where several services, and servers, need to be provided for different groups of people, and more often than not, maintained by these groups. This leads to an inconsistency in management and security procedures, where servers poorly configured, with outdated services, or both may become potential targets for attacks.

In this context, the traditional approach for network security, in which firewalls are deployed on the border of the network is no longer effective, as centralized border firewalls are not able to deal with attacks originated from inside the security perimeter [1]. Today's technology movements, such as Bring Your Own Device (BYOD) and the availability of 3G/4G connections, mean that a malicious user has already penetrated the border defense. This is exacerbated when we consider university environments, which are usually open to the public in general, and contains some servers maintained by researchers, with outdated and potentially vulnerable services.

Distributed firewall systems [2] have emerged as a solution for dealing with incidents originated from inside the secure perimeter, by including firewalls in different points of the network and servers. In such systems, a centralized control mechanism is responsible for distributing firewall rules to each point of the network, and hence it is possible to control what services running on those servers are exposed on the network, and only for specific client hosts.

However, the application of distributed firewall also brings some challenges, such as the management of these firewalls and their rules, and the response time in case of an incident. Traditional solutions for intrusion detection

*Correspondence: kaduardo@imd.ufrn.br
[1] Digital Metropolis Institute, Federal University of Rio Grande do Norte (UFRN), Natal, RN, Brazil
Full list of author information is available at the end of the article

or vulnerability assessment usually notify an administrator, who then assesses and decides how to respond to deal with the situation [3]. However, this approach is usually not fast enough for avoiding information theft, the infection of new systems/servers, or even service unavailability, mainly for attacks conducted during strategic times, such as the middle of the night or weekends, when the IT team is usually out of service.

Moreover, it is possible to identify a gap on the integration between the several tools involved in securing a network environment. For example, a Vulnerability Assessment System (VAS) may detect a vulnerability on a particular server, but the firewall of such system may not react to such detection, as both systems are not integrated. Such problem becomes more evident when we consider the dynamic nature of a complex network environment, in which new devices and services are constantly added/removed, and new vulnerabilities are discovered and patched. Finally, it is worth mentioning that, although there are firewall solutions that inspect application protocols, in 2016 more than half of the corporate networks were still using conventional firewalls [4].

In this context, the main contribution of this paper is an architecture for network security based on self-protection, named Self-Adaptive Distributed Firewall (SADF). SADF integrates different components that are part of a network for supporting the autonomic management of its infrastructure in response to security-related incidents. SADF has been deployed on a prototype integrating a configuration management system with a VAS for managing a distributed firewall, in which possible threats can be detected (i.e., servers with vulnerabilities) and appropriate decisions be made for mitigating their impacts with minimal human intervention.

The motivation for using self-adaptation is the proven effectiveness and efficiency of self-adaptation in dealing with uncertainty in a wide range of applications, including those related to security [5–7]. Our current prototype of SADF monitors the hosts of a network, which are then scanned by a VAS in the search for known vulnerabilities. Once a vulnerability is discovered, firewall rules for reacting to it are defined based on high-level policies. These rules are then applied to individual hosts, effectively mitigating the exposure window for the vulnerable server.

SADF was first proposed in [8], in which the main issues related to the theme were presented, together with a proposal of architecture and a prototype implementation that demonstrated its viability. Compared to our previous work, this article further details the proposed architecture, which now fully implements a Monitor-Analyse-Plan-Execute-Knowledge (MAPE-K) feedback control loop [9] for managing the self-adaptation process. This article also presents details about high-level policies and the decision-making process used for controlling adaptation, and an evaluation of the whole implementation in a controlled environment.

The remaining of this paper is organized as follows: Section 2 contextualizes our work defining its scope and presenting some background on self-protection. Section 3 presents a conceptual view of the proposed SADF architecture. Section 4 describes a prototype that has been implemented to demonstrate our approach feasibility. Results from the experiments conducted in a controlled environment to evaluate the proposed approach are presented and discussed in section 5. Section 6 discusses some related work. Section 7 concludes the paper.

2 Contextualization

In a university network infrastructure, such as the Federal University of Rio Grande do Norte (UFRN) in Brazil, it is common to find several groups of servers hosting from basic services, like e-mail, Web, and DNS (Domain Name System), to specific applications. These institutions usually maintain a network team responsible for managing these services. However, it is also common to find other servers and equipment providing a set of local services used and maintained by researchers in their respective laboratories. Such a diverse environment is prone to inconsistency in management and security procedures, causing it to be likely susceptible to vulnerable servers due to misconfiguration or outdated software. Moreover, an university network contains some workstations and wireless access points, which together with the trend of BYOD and the availability of 3G/4G connectivity, constitute a plethora of equipment outside the control of the central management team.

The firewall is usually treated as the first line of defense of computer networks [10]. A firewall is a trusted host that acts as a choking point of one or more networks, usually at the border between a public and a private network. The traffic between the networks passes through the firewall, which decides based on a set of rules, which network packet should be allowed to continue or blocked. However, as previously mentioned, the limitations of a centralized firewall, which is not able to protect against internal attacks, has motivated the definition of a distributed firewall model [2]. In a distributed firewall, security policies are defined in a centralized fashion using specific language, and then distributed, by secure means, to be applied to different enforcement points. These enforcers can either be located on different segregation points inside the network, such as routers and switches, or on each host of the network [1]. Figure 1 presents a general view of a network infrastructure where we can identify a Rules Management Server, which is responsible for dealing with and distributing the firewall rules into the different enforcement points, such as switches, servers, or both.

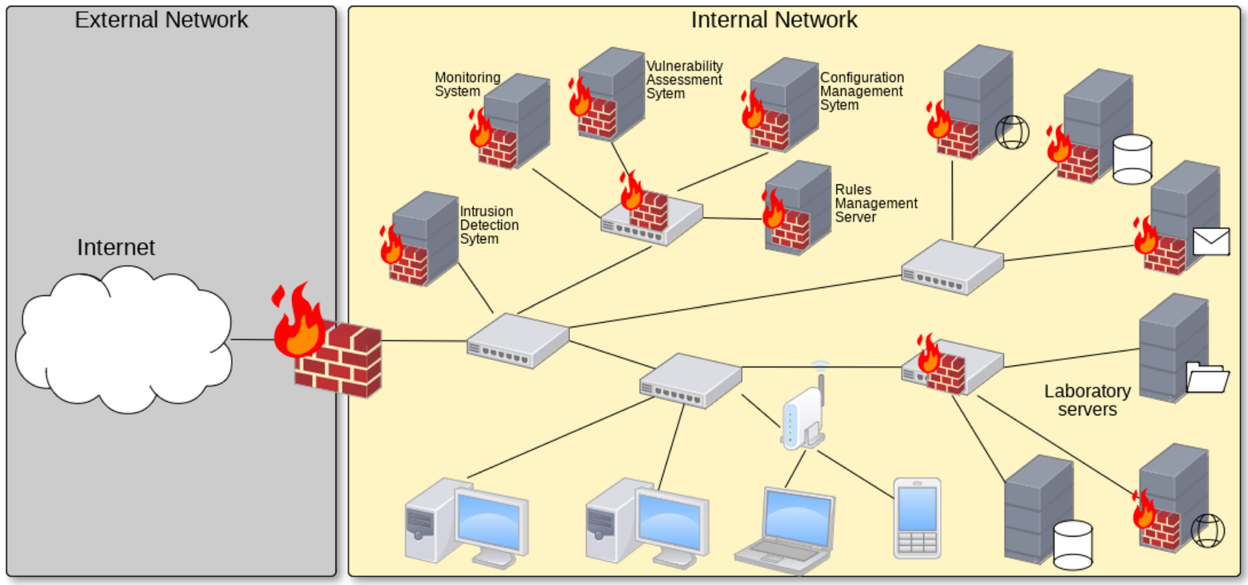


Fig. 1 General view of the distributed firewall scope considered in this paper

Operating and maintaining this infrastructure requires a continuous effort as, even though there are different tools for facilitating management and maintenance actions, it is common to find out that most of these operations are still manually conducted. For example, the majority of institutions use, apart from firewalls, some tool for configuration management, resource and service monitoring, intrusion detection and vulnerability assessment systems, which scans the network pointing out vulnerable services. These are important tools for maintaining the network infrastructure, but there is a lack of integration among them, requiring human intervention for conducting some tasks, and wasting valuable time between the moment an incident is detected, and an administrator performs some corrective action to mitigate its impact.

In this context, software self-adaptation can be used for integrating such tools, contributing for automating the security management of the network, with minimal human intervention.

A self-adaptive software system is able to modify its own structure or behaviour during run-time in order to deal with changes in its requirements, the environment in which it is deployed, or the system itself [9]. Among the different properties of a self-adaptive system, self-protection has been identified as a key concept for building autonomous self-managed systems. While systems' architectures are becoming more dynamic and adaptive, the majority of the protection mechanisms have kept simple, with security policies usually manually defined, in a slow and costly way.

One way for achieving self-adaptation is through the Monitor-Analyse-Plan-Execute-Knowledge (MAPE-K) feedback control loop over a target system [11]. In this way, a self-protection mechanism allows the protected system to monitor and analyze its resources to detect possible problems, being able to react accordingly to deal with the detected problem. This reaction depends on the type of monitoring and analysis technique being employed, type of incident and the type of system being protected, and can range from emergency system shutdown, deactivation of damaged module and replacement for a new instance, user or connection blocking, etc. [7].

Figure 2 presents a reference architecture for a system that implements self-protection. At a meta-level, we have a *protecting sub-system*, responsible for implementing the MAPE-K feedback control loop that protects the *protected sub-system* at the base level. The *protected sub-system* contains the system functionality associated with the main application logic and may incorporate different security mechanisms, such as access control and cryptography, and different execution environment such as Software Defined Networks with or without support for Network Functions Virtualization. The meta-level subsystem is responsible for detecting security-related incidents and for the decision-making associated with the use of the mechanisms available at the base-level [7]. The base-level sub-system runs over and interacts with, a domain, which can also be monitored for helping in the decision making of the MAPE-K at the meta-level.

Thinking along these lines, a SADF solution can be employed as a preventive mechanism for dealing with

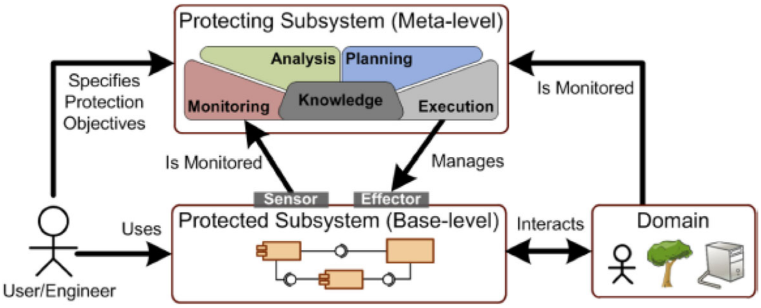


Fig. 2 Self-protection reference architecture [7]

well-known vulnerabilities. For example, whenever a particular server contains a vulnerability with a score greater than a pre-defined value, the firewall could be configured to only allow access to the server from clients in the same network.

3 Architecture for self-adaptive distributed firewall

Our solution for a Self-Adaptive Distributed Firewall (SADF) is built on top of the MAPE-K reference model as the means for logically structuring the different tasks involved in the management of the security aspects for network infrastructure, and for integrating the different tools usually involved in those tasks, allowing for their automation. Figure 3 presents a conceptual view of SADF architecture.

Each phase of the MAPE-K feedback control loop is implemented by an engine, which encapsulates the concrete components that allow for each engine functionality. To perform self-adaptation, the Monitor, Analyze, Plan and Execute engine components use different models that provide an abstraction of relevant aspects of the managed system, its environment, and the self-adaptation goals [12]. These models are maintained by a knowledge base represented in Fig. 3.

The *Monitoring engine* is responsible for collecting information about the different servers of the network infrastructure. This collection happens through *Sensor*

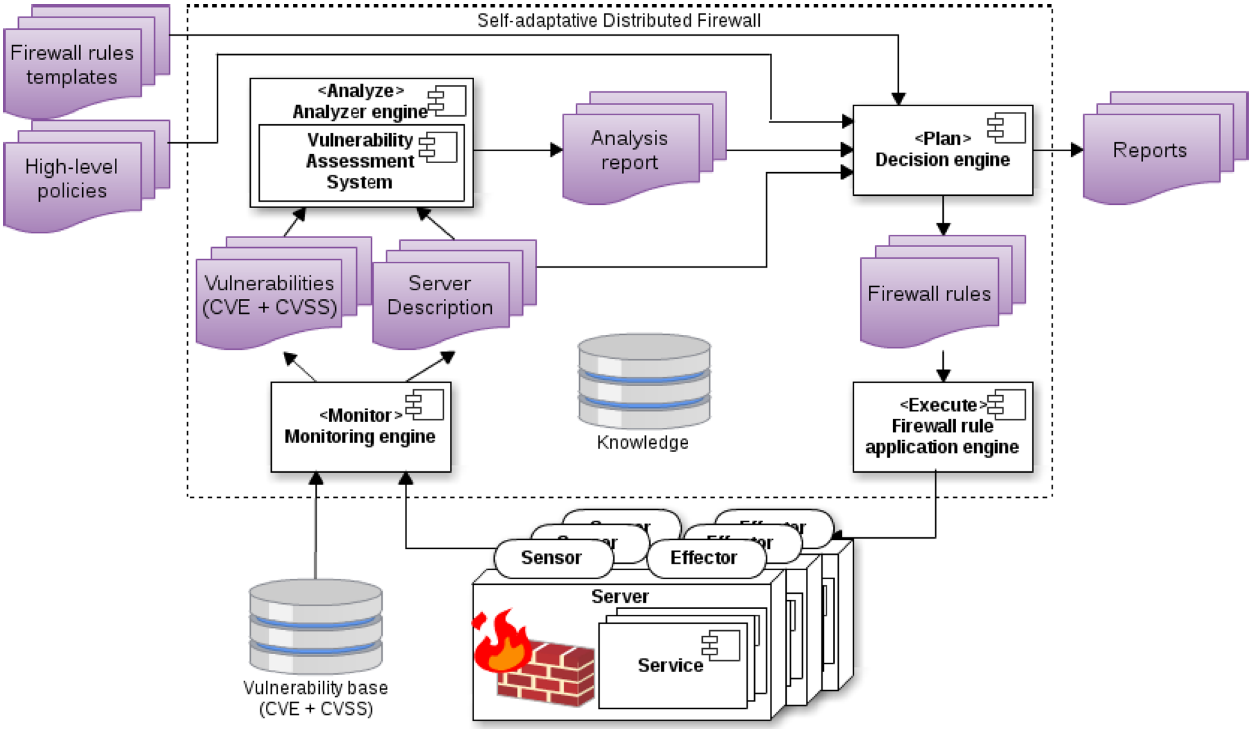

Self-adaptative Distributed Firewall
Firewall rules templates
High-level policies
<Analyze> Analyzer engine
Vulnerability Assessment System
Analysis report
<Plan> Decision engine
Reports
Vulnerabilities (CVE + CVSS)
Server Description
Firewall rules
Knowledge
<Monitor> Monitoring engine
<Execute> Firewall rule application engine
Sensor
Effector
Server
Service
Vulnerability base (CVE + CVSS)

Fig. 3 Conceptual architecture of the proposed solution

interfaces in each server. This data is represented by a *Server description* model, which is a format that can be manipulated and reasoned upon by the components of SADF. A service description model contains, among other information, details about the operating system, IP Address, services' names, versions and network port. Furthermore, it captures the firewall rules currently in effect on the server. The *Monitoring engine* is also responsible for obtaining *Vulnerabilities* descriptions from an external *Vulnerability base*. Vulnerabilities are represented through CVE[1], which defines a dictionary and standard representation format for vulnerabilities descriptions. These descriptions are published through the CVE List and maintained by different vulnerabilities databases (e.g., the NVD[2]). Vulnerabilities have an associated severity score calculated based on the CVSS[3], which defines metrics and formulas for deriving a vulnerability score, and a standard format representation.

The *Analyzer engine* relies on a Vulnerability Assessment System (VAS) to search for known vulnerabilities on the services currently running on the network. A VAS works by scanning the network and conducting different tests to find vulnerabilities in systems and servers, producing a vulnerability report for each server. Based on the server descriptions, the VAS can be employed with higher priority to scan known services running on each server, usually, when there are changes in the server descriptions or new vulnerabilities have been published. In the meantime, full vulnerability analysis of servers can still be performed. A *High-level policy* captures the requirements of the administrator, and together with the VAS report and the server description, is used for detection of policy violations. For example, servers with a vulnerability score greater than a particular threshold should only be accessible from machines in the same network, but the current firewall rules allow access from anywhere. All this data is used by the *Analyzer engine* component for producing *Analysis report*, which indicates, for example, servers with known vulnerabilities.

The *Decision engine* component is responsible for the plan phase of the MAPE-K loop. This component is responsible for making decisions on how to respond to the encountered situation based on the analysis report, the server descriptions, the high-level policies, and a set of *Firewall rules templates*. These templates provide a sort of parametrized firewall rules for different services, which can then be employed by the Decision engine for defining specific firewall rules to be applied. The creation of firewall rules must employ mechanisms for avoiding conflicts between rules. Besides creating firewall rules to be applied to the servers of the network, the *Decision engine* also produces a report intended for a human administrator.

At the execute phase we have a *Firewall rule application engine* component, which is responsible for effecting the new firewall rules on the servers. This component must take in consideration mechanisms for guaranteeing secure communication with each server, and configuration management techniques.

The MAPE-K reference architecture allows for the use of different mechanisms for each of its phases, which can be integrated in a number of different interaction patterns [13, 14]. Thinking along these lines, an SADF-based solution can be deployed with different components. In our particular architecture, we employed a configuration management system and a VAS with the roles of monitoring and analyzing the network infrastructure. These can be easily replaced by a metrics monitoring system, such as Zabbix, with threshold-based policies for identifying the necessity of adaptation, an IDS or any other analysis product that can detect abnormal situation on the network infrastructure. The decision-making process of our solution is based on high-level policies defined by an administrator, which must consider the execution environment to be controlled. One advantage of such approach is the separation of concerns between the functions of the MAPE-K loop. SADF controls a distributed firewall, but our architecture allows the management of more sophisticated environments such as Software Defined Networks with Network Function Virtualization capabilities when those are available. Since we have taken as a basis a real scenario, in which there is no support to SDN/NFV, we chose to focus on the control of a distributed firewall, acting as a proof-of-concept for deployment in such environment.

4 Instantiating the SADF

The proposed SADF architecture has been instantiated into a prototype implementation using a combination of existing open source and in-house developed components. This instantiation has been used to build a case study to demonstrate the feasibility of our approach. As a scenario for presenting its instantiation, in this paper we consider the protection of a Web server running the Apache HTTPD software and the JBoss Application Server.

In this section we present details about the different representation models employed in our instantiation, followed by a description of the developed prototype.

4.1 Model representation

One aspect that must be considered for a self-protection solution is the representation of the protected environment, such as servers, services, and firewall rules.

For representing servers and their deployed services we chose the representation language defined by the *Puppet*[4] configuration management tool. The *Puppet* language allows the description of servers, services, and configurations using a parametrized approach and well-defined semantics. *Puppet* configures systems in two main stages: compiling and applying a catalog. A catalog is a document that describes the desired system state. It lists all resources that need to be managed, as well as any dependencies between those resources. The core of the *Puppet* language is declaring resources. Groups of resources can be organized into classes, which are larger units of configuration. While a resource may describe a single file or package, a class may describe everything needed to configure an entire service or application.

An example of representation for a server and its services is shown in Listing 1. The node named *sadf-target.info.ufrn.br* was previously added to the *Puppet* ecosystem, which allows to describe and apply new configurations to this server. As shown in Listing 1, the node has two monitored services: the Apache HTTPD - a well-known open source HTTP server -, and the JBoss - an application server to the *Java Platform, Enterprise Edition (Java EE)*.

Listing 1 Example of a node description using the *Puppet* language

```
 1 node 'sadf-target.info.ufrn.br' {
 2
 3  include apache::mod::php
 4  apache::vhost { 'apache':
 5  port => '80',
 6  docroot => '/var/www/html',
 7  }
 8
 9  include jboss
10  jboss::default { 'jboss':
11  http_port => 8080,
12  ajp_port => 8009,
13  jmx_porp => 9090,
14  xms => 1024,
15  xmx => 1024,
16  }
17
18  include fw_sadf_target
19 }
```

Briefly, the settings on Listing 1 define that the Apache HTTPD server must run the PHP module (line 3), and create a VirtualHost listening to port 80 from *DocumentRoot /var/www/html* (lines 4 to 7). Regarding JBoss service, three ports must be configured: 8080 which is bound to the *HTTP connector*, 8009 to the *AJP connector*, and 9090 that works as a managing interface to the JMX (lines 11 to 13). Moreover, the parameters *xms* and *xmx* are used to determine the min and max memory size, respectively, to be allocated on the *HEAP* (lines 14 and 15). The class *fw_sadf_target* (line 18), which will be detailed later on, is applied to this node, defining a specific firewall rules for the *sadf-target.info.ufrn.br* node.

Similarly, it is necessary to represent firewall rules in a format that can be reasoned upon. For this purpose, we decided to employ the FLIP language [15, 16] for defining the firewall rules templates that SADF receives as input. In FLIP, firewall rules are defined using a high-level language that can be automatically translated into device-specific format. FLIP provides a well defined language with formal semantics, together with proven sound and complete algorithms for conflict resolution and translation into device specific firewall rules. Its formalism was one of the main reasons for choosing FLIP.

Listing 2 Example of firewall rules using the FLIP language

```
 1 domain sadf-target.info.ufrn.br =
       [10.3.128.12],
 2  local-network = [10.3.128.0/24],
 3
 4 service apache = tcp.[port=80],
 5  jboss = tcp.[port=8080, port=8009, port
       =9090],
 6
 7 policy_group vulnerable_jboss{
 8  incoming:
 9  apache {allow *}
10  jboss {deny * except local-network}
11 }
12
13 apply vulnerable_jboss on sadf-target.
       info.ufrn.br;
```

Listing 2 presents an example of a rule in FLIP. The first block (lines 1-2) defines the domains, which can be networks or hosts. We define a target server (*sadf-target.info.ufrn.br*) and a network (*local-network*) that represents the network in which the target is running. The second block (lines 4-5) of FLIP defines services, which may have one or more ports. In this example, Apache and JBoss were specified. In the sequence, we define a policy group (lines 7-11), which specifies the behavior that will be taken about services in a given scenario. One group was created that allows access to Apache HTTPD (line 9) and blocks access to JBoss (line 10) except when the connection comes from *sadf-engine.info.ufrn.br*. Finally, it is necessary to make a connection between the group and the protected domain (line 13).

Figure 4 presents a class diagram with the meta-model created for manipulating FLIP firewall rules as objects. The *Model* class represents the language model that consists of declarations. A *Declaration* is a generic block that represents each of the terms supported by FLIP. The *Domain* class describes a network or host, which has the attributes *name* and *address* to represent the name of the domain and the IP address. The *Service* class represents a service with names and ports. The *Policy_group* class defines the policy that has a name, traffic directions, which can be incoming or outgoing, policy-related

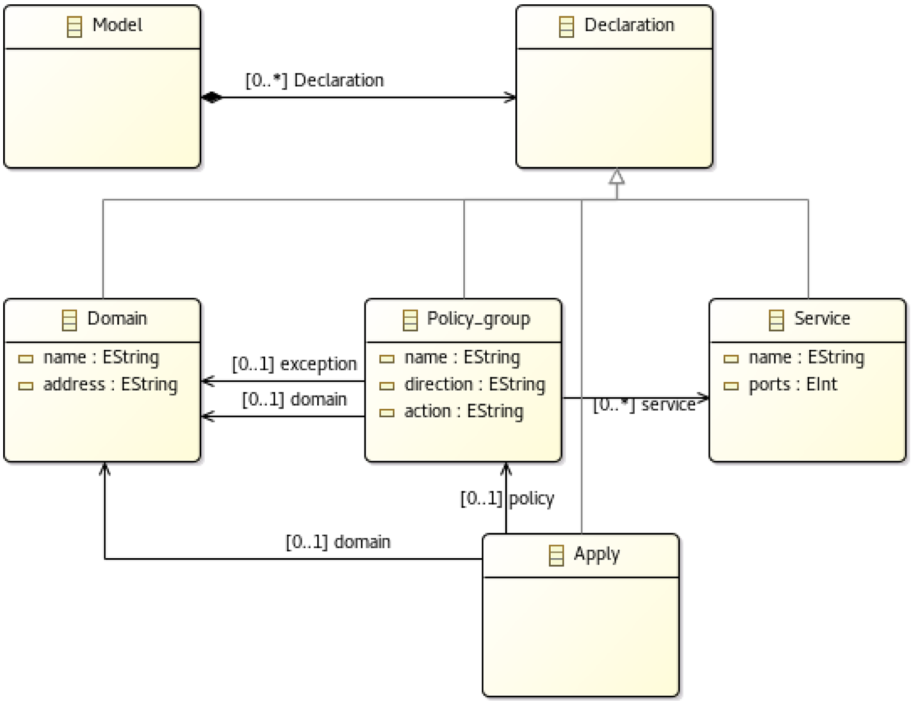


Fig. 4 Class diagram of FLIP implementation in Xtext

services and domains that can be inserted into the policy as the target domain of the action or exception to the target group. Finally, the Apply class assigns policies to domains, causing their traffic to meet the conditions described in the FLIP policy.

In order to apply firewall rules, FLIP models need to be translated into *Concrete firewall rules* . Since we employ the *Puppet* language for describing and managing servers and services, we employed *Puppet*'s firewall module[5] for representing concrete firewall rules, and hence the firewall rules can be applied by *Puppet* agents. To achieve that, FLIP rules expressed in text files are instantiated into Java objects, which are then used to create *Puppet* classes. During the translation process, some fields are extracted from the FLIP rules' fields and then written as a *Puppet* class describing each host and its services. This translation has been implemented as a Java class that handles the FLIP Model and makes it possible to obtain its respective rule declarations as domains, services, policy_groups and actions to perform the assembly of *Puppet*'s classes. Listing 3 presents an example of the class that specifies the firewall rules to the *sadf-target.info.ufrn.br* node, which have been generated from the example presented in Listing 2. The first two rules (lines 3-12) are included by SADF to guarantee access from its components to the targeted host. The following rules (lines 13-30) blocks access to JBoss ports (8009, 8080, 9090) except when the connection comes from the network 10.3.128.0/24.

Listing 3 Firewall rule created by using *Puppet*

```
 1 class fw_sadf_target {
 2  include firewall_module
 3  firewall {’001 accept connections from
       puppet master’:
 4  proto => ’all’,
 5  source => ’10.3.225.163’,
 6  action => ’accept’,
 7  }
 8  firewall {’002 accept connections from
       sadf-scanner’:
 9  proto => ’all’,
10  source => ’10.3.227.77’,
11  action => ’accept’,
12  }
13  firewall {’100 deny access to port
       8009’:
14  dport => ’8009’,
15  proto => tcp,
16  action => drop,
17  source => !10.3.128.0/24,
18  }
19  firewall {’101 deny access to port
       8080’:
20  dport => ’8080’,
21  proto => tcp,
22  action => drop,
23  source => !10.3.128.0/24,
24  }
25  firewall {’102 deny access to port
       9090’:
26  dport => ’9090’,
27  proto => tcp,
28  action => drop,
29  source => !10.3.128.0/24,
30  }
31 }
```

We employ high-level policies to define how to respond to found vulnerabilities in the monitored environment. These policies are based on Event-Condition-Action (ECA) rules. The basics of an ECA rule imply that whenever an event occurs, a predefined condition is evaluated, which trigger a specific action when the condition is true. The event may be represented by a complex structure involving a number of sub-events. Similarly to the event, the condition may be formed by several sub-conditions that must be evaluated under a specific logic. Finally, an action may be a composition of actions according to the condition and event that activated it [17].

In our approach, an event is a discovered vulnerability, while the condition captures the context of this vulnerability in terms of its severity and other information that can be used for decision making on how to respond, i.e., the action.

The syntax used to define a policy is given by the set of fields *{server, service, port, CVSS, CVE, action}* that can be mapped to four scopes: target, score, vulnerability, and execution, as described in Table 1. The target scope defines the policy application range and is formed by the fields *server*, *service*, and *port*. The score scope - given by the *CVSS* field - specifies the CVSS threshold value, hence addressing vulnerabilities with CVSS equal or higher than this value. In contrast, the vulnerability scope - given by the *CVE* field - directly addresses the CVE, allowing to act only for specific vulnerabilities. The score and vulnerability scopes may be optional, i.e., as long as one is provided, the other may be omitted.

The target, score and vulnerability scopes comprise the condition of our policy, while the action scope defines how to respond when the specified condition is evaluated to true. Action in our policy refers to a firewall rule template, which must be provided together with the policy. In this way, the proposed system can dynamically select the appropriate parametric rule template for the detected situation, which is then populated based on the information about the affected host description.

4.2 Prototype implementation

In this section, we describe our prototype implementation, whose concrete architecture is illustrated in Fig. 5. Implementation and functioning details for each developed module are also presented. Moreover, the iterations between the modules and the tools/models employed in the solution are discussed.

As previously mentioned, we employ the *Puppet* configuration management language for describing servers' configurations. *Puppet* provides tools for applying configurations, and for obtaining the current status of a host. A *Puppet agent* component runs on each host, and reports to (and receive commands from) the *puppet-master* component, which stores servers description into the *Puppet* catalog. Hence, *Puppet* agents fulfill the roles of sensor and effector of servers, while the Puppet master is responsible for the monitor and execute phases of the MAPE-K.

4.2.1 Monitor engine

The function of this module is to collect information regarding all *hosts* members and so creating the *Server description* which is used by SADF to represent the servers and its services. All *hosts* managed by the *Puppet* are taken as members of our scheme. The *Puppet* collects information regarding the servers by pooling agents (see Fig. 5) - that plays the role of a *sensor* on SADF architecture - and store it in its *Catalog*. Thus, the monitoring module can gather information about all servers by communicating to the *PuppetDB*[6] - a *Data Warehouse* that stores information and allows the access to it through a specific API[7].

The UML sequence diagram in Fig. 6 illustrates the interaction between the Monitor engine and the *PuppetDB*. The *Monitor engine* first creates an empty list of descriptions (call 1), then it contacts the *PuppetDB* to collect the list of managed nodes (call 2), and finally, it starts a loop to recover the list of services to each node (call 3). The *Monitor engine* maintains a list of servers previously collected to detect when there is a change in one of the servers. Thus, once all servers' descriptions have been obtained, the *Monitor engine* performs internal processing

Table 1 High-level policy's fields description

Field	Description	Mandatory	Input syntax
Server	Target server for the policy	Yes	Hostname, network range or just 'any' to represent any server.
Service	Target service for the policy	Yes	Server name or 'any' to represent any service.
Port	Target port for the policy	Yes	List of integer numbers or 'any' to represent any port.
CVSS	Score's threshold value	No	A decimal value between 0.0 and 10.0
CVE	Vulnerability ID.	No	List of CVEs.
Action	Action to be executed for the policy.	Yes	List of supported action names.

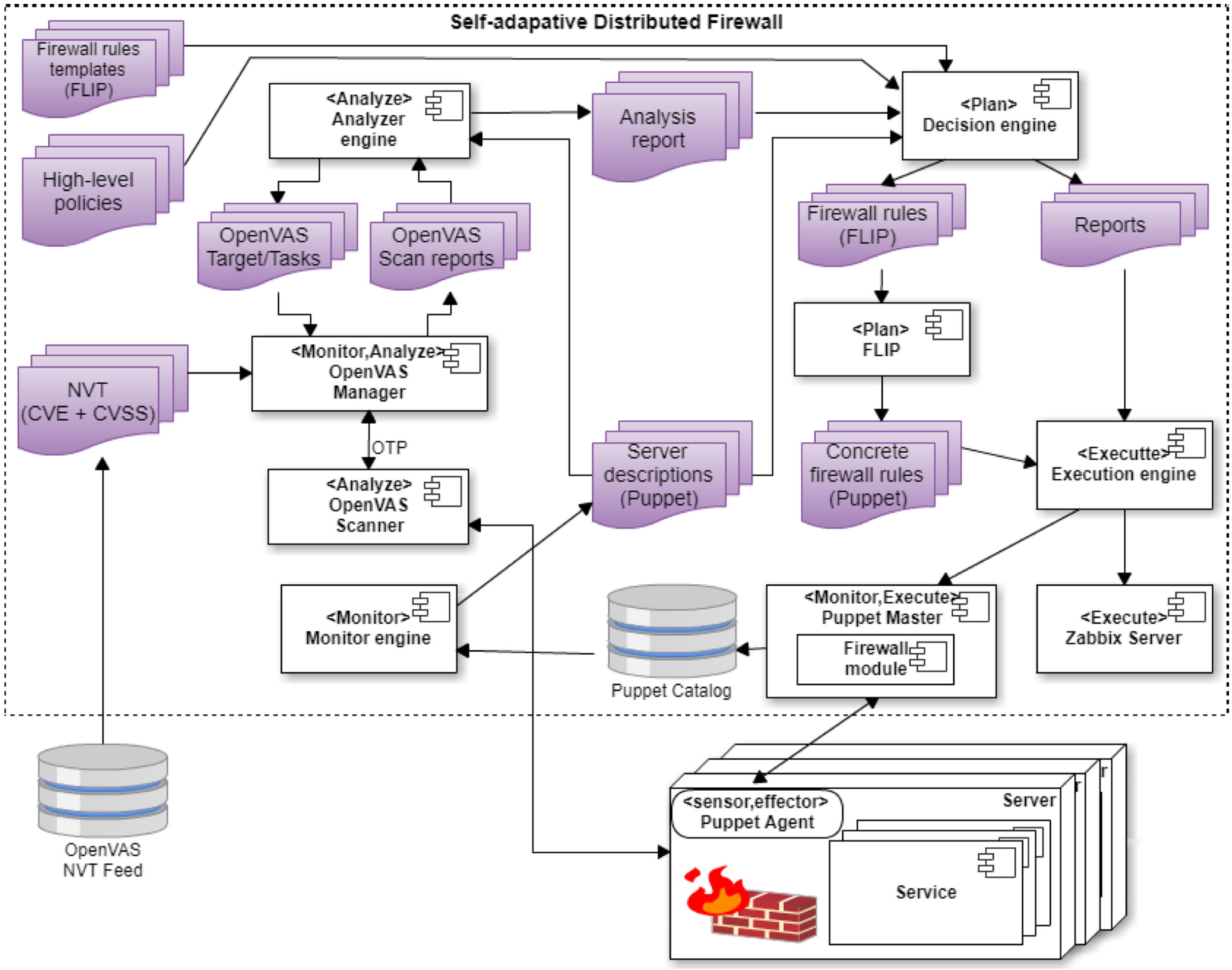

Self-adapative Distributed Firewall
Firewall rules templates (FLIP)
High-level policies
<Analyze> Analyzer engine
Analysis report
<Plan> Decision engine
OpenVAS Target/Tasks
OpenVAS Scan reports
Firewall rules (FLIP)
Reports
<Monitor,Analyze> OpenVAS Manager
<Plan> FLIP
NVT (CVE + CVSS)
OTP
Server descriptions (Puppet)
Concrete firewall rules (Puppet)
<Executte> Execution engine
<Analyze> OpenVAS Scanner
<Monitor> Monitor engine
Puppet Catalog
<Monitor,Execute> Puppet Master
Firewall module
<Execute> Zabbix Server
OpenVAS NVT Feed
<sensor,effector> Puppet Agent
Server
Service

Fig. 5 Architectural view of the prototype implementation

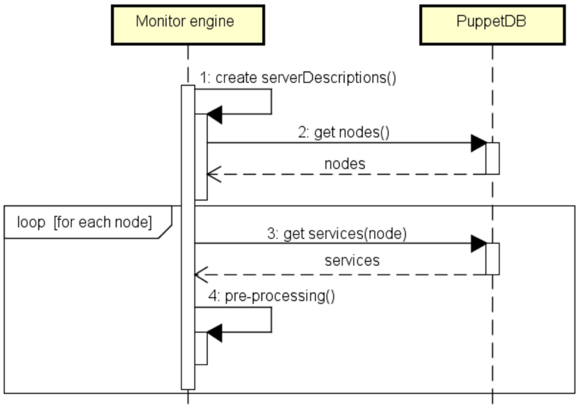


Fig. 6 UML sequence diagram for the *Monitor engine*

(call 4) in order to identify servers that need to be scanned again due to changes in their configuration.

It is important to mention that the description files definition for servers in *Puppet* is out of the scope of this work. So there is an assumption that this task was previously performed by the network administrator team. Thereby the focus of this work is to define and apply the firewall rules according to the specified configuration, as well as targeting well-known vulnerabilities on servers.

4.2.2 *Analyzer engine*

Considering that a list of all services running on each server is known, a vulnerability evaluation may be used as the trigger to an adaptation. The OpenVAS[8] is used to scan the servers. It provides an open source solution to identify, quantify and prioritize the vulnerabilities. These properties were explored by the *Analyzer engine*. In our solution, the *OpenVAS Scanner* conducts Network Vulnerability Tests (NVT)[9] for all hosts. An NVT is formed by a script for vulnerability detection, and a specification of the CVE and CVSS for each vulnerability detected. *OpenVAS Manager* gets NVTs through the *OpenVAS NVT Feed*, which is developed based on the CVEs from the NVD.

The *OpenVAS Manager* is responsible for controlling the *OpenVAS Scanner* actions, by providing its inputs and processing its outputs. All the interactions between the *Manager* and the *Scanner* are performed through the *OpenVAS Transfer Protocol* (OTP). OTP supplies all the resources to control the scan's executions. The *Manager* also includes the OpenVAS Management Protocol (OMP), an XML-based stateless API, which may be used to interact with and control the *OpenVAS Manager*.

The *Analyzer engine* configures and runs the OpenVAS scans by using the server description. This interaction is allowed by the OMP, which receives an XML entry describing the target host and the scan tasks. A detailed view of the interaction between the *Analyzer engine* and the *OpenVAS* is presented in Fig. 7. The *Analyzer engine* obtains the *Server descriptions* of those hosts that need to be scanned (call 1) and builds scan tasks for each server (see calls 2 and 3 in the figure). Each target defines a host, listing the ports for all detected services, and thereby all services managed by *Puppet* in each host can be scanned. A *task* is a scan configuration that defines how the target will be tested.

The *Analyzer engine* starts each scan (call 4 of the UML sequence diagram) and stores its ID. A scan may take several minutes to complete, as the current *OpenVAS* database includes more than 50,000 entries. Besides, the scan duration may vary according to the number of services on the target and the processing load on the *Scanner* and the Server being scanned. Currently, the VAS is queried every five seconds to check whether the scan has finished, at which point the *Analyzer engine* uses the stored ID as a key to recover its report (call 5). Each report from the *OpenVAS* includes, among other data: the CVE, the CVSS, the host, the port bound to the affected service, the protocol, the suggested solution type, and the affected software name. The *Analyzer engine* process the received report (call 6) to check whether a vulnerability has been detected, or a vulnerability previously detected is no longer present. When a vulnerability is detected the *Decision engine* is activated to handle it (call 7).

4.2.3 Decision engine

Once activated, the *Decision engine* must decide how to properly respond to the detected vulnerability. Its behavior is presented in Fig. 8. Initially, it loads the high-level policies, the *Analysis report*, and the *Server descriptions*

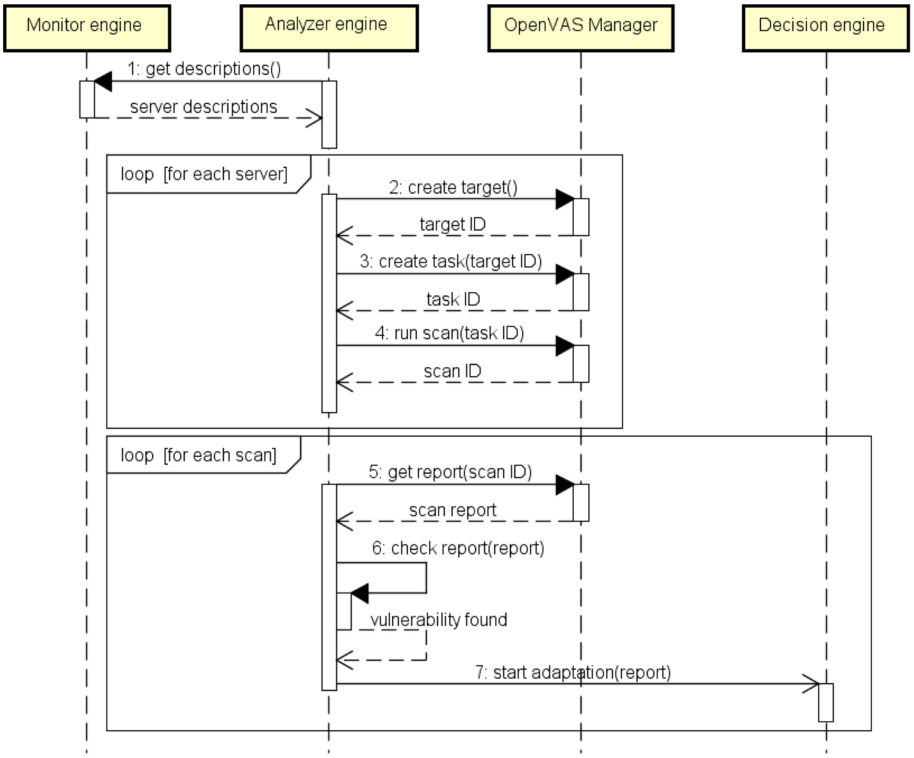


Fig. 7 UML sequence diagram for the Analyzer Engine

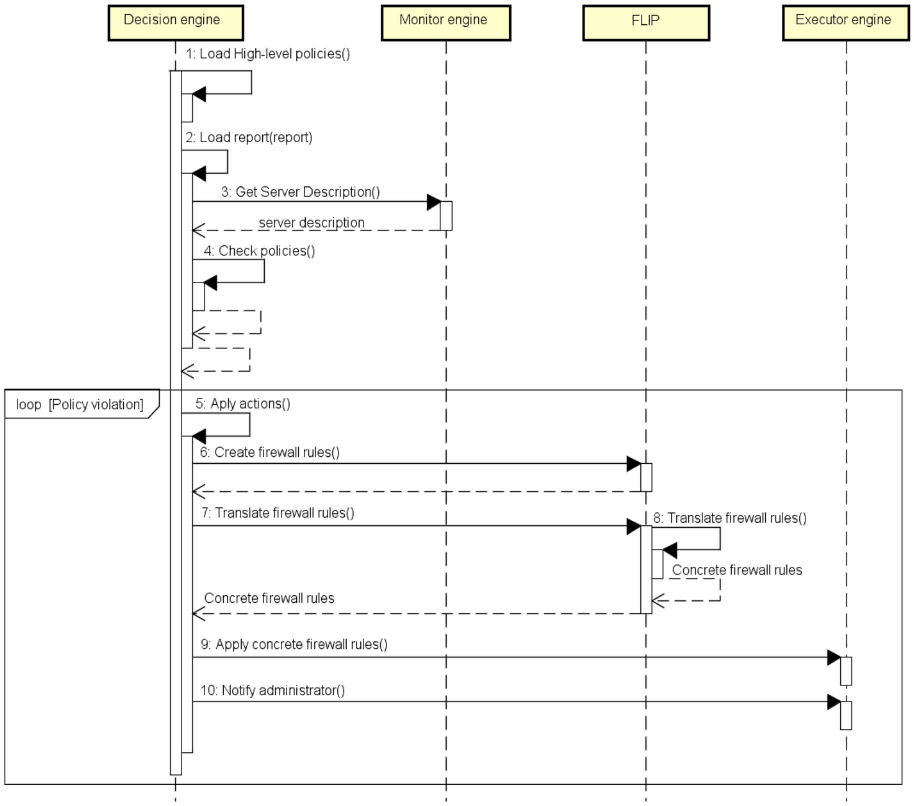

Decision engine
Monitor engine
FLIP
Executor engine
1: Load High-level policies()
2: Load report(report)
3: Get Server Description()
server description
4: Check policies()
loop [Policy violation]
5: Aply actions()
6: Create firewall rules()
7: Translate firewall rules()
8: Translate firewall rules()
Concrete firewall rules
Concrete firewall rules
9: Apply concrete firewall rules()
10: Notify administrator()

Fig. 8 UML sequence diagram for the *Decision engine*

(calls 1, 2, and 3). Then the policies are evaluated to detect any violation (call 4). This verification involves a set of iterations and conditional operations used to assert the conditions according to the data collected from the *Analysis report*.

When a violation is detected, an adaptation loop is started. In the current prototype implementation, the execution module allows two actions: (1) the configuration of the firewall rules on the affected servers, and (2) to send an alert to the system administrator. The action to be taken is defined by the high-level policy. If the selected action requires blocking a port, then the *Decision engine* creates the firewall rules using *FLIP* syntax and then request its translation (calls 6 and 7). Following, the *FLIP* translates the rules and creates the *Concrete firewall rules* model, which is written using the *Puppet* notation (call 8). The corresponding action is then requested to the *Executor engine*, either to apply new firewall rules to a server (call 9) or to notify the system administrator (call 10).

Figure 9 presents a simplified algorithm describing the policy evaluation mechanism. Essentially, the evaluation consists in looping through each server in which a vulnerability has been detected, and checking whether the conditions of the high-level policies hold and apply.

```
Procedure verifyViolations:
$serverDescriptions \leftarrow$ list of servers
$policies \leftarrow$ list of policies
$analysisReport \leftarrow$ list of vulnerabilities
foreach $server$ in $serverDescriptions$ do
    foreach $policy$ in $policies$ do
        if $server \in policy.servers$ then
            foreach $vulnerability$ in $analysisReport$ do
                if $vulnerability.port \in policy.port$ or $vulnerability.service \in policy.service$ then
                    if $vulnerability.cve == policy.cve$ and $vulnerability.cvss >= policy.cvss$ then
                        apply($policy.action$)
                    else if $vulnerability.cve == policy.cve$ then
                        apply($policy.action$)
                    else if $vulnerability.cvss >= policy.cvss$ then
                        apply($policy.action$)
                    end
                end
            end
        end
    end
end
```

Fig. 9 Algorithm for policy evaluation

4.2.4 Executor engine

Once the planning phase is finished, the *Executor engine* is invoked to apply all the necessary actions.

To block ports, it would apply the *Concrete firewall rules* by sending the generated classes to the directory */etc/puppetlabs/code/environments/production/manifests* to be read by the *puppet-master*. Following this step, each agent can contact the master to apply the scheduled firewall rules. The *puppet-master* together with the firewall module manages and configures the generated iptables and ip6tables rules to be applied directly to each host.

The Zabbix[10] monitoring system is employed in order to support alerting system administrators. Zabbix is a network monitoring system with support to triggered notifications. SADF creates a list of alerts, which is used as the basis for populating a monitoring template configured into Zabbix. This template includes information about the affected host, its service, and the discovered vulnerability, with its respective CVSS. Through the template, Zabbix can detect the alerts sent by SADF, creating triggers for each alert into Zabbix monitoring screen that should be permanently exhibited on the Network Operation Center. We have employed Zabbix on SADF because this is the monitoring solution used by the network administration team of where SADF is being deployed.

The behavior of the *Executor engine* is presented in Fig. 10. As previously mentioned, our current prototype supports two actions, the notification of administrators (call 1), or the application of firewall rules through *puppet-master* (call 2). In case of applying firewall rules, once the new rules have been saved, each *puppet-agent* obtain them through the *Puppet* catalog (call 3), and then applies the rules on their corresponding hosts (call 4).

5 Evaluation

This section presents an evaluation of SADF that has been conducted in a controlled environment.

First, we present the complete operational flow of SADF to demonstrate its feasibility (section 5.1). In the sequence, we present a set of experiments to evaluate SADF resource consumption (section 5.2). Finally, we discuss the achieved results (section 5.3).

5.1 Demonstration

A set of servers and services in a controlled environment were deployed in order to demonstrate the functionalities of the proposed SADF. All servers run the CentOS 7. Such services are similar to what can be found in a typical network infrastructure of a university. Figure 11 presents a general view of the deployed environment, in which we can find two groups of servers. The first group corresponds to SADF components and is composed by three servers: (i) *puppet-master* agent that acts as a configuration management server; (ii) *sadf-engine*, which contains all components of SADF; and (iii) *sadf-scanner* corresponding to the OpenVAS vulnerability assessment system. The second group of servers represents the target environment, which is controlled by SADF, and corresponds to the different servers that will have their firewall rules managed by our solution. For this first demonstration, we have instantiated a total of 10 servers (*sadf-target01* to *sadf-target10*), in which we have services managed by *Puppet*.

Several services that are commonly found on complex network infrastructure were selected in order to demonstrate SADF functionality. The selection of services was based on past experiences with incidents that occurred in the UFRN[11] network infrastructure. Therefore, we have some services with vulnerabilities of different severity

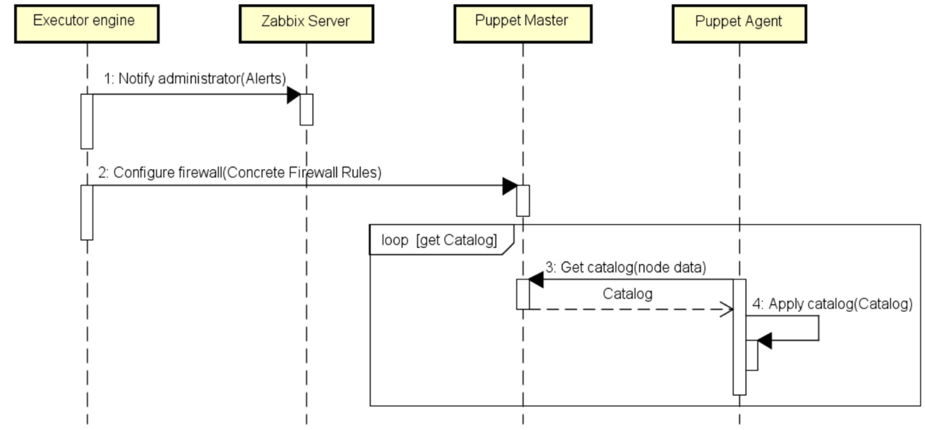

Executor engine
Zabbix Server
Puppet Master
Puppet Agent
1: Notify administrator(Alerts)
2: Configure firewall(Concrete Firewall Rules)
loop [get Catalog]
3: Get catalog(node data)
Catalog
4: Apply catalog(Catalog)

Fig. 10 UML sequence diagram of *Executor engine*

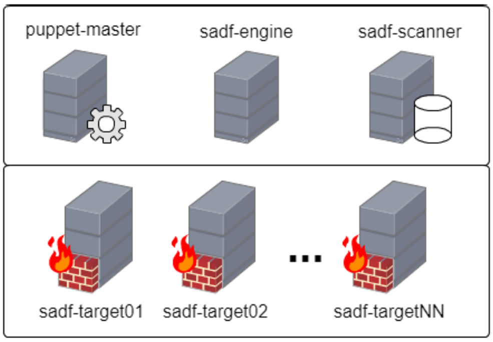


Fig. 11 Environment deployed for testing SADF

levels. Table 2 presents details about versions, ports, and vulnerabilities of the services considered in this demonstration.

These services have been distributed in 10 servers *sadf-target* according to Table 3. The column *Vulnerability Severity* represents the server status based on the vulnerabilities found on its deployed services. The classification of vulnerabilities is based on the CVSS V2 specification[12], which defines the intervals 0.0 − 3.9 as *LOW*, 4.0 − 6.9 as *MEDIUM*, and 7.0 − 10.0 as *HIGH*.

Once all target servers are configured and managed by *puppet-master*, SADF is capable of extracting their *Server descriptions* to monitor and control them. The next step is to configure the *high-level policies* that define the behaviour of SADF. For this demonstration, we have defined four policies that either block vulnerable services or alert network administrators in different contexts. The policies defined are listed below with an informal textual description, and then using the policy definition rules of SADF.

P1: Alert administrator if any vulnerability with CVSS higher than 5.0 is found in any server:
"Server = any", "Service = any", "Port = any", "CVSS = 5.0", "Action = alert"

P2: Block any port on any server assigned to any service in which with vulnerability "CVE-2013-4810" (regarding to JBoss 5.1.0) has been detected:
"Server = any", "Service = any", "Port = any", "CVE = CVE-2013-4810", "Action = block"

P3: Block any port assigned to any service with CVSS equal or higher than 7.0 running on the server *sadf-target06.info.ufrn.br*:
"Server = sadf-target06.info.ufrn.br", "Service = any", "Port = any", "CVSS = 7.0", "Action = block"

P4: Block any server or port assigned to the service *ftp* with CVSS equal or higher than 9.0:
"Server = any", "Service = Ftp", "Port = any", "CVSS = 9.0", "Action = block"

P5: Only allow connections from a particular network on any server that belongs to this network in case any vulnerability is found.
"Server = 10.2.59.0/24", "Service = any", "Port = any", "CVSS = any", "Action = isolate"

Once the high-level policies are in place, SADF is ready to start monitoring and controlling the target servers, according to the MAPE-K feedback control loop. It starts by obtaining the *server descriptions* of the target servers and interacting with OpenVAS for creating the respective scan routines. Based on OpenVAS results, which identified the existing vulnerabilities in the testing environment, SADF employed the high-level policies to decide how to respond. Following our case study, the OpenVAS found a vulnerability with CVSS score 7.8 in HTTP service that allows remote attackers to cause a denial of service (memory and CPU consumption). An extract of the XML report is presented in Listing 4.

Listing 4 Extraction of OpenVAS report to a vulnerable HTTP server

```
  ... <port>80/tcp
   <host>10.3.128.20</host>
   <severity>7.8</severity>
   <threat>High</threat>
  </port> <nvt oid
      ="1.3.6.1.4.1.25623.1.0.901203">
   <name>
   Apache httpd Web Server Range Header
      Denial of Service Vulnerability
   </name>
   <family>Denial of Service</family>
   <cvss_base>7.8</cvss_base>
   <cve>CVE-2011-3192</cve>
   <bid>49303</bid>
  ...
```

Based on policy P1, which defines the trigger of alerts for vulnerabilities with CVSS above 5.0, we have the list of alerts presented in Fig. 12, extracted from Zabbix. SADF creates several objects and triggers in Zabbix with information about the vulnerable services, using a color classification according to the severity of the vulnerability found. We can notice that some servers appeared more than once in the list (such as sadf-target10), which indicates that either the server contains more than one vulnerability, or that the same vulnerability has been found in different ports.

Based on the other policies, SADF activated the *Executor engine* for blocking several ports. Policy P2 blocked port 8080 on server *sadf-target09*, while policy P3 blocked port 80 on server *sadf-target06*. policies P2 and P4 caused the blocking of ports 21 and 8080 on server *sadf-target01*, and policy P4 blocked port 21 on server *sadf-target03*.

Policy P5, in particular, is an example of a policy which could be applied for isolating servers of a particular network by only accepting connections from their network

Table 2 Services and vulnerabilities used to test the prototype

Service	Port	CVE	CVSS	Vulnerability
Apache 2.4.6	443	CVE-2016-2183, CVE-2016-6329	5.0	Confidential information leak.
Apache 2.4.6	80	CVE-2004-2320, CVE-2003-1567	5.8	Cross site
Apache 2.2.15	80	CVE-2004-2320, CVE-2003-1567	5.8	Cross site
Apache 2.2.15	80	CVE-2011-3192	7.8	Denial-of-service (DoS)
MySQL 5.5.52	3306	-	-	-
ProFTPD 1.3.4a	21	CVE-2015-3306	10.0	Arbitrary remote code execution.
JBoss 5.1.0	8080	CVE-2013-4810	10.0	Arbitrary remote code execution.
JBoss 5.1.0	8009	-	-	-
JBoss 5.1.0	9090	-	-	-
Tomcat 8.5.16	8080	-	-	-
Tomcat 8.5.16	8009	-	-	-
Tomcat 8.5.16	9090	-	-	-

(*10.2.59.0/24*) when any vulnerability is found. Such policy triggers the application of firewall rules on all servers managed by SADF that belong to the identified network.

The blocking of ports is achieved by manipulating objects based on the FLIP meta-model defined by us. These objects are then translated into Puppet classes containing concrete firewall rules. Each server has its own *Puppet* class with its respective firewall rules. Listing 5 presents an example of class for server *sadf_target01*.

Listing 5 New class created by the SADF to block ports 21 and 8080 on the server sadf-target01

```
class fw_sadf_target01 {
 include firewall_module
 firewall { ’100’:
 dport => ’8080’,
 proto => tcp,
 action => drop,
 }
 firewall { ’101’:
 dport => ’21’,
 proto => tcp,
 action => drop,
 }
}
```

The iptables tool was used on the affected servers in order to confirm if the firewall rules have been correctly applied.

Figure 13 presents the output for server *sadf-target01* before SADF has run. We can notice explicit rules allowing (*ACCEPT*) packets from the servers that play the role of *puppet-master* (IP address 10.3.225.163) and *sadf-scanner* (IP address 10.3.227.77), and no closed ports (indicated by the absence of rules, which means that all connection are accepted). These rules are maintained by the firewall module of *Puppet*, defining the servers that are part of SADF as trusted. Figure 14 presents the iptables output for the same server (*sadf-target01*) after SADF detected a vulnerability and reacted according to its policies. We can notice rules allowing connections from SADF components (*puppet-master* and *sadf-scanner*), similar to Fig. 13, and rules for blocking (*DROP*) incoming packets (*INPUT* chain) on ports 21/tcp and 8080/tcp.

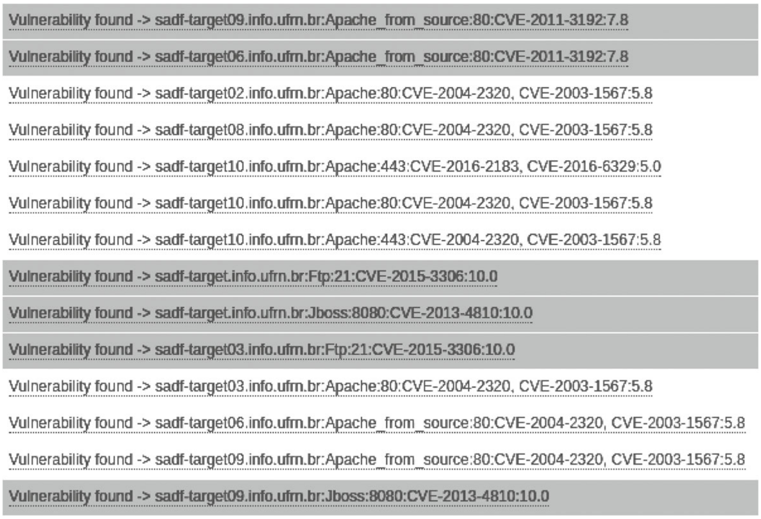
This procedure was repeated to all servers where we confirmed the application of firewall rules according to the high-level policies defined. We have repeated these experiments, varying the services versions and known vulnerabilities, services, and high-level policies. In all occasions SADF worked as expected, demonstrating its effectiveness in blocking vulnerable services and triggering alarms for alerting administrators when the blocking is deemed a too harsh response.

5.2 Resource consumption experiments

After demonstrating the operational viability of SADF, we have conducted a set of experiments for evaluating its resource consumption. These experiments have been conducted with the objective of providing subsides for

Table 3 Servers and services tested

Server	Vulnerability severity	Running services
sadf-target01	HIGH	JBoss 5.1.0; ProFTPD 1.3.4a
sadf-target02	MEDIUM	Apache 2.4.6; Tomcat 8.5.16
sadf-target03	HIGH	Apache 2.4.6; ProFTPD 1.3.4a
sadf-target04	MEDIUM	Apache 2.4.6; MySQL 5.5.52
sadf-target05	NONE	Tomcat 8.5.16
sadf-target06	MEDIUM	Apache 2.2.15
sadf-target07	NONE	MySQL 5.5.52
sadf-target08	MEDIUM	Apache 2.4.6
sadf-target09	HIGH	JBoss 5.1.0; Apache 2.2.15
sadf-target10	MEDIUM	Apache 2.2.15; MySQL 5.5.52



```
Vulnerability found -> sadf-target09.info.ufrn.br:Apache_from_source:80:CVE-2011-3192:7.8
Vulnerability found -> sadf-target06.info.ufrn.br:Apache_from_source:80:CVE-2011-3192:7.8
Vulnerability found -> sadf-target02.info.ufrn.br:Apache:80:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target08.info.ufrn.br:Apache:80:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target10.info.ufrn.br:Apache:443:CVE-2016-2183, CVE-2016-6329:5.0
Vulnerability found -> sadf-target10.info.ufrn.br:Apache:80:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target10.info.ufrn.br:Apache:443:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target.info.ufrn.br:Ftp:21:CVE-2015-3306:10.0
Vulnerability found -> sadf-target.info.ufrn.br:Jboss:8080:CVE-2013-4810:10.0
Vulnerability found -> sadf-target03.info.ufrn.br:Ftp:21:CVE-2015-3306:10.0
Vulnerability found -> sadf-target03.info.ufrn.br:Apache:80:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target06.info.ufrn.br:Apache_from_source:80:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target09.info.ufrn.br:Apache_from_source:80:CVE-2004-2320, CVE-2003-1567:5.8
Vulnerability found -> sadf-target09.info.ufrn.br:Jboss:8080:CVE-2013-4810:10.0
```

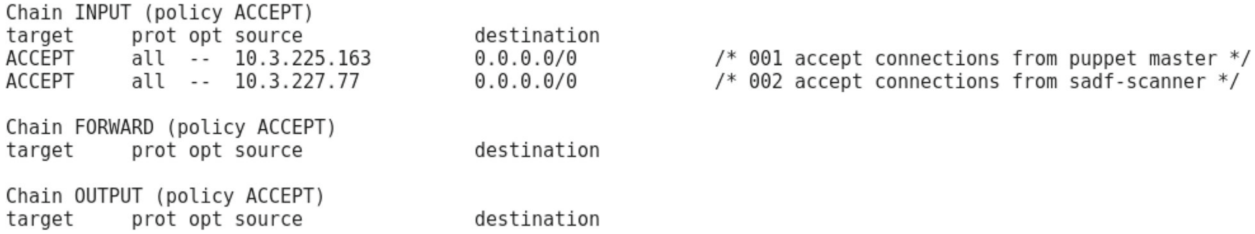
Fig. 12 Vulnerability alerts triggered on Zabbix system

the allocation of resources for SADF deployment. A first observation allowed us to notice that the biggest overhead of SADF is related to the vulnerability analysis. Based on this, the experiments conducted have been divided into two groups. The first group considered whether there is an improvement in the use of OpenVAS when the ports to be scanned are explicitly indicated (based on the services' ports configured via *Puppet*). The second group considered the impact of increasing the number of servers monitored and controlled.

These experiments have been conducted on the same environment used for demonstrating SADF (which has been presented in Fig. 11). The configurations of each server used in these experiments are presented in Table 4. The resources allocated to *puppet-master* server are based on the official documentation of puppet[13], which recommends a server with four cores and at least 4GB of RAM for attending a number of 1000 servers. The resources allocated for *sadf-engine* correspond to the server used for its development. The server responsible for *OpenVAS*, *sadf-scanner*, demands more computational power as it will potentially conduct more than 50,000 NVTs in each monitored host. The resources allocated to this server are based on the resource allocation to similar services in our real network infrastructure.

To run the first group of experiments, we replicated the 10 servers used in the demonstration of SADF, considering a growing number of servers, respectively, 1, 10, 20, 30, 40 and 50 servers, while employing the same distribution of services showed on Table 3. For these experiments, *OpenVAS* has been used in its default configuration, up to 10 NVTs running in parallel for each host, and a maximum of 30 hosts that can be verified simultaneously. In this context, in case there are more than 30 servers to be verified, the exceeding ones will be queued until the end of the scans in execution. *OpenVAS* has been restarted between each test, and the tests have been repeated three times for obtaining the average execution time for each run.

When considering the consumption of hardware resources, we did not identify any significant difference with and without the definition of ports. However, *OpenVAS* took around 2 hours and 43 minutes to scan 50 servers in its default configuration. When comparing the time necessary to run the tests with (*Scan on defined ports*) and without (*Scan without ports definition*) port

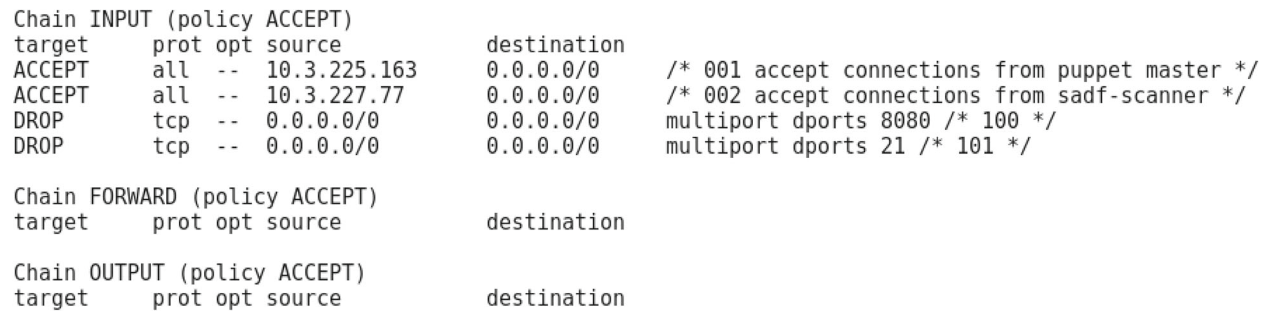


```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  10.3.225.163         0.0.0.0/0            /* 001 accept connections from puppet master */
ACCEPT     all  --  10.3.227.77          0.0.0.0/0            /* 002 accept connections from sadf-scanner */

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Fig. 13 Iptables output of firewall rules that are managed by *Puppet* on the server *sadf-target01* before running SADF



```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  10.3.225.163         0.0.0.0/0            /* 001 accept connections from puppet master */
ACCEPT     all  --  10.3.227.77          0.0.0.0/0            /* 002 accept connections from sadf-scanner */
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0            multiport dports 8080 /* 100 */
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0            multiport dports 21 /* 101 */

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Fig. 14 Iptables output of firewall rules that are managed by *Puppet* on the server *sadf-target01* after running SADF

indication, we have a difference up to 20.53% for 50 servers (2 h and 25 min with ports defined). We considered these results as a substantial indication of the benefits of directing *OpenVAS* scans as done by SADF, which encouraged the execution of the second group of experiments.

After that, we started to experiment with the full SADF solution. As previously mentioned, the biggest overhead of SADF is related to the vulnerability analysis. Thus, our focus was on evaluating the analysis phase of SADF, which includes OpenVAS to scan hosts on specific ports.

During these experiments, we have observed the resource consumption of *sadf-scanner* to understand its behavior and profile its hardware requirements. We present the measurements for the experiments involving 50 hosts. Figure 15 presents CPU load, Fig. 16 presents the RAM utilization, and Fig. 17 presents the network usage.

The higher CPU-load is expected, due to the number of NVT executing simultaneously. Regarding memory and network load, we notice a low usage, with some peaks caused by the inner workings of NVTs, which involves discovery and test of ports and service, followed by loading and running of tests. These peaks represent when the number of servers tested in parallel dropped below 30, and the tests for the next servers in the queue have been started. Regarding network consumption, a maximum of 6.78 Mbits/s is viewed as a good result, considering that the datacenter infrastructure is connected to at least 1Gbits/s, with 10 Gbits/s in several segments, and the duration of the peak when compared to the whole testing time.

Table 4 SADF servers resource configuration

Server	Memory	Processor
puppet-master	4 GB	4 Cores
sadf-engine	16 GB	8 Cores
sadf-scanner	16 GB	16 Cores

The results obtained with these experiments have been considered satisfactory. The time of 2 hours and 25 min to scan 50 hosts is within the expected time given the complexity of the vulnerability analysis process, and the experience of the network administration team.

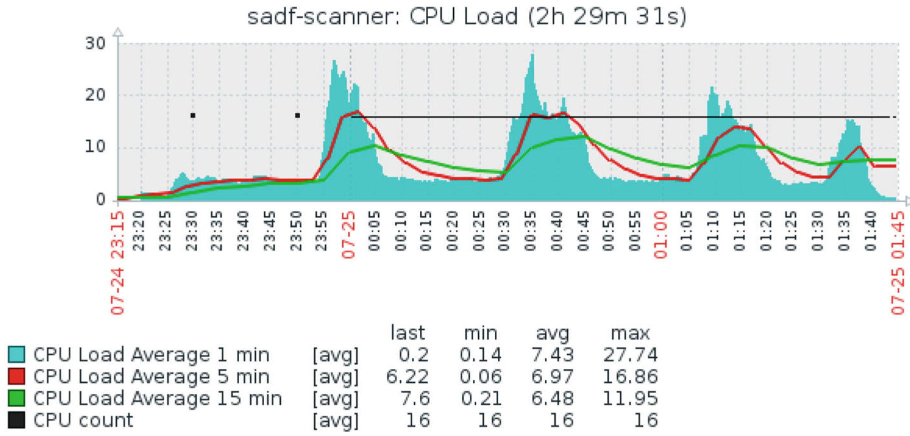
A more detailed discussion about the results obtained is presented in the sequence.

5.3 Discussion

The experiments conducted have shown that SADF can dynamically update firewall rules in response to discovered vulnerabilities, following the high-level policies defined by an administrator. Based on this, we conclude that SADF indeed provides a significant improvement in the administrative procedures of the network administration team of UFRN in detecting and protecting vulnerable services. This is evidenced when we consider that such procedure used to be conducted manually by network administrators, usually by different teams, and the response to a detected vulnerability took hours (sometimes days) to be implemented.

In its current implementation stage, SADF is only able to deal with hosts. Although SADF supports the specification of more complex firewall rules using FLIP and our high-level policies, we do not consider the deployment of firewall rules in other points of the network infrastructure, such as routers and switches. This is due to a simple decision-making implementation, which only targets hosts. Through the use of adaptation rules, the response can be anything supported by the controlled environment. Thus, although the proposed SADF can, for example, isolate a network employing firewall rules, such isolation is achieved by applying firewall rules at each host of the affected network.

The resource consumption experiments have confirmed the overhead caused by the vulnerability scanner, and the improvement of its use when the ports to be scanned are identified. Although the time to scan 50 hosts is above two hours, it is still substantially faster than the traditional



		last	min	avg	max
🟦 CPU Load Average 1 min	[avg]	0.2	0.14	7.43	27.74
🟥 CPU Load Average 5 min	[avg]	6.22	0.06	6.97	16.86
🟩 CPU Load Average 15 min	[avg]	7.6	0.21	6.48	11.95
⬛ CPU count	[avg]	16	16	16	16

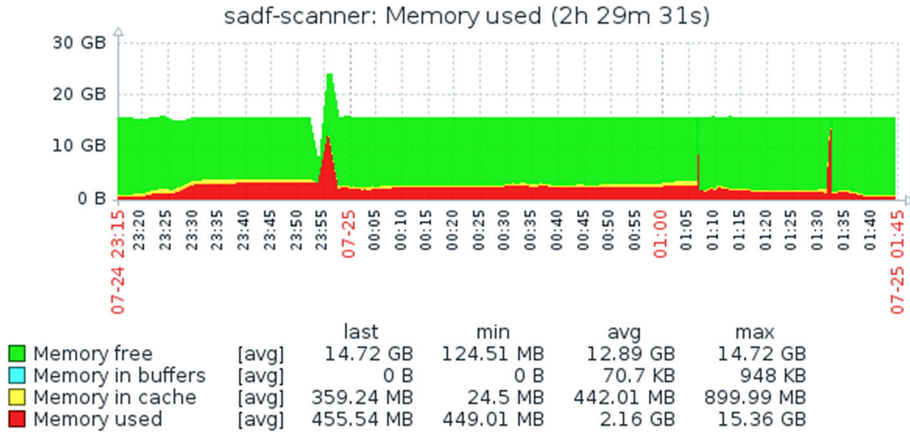
Fig. 15 Load average on the SADF server for 50 servers

manual approach. One aspect that has not been explored is to deploy OpenVAS as a cluster using a master-slave architecture, which has the potential to provide horizontal scalability to our solution without any changes to its implementation.

Another aspect worth mentioning concerns the *Puppet* configuration management tool. *Puppet* presents a pull-based model, in which agents query the master at predetermined intervals in search of new commands (usually, 30 minutes). It might be considered an issue when responding on-the-fly to detected situations, which is not the case in this paper. We focused on preventing the exploitation of known vulnerabilities before they happen, opposed to traditional IDPS systems, which focus on responding to incidents in real-time. Hence, we consider that 30 minutes is a reasonable time for applying firewall rules blocking vulnerable services, although this time could be reduced. We have changed this configuration to 10 minutes in our experiments, which is still considered as a satisfactory time, since the blocking action occurs in a preventive way, upon discovery of a vulnerable server.

Regarding the secure communication between the SADF and protected servers, we employ the certificate based security provided by the *Puppet* tool, which takes care of authentication and secure transit between the *puppet-master* and its agents through SSL certificates.

The number of 50 hosts have been used for the tests due to limitation of the infrastructure available for this project at the time of the experiments. Nevertheless, this number has been considered relevant for the evaluation of the proposal when taking into account feedback from the network management team, simulating an infrastructure capable of providing many applications and services. Also, this is the current number of servers that will be initially monitored when deployed in the UFRN data center. Besides that, the results obtained provide us guidelines on how to allocate resources to SADF components to reproduce the experiments with a larger number of servers.



		last	min	avg	max
🟩 Memory free	[avg]	14.72 GB	124.51 MB	12.89 GB	14.72 GB
🟦 Memory in buffers	[avg]	0 B	0 B	70.7 KB	948 KB
🟨 Memory in cache	[avg]	359.24 MB	24.5 MB	442.01 MB	899.99 MB
🟥 Memory used	[avg]	455.54 MB	449.01 MB	2.16 GB	15.36 GB

Fig. 16 RAM memory utilization on the SADF server for 50 servers

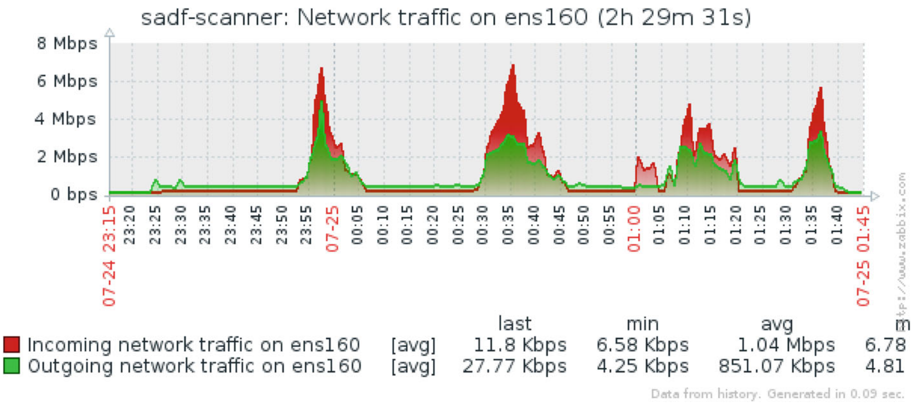


Fig. 17 Network traffic on the SADF server for 50 servers

6 Related work

The discussion on centralized and distributed firewalls is well established in the literature [2, 18]. One of its first implementation proposals has been presented by Ioannidis [1], in which kernel extensions have been developed for the OpenBSD distribution, together with a policy definition language (named *KeyNote*) and use of IPsec for secure traffic amongst the hosts of the network. More recently [19] introduces a distributed firewall system for Linux platform that works upon Iptables/Netfilter for IPv6 networks with IPsec support. A distributed firewall architecture was proposed in order to improve performance, handling the additional costs of encryption of packages with IPsec.

Autonomic computing and self-protection have been gaining traction as the means for dealing with new security challenges and systems, in which static and rigid security practices are not enough to deal with security threats that need to be detected and mitigated at runtime [7]. In this context, Yuan et al. [7] have done an extensive systematic survey of the state of the art on self-protecting software, identifying trends, patterns, and gaps. Some works focus on adapting authorization policies, such as the Self-Adaptive Authorisation Framework (SAAF) [5] that focus on adapting access control policies on the PERMIS system [20], and SecuriTAS [6], a tool that enables dynamic decisions in awarding physical access, based on a perceived state of the system and its environment.

Several researchers focus on Intrusion Detection and Response Systems. For example, Uribe and Cheung [21] have looked into the integration between IDSs and firewalls, proposing an approach for optimizing IDS configuration by only analyzing traffic that is not considered by the firewalls' rules. Zhang and Shen [22] employ a statistical learning based approach to reduce false positives on IDSs. Rahbarinia et al. [23] use graph mining techniques for analyzing download events for detecting malware download. These works are concentrated on improving an IDS, and present interesting discussions that could complement SADF in a possible future work integrating IDS into its architecture. Iannucci and Abdelwahed [24] employ a Markov Decision Process together with an Intrusion Response System for deciding which action to perform in response to a detected intruder. Compared to our work, their approach presents a formalism that can be used for deciding which action to perform, considering cost and impact, from a list of possible actions. Of-IDPS [25] is a system that considers network usage history and IDS alerts for extracting security rules that are applied through a Software Defined Network (SDN) based on *OpenFlow* controllers.

As stated by Shin et al. [26], SDN environment can be combined with traditional security mechanisms to reinforce the detection of attacks and to quickly react to them. In the context of our work, the major impact of SDN on a SADF solution would be twofold: at first, the decision making now needs to deal with flows fate besides firewall rules, and second, SADF can act upon switches through Access Control Lists (ACL) which allows creating rules connected to the network traffic [27]. In fact, our architecture would fit perfectly with an SDN environment, and with the range of possibilities pointed out in the literature [26, 27], being able to provide the link between a detection tool, the decision making process for deciding how to react to the detected situation, and the commands that will change the SDN environment. Similarly, an environment with NFV capabilities can provide more response options for our architecture.

There are also works that aim to facilitate the configuration of security mechanisms as part of the instantiation process of virtual machines in cloud computing environment [28, 29]. This approach provides IDS/IPS, firewall, and anti-malware in a solution Security-as-a-Service.

Few works consider the analysis of vulnerabilities for making a decision at the network level (i.e., firewall rules). Debar et al. [30] present formalisms for the definition of security policies that can be dynamically modified in response to detected threats. The formalism presented in their paper is at an abstract level and may consider vulnerability analysis in the threat detection process. Compared to our work, their approach can be considered as complementary, providing formalisms that could improve the robustness of our approach regarding the definition of high-level policies.

To the best of our knowledge, the closest work to ours has been presented by Sheridan et al. [31] and involves the automatic security of virtual machines in cloud computing environments. Their approach is based on three flows: First, a VAS analyses a host during its instantiation, notifying an administrator by e-mail in case a vulnerability is found. Second, the firewall provided by the cloud platform is activated for allowing access to services known during instantiation. Third, the Chef configuration management system is employed for automatically installing and upgrading software packages for hardening and updating the operating system. Although their approach also employs a VAS and activate firewall rules, the VAS output is not used for decision making on which firewall rules to apply, as we propose. Moreover, their approach only considers a virtual machine during its creation, not presenting any means for constant monitoring, and the automatic update of software packages without any supervision (or an intelligence level) might be dangerous in a critical environment.

An aspect that must be considered for a self-protection solution is the representation of the protected environment, such as servers, services, and firewall rules. We present here some of the options found in the literature during our searches. The Network Markup Language (NML) [32] is a generic model defined by the OpenGrid Forum (OGF)[14] as a standard for modeling networks, such as switches and links, which is out of the scope considered in this paper. Regarding the representation of firewall rules, apart from the FLIP language (already presented), there is also the AFPL2 [33] (Abstract Firewall Policy Language 2), a domain-specific language that provides an XML Schema for the definition of firewall rules independent of firewall product. Although its support of NAT rules, AFPL2 has not evolved as FLIP and does not provide conflict resolution of firewall rules. The Distributed Management Task Force (DMTF) proposed the Common Information Model (CIM), a specification aimed at allowing the interoperation of management information. The CIM also provides an extensible XML model and, although it has been employed by different vendors, its extension for Network Policy Management [34] is still considered work-in-progress, and may be subject to changes.

The research in the area of self-protection, when considering the network level, tend to focus on IDS/IRS. To the best of our knowledge, we have not found any work that employs vulnerability detection as the trigger for self-adaptation, or that consider the addition of self-adaptive capabilities to a distributed firewall.

7 Conclusions & future work

This article presented an approach for Self-Adaptive Distributed Firewall (SADF) based on the synergies between the MAPE-K feedback control loop of self-adaptive systems with the increased network security provided by distributed firewalls. The MAPE-K reference model is used as the means for logically structuring the different tasks involved in the management of network security, employing a vulnerability analysis system to detect vulnerable hosts on the network infrastructure. Along these lines, our approach can cope with the complexity on the management of distributed firewalls, while reducing the exposure windows of vulnerable hosts by automatically applying firewall rules during run-time, according to specified high-level policies, in response to detected vulnerabilities.

We have developed a prototype of SADF using a combination of existing open source and in-house developed components, which has been used to conduct a series of experiments, involving different services with a varied degree of vulnerabilities. These experiments have demonstrated the feasibility of SADF, which is able to dynamically modify the firewalls on protected servers, and presented a satisfactory performance of the environment in which it is being deployed. For example, SADF managed to achieve a 20% reduction on the scanning time for a group of 50 servers using the default configurations of the OpenVAS vulnerability scanning system, a significant performance gain which reduces the exposure window of a detected vulnerability. These experiments provided interesting feedback about its configuration and operation that has been incorporated into our solution.

Although we obtained very encouraging results, SADF presents some limitations. We employ the *Puppet* configuration management tool for describing servers and services, and for applying firewall rules on the affected hosts. Accordingly, we assume the infrastructure has been previously configured using this tool, as we employ *Puppet* descriptions as part of our monitoring and execution engine.

Another limitation is related to the use of NVTs. Each NVT is updated based on the NVT Feed, maintained by *Greenbone Community*[15]. SADF is only able to react to known vulnerabilities that have been published to NVT Feed. Moreover, recently *Greenbone* has adopted measures to increase paid subscription to its service, which might delay the availability of new NVTs.

In its current stage, SADF only deal with hosts. Although SADF supports the specification of more complex firewall rules using FLIP and our high-level policies, we do not consider the deployment of firewall rules in other points of the network infrastructure, such as routers and switches. This is caused by a simple decision-making process, which only targets hosts. Although it is possible to isolate a network through firewall rules, such isolation is achieved at each host of the affected network.

Other improvements on the decision making might allow more complex responses. For example, a host firewall might redirect all incoming connection to an application level firewall or IDS when a particular vulnerability has been detected, adding an extra layer of security while the vulnerability has not been fixed. Another future work involves the integration of our solution with traditional IDPSs on the monitoring and analyses phases, allowing it to react to attacks exploiting zero-day vulnerabilities. This would allow the use of less disruptive firewall rules, such as blocking of vulnerable services only from suspect sources. SADF can also be easily scaled up by deploying OpenVAS into a cluster configuration to deal with an elevated number of monitored servers.

Another possible future work is related to Software-Defined Networks (SDN) and virtualization. In our setup, we have not considered the impact of such technologies. An SDN-enabled infrastructure, combined with support for virtualized network functions, would allow for more sophisticated responses without direct host intervention. New challenges arise from the dynamicity of the Virtual Machines (VMs), which imposes frequent changes on the virtual network topology, where VMs might be migrated for resource management and optimization. We intent on conducting further research in this direction in order to support new responses, such as the dynamic redirection of traffic to an application proxy that could perform a second authentication.

Endnotes

[1] Common Vulnerabilities and Exposures sponsored by US-CERT - https://cve.mitre.org

[2] National Vulnerability Database maintained by NIST - https://nvd.nist.gov/

[3] Common Vulnerability Scoring System sponsored by FIRST - https://www.first.org/cvss

[4] https://puppet.com/

[5] https://forge.puppet.com/puppetlabs/firewall

[6] *PuppetDB* - Overview https://docs.puppet.com/puppetdb/

[7] *PuppetDB* 5.0: API overview https://docs.puppet.com/puppetdb/latest/api/index.html

[8] OpenVAS http://www.openvas.org/

[9] http://www.openvas.org/nvt-dev.html

[10] Zabbix is an enterprise-class open source monitoring solution for network monitoring and application monitoring of millions of metrics - https://www.zabbix.com/

[11] Federal University of Rio Grande do Norte, Brazil

[12] NVD CVSS Support https://nvd.nist.gov/vuln-metrics/cvss

[13] *Puppet* system requirements https://docs.puppet.com/puppet/5.0/system_requirements.html

[14] https://www.ogf.org/

[15] Greenbone Community https://www.greenbone.net/en/community-edition/

Availability of data and materials
The datasets generated during the current work are not publicly available due to its private nature, as it is related to a real University, but are available from the corresponding author on reasonable request.

Authors' contributions
EPCJ was responsible for the majority of the technical work, implementing the prototype, conducting experiments and collecting the obtained results, drafting the initial version of the article. CEDS made contributions to the conception and design of the proposed solution, validating the prototype implementation, and contributing for critically reviewing the manuscript. MP made contributions to the conception and design of the proposed solution, to designing the experiments conducted, and writing the manuscript. SCS participated in the drafting and critically reviewing the article for important intellectual content, and made substantial contributions to the analysis of the experimental data. All authors read and approved the final manuscript.

Competing interests
The author(s) declare(s) that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details
[1] Digital Metropolis Institute, Federal University of Rio Grande do Norte (UFRN), Natal, RN, Brazil. [2] Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte (UFRN), Natal, RN, Brazil.

Received: 24 August 2017 Accepted: 10 April 2018
Published online: 04 June 2018

References
1. Ioannidis S, Keromytis AD, Bellovin SM, Smith JM. Implementing a Distributed Firewall. In: Proceedings of the 7th ACM Conference on Computer and Communications Security. New York: ACM; 2000. p. 190–9. CCS '00. https://doi.org/10.1145/352600.353052.
2. Bellovin SM. Distributed Firewalls. J Login. 1999;24(5):39–47. https://www.cs.columbia.edu/~smb/papers/distfw.pdf.
3. Meng G, Liu Y, Zhang J, Pokluda A, Boutaba R. Collaborative Security: A Survey and Taxonomy. ACM Comput Surv. 2015;48(1):1:1–1:42. https://doi.org/10.1145/2785733.
4. Young G, Pescatore J. Magic quadrant for enterprise network firewalls. Tech. Rep. 141050, Gartner RAS Core G. 2008. http://www.eircomictdirect.ie/docs/juniper/gartner_magic_quadrant_2008.pdf.
5. Bailey C, Chadwick DW, de Lemos R. Self-adaptive federated authorization infrastructures. J Comput Syst Sci. 2014;80(5):935–52. https://doi.org/10.1016/j.jcss.2014.02.003.

6. Pasquale L, Menghi C, Salehie M, Cavallaro L, Omoronyia I, Nuseibeh B. SecuriTAS: A Tool for Engineering Adaptive Security. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York: ACM; 2012. p. 19:1–19:4. FSE '12. https://doi.org/10.1145/2393596.2393618.
7. Yuan E, Malek S, Schmerl B, Garlan D, Gennari J. Architecture-based Self-protecting Software Systems. In: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures. New York: ACM; 2013. p 33–42. QoSA '13. https://doi.org/10.1145/2465478.2465479.
8. da Costa Jr. EP, da Silva CE, Madruga M, Medeiros ST. XVI Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg2016). 2016. http://sbseg2016.ic.uff.br/pt/files/anais/completos/ST7-1.pdf. Accessed 15 Mar 2018.
9. Cheng BH, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng BH, de Lemos R, Giese H, Inverardi P, Magee J, editors. Software Engineering for Self-Adaptive Systems. Berlin: Springer-Verlag; 2009. p. 1–26. https://doi.org/10.1007/978-3-642-02161-9_1.
10. Kurose JF, Ross KW. Computer Networking: A Top-Down Approach. 6th ed. New York: Pearson; 2012.
11. Kephart JO, Chess DM. The Vision of Autonomic Computing. IEEE Comput. 2003;36(1):41–50. https://doi.org/10.1109/MC.2003.1160055.
12. Iglesia DGDL, Weyns D. MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. ACM Trans Auton Adapt Syst. 2015;10(3):15:1–15:31. https://doi.org/10.1145/2724719.
13. Sweitzer JW, Draper C. Architecture Overview for Autonomic Computing. In: Parashar M, Hariri S, editors. Autonomic Computing: Concepts, Infrastructure, and Applications. CRC Press; 2006. p. 71–98.
14. Vromant P, Weyns D, Malek S, Andersson J. On Interacting Control Loops in Self-Adaptative Systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems; 2011. p. 202–7. https://doi.org/10.1145/1988008.1988037.
15. Zhang B, Al-Shaer E, Jagadeesan R, Riely J, Pitcher C. Specifications of a High-level Conflict-free Firewall Policy Language for Multi-domain Networks. In: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies. New York: ACM; 2007. p. 185–94. SACMAT '07. https://doi.org/10.1145/1266840.1266871.
16. Al-Shaer E. In: Automated Firewall Analytics: Design, Configuration and Optimization. Springer International Publishing; 2014. p. 49–74. chap. Specification and Refinement of a Conflict-Free Distributed Firewall Configuration Language. https://doi.org/10.1007/978-3-319-10371-6_3.
17. Alferes JJ, Banti F, Brogi A. An Event-Condition-Action Logic Programming Language. Berlin, Heidelberg: Springer Berlin Heidelberg; 2006, pp. 29–42. https://doi.org/10.1007/11853886_5.
18. Stallings W. Network Security Essentials: Applications and Standards, 4th edn. Englewood Cliffs: Prentice Hall; 2010.
19. Lai Y, Jiang G, Li J, Yang Z. Design and Implementation of Distributed Firewall System for IPv6. In: Communication Software and Networks, 2009. ICCSN '09. International Conference on; 2009. p. 428–32. https://doi.org/10.1109/ICCSN.2009.121.
20. Chadwick DW, Zhao G, Otenko S, Laborde R, Su L, Nguyen TA. PERMIS: a modular authorization infrastructure, Concurrency and Computation: Practice & Experience. 2008;20(11):1341–57. https://doi.org/10.1002/cpe.v20:11.
21. Uribe TE, Cheung S. Automatic Analysis of Firewall and Network Intrusion Detection System Configurations. In: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering (FMSE 2004); 2004. p. 66–74. https://doi.org/10.1145/1029133.1029143.
22. Zhang Z, Shen H. M-AID: An Adaptive Middleware Built Upon Anomaly Detectors for Intrusion Detection and Rational Response. ACM Trans Auton Adapt Syst. 2009;4(4):24:1–24:35. https://doi.org/10.1145/1636665.1636670.
23. Rahbarinia B, Balduzzi M, Perdisci R. Real-Time Detection of Malware Downloads via Large-Scale URL ⟶ File ⟶ Machine Graph Mining. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. New York: ACM; 2016. p. 783–94. ASIA CCS '16. https://doi.org/10.1145/2897845.2897918.
24. Iannucci S, Abdelwahed S. A Probabilistic Approach to Autonomic Security Management. In: 2016 IEEE International Conference on Autonomic Computing (ICAC); 2016. p. 157–66. https://doi.org/10.1109/ICAC.2016.12.
25. dos Santos LAF, Campiolo R, Monteverde WA, Batista DM. Abordagem autonômica para mitigar ciberataques em LANs. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC); 2016. p. 600–13. http://www.sbrc2016.ufba.br/downloads/SessoesTecnicas/152417.pdf.
26. Shin S, Xu L, Hong S, Gu G. Enhancing Network Security through Software Defined Networking (SDN). In: Proceedings of the 25th International Conference on Computer Communication and Networks ICCCN '16, vol. 1; 2016. p. 1–9. https://doi.org/10.1109/ICCCN.2016.7568520.
27. Alsmadi I, Xu D. Security of Software Defined Networks: A survey. Comput Secur. 2015;53:79–108. https://doi.org/10.1016/j.cose.2015.05.006.
28. Daniel J, Dimitrakos T, El-Moussa F, Ducatel G, Pawar P, Sajjad A. Seamless Enablement of Intelligent Protection for Enterprise Cloud Applications through Service Store. In: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science; 2014. p. 1021–6. https://doi.org/10.1109/CloudCom.2014.92.
29. Pawar PS, Sajjad A, Dimitrakos T, Chadwick DW. Security-as-a-Service in Multi-cloud and Federated Cloud Environments. In: Damsgaard Jensen C, Marsh S, Dimitrakos T, Murayama Y, editors. Trust Management IX. Cham: Springer International Publishing; 2015. p. 251–61. https://doi.org/10.1007/978-3-319-18491-3_21.
30. Debar H, Thomas Y, Cuppens F, Cuppens-Boulahia N. Enabling automated threat response through the use of a dynamic security policy. J Comput Virol. 2007;3(3):195–210. https://doi.org/10.1007/s11416-007-0039-z.
31. Sheridan C, Massonet P, Phee A. Deployment-Time Multi-Cloud Application Security. In: 2017 IEEE International Conference on Smart Computing (SMARTCOMP); 2017. p. 1–5. https://doi.org/10.1109/SMARTCOMP.2017.7947000.
32. van der Ham J, Dijkstra F, Apacz R, Zurawski J. Network markup language base schema version 1. 2013. Grid Final Draft (GFD), Proposed Recommendation (R-P) GFD-R-P.206, Open Grid Forum. https://www.ogf.org/documents/GFD.206.pdf. Accessed 15 Mar 2018.
33. Pozo S, Varela-Vaca AJ, Gasca RM. AFPL2, an Abstract Language for Firewall ACLs with NAT Support. In: Second International Conference on Dependability (DEPEND 2009); 2009. p. 52–9. https://doi.org/10.1109/DEPEND.2009.14.
34. DMTF. Network policy management profile. 2016. https://www.dmtf.org/sites/default/files/standards/documents/DSP1048_1.0.0c_0.pdf. Accessed 15 Mar 2018.