Project Number 732223

D4.4 System Configuration Code Analyser - Final Version

Version 1.0
28 June 2019
Final

Public Distribution

Athens University of Economics and Business (AUEB)

Project Partners: **Athens University of Economics & Business**, **Bitergia**, **Castalia Solutions**, **Centrum Wiskunde & Informatica**, **Eclipse Foundation Europe**, **Edge Hill University**, **FrontEndART**, **OW2**, **SOFTEAM**, **The Open Group**, **University of L'Aquila**, **University of York**, **Unparallel Innovation**

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L'Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	Initial outline	5 June 2019
0.5	First complete draft	17 June 2019
0.8	Internal release to partners for review	24 June 2019
1.0	Final version incorporating internal review comments	28 June 2019

Table of Contents

Executive Summary

In this deliverable, we report the final implementation of the System Configuration Code Analyser. Its purpose is to provide an overview of its components and its functionalities. It builds upon and continues from Deliverable D.4.2 - System Configuration Code Analyser - Interim Version and focuses on the two main tools that are implemented in the context of Task 4.2: Development of the System Configuration Code Analyser, as described in the CROSSMINER work plan, Puppeteer, that analyses Puppet files and manifests, and Jadolint, a tool that analyses Dockerfiles. We describe the main functionalities of these tools, we present the smells that each tool detects and we give along concrete examples of these smells.

In the context of CROSSMINER, these outputs of our tools are translated into metrics of the Metric Platform. These metrics reveal insights related to configuration code quality and security. The metrics address a part of the requirements that are specified in Deliverable D1.1 and are exploited by the DevOps Dashboard, which is presented in Deliverable D4.3.

1 Introduction

This document reports the results that directly address Task 4.2: Development of the System Configuration Code Analyser, as described in the CROSSMINER work plan. To this end, D4.4 reports on the implementation of the final version of the above component, its functionality and its integration with the CROSSMINER platform.

This section presents an overview of the structure of the current document and an overview of System Configuration Code Analyser.

1.1 Purpose of this document

Infrastructure as Code (IaC) [12] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. Infrastructure as Code practices treat configuration code similar to the production code [13]. In equivalence to production code, configuration code should also follow certain standards in order to remain maintainable [10, 21] and adhere to design quality standards [30]. The CROSSMINER configuration code analysis is based on Puppet and Docker, two of the most popular configuration management tools. In this document we provide an overview of Puppet and Docker, the files that they handle, and define quality problems that can be encountered in their code. Moreover, we describe the tools that we developed to analyse such files and how we integrated the outputs of these tool in the CROSSMINER platform.

1.2 Structure of this document

This document focuses on the presentation of the architecture of the CROSSMINER configuration code analysis, therefore, the report mainly elaborates this in the related sections. The document is structured as follows:

- Section 2 overviews the CROSSMINER system configuration analysis component and describes how it addresses the "future work" part from D4.2. This section also reviews the stat- of-the-art on work related to system configuration analysis, focusing on tools relevant to the CROSSMINER analyser, namely, Puppet and Docker.
- Section 3 overviews the Puppet[1] software configuration management tool and lists the quality problems that the System Configuration Code Analyser detects in Puppet files.
- Section 4 overviews the Docker[2] software configuration management tool and lists the quality problems that the System Configuration Code Analyser detects in Docker files.
- Section 5 describes the components about the detection of new versions of third-party dependencies and the detection about relations between handled projects.
- Section 6 presents the integration of the System Configuration Code Analyser into the CROSSMINER platform and how and where its results are stored.
- Section 7 lists the requirements that are addressed by the System Configuration Code Analyser.
- Section 8 concludes the document.

[1] `https://puppet.com/`
[2] `https://www.docker.com/`

2 CROSSMINER System Configuration Code Analyser

2.1 Overview

The objective of this deliverable is to present the implementation of the CROSSMINER system configuration code analyser. The analyser detects code smells, i.e., quality/security problems, in system configuration files. Detecting such smells will help developers keep the code in these files reusable and maintainable. As well, by analysing candidate third-party software, developers will be able to decide if such software is worth to be included in their projects. In this deliverable, we focus on Puppet configuration files and Dockerfiles. We define the types of smells that are detected by the analyser. We introduce the methods employed for detecting these smells and describe how the analyser is integrated into the metric platform of CROSSMINER as new metric providers. This deliverable builds upon D4.2 based on the future work that has been described in it, whose main points are the integration of the metrics that were implemented about Puppet and the addition of metric providers about Docker. Both of these goals are achieved and their implementation is described in this deliverable.

2.2 Related Work

Our work is related to studies of code quality practices in traditional software engineering and system configuration management.

2.3 Code Quality Practices in Traditional Software Engineering

Fowler [10] characterized code smells as poor design and implementation decisions and identified numerous of them. Girish et al. [30] provided a comprehensive catalog of structural design smells classified based on the principle that they violate. Similarly, Garcia et al. [11] provided a catalog of smells that may arise at the architectural level. Many of our cataloged smells have names similar to ones in the literature [10, 30]; however, their meaning and context is aligned to the configuration domain. Additionally, we have added a few new ones; for example unstructured module and dense structure.

Smells lead to technical debt and affect maintainability negatively [14]. A popular approach for detecting them is static code analysis, which we also employ in this work. Metrics-based rules [19] identify code smells, such as God class or blob, by comparing computed metrics to specified thresholds. Decor [22] provides a domain specific language for formulating rules that detect smells such as blob, functional decomposition, and swiss army knife. Designite [28] detects numerous implementation and design smells. Along similar lines, various tools to identify refactoring candidates have been proposed; for example, extract method refactoring [31, 26] and extract class refactoring [29, 9].

In addition to the above approaches that base their analysis on a given code snapshot, hist [23] and Clio [33] consider change history information. [23] analyses changes between software components to detect blob, divergent change, shotgun surgery, parallel inheritance, and feature envy smells. Clio [33] detects software modularity violations with the study of co-evolving software components throughout project history.

2.4 Code Quality Practices in System Configuration Management

There is a paucity of empirical studies about system configuration management tools. Jiang et al. [13] study the co-evolution of Puppet and Chef configuration files with source, test and build code. Sharma et al. [27]

proposed design and implementation smells about Puppet and analysed Puppet repositories in order to detect smells. Cito et al. [7] discussed about the rise of Docker adoption in industry and performed an empirical study on Dockerfiles.

van der Bent et al. [32] developed a measurement model from quality metrics, derived from a survey among Puppet developers and from existing software quality models in order to provide a definition of code quality for Puppet code and an automated tool for measuring it. Lu et al. [18] defined the term “temporary file smell" in Dockerfiles. It is a problem that leads to file redundancy and eventually to larger-size images. This affects the scalability of Docker services. A static analysis method to detect this kind of smells is proposed along with fixing methods for them. Schwarz et al. [25] extended Sharma’s work [27] by transferring Puppet smells to Chef, showing that IaC code definitions are general and technology agnostic. Furthermore, a new set of smells is proposed.

Puppet Forge [2]– the repository of Puppet modules, provides an evaluation of configuration code quality through a quality score based on three aspects: code quality score provided by Puppet-Lint [3], compatibility with Puppet, and metadata quality. Metadata quality is subject to a set of guidelines that metadata files should adhere to.

Sonar-Puppet [4] is a SonarQube [1] plug-in that has numerous rules to detect quality violations in Puppet code; most of the rules applied by Sonar-Puppet are common with Puppet-Lint. We have included Puppet-Lint and the additional rules checked by Sonar-Puppet in our analysis and mapped the rules to implementation configuration smells. We have also implemented Puppeteer, a tool that identifies all the cataloged design configuration smells.

Various authors have published their ideas describing best practices for configuration code in the form of blog-posts, articles, and technical talks [15, 16, 17, 24].

3 Puppet Configuration Code Analysis

This section focuses on the analysis of Puppet configuration files, the smells that are detected and the metrics that are produces based on them.

3.1 Puppet Background

Puppet is a configuration management tool, which is developed in order to automate infrastructure management and configuration. Puppet facilitates the concept of IaC and helps modeling the system configuration of a complete infrastructure as code. Puppet code can be treated and, by extension infrastructure's configuration, just like any other code, which can be easily managed, configured, tracked, tested, deployed, reproduced and scaled.

Puppet follows the client-server model, where one or more machines in any cluster act as server known as Puppet master and the all the managed machines act as clients known as agents. Puppet has the capability to manage any system from scratch, starting from initial configuration till the end of life of any particular machine.

Puppet uses its own declarative programming language in order to orchestrate the configuration of a system. The Puppet language describes the state of a computer system in terms of "resources", which represent underlying network and operating system constructs. The user assembles resources into manifests that describe the desired state of the system and they are essentially the place where Puppet code resides. These manifests are stored on the server and compiled into configuration instructions for agents on request.

The foundation of the Puppet language is its declaration of resources. The notion of resource is the fundamental unit of Puppet, which is used to design and build any particular infrastructure or machine. A resource can define any component of the managed system such as users, groups, packages, files, cron, mounts and services.

The resource-oriented modelling is based on Puppet's Resource Abstraction Layer, which can be considered as its core conceptualised model. Abstraction can be considered as a key feature where the resources are defined independently from the target operating system. In other words, users can configure systems in a platform-agnostic way by representing operating system concepts as structured data. Rather than specifying the exact commands to perform a system action, the user creates a resources, which Puppet then translates into system-specific instructions which are sent to the machine being configured.

3.2 Analysis of Puppet System Configuration Files

Puppet programs are called manifests. Manifests are composed of Puppet code and their filenames use the .pp extension. The core of Puppet is the way resources are declared and how these resources are representing their state. In any manifest, the user can have a collection of different kind of resources which are grouped together using classes, which are larger units of configuration.

Classes are used in order to separate code blocks, to combine resources in larger units, to re-use code, and to make reading manifests easier. Classes are named blocks of Puppet code that are stored in modules for later use and are not applied until they are invoked by name.

```
user { 'crossminer_user':
   ensure => present,
   uid => '1000',
   gid => '1000',
   shell => '/bin/bash',
   content => '/home/crossminer_user'
}
```

Listing 1: Example of a Puppet manifest

Modules are self-contained bundles of code and data that have a specific directory structure. Modules are useful for organising Puppet configuration code because they allow splitting of code into multiple manifests. Puppet supports easy re-distribution of modules, which is very helpful in modularity of code as one can write a specified generic module and can use it multiple times with very few simple code changes.

At the bottom line, classes and modules are ways to organise manifests, which contains the declarations of resources. Every resource is associated with a resource type, which determines the kind of configuration it manages. A resource has a resource type, a title, and a set of attributes. An example of a manifest file is shown in Listing 1. The resource type it describes is a user, its title is "crossminer_user" and it has some variables, like its user id, its group id, the type of the Unix shell it uses, its home folder and the "ensure" variable defines that this user should exist.

```
exec { 'apt-update':
  command => '/usr/bin/apt-get update'
}

package { 'apache2':
  require => Exec['apt-update'],
  ensure => installed
}

service { 'apache2':
  ensure => running
}

package { 'php5'':
  require => Exec['apt-update'],
  ensure => installed
}

package { '/var/www/html/info.php':
  ensure => file,
  content => '<?php phpinfo(); ?>',
  require => Package['apache2'],
}
```

Listing 2: Another example of a Puppet manifest

Another example of a manifest is shown in Listing 2. This manifest file defines an exec resource, which contains the command this resource will run if needed, it defines two package resources that instruct the system to install packages "apache2" and "php5" and requires the previous exec resource to be executed before the installation of the packages, it defines a service resource that instructs the system that the service "apache2" should be in running state and finally it defines a file resource that instructs the system that this file should exist, it declares the content of this file and requires that the pack "apache2" should be installed before the creation of the file.

3.3 Detected Quality Problems

As mentioned earlier, IaC practices treat configuration code similar to the production code. As production code should follow certain standards in order to keep its maintainability design quality, so does the configuration code. For this reason, we defined a set of quality problems that can be encountered in Puppet code [27]. To achieve that, we studied available resources that highlight the best practices to be followed for configuration code such as the Puppet style guide [24] and rules implemented by puppet-lint.[3] In traditional software engineering practices, quality problems are classified as implementation/code problems, design problems and architectural problems. Based on that, we focused on two of these categories, so we divided the set of problems (or smells) that we defined into two subsets: implementation configuration problems and design configuration problems. To detect these problems, we implemented a static analysis tool of Puppet files, Puppeteer.

3.3.1 Implementation Configuration Problems

Implementation configuration smells are quality issues such as naming convention, style, formatting and indentation in configuration code. We list these problems with a brief description for each and the metrics that we compute to detect them.

- **Missing Default Case:** A default case is missing in a case or selector statement.
  - Case and Selector statements should have a default case otherwise we detect the smell.
- **Inconsistent Naming Convention:** The used naming convention deviates from the recommended naming convention.
  - The names of classes and variables should not contain a dash otherwise we detect the smell.
- **Complex Expression:** A program contains a difficult to understand complex expression.
  - The smell is detected if we find more than two logical operators in a conditional expression.
- **Duplicate Entity:** Duplicate parameters present in the configuration code.
  - We detect the smell if we find duplicate parameters in the code.
- **Misplaced Attribute:** Attribute placement within a resource or a class has not followed a recommended order (for example, mandatory attributes should be specified before the optional attributes).
  - The smell is detected if an “ensure” attribute is not the first attribute specified or if required parameters for a class or ‘define’ are not listed before optional parameters.
- **Improper Alignment:** The code is not properly aligned (such as all the arrows in a resource declaration) or tabulation characters are used.
  - The smell is detected if right-to-left chaining arrows are found, indentation of arrows is not properly aligned (arrows are not all placed one space ahead of the longest attribute) or tabulation characters are found.

[3] `http://puppet-lint.com/`

- **Invalid Property Value:** An invalid value of a property or attribute is used (such as a file mode specified using 3-digit octal value rather than 4-digit).
  - The smell is detected if file modes are not represented by a valid 4-digit octal value (rather than 3) or symbolically.
- **Incomplete Tasks:** The code has "fixme" and "todo" tags indicating incomplete tasks.
  - The smell is detected if "fixme" and "todo" tags are found in the code.
- **Deprecated Statement Usage:** The configuration code uses one of the deprecated statements (such as "import").
  - The smell is detected if an "import" statement is found in the code.
- **Improper Quote Usage:** Single and double quotes are not used properly. For example, boolean values should not be quoted and variable names should not be used in single quoted strings.
  - The smell is detected if we find double quoted strings that contain no variables, unquoted file modes, quoted boolean values, strings that contain only a single variable, unquoted resource titles or single quoted string s that contain a variable.
- **Long Statement:** The code contains long statements (that typically do not fit in a screen).
  - The smell is detected if a line has more than 80 characters.
- **Incomplete Conditional:** An "if . . . elseif" construct used without a terminating "else" clause.
  - The smell is detected if an "if . . . elseif" construct terminates without an "else" clause.
- **Unguarded Variable:** A variable is not enclosed in braces when being interpolated in a string.
  - The smell is detected when a variable which is not enclosed in braces ({}) is found.

A cumulative example of a manifest that contains all the implementation configuration smells and their respective detection is shown in Figure 3.

3.3.2 Design configuration problems

Design configuration smells reveal quality issues in the module design or structure of a configuration project. We list these problems with a brief description for each and the metrics that we compute to detect them, along with an example.

- **Multifaceted Abstraction:** Each abstraction (e.g., a resource, class, "define", or module) should be designed to specify the properties of a single piece of software. In other words, each abstraction should follow the single responsibility principle [20]. An abstraction suffers from multifaceted abstraction when the elements of the abstraction are not cohesive. The smell may occur in the following two forms:
  - a resource (file, package, or service) declaration specifies attributes of more than one physical resources, or
  - all the language elements declared in a class, "define", or a module are not cohesive.

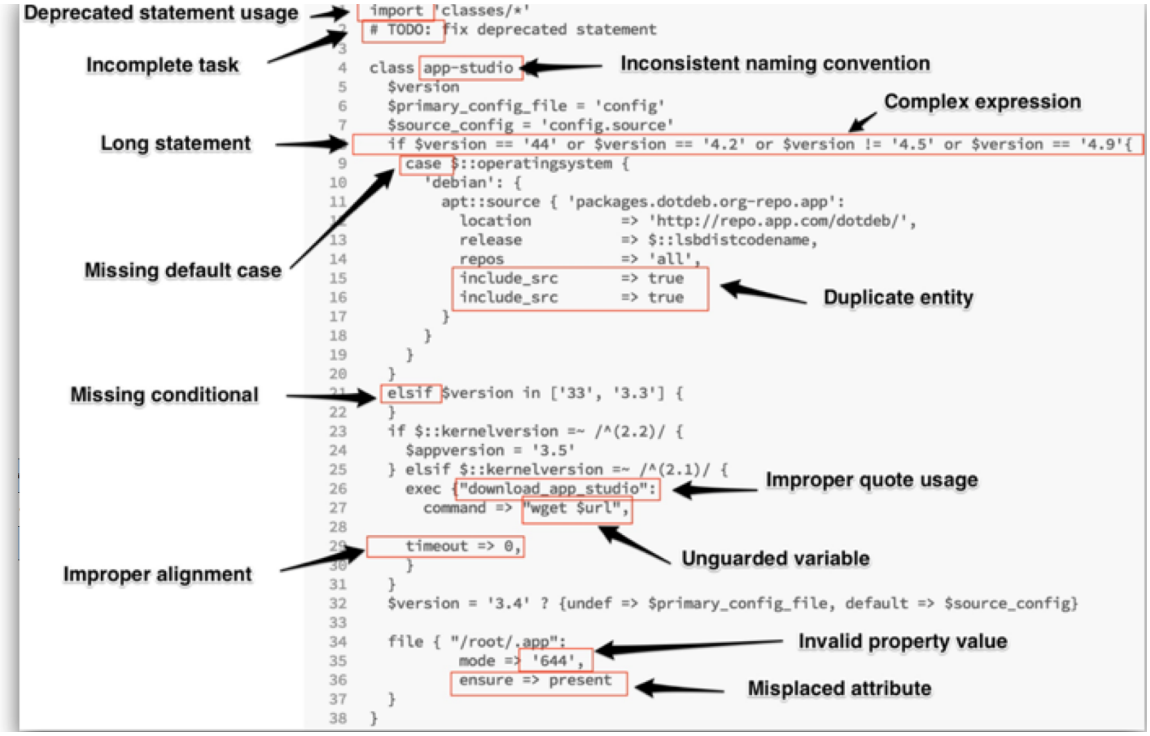

Figure 1: A manifest where all the implementation configuration smells are detected

The detection strategy for the two forms of the smell is as follows.

1. We compute a metric, “Physical resources defined per resource declaration”, for each declared resource. We report the smell when the metric value is more than one.
2. We compute lack of cohesion for the configuration abstractions to detect the second form of the smell. Traditional software engineering uses the LCOM (Lack of Cohesion Of Methods) [6] metric to compute lack of cohesion for an abstraction. The same metric cannot be used for configuration code due to its different structure and characteristics. We use the following algorithm to compute LCOM in a configuration code abstraction.
   (a) Consider each declared element (such as file, package, service resources and exec statements) as a node in a graph. Initially, the graph contains the disconnected components (DC) equal to the number of elements.
   (b) Identify the parameters of the abstraction, used variables, and literals (such as file name). Call them as data members collectively.
   (c) For each data member, repeat the following: identify the components that use the data member. Merge the identified components in a single component.
   (d) Compute LCOM: Note that we compute LCOM for each class, “define”, and file. Therefore, it is quite possible that the metric reports more than one instance of this smell in a single Puppet file

$$LCOM = \begin{cases} \frac{1}{|DC|}, & \text{if } DC > 0 \\ 1, & \text{otherwise} \end{cases}$$

Example:

```
1    package {['libssl', 'apache2']:
2       ensure => installed
3    }
```

- **Unnecessary Abstraction:** A class, "define", or module must contain declarations or statements specifying the properties of a desired system. An empty class, "define", or module shows the presence of unnecessary abstraction smell and thus must be removed.
  - We compute a metric namely "Size of the abstraction body". A zero value of the metric shows that the abstraction does not contain any declarations and thus suffers from unnecessary abstraction smell.

  Example:

```
1    class apache {
2
3    }
```

- **Imperative Abstraction:** Puppet is declarative in nature. The presence of imperative statements (such as "exec") defies the purpose of the language. An abstraction containing numerous imperative statements suffers from imperative abstraction smell.
  - We compute a metric namely 'Total "exec" declarations' in a given abstraction. The tool reports the imperative abstraction smell when the abstraction has more than two "exec" declarations and ratio of the "exec" declarations against all the elements in the abstraction is more than 20%.

```
1    class install {
2      # This can be replaced with the "package{"apache2"...}" resource.
3      exec {'Install Apache':
4         command => 'apt-get install apache2'
5      }
6    }
```

- **Missing Abstraction:** Resource declarations and statements are easy to use and reuse when they are encapsulated in an abstraction such as a class or "define". A module suffers from the missing abstraction smell when resources and language elements are declared and used without encapsulating them in an abstraction.
  - We identify total number of configuration elements except classes or defines that are not encapsulated in a class or "define". A module suffers from the smell if there are more than two such elements in the module.

```
1    # All the following resources are declared at top-level.
2    # We could define either a custom abstraction through "define"
        primitive or a class.
3    package {'apache2':
4       ensure => installed
5    }
```

```
6
7     file {'/etc/apache2/apache2.conf':
8        ensure => file,
9        require => Package['apache2']
10    }
11
12    service {'apache2':
13       ensure => running,
14       subscribe => [
15          Package['apache2'],
16          Filep['/etc/apache2/apache2.conf']
17       ]
18    }
```

- **Insufficient Modularization:** An abstraction suffers from this smell when it is large or complex and thus can be modularized further. This smell arises in following forms:
  - if a file contains a declaration of more than one class or "define", or
  - if the size of a class declaration is crossing a certain threshold, or
  - the complexity of a class or "define" is high.

  The detection strategy for the three forms of the smell is as follows:

  1. We count the number of classes and defines declared in a manifest file. We report the smell if a file defines more than one class and 'define'.
  2. We count the number of lines in an abstraction. If a class or 'define' contains more than 40 lines of code, it suffers from the smell.
  3. We compute maximum nesting depth for an abstraction. An abstraction with maximum nesting depth more than three suffers from this smell.

```
1     class install_apache {
2        package{'apache2':
3          ensure => installed
4        }
5     }
6
7     class install_ntp {
8        package{'ntp':
9          ensure => installed
10       }
11    }
```

- **Duplicate Block:** A duplicate block of statements indicates that probably a suitable abstraction definition is missing. Thus a module containing such a duplicate block suffers from duplicate block smell.
  - We use the pmd-cpd[4] tool to identify code clones. A module suffers from this smell when a code clone of larger than 150 tokens gets identified in the module.

[4] https://pmd.github.io

```
1  # To configure every apache module, we declare a "file" resource.
2  # To remove duplicate blocks, we could define a custom resource that
3  # can be parameterized with the content and the name of each apache
       module.
4  file {'/etc/apache2/modules-available/foo.mod':
5     ensure => file,
6     mode => 0777,
7     content => "Content of the module foo"
8  }
9
10 file {'/etc/apache2/modules-available/bar.mod':
11    ensure => file,
12    mode => 0777,
13    content => "Content of the module bar"
14 }
15
16 file {'/etc/apache2/modules-available/baz.mod':
17    ensure => file,
18    mode => 0777,
19    content => "Content of the module baz"
20 }
21
22 file {'/etc/apache2/modules-available/qux.mod':
23    ensure => file,
24    mode => 0777,
25    content => "Content of the module qux"
26 }
```

- **Broken Hierarchy:** The use of inheritance must be limited to the same module. The smell occurs when, the inheritance is used across namespaces where inheritance is not natural ("is-a" relationship is not followed).
  - For all the class definitions, we identify the inherited class (if any). If the inherited class is defined in any other module, the class suffers from "broken hierarchy" smell.

```
1  class apache_setup inherits apache {
2     # The class apache is defined in another module (i.e., puppetlabs-
          apache)
3  }
```

- **Unstructured Module:** Each module in a configuration repository must have a well-defined and consistent module structure. A recommended structure for a module is the following:

```
Module name:
  manifests
  files
  templates
  lib
  facts.d
  examples
  spec
```

An ad-hoc structure of a repository suffers from unstructured module smell that impacts understandability and predictability of the repository.

The detection strategy for the three forms of the smell is as follows:

1. We search for a folder named “manifests” in the root folder of the repository. If the total number of Puppet files in the folder is more than five while there is no folder containing the string “modules”, the smell is detected.
2. We find a folder containing the string “modules” and treat all the sub-folders as separate modules. Each module must have a folder named “manifests”. Absence of the folder shows the presence of the smell.
3. In each module, we count the unexpected files and folders. Expected files and folders are: “manifests”, “files”, “templates”, “lib”, “tests”, “spec”, “readme”, “license”, and “metadata”. A module with more than three such unexpected files or folders suffers from the smell.

- **Dense Structure:** This smell arises when a configuration code repository has excessive and dense dependencies without any particular structure.
  - We prepare a graph for each repository to detect the smell. Each module is treated as a node and any reference from the module to another module is treated as an edge. We, then compute average degree of the graph.

$$AvgDegree(G) = \frac{2 \times |E|}{V}$$

    where $|E|$ and $|V|$ are number of edges and nodes respectively. A graph more than 0.5 average degree suffers from Dense structure smell.

- **Deficient Encapsulation:** This smell arises when a node definition or External Node Classifier (ENC) declares a set of global variables to be picked up by the included classes in the definition.
  - We count the number of global variables declared for each node declaration, followed by at least one include statement. If a node declaration has one or more such global variables, the module suffers from deficient encapsulation smell.

```
1   # The file site.pp
2   node default {
3      $conf_content = "Configuration content..."
4      include apache_setup
5   }
6
7   # The file setup.pp
8   class apache_setup {
9      package {'apache2':
10        ensure => installed
11     }
12
13     file {'/etc/apache2/apache2.conf':
14        ensure => file,
15        require => Package['apache2'],
16        content => $conf_content
17     }
```

```
18
19          service {'apache2':
20              ensure => running,
21              subscribe => [
22                Package['apache2'],
23                Filep['/etc/apache2/apache2.conf']
24              ]
25          }
26      }
```

- **Weakened Modularity:** Each module must strive for high cohesion and low coupling. This smell arises when a module exhibits high coupling and low cohesion.
  - We compute modularity ratio [5] for each module as follows:

$$ModularityRatio(A) = \frac{Cohesion(A)}{Coupling(A)}$$

    where, $Cohesion(A)$ refers to the number of intramodule references and $Coupling(A)$ refers to the number of inter-module references from module A. We report the smell if the ratio is less than one.

4 Docker Configuration Code Analysis

In this section we focus on the analysis of Dockerfiles, the smells that are detected and the metrics that are produced based on them.

4.1 Docker Background

Containerisation is a increasingly popular lightweight virtualisation technology that allows the definition of software infrastructure. Containers allow to package an application with its dependencies and execution environment into a standardised, self-contained unit and to run the packaged application on any system. The leader of this field and the most widely used -by both software engineers and DevOps specialists- containerization technology is Docker. Its aim is the improvement of reproducibility of software by the deployment of a single object across different platforms.

Docker is developed primarily for Linux, where it uses the resource isolation features of the Linux kernel. This isolation is achieved by two main kernel features, kernel namespaces and control groups (cgroups) and allows independent Docker containers creating wrapped and controlled environments on the host machine, where their code can be run. In contrast with virtual machines, containers do not require or include another operating system, but rather share the underlying kernel with the host. This minimizes the overhead that is created by the use of virtual machines.

The two main objects of Docker are containers and images. Containers provides operating-system-level virtualization by abstracting the “user space”. They are standard units of software that package up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. They include everything needed to run an application: code, runtime, system tools, system libraries and settings. Docker images are the basis of containers. Images become containers at runtime, thus an instance of an image is called a container. Images are stored in registries, which essentially are repository for Docker images, like

Docker Hub. Images can be changed and committed to these registries and have multiple versions. Docker images have intermediate layers that increase reusability, decrease disk usage and speed up the building of containers by allowing each step to be cached. Images are divided in two types. The base images and the child images. Base images are images that have no parent image, usually images containing an operating system. Child images are images that build on base images and add additional functionality.

4.2 Analysis of Docker System Configuration Files

Docker builds images automatically by reading the instructions from a Dockerfile. Dockerfile is a text file that contains all commands, in order, needed to build a given image. It is similar with Makefile, as both are are used in order to automate building. Dockerfiles automate the building of docker images while Makefiles automate building of executable programs and libraries. A Dockerfile adheres to a specific format and set of instructions. A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer.

An example of a Dockerfile is shown in Figure 4.1. It contains four commands and each of them creates a layer. The FROM instruction creates a layer from the ubuntu:18.04 image. The COPY adds a folder and its contents from the Docker client's current directory. The RUN instruction calls make command in order to build the application that exists into the folder. Finally, the last layer is specified by the CMD instruction, which defines that the executable, created by the previous command, will be executed.

The instructions that can be included in a Dockerfile, along with a description for each, are the following:

- The FROM instruction initialises a new build stage and sets the Base Image for the Dockerfile. As such, a valid Dockerfile must start with a FROM instruction.
- RUN creates a layer at build-time by running a specified command. Docker commits the state of the image after each RUN.
- CMD provides Docker a command to run when a container is started. It provides defaults for an executing container and there can be only one such instruction in a Dockerfile.
- The LABEL instruction adds metadata to an image, like the creator or the version of the Dockerfile.
- The EXPOSE instruction is used to declare on which port the container listens at runtime.
- ENV sets a persistent environment variable that is available at container run time.
- The ADD instruction copies new files, directories or remote file URLs and adds them to the filesystem of the image at a specified path.
- The COPY instruction tells Docker to copy the files and folders in the local build context and add them to the Docker image's current working directory.
- ENTRYPOINT allows the configuration of a container that will run as an executable by providing a default command and arguments when the container starts.
- VOLUME specifies where your container will store and/or access persistent data by creating a mount point with a specified name and marks it as holding externally mounted volumes from native host or other containers.

- The USER instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any following RUN, CMD and ENTRYPOINT instructions.
- The WORKDIR instruction sets the working directory for any following RUN, CMD, ENTRYPOINT, COPY and ADD instructions.
- ARG defines a variable to pass from the command line to the image at build-time.
- The ONBUILD instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build.
- The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit.
- The HEALTHCHECK instruction tells Docker how to test a container to check that it is still working.
- The SHELL instruction allows the default shell used for the shell form of commands to be overridden.

To detect quality and security issues about the use of the aforementioned instructions of Dockerfiles, we implemented Jadolint (Java Dockerfile Linter). Jadolint parses Dockerfiles line by line, detects the instruction that each line contains and examines if certain rules are followed for each type of instruction. Additionally, Jadolint extracts the third-party dependencies that are defined in a Dockerfile. Jadolint is based on Hadolint (Haskell Dockerfile Linter). We decided to implement our tool in Java and not utilize Hadolint because the integration of a Java based tool is smoother in CROSSMINER Platform than the integration of a Haskell application and the freedom that our own implementation gives us in order to extend our tool, like we did for the dependencies extraction. Also, we can add our own smell detection rules in Jadolint.

4.3 Detected Quality Problems

The smells that we detect in Dockerfiles are based mostly on the best practices for writing Dockerfiles [8]. They cover both security and quality issues and they are the following:

- **Use COPY instead of ADD for files and folders.**
  - For other items (files, directories) that do not require ADD's tar auto-extraction capability, you should always use COPY.

    Problematic code:
    FROM python:3.4
    ADD requirements.txt /usr/src/app/

- **Use arguments JSON notation for CMD and ENTRYPOINT arguments.**
  - When using the plain text version of passing arguments, signals from the OS are not correctly passed to the executables, which is in the majority of the cases what you would expect.

    Problematic code:
    FROM busybox
    CMD my-service server

- **Use ADD for extracting archives into an image.**
  - Although ADD and COPY are functionally similar, generally speaking, COPY is preferred for extracting archives, as its more transparent than ADD.

    Problematic code:
    COPY rootfs.tar.xz /.

- **COPY with more than 2 arguments requires the last argument to end with /**
  - If multiple resources are specified, either directly or due to the use of a wildcard, then must be a directory, and it must end with a slash /

    Problematic code:
    FROM node:carbon
    COPY package.json yarn.lock myApp

- **COPY –from should reference a previously defined FROM alias.**
  - Trying to copy from a missing image alias results in an error.

    Problematic code:
    FROM debian:jesse
    RUN stuff
    FROM debian:jesse
    COPY –from=build some stuff ./

- **COPY –from cannot reference its own FROM alias.**
  - Trying to copy from the same image the instruction is running in results in an error.

    Problematic code:
    FROM debian:jesse as build
    COPY –from=build some stuff ./

- **Valid UNIX ports range from 0 to 65535.**
  - This is the specified range of ports in TCP and UDP protocols.

    Problematic code:
    FROM busybox
    EXPOSE 80000

- **Always tag the version of an image explicitly.**
  - User can never rely that the latest tags is a specific version.

    Problematic code:
    FROM debian

- **Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag.**
  - User can never rely that the latest tags is a specific version.

    Problematic code:
    FROM debian:latest

- **FROM aliases (stage names) must be unique.**
  - Defining duplicate stage names results in an error.

    Problematic code:
    FROM debian:jesse as build
    RUN stuff
    FROM debian:jesse as build
    RUN moreStuff

- **Command does not make sense in a container.**
  - For some POSIX commands it makes no sense to run them in a Docker container because they are bound to the host or are otherwise dangerous (like `´shutdown´`, `´service´`, `´ps´`, `´free´`, `´top´`, `´kill´`, `´mount´`, `´ifconfig´`). Interactive utilities also don't make much sense.

Problematic code:
FROM busybox
RUN top

- **Use WORKDIR to switch to a directory.**
  - Only use cd in a subshell. Most commands can work with absolute paths and it in most cases not necessary to change directories. Docker provides the WORKDIR instruction if you really need to change the current working directory.

    Problematic code:
    FROM busybox
    RUN cd /usr/src/app && git clone git@github.com:crossminer/scava.git

- **Do not use sudo.**
  - Do not use sudo as it leads to unpredictable behavior. Use a tool like gosu to enforce root.

    Problematic code:
    FROM busybox
    RUN sudo apt-get install

- **Do not use apt-get upgrade or dist-upgrade.**
  - User should avoid RUN apt-get upgrade or dist-upgrade, as many of the "essential" packages from the base images won't upgrade inside an unprivileged container. If a package contained in the base image is out-of-date, you should contact its maintainers.

    Problematic code:
    FROM debian
    RUN apt-get update && apt-get upgrade

- **Pin versions in apt get install.**
  - Version pinning forces the build to retrieve a particular version regardless of what's in the cache. This technique can also reduce failures due to unanticipated changes in required packages.

    Problematic code:
    FROM busybox
    RUN apt-get install python

- **Delete the apt-get lists after installing something.**
  - Cleaning up the apt cache and removing /var/lib/apt/lists helps keep the image size down. Since the RUN statement starts with apt-get update, the package cache will always be refreshed prior to apt-get install.

    Problematic code:
    RUN apt-get update && apt-get install -y python

- **Pin versions in pip.**
  - Version pinning forces the build to retrieve a particular version regardless of what's in the cache. This technique can also reduce failures due to unanticipated changes in required packages.

    Problematic code:
    FROM python:3.4
    RUN pip install django
    RUN pip install https://github.com/Banno/carbon/tarball/0.9.x-fix-events-callback

- **Use the -y switch.**
  - Without the -y option it might be possible that the build breaks without human intervention.

    Problematic code:
    FROM debian
    RUN apt-get install python=2.7

- **Avoid additional packages by specifying –no-install-recommends.**
  - Avoid installing additional packages that you did not explicitly want.

    Problematic code:
    FROM busybox
    RUN apt-get install -y python=2.7

- **Do not use apk upgrade.**
  - User should avoid RUN apk upgrade, as many of the "essential" packages from the parent images won't upgrade inside an unprivileged container.

    Problematic code:
    FROM alpine:3.7
    RUN apk update && apk upgrade && apk add foo=1.0 && rm -rf /var/cache/apk/

- **Pin versions in apk add.**
  - Version pinning forces the build to retrieve a limited range of versions, or an exact particular version, regardless of what's in the cache. This technique can also reduce failures due to unanticipated changes in required packages.

    Problematic code:
    FROM alpine:3.7
    RUN apk –no-cache add foo

- **Use the –no-cache switch.**
  - As of Alpine Linux 3.3 there exists a new –no-cache option for apk. It allows users to install packages with an index that is updated and used on-the-fly and not cached locally: This avoids the need to use –update and remove /var/cache/apk/ when done installing packages.

    Problematic code:
    FROM alpine:3.7
    RUN apk add –update foo=1.0 && rm -rf /var/cache/apk/

- **Use absolute WORKDIR.**
  - By using absolute paths user will not run into problems when a previous WORKDIR instruction changes. User also often do not know the WORKDIR context of their base container.

    Problematic code:
    FROM busybox
    WORKDIR usr/src/app

New dependencies versions

Dependency	Current version	New version	datetime per month
curl	5.0.0	7.64.0	2019-06-01
nginx	0.0.0	1.15.9	2019-06-01
build-essential	4.0.0	12.6ubuntu1	2019-06-01
gcc	7.0.0	8.3.0	2019-06-01
python-dev	0.0.0	2.7.16	2019-06-01
vim	3.0.0	8.1.0320	2019-06-01

Figure 2: A dashboard view about new versions of third-party components

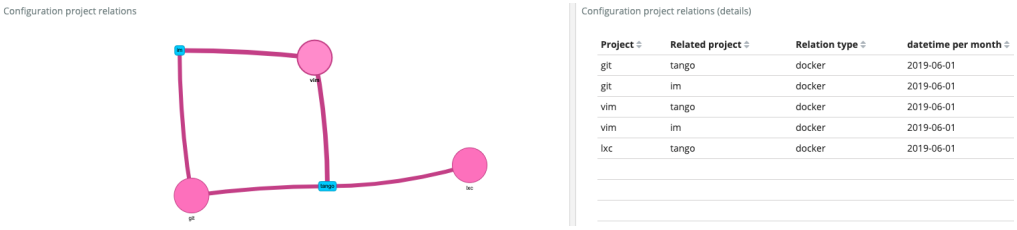

Figure 3: A dashboard view about the relations between handled projects and a graph representation of them

5 New Versions and Project Relations Components

Apart from the two main tools that we implemented, we created two additional components in order to address the remaining requirements of the System Configuration Code Analyser. The first one detects if there are new versions of the dependencies that are detected form Puppeteer and Jadolint. This component checks for new versions of dependencies extracted from the tools that were implemented in the context of WP2 about Maven and OSGi. The component queries the Maven Central Repository through its API to detect new versions of libraries declared in Maven and OSGi projects. About Puppet and Docker based projects, we use the APT package manager to find if there are new candidate versions of the detected dependencies.

The second component detects if an already analysed project is declared from other analysed projects of the platform as third party dependency. It achieves that by searching all the dependencies of the other already analysed projects.

An example of how new versions of third-party components is shown though the DevOps Dashboard of CROSSMINER platform, is depicted in Figure 2. In Figure 3, the relations between handles projects are shown along with a graph representation of them.

6 CROSSMINER System Configuration Code Analyser Integration

Both Puppeteer and Jadolint are integrated in the CROSSMINER Platform. Based on them, we implemented metric providers as part of the Metric Platform of CROSSMINER, as it is shown in Figure 4 and as is described in Deliverable D8.1. As every other metric provider of the metric platform, our providers are automatically scheduled and invoked by CROSSMINER platform on a daily basis. They compute their metrics by running through the Working Copies of the repositories of each imported project of the platform and analyse Puppet files or Dockerfiles, if any.

The metrics are divided in two parts: the transient metrics and the historic metrics. Our historic metric providers reside under the org.eclipse.scava.metricprovider.historic.configuration.docker.* and org.eclipse.scava.metricprovider.historic.configuration.puppet.* packages. The metrics, along with their description and their json representation as the plaform's API produces it, are the following:

- org.eclipse.scava.metricprovider.historic.configuration.puppet.designsmells

```
{
  id: "puppet.design.smells",
  projectId: "puppetpython",
  metricId: "org.eclipse.scava.metricprovider.historic.
     configuration.puppet.designsmells",
  name: "Sum of Puppet Design Smells",
  description: "The number of Puppet design smells per day, up
     to and including the current date.",
  type: "LineChart",
  datatable: [
    {
      Date: "20190401",
      DesignSmells: 12
    },
    {
      Date: "20190402",
      DesignSmells: 10
    },
    ...
}
```

- org.eclipse.scava.metricprovider.historic.configuration.puppet.implementationsmells

```
{
  id: "puppet.implementation.smells",
  projectId: "puppetpython",
  metricId: "org.eclipse.scava.metricprovider.historic.
     configuration.puppet.implementationsmells",
  name: "Sum of Puppet Implementation Smells",
  description: "The number of Puppet implementation smells per
     day, up to and including the current date.",
```

```
    type: "LineChart",
    datatable: [
      {
        Date: "20190401",
        ImplementationSmells: 14
      },
      {
        Date: "20190402",
        ImplementationSmells: 16
      },
      ...
  }
```

- org.eclipse.scava.metricprovider.historic.configuration.docker.smells

```
  {
    id: "docker.smells",
    projectId: "tango",
    metricId: "org.eclipse.scava.metricprovider.historic.
       configuration.docker.smells",
    name: "Sum of Docker Smells",
    description: "The number of Docker smells per day, up to and
       including the current date.",
    type: "LineChart",
    datatable: [
      {
        Date: "20190102",
        DockerSmells: 34
      },
      {
        Date: "20190103",
        DockerSmells: 34
      },
      ...
  }
```

The above metrics are the historic metrics about Puppet and Docker smells. They show the evolution of the number of smells of an analysed project through time and allows the user to determine if the quality of the configuration code of the projects improves or deteriorates as time passes.

- org.eclipse.scava.metricprovider.trans.configuration.puppet.designsmells

```
  [
    {
      _id: "d9ec7498-0226-40ca-84b0-b9781d4958e2",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.designsmells.model.Smell",
      smellName: "Multifaceted Abstraction - Form 2",
```

```
      reason: "Define",
      fileName: "/manifests/pip.pp"
    },
    {
      _id: "ed276cd8-5854-4670-a165-305cf2915529",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.designsmells.model.Smell",
      smellName: "Multifaceted Abstraction - Form 2",
      reason: "Define",
      fileName: "/manifests/requirements.pp"
    },
    ...
  ]
```


- org.eclipse.scava.metricprovider.trans.configuration.puppet.implementationsmells


```
  [
    {
      _id: "803087a0-0d8f-4d49-b4c7-2add1fbf60ec",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.implementationsmells.model.Smell",
      line: "207",
      reason: "line has more than 140characters ",
      fileName: "/manifests/pip.pp "
    },
    {
      _id: "33dee7bc-cd46-4406-84e2-29f1a639727b",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.implementationsmells.model.Smell",
      line: "50",
      reason: "python::pip not in autoload module layout ",
      fileName: "/manifests/pip.pp ",
      smellName: "Inconsistent naming convention"
    },
    ...
  ]
```


- org.eclipse.scava.metricprovider.trans.configuration.docker.smells


```
  [
    {
      _id: "b4e2b3ba-8d0a-44c7-a37c-232b64fe03f3",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.docker.smells.model.Smell",
      smellName: "Improper Upgrade",
      reason: "Delete the apt-get lists after installing
         something",
```

```
        code: "DL3009",
        fileName: "/Dockerfile",
        line: "6"
      },
      {
        _id: "3b931d4a-abd5-4b9f-8225-2095be26a5da",
        _type: "org.eclipse.scava.metricprovider.trans.
           configuration.docker.smells.model.Smell",
        smellName: "Improper Upgrade",
        reason: "Delete the apt-get lists after installing
           something",
        code: "DL3009",
        fileName: "/Dockerfile",
        line: "6"
    },
    ...
  ]
```


The above metrics are the transient metrics about Puppet and Docker smells. They contain actual information about the smells detected over the last day of the analysed project. The information that they contain includes the type of smell detected, the file and line number where it is located and the reason why this has been identified as a smell. They allow the user to examine the current state of the configuration code of the analysed project.

- org.eclipse.scava.metricprovider.historic.configuration.puppet.dependencies


```
  {
    id: "puppet.dependencies",
    projectId: "puppetpython",
    metricId: "org.eclipse.scava.metricprovider.historic.
       configuration.puppet.dependencies",
    name: "Sum of Puppet Dependencies",
    description: "The number of Puppet dependencies per day, up
       to and including the current date.",
    type: "LineChart",
    datatable: [
      {
        Date: "20190102",
        PuppetDependencies: 20
      },
      {
        Date: "20190103",
        PuppetDependencies: 20
      },
      ...
  }
```

- org.eclipse.scava.metricprovider.historic.configuration.docker.dependencies

```
{
  id: "docker.dependencies",
  projectId: "tango",
  metricId: "org.eclipse.scava.metricprovider.historic.
     configuration.docker.dependencies",
  name: "Sum of Docker Dependencies",
  description: "The number of Docker dependencies per day, up
     to and including the current date.",
  type: "LineChart",
  datatable: [
    {
      Date: "20190102",
      DockerDependencies: 12
    },
    {
      Date: "20190103",
      DockerDependencies: 15
    },
    ...
}
```

These metrics are the historic metrics about the number of dependencies of Puppet and Docker based projects. They show the evolution of the number of dependencies of an analysed project through time.

- org.eclipse.scava.metricprovider.trans.configuration.puppet.dependencies

```
[
  {
    _id: "f3fbfc86-f602-4c01-b78c-d91c87b5c8aa",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.puppet.dependencies.model.PuppetDependency
       ",
    dependencyName: "python-dev",
    dependencyVersion: "N/A",
    type: "puppet"
  },
  {
    _id: "f9bc3e06-cfab-430e-a6b1-ba0a138c18e2",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.puppet.dependencies.model.PuppetDependency
       ",
    dependencyName: "nginx",
    dependencyVersion: "N/A",
    type: "puppet"
  },
  ...
]
```

- org.eclipse.scava.metricprovider.trans.configuration.docker.dependencies


```
[
  {
    _id: "b1c4bcfb-3fb0-4c4f-82f9-56e4560bd98d",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.docker.dependencies.model.DockerDependency
       ",
    dependencyName: "ubuntu",
    dependencyVersion: "18.04",
    type: "docker",
    subType: "image"
  },
  {
    _id: "c1a92582-7af6-4801-9e3d-d4f2f6fae354",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.docker.dependencies.model.DockerDependency
       ",
    dependencyName: "gcc",
    dependencyVersion: "9.1",
    type: "docker",
    subType: "package"
  },
  ...
]
```


These metrics are the transient metrics about the dependencies detected in Puppet and Docker based projects. They give information about the name and the version (where available) of the dependencies.

- org.eclipse.scava.metricprovider.trans.configuration.puppet.designantipatterns


```
[
  {
    _id: "f92dbf8c-49ba-40c8-9cac-dc8437b62aa8",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.puppet.designantipatterns.model.
       DesignAntipattern",
    smellName: "Multifaceted Abstraction - Form 2",
    reason: "Define",
    fileName: "/manifests/pip.pp",
    commit: "7eae51bd3447c9db9317ea081ef31db2180dd610",
    date: {
      $date: "2019-04-03T13:01:30.000Z"
    }
  },
  {
    _id: "2a81310b-227a-4235-88ee-0d060cdd0d22",
```

```
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.designantipatterns.model.
         DesignAntipattern",
      smellName: "Multifaceted Abstraction - Form 2",
      reason: "Define",
      fileName: "/manifests/requirements.pp",
      commit: "7eae51bd3447c9db9317ea081ef31db2180dd610",
      date: {
        $date: "2019-04-03T13:01:30.000Z"
      }
    },
    ...
  ]
```

- org.eclipse.scava.metricprovider.trans.configuration.puppet.implementationantipatterns

```
  [
    {
      _id: "2df77867-ebd2-41e3-8186-89f99e3d3c76",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.implementationantipatterns.model.
         ImplementationAntipattern",
      line: "178",
      reason: "line has more than 140characters ",
      fileName: "/manifests/pip.pp ",
      commit: "7eae51bd3447c9db9317ea081ef31db2180dd610",
      date: {
        $date: "2019-04-03T13:01:30.000Z"
      }
    },
    {
      _id: "cb9d03c0-c1b7-4945-8642-27fb6aec3831",
      _type: "org.eclipse.scava.metricprovider.trans.
         configuration.puppet.implementationantipatterns.model.
         ImplementationAntipattern",
      line: "187",
      reason: "line has more than 140characters ",
      fileName: "/manifests/pip.pp ",
      commit: "7eae51bd3447c9db9317ea081ef31db2180dd610",
      date: {
        $date: "2019-04-03T13:01:30.000Z"
      }
    },
    ...
  ]
```

- org.eclipse.scava.metricprovider.trans.configuration.docker.antipatterns


```
[
  {
    _id: "2df77867-ebd2-41e3-8186-89f99e3d3c76",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.docker.antipatterns.model.
       DockerAntipattern",
    smellName: "Improper Upgrade",
    reason: "Delete the apt-get lists after installing
       something",
    code: "DL3009",
    fileName: "/Dockerfile",
    line: "6"
    date: {
      $date: "2019-04-03T13:01:30.000Z"
    }
  },
  {
    _id: "cb9d03c0-c1b7-4945-8642-27fb6aec3831",
    _type: "org.eclipse.scava.metricprovider.trans.
       configuration.docker.antipatterns.model.
       DockerAntipattern",
    smellName: "Improper Upgrade",
    reason: "Delete the apt-get lists after installing
       something",
    code: "DL3009",
    fileName: "/Dockerfile",
    line: "17"
    date: {
      $date: "2019-04-03T13:01:30.000Z"
    }
  },
  ...
]
```


The above metrics contain the whole history of the smells of a project and allow the user to examine the smells of the last commit of each day of the project.

Apart from metrics produced by Puppeteer and Jadolint, we provide metrics that come from the New Versions and Project Relations Component. Their metric providers reside in org.eclipse.scava.metricprovider.trans.newversion.* package range and in org.eclipse.scava.metricprovider.trans.configuration.projects.relations package respectively.

- org.eclipse.scava.metricprovider.trans.newversion.docker


```
[
  {
    _id: "8b323b25-3a9d-4c1d-b438-fea0f3c9a80b",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "gcc",
    oldVersion: "7.0.0",
    newVersion: "8.3.0"
  },
  {
    _id: "1f2619b4-04cb-4bf3-ac58-226e390934c4",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "build-essential",
    oldVersion: "11.4",
    newVersion: "12.6ubuntu1"
  },
  ...
]
```


- org.eclipse.scava.metricprovider.trans.newversion.maven


```
[
  {
    _id: "8b323b25-3a9d-4c1d-b438-fea0f3c9a80b",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "junit",
    oldVersion: "5.3.1",
    newVersion: "5.4.2"
  },
  {
    _id: "1f2619b4-04cb-4bf3-ac58-226e390934c4",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "scalate",
    oldVersion: "1.2.5",
    newVersion: "1.9.4"
  },
  ...
]
```

- org.eclipse.scava.metricprovider.trans.newversion.osgi


```
[
  {
    _id: "8b323b25-3a9d-4c1d-b438-fea0f3c9a80b",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "junit",
    oldVersion: "5.3.1",
    newVersion: "5.4.2"
  },
  {
    _id: "1f2619b4-04cb-4bf3-ac58-226e390934c4",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "scalate",
    oldVersion: "1.2.5",
    newVersion: "1.9.4"
  },
  ...
]
```


- org.eclipse.scava.metricprovider.trans.newversion.puppet


```
[
  {
    _id: "8b323b25-3a9d-4c1d-b438-fea0f3c9a80b",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "python",
    oldVersion: "3.5.2",
    newVersion: "3.7.3"
  },
  {
    _id: "1f2619b4-04cb-4bf3-ac58-226e390934c4",
    _type: "org.eclipse.scava.metricprovider.trans.newversion.
       docker.model.NewDockerVersion",
    packageName: "nginx",
    oldVersion: "1.14.0",
    newVersion: "1.16.0"
  },
  ...
]
```


These metrics are the metrics about the new versions of dependencies of Docker, Maven, Osgi and Puppet based projects respectively. They contain information about the name of the dependency, the number of the current version and the number of the new version. They inform the user about detected new versions and allow him to determine how up-to-date an project is.

- org.eclipse.scava.metricprovider.trans.configuration.projects.relations

```
[
  {
     _id: "eea0db9c-41bd-42dc-baf0-c1f22f69fe96",
     _type: "org.eclipse.scava.metricprovider.trans.
        configuration.projects.relations.model.ProjectRelation
        ",
     relationName: "tango",
     dependencyType: "docker"
   },
   {
     _id: "5cea8a17-724c-4491-94bc-ab25d7bcc402",
     _type: "org.eclipse.scava.metricprovider.trans.
        configuration.projects.relations.model.ProjectRelation
        ",
     relationName: "git",
     dependencyType: "docker"
  },
  ...
]
```

This metric contains information about the relations between handled projects of the platform. It provides information about the name of the two interconnected projects.

The DevOps Dashboard that is presented in deliverable D4.3 DevOps Dashboard - Final Version, relies on all these metrics in order to produce its visualisations. It incorporates all the smells of the System Configuration Code Analyser and allows the user to browse through them in an intuitive and graphical way.

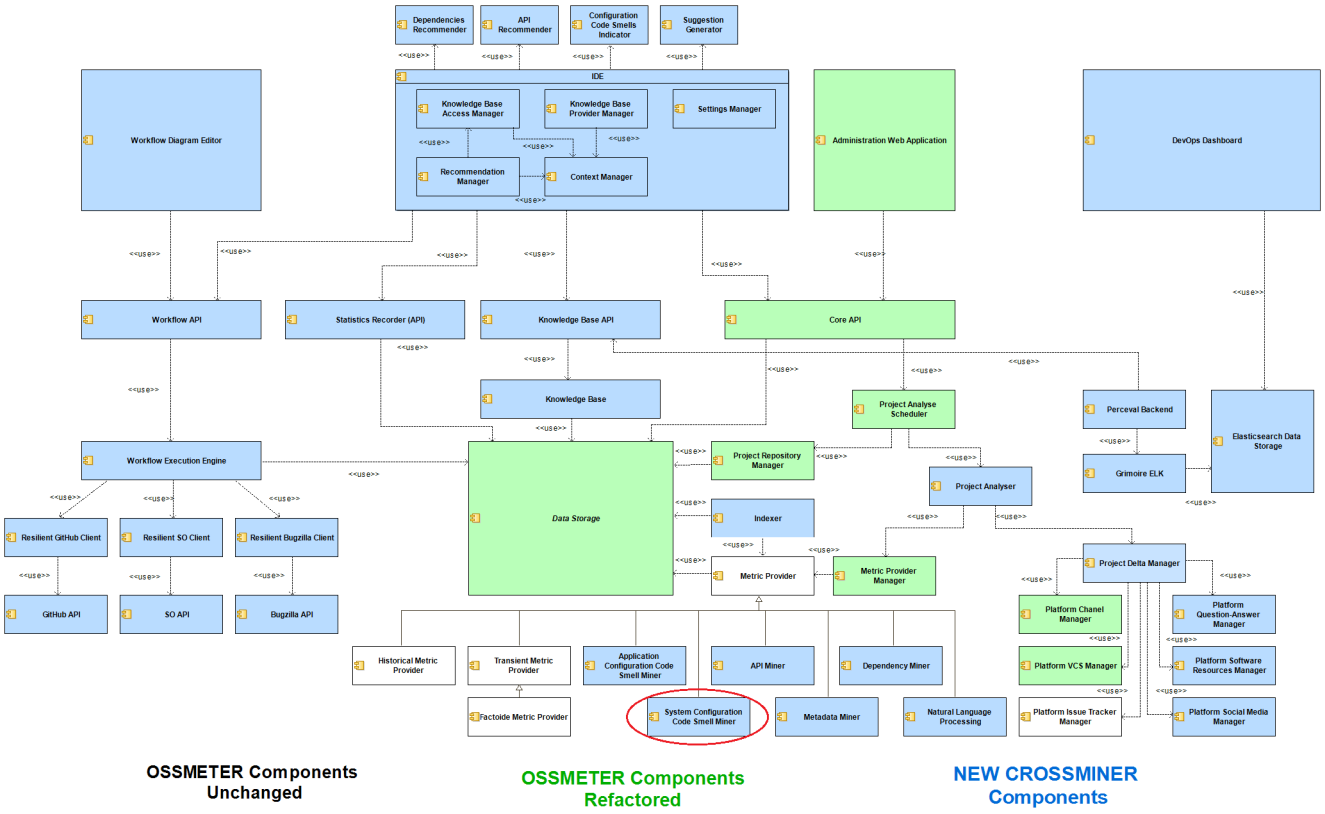

Figure 4: Logical view of the CROSSMINER components. The System Configuration Code Analyser metrics will be part of the System Configuration Code Smell Miner Component

7 Satisfaction of CROSSMINER Requirements

Req. No.	Requirement	Priority	Status
D40	The configuration code analysis tool shall create a directed acyclic graph of all the third party libraries on which a project depends	Shall	Full: The dependency extraction that is carried out by Puppeteer and Jadolint creates this graph in the data storage by the way that it stored there and these data are exploited by the DevOps Dependencies Dashboard for the graph representation that it produces
D42	The configuration code analysis tool shall detect dependencies between handled projects	Shall	Full: It is addressed by the Projects Relations component
U75	Provides a security assessment of the configuration dependencies mechanism/files	Shall	Full: It is addressed by the smell detection of Docker and Puppeteer
U76	Provides a quality assessment of the configuration dependencies mechanism/files	Shall	Full: It is addressed by the smell detection of Docker and Puppeteer
D79	Given a file changes from a commit provides metrics for patterns and antipatterns related to this commit	Shall	Full: It is addressed by the smell detection of Docker and Puppeteer
U80	Able to detect from the configuration settings if a new version of a used third-party library is available	Shall	Full: It is addressed by the New Versions component
U81	Able to identify and list the third-party components used in a project	Shall	Full: It is addressed by the dependency extraction of Docker and Puppeteer

Table 1: Satisfaction of CROSSMINER requirements extracted from **D1.1 – Project Requirements**.

8 Conclusions

In this deliverable we presented the work done regarding the Task 4.2 and the implementation of the System Configuration Code Analyser. We discussed how the notion of IaC, which treats configuration code with the same manner and with similar software engineering techniques, gave us the incentive to develop techniques that can detect quality problems in the configuration code. We implemented two tools, Puppeteer and Jadolint, that analyse Puppet and Dockerfiles. Also, we presented the smells that we detect and the metrics that we provide based on them along with metrics about dependencies. Finally, we described how we integrated the System Configuration Code Analyser into the CROSSMINER platform, in a manner that its results can be exploited from other components of the platform, such as the DevOps Dashboard.

References

[1] Continuous inspection | sonar qube. https://www.sonarqube.org/. [last accessed June 24, 2019].

[2] Puppet forge. https://forge.puppet.com/. [last accessed June 24, 2019].

[3] puppet-lint. http://puppet-lint.com/. [last accessed June 24, 2019].

[4] sonar-puppet. https://github.com/iwarapter/sonar-puppet. [last accessed June 24, 2019].

[5] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429. IEEE, 2012.

[6] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[7] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the docker container ecosystem on github. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 323–333. IEEE, 2017.

[8] Docker. Docker best practices. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, jun 2019. [last accessed June 24, 2019].

[9] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039. IEEE, 2011.

[10] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[11] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *International Conference on the Quality of Software Architectures*, pages 146–162. Springer, 2009.

[12] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[13] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code: An empirical study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 45–55. IEEE Press, 2015.

[14] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[15] Garry Larizza. Building a functional puppet workflow part 1: Module structure. http://garylarizza.com/blog/2014/02/17/puppet-workflow-part-1/. [last accessed June 24, 2019].

[16] Garry Larizza. Building a functional puppet workflow part 2: Module structure. http://garylarizza.com/blog/2014/02/17/puppet-workflow-part-2/. [last accessed June 24, 2019].

[17] Garry Larizza. Puppetconf 2014 talk - the refactor dance. `http://garylarizza.com/blog/2014/10/23/puppetconf-2014-talk/`. [last accessed June 24, 2019].

[18] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang. An empirical case study on the temporary file smell in dockerfiles. *IEEE Access*, 7:63650–63659, 2019.

[19] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[20] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[21] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.

[22] Naouel Moha, Yann-Gael Gueheneuc, Anne-Fran Duchien, et al. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36, 2010.

[23] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 268–278. IEEE Press, 2013.

[24] The puppet language style guide. `https://puppet.com/docs/puppet/5.3/style_guide.html`. [last accessed June 24, 2019].

[25] J. Schwarz, A. Steffens, and H. Lichter. Code smells in infrastructure as code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228, Sep. 2018.

[26] Tushar Sharma. Identifying extract-method refactoring candidates automatically. In *Proceedings of the Fifth Workshop on Refactoring Tools*, pages 50–53. ACM, 2012.

[27] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 189–200. IEEE, 2016.

[28] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: a software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pages 1–4. ACM, 2016.

[29] Tushar Sharma and Pvr Murthy. Esa: the exclusive-similarity algorithm for identifying extract-class refactoring candidates automatically. In *Proceedings of the 7th India Software Engineering Conference*, page 15. ACM, 2014.

[30] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[31] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[32] E. van der Bent, J. Hage, J. Visser, and G. Gousios. How good is your puppet? an empirically defined and validated quality model for puppet. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 164–174, March 2018.

[33] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 411–420. IEEE, 2011.

Appendix A: Related work of WP4

In addition to the work described in this document, which concerns the functionalities of the CROSSMINER system Configuration Code Analyser, further research has been conducted in the context of WP4. This work investigates novel methods for dynamic analysis of Puppet–based projects and could not be incorporated currently in the CROSSMINER platform since the platform is designed to support only static analysis. This research had been drafted as a research paper, which has been submitted to ESEC/FSE'19 and is currently under resubmission to ICSE'20.

Detecting Missing Dependencies and Notifiers in Puppet Programs

Thodoris Sotiropoulos
theosotr@aueb.gr
Athens University of Economics and Business

Dimitris Mitropoulos
dimitro@aueb.gr
Athens University of Economics and Business

Diomidis Spinellis
dds@aueb.gr
Athens University of Economics and Business

ABSTRACT

Puppet is a popular computer system configuration management tool. It provides abstractions that enable administrators to setup their computer systems declaratively. Its use suffers from two potential pitfalls. First, if ordering constraints are not specified whenever an abstraction depends on another, the non-deterministic application of abstractions can lead to race conditions. Second, if a service is not tied to its resources through notification constructs, the system may operate in a stale state whenever a resource gets modified. Such faults can degrade a computing infrastructure's availability and functionality.

We have developed an approach that identifies these issues through the analysis of a Puppet program and its system call trace. Specifically, we present a formal model for traces, which allows us to capture the interactions of Puppet abstractions with the file system. By analyzing these interactions we identify (1) abstractions that are related to each other (e.g., operate on the same file), and (2) abstractions that should act as notifiers so that changes are correctly propagated. We then check the relationships from the trace's analysis against the program's dependency graph: a representation containing all the ordering constraints and notifications declared in the program. If a mismatch is detected, our system reports a potential fault.

We have evaluated our method on a large set of Puppet modules, and discovered 57 previously unknown issues in 30 of them. Benchmarking further shows that our approach can analyze in minutes real-world configurations with a magnitude measured in thousands of lines and millions of system calls.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; **Software testing and debugging**; *File systems management*.

KEYWORDS

Puppet, ordering, notifiers, system calls, traces, dynamic analysis

1 INTRODUCTION

The prevalence of cloud computing and the advent of microservices have made the management of multiple deployment and testing environments a challenging and time-consuming task [5, 29, 32, 39]. *Infrastructure as Code* (IaC) methods and tools automate the setup and provision of these environments, promoting reliability, documentation, and reuse [39]. Specifically, IaC (1) boosts the reliability of an infrastructure, because it minimizes the human intervention which is both time-consuming and error-prone; (2) ensures the predictability and consistency of the final product, because it eases the repetition of the steps followed to produce a specific outcome; and (3) allows the documentation and reuse of a system's configuration, because it associates the system's configuration with modular code [22, 29, 39, 41].

Puppet [27] is one of the most popular system configuration tools used in the IaC context [33, 37]. Puppet abstracts the actual system resources through a declarative approach. It collects all the declared abstractions from a program, and applies them one-by-one so that the system eventually reaches the desired state.

By default, any execution sequence of abstractions is valid, unless there are specific ordering constraints imposed by the interdependencies among them. For example, an Apache service should run only after the installation of the corresponding package. Therefore, developers need to declare any ordering constraints between abstractions in their programs to remove erroneous execution sequences, e.g., trying to start a service before the installation of its package. Conceptually, Puppet captures all the ordering relationships defined in a program through a directed acyclic graph and applies each abstraction in topological ordering. In this context, all the unrelated abstractions are processed *non-deterministically*. Furthermore, Puppet allows programmers to apply certain abstractions whenever specific events take place via a feature called *notification*. Notifications propagate changes to related resources, ensuring that their state is up-to-date. For instance, when a configuration file changes the corresponding service has to be notified so that it will run with the new settings.

Tracking all the required ordering constraints and notifications is a complicated task though, mostly because developers are not always aware of the actual interactions of Puppet abstractions with the underlying operating system. Notably, such errors can have a negative impact on the reliability of an organization's infrastructure leading to inconsistencies [37] and outages [18]. For example, the Github's services became unavailable when a missing notifier in their Puppet codebase caused a chain of failures such as DNS timeouts [18].

Approaches that automatically detect these issues in production code [20, 37] have significant room for improvement, facing limitations that prevent them from being practical. *Rehearsal* [37] employs static code verification and cannot handle realistic Puppet programs. In particular, it cannot reason about programs that abstract arbitrary shell commands. Additionally, the model-based testing approach adopted by *Citac* [20] imposes a significant overhead and restrictions on the supported Puppet programs under test (they must be able to run in Docker[1] containers). It also requires extra instrumentation on the execution engine of Puppet. Finally, none of those tools addresses missing notification issues.

[1] https://www.docker.com/

We have developed a *practical* and *effective* approach to identify faults involving ordering violations and notifiers in Puppet programs. To do so, we examine the system call trace produced by a single execution. The stepping stone of our approach is *FStrace*; a language for modeling a sequence of system call traces. We employ FStrace and operate in the following steps. First, we *model* the system call trace of a Puppet program in FStrace. Through FStrace, we derive an analysis that *captures* the interactions of higher-level programming constructs (Puppet abstractions) with the file system, and *estimates* the set of the expected relationships among them. Then, for a given Puppet program, we *build* the *dependency graph* which reflects all the ordering relationships and notifications that have been specified by the developer. Finally, we *verify* whether the expected relationships (as specified from the analysis of traces) hold with respect to the dependency graph. Unlike previous tools [20, 37], our approach (1) can reason about which system resources are affected by the execution of a program and how, and (2) requires only a single Puppet run for discovering issues.

Contributions. Our work makes the following contributions:

- We introduce FStrace, a language for modeling system call traces. The interpretation rules of FStrace allow us to infer the impact that higher-level building blocks have on the file system. The model proposed is generic and can be leveraged—apart from Puppet programs—by other domains (Section 3).
- We design a framework for detecting faults regarding ordering violations and notifiers in Puppet programs. To the best of our knowledge, it is the first approach to deal with issues involving notifiers. (Section 4).
- We provide an open-source implementation of our approach (Section 5).
- We demonstrate the effectiveness and performance of our tool on a large set of Puppet modules. Specifically, our tool was able to detect 57 previously unknown faults in 30 Puppet modules. We provided fixes for 21 projects and 16 of them were accepted and integrated. This implies that our tool is capable of discovering issues that are useful to developers (Section 6).

2 OVERVIEW

Here is a brief overview of Puppet, a motivating example that demonstrates the types of defects our approach detects, and how our approach is structured.

Puppet. Puppet enables developers to describe the desired state of a system declaratively.

```
1  package {"apache2": ensure => "installed"}
2  file {"/etc/apache2/apache2.conf": ensure => "file"}
3  service {"apache2": ensure => "running"}
```

The code above indicates that the apache2 package should be installed in the host, the file apache2.conf should exist in the /etc/apache2/ path, and that the Apache server should be running. There are different types for abstracting system resources, including but not limited to, file, package, service, exec. Beyond that, the Puppet language provides conditionals, loops, and—for reusability—supports the creation of new abstractions and classes.

Puppet code is stored in files called *manifests*. Puppet compiles manifests into *catalogs* that specify all the abstractions it needs to apply in a particular system to reach the desired state [26]. Then, it evaluates the compiled catalogs and applies potential changes, if the system is not in the appropriate state. For example, if a file does not exist at a certain location, Puppet will create it. The execution of a catalog must be *idempotent* [25], so that the evaluation proceeds only if the current and the desired state of the system do not match.

```
1  class ntp (..., String $defaults_file = "/etc/default/ntp") {
2    package { "ntp": ensure => $package_ensure }
3    file { "/etc/ntp.conf":
4      ensure => "present",
5      require => Package[$package],
6    }
7    if $defaults_file {
8      file { $defaults_file:
9        ensure => "present",
10       content => "conf content..."
11       require => Package["ntp"],
12     }
13   }
14   $service_subscribe = $defaults_file ? {
15     true => [ File[$defaults_file], File["/etc/ntp.conf"] ],
16     default => [ File["/etc/ntp.conf"] ],
17   }
18   service { "ntp":
19     ensure => "running",
20     require => File[/etc/ntp.conf],
21     subscribe => $service_subscribe,
22   }}
```

Figure 1: A Puppet program that manifests a missing ordering relationship and notifier. We omit irrelevant code.

Motivating Example. In the following, we present a motivating example that demonstrates the issues that our approach addresses.

Missing Ordering Relationships (MOR) occur when a developer fails to define a *happens-before* relation between two Puppet abstractions that depend on each other. This can lead to unstable code that behaves correctly in some circumstances, but breaks in others depending on the order that Puppet processes resources.

Consider the code snippet shown in Figure 1. This fragment is taken from a real-world Puppet module [12], which defines a class that setups a Network Time Protocol (NTP) service. This class expects the String parameter $defaults_file as an argument (line 1), which stands for the path where the service's default configuration file is created. Notice that the default value of $defaults_file is /etc/default/ntp. Initially, at line 2, the program installs the ntp package. Then, it creates a configuration file at the path /etc/ntp.conf (lines 3–6). If the variable $defaults_file is *defined*, Puppet generates a file at the location specified by $defaults_file (lines 7–13). Note that if Puppet finds an already defined variable in the condition of an if statement, it implicitly coerces it to true. Puppet evaluates both file abstractions *after* the package resource. This is expressed through the require property at lines 5, 11.

In lines 18 to 22, the program declares that an NTP service should be in a running state. Notice the subscribe parameter at line 21, where the NTP service subscribes to the variable $service_subscribe which in turn is computed at lines 14–17. The subscribe construct states that the service depends on the Puppet abstractions included in the service_subscribe variable.

The initial intention of the programmer is that when the $defaults_file is defined, then the service should subscribe to both File[$default_file] and File["/etc/ntp.conf"] (line

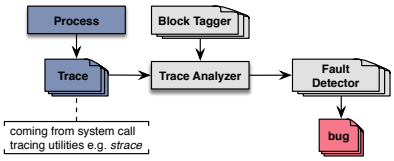

Figure 2: The Abstract Architecture of our Framework.

15). However, unlike `if` statements, the operand of the "?" operator (i.e, `$defaults_file`) at line 14 *never* evaluates to `true` because it is a `String` variable. Therefore, the program considers only the default case (line 16) where the subscribers' list only contains `File[/etc/ntp.conf]`. As a result, the dependency between the `service` and the `/etc/default/ntp` file is never created. In other words, Puppet might apply `service` before the configuration of `/etc/default/ntp`. A fix to this problem is to replace "?" operator with an `if` statement.

Missing Notifiers (MN). Notifiers are necessary for many entities such as configuration files and services. An update to a configuration file should trigger the restart of the corresponding service, because these files typically describe settings processed during a service's startup. For example the configuration file of an Apache service lists additional modules that should be loaded into memory.

A missing notifier issue is illustrated in Figure 1. The `subscribe` primitive (line 21) creates a notifier that restarts the NTP service whenever there is a change to the resources included in the `$service_subscribe` list. Although the programmer's intention was correct, the programming error at lines 14–17 causes an unexpected behavior: the service does not restart even if the configuration `/etc/default/ntp` changes because the service subscribes only to `/etc/ntp.conf`.

Framework. To address these issues, we propose a framework—illustrated in Figure 2—that operates as follows. First, it monitors the system calls of the Puppet process and its descendants. Then, the framework employs two components: the *trace analyzer*, and the *fault detector*. The trace analyzer takes as input a system cal trace derived from the application of a Puppet configuration, and it interprets each system call based on the model described in Section 3. The analyzer is instantiated with the *block tagger* component, which splits system calls into different blocks that correspond to Puppet abstractions. The analysis output is the set of the effects that Puppet abstractions have on the file system. The fault detector, generates the directed acyclic graph containing all the ordering constraints and notifications declared in a Puppet program, and compares it against the expected relationships inferred from the output of the trace analysis. If a mismatch is identified, the fault detector reports a potential fault.

Trace Example. To generates traces, we exploit a system call tracing program [28, 35], namely, `strace`. Figure 3 presents an excerpt from the trace of the program of Figure 1. Each line denotes an invocation of a system call along with the process (PID) that triggered it. For example, the entry `103 close(7) = 0` states that the process with ID = 103, invoked `close` with 7 as an argument, and that system call returned 0. By further inspecting Figure 3, we observe that Puppet initially processes the `File[/etc/default/ntp]` resource (lines 2–7), and then the NTP service (lines 8–16). Observe the calls of `write` at lines 2, 7–8, 16. These calls correspond to messages printed to the standard output by Puppet indicating the points where the application of each abstraction starts and ends respectively. We exploit these points to classify system calls according to the Puppet abstraction they come from (Section 3.2).

3 MODELING SYSTEM CALL TRACES

We formally introduce *FStrace*, the language we use to model system calls in it, and discuss how we model traces stemming from Puppet programs.

3.1 The FStrace Language

FStrace primitives are designed to model system calls that operate on file system resources. Some of the constructs have direct correspondence with the actual system calls, while others are generic enough so that they can represent a family of system calls. We group system calls into execution blocks, with a unique ID. FStrace assumes that within a block, all system calls are processed sequentially. However, the execution order at the level of blocks is *not* deterministic. Therefore, there is no guaarantee that a block b_1 is always processed before b_2, even if the former appears before the latter in traces. FStrace processes every execution block *atomically*, and nested blocks are not allowed.

3.1.1 Syntax and Domains. Figure 4 shows the syntax of FStrace. The language constists of file names, paths—which are sequences of file names—and file descriptors represented by either an integer or the **at_fdcwd** construct. We also include (1) flags (e.g., read-mode, write-mode, **o_trunc**) that indicate how a file is opened, (2) the constructs **fd** and **cwd** that provide information for cloning a process, (3) primitives (**consumed**, **produced**, **expunged**), that stand for the types of the effect that a system call has on a file, and (4) an infinite set of unique identifiers for execution blocks. A trace is a sequence of blocks. A block has the following syntax: **begin** b $(z, s)^*$ **end**, where b implies its ID and $(z, s)^*$ is a sequence of trace entries. Each pair (z, s) is a process ID (PID), which is a positive integer, and a system call. Finally, FStrace models every system call $s \in Sys$ using the following eleven constructs.

chdir p changes the working directory of the current process to p.
clone c^* f spawns a new process whose PID is f. The given flags c^* reveal what kind of information is shared between the parent and the child process.
close f disassociates the file descriptor f from the corresponding resource.
dupfd f_1 f_2 creates a new file descriptor f_2 as a copy of f_1. This construct models a number of system calls such as `dup`, `dup2`, `dup3`, `fcntl(fd, F_DUPFD)`.
hpath d p m captures the effect m that a system call has on the path p. If the given path name is not absolute, we interpret it as relative to the file descriptor d. If the value of d is **at_fdcwd**, we consider p relative to the current working directory. Otherwise, if p is absolute, we ignore d. In this way, we can represent the system calls whose suffix is *"at"* (e.g., `linkat`, `renameat`) or system calls that take relative paths as arguments. For instance, the system call `stat("foo",...)`—which retrieves the main information and attributes of the file `foo`—is represented as **hpath at_fdcwd** `foo` **consumed**. On the other hand, we represent the system call `mkdir("/foo/bar")`—which creates a new directory at path `/foo/bar`—as **hpath at_fdcwd** ("/", "foo", "bar") **produced**.

```
1   ...
2   103 write(1, "Info: /Stage[main]/Ntp/File[/etc/default/ntp]: Starting to evaluate the resource", 91) = 91
3   103 open("/etc/default/ntp20190128-32-15kba2r", O_RDWR|O_CREAT, 0600) = 7
4   103 write(7, "conf content"..., 44) = 44
5   103 close(7) = 0
6   103 rename("/etc/default/ntp20190128-32-15kba2r", "/etc/default/ntp") = 0
7   103 write(1, "Info: /Stage[main]/Ntp/File[/etc/default/ntp]: Evaluated in 0.06 seconds", 83) = 83
8   103 write(1, "Info: /Stage[main]/Ntp/Service[ntp]: Starting to evaluate the resource", 81) = 81
9   ...
10  103 execve("/etc/init.d/ntp", ["/etc/init.d/ntp", "start"], ...) = 0
11  ...
12  650 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f70159c39d0) = 660
13  ...
14  660 open("/etc/default/ntp", O_RDONLY) = 3
15  660 read(3, "conf content..."..., 44) = 44
16  103 write(1, "Info: /Stage[main]/Ntp/Service[ntp]: Evaluated in 1.85 seconds", 73) = 73
```

Figure 3: An example of trace produced by the `strace`.

$f \in F = \mathbb{Z},\ \ z \in Proc = \mathbb{Z}^*,\ \ b \in BlockID,\ \ v \in Filename$
$e \in Trace ::= x^*$
$x \in Block ::= \textbf{begin}\ b\ (z, s)^*\ \textbf{end}$
$s \in Sys ::= \textbf{chdir}\ p \mid \textbf{clone}\ c^*\ f \mid \textbf{close}\ f \mid \textbf{dupfd}\ f_1\ f_2 \mid$
$\quad \textbf{hpath}\ d\ p\ m \mid \textbf{hpathsym}\ d\ p\ m \mid \textbf{link}\ d_1\ p_1\ d_2\ p_2 \mid$
$\quad \textbf{open}\ d\ p\ o^*\ f \mid \textbf{rename}\ d_1\ p_1\ d_2\ p_2 \mid \textbf{symlink}\ d\ p_1\ p_2 \mid \textbf{nop}$
$m \in Eff ::= \textbf{consumed} \mid \textbf{produced} \mid \textbf{expunged}$
$c \in CloneFlags ::= \textbf{fd} \mid \textbf{cwd}$
$m \in OpenFlags ::= \textbf{read} \mid \textbf{write} \mid \textbf{trunc} \mid \textbf{creat}$
$d \in DirFd ::= f \mid \textbf{at\_fdcwd}$
$p \in Path ::= v^*$

Figure 4: The syntax of FStrace.

hpathsym $d\ p\ m$ operates in a way similar to **hpath**. In **hpathsym** though, if the given path p is a symbolic link, we do not dereference it. Through **hpathsym** we express system calls that do not follow symbolic links such as `lstat`, `lchown`, `lgetxattr`.
link $d_1\ p_1\ d_2\ p_2$ creates a hard link that points to the same resource as the file indicated by the file descriptor d_1 and the path p_1.
open $d\ p\ o^*\ f$ associates the file indicated by the path p with the file descriptor f. A sequence of flags o^* captures the operations that can be performed on the file.
rename $d_1\ p_1\ d_2\ p_2$ arranges that a given file, specified through the path defined by the file descriptor d_1 and path p_1, is accessed through the new path defined by d_2 and p_2.
symlink $p_1\ d\ p_2$ creates a symbolic link file at the location specified by the file descriptor d and path p_2 pointing to the path p_1.
nop (no operation) does not affect the state. We use **nop** to model all system calls that we do not need to take into account, e.g., `getpid`, `sync`.

Figure 5 illustrates the semantic domains of FStrace. FStrace introduces six major components: An `inode` table $\tau \in INodeT$ is a map of a pair, consisting of an `inode` and a file name to another `inode`. An `inode` is a positive integer that acts as the *identifier* for a certain file system resource. Note that we also keep the special `inode` ι_r which corresponds to the `inode` of the root directory "/". The `inode` table mimics the `inode` structure implemented in Unix-like operating systems. In this context, the first element of the key is the `inode` of the directory where the file name exists. For example, the `inode` of the `/foo` file, whose value is 3, is stored as follows:

$$\iota \in INode = \{\iota_i \mid i \in \mathbb{Z}^*\} \cup \{\iota_r\}$$
$$\alpha \in Addr = \{\alpha_i \mid i \in \mathbb{Z}^*\}$$
$$\tau \in INodeT = (INode \times Filename) \rightharpoonup INode$$
$$\pi \in FdT = Addr \hookrightarrow (F \hookrightarrow INode)$$
$$\nu \in ProcT = Proc \rightharpoonup (Addr \times Addr)$$
$$\phi \in CwdT = Addr \hookrightarrow INode$$
$$\kappa \in SymT = INode \hookrightarrow Path$$
$$\rho \in Res = Path \hookrightarrow \mathcal{P}(Eff \times Block)$$

Figure 5: Semantic domains for FStrace.

$[(\iota_r, \text{"foo"}) \rightarrow 3)]$. A file descriptor table $\pi \in FdT$ maps an address and a file descriptor to an `inode`. We use this component to map open file descriptors of a process to the resource they handle. The $CwdT$ element maps an address to an `inode`. That `inode` stands for the current working directory of a process.

Observe that we do not use the PID found in the trace entries as the key of the two definitions above. Instead, we have an indirection: each process points to a pair of addresses (e.g., see the domain $ProcT$). The first element of the pair is the address that stores the file descriptor table of the process. The second element of the pair reflects the address where the current working directory of the process is located. Therefore, two different processes might share the same file descriptor table or working directory. For example, in the following entries: $[(z_1 \rightarrow (\alpha_1, \alpha_2)),\ z_2 \rightarrow (\alpha_1, \alpha_3)]$, the processes z_1 and z_2 point to the same file descriptor table because the first elements of their pairs are identical (i.e., α_1). Similarly, since their second addresses do not match (i.e, $\alpha_2 \neq \alpha_3$), we presume that they do not share the same working directory; thus, a change imposed by any process does not affect the other one.

A table of symbolic links $\kappa \in SymT$ is a partial map of `inodes` to paths. This domain holds the path names that symbolic links—identified by their `inodes`—point to. The last component of FStrace ($\rho \in Res$) maps path names to an element of the power set of blocks and effects. Specifically, this component tracks where and how each path is accessed. For example, the entry `/foo` $\rightarrow$ $\{(\textbf{produced}, b_1), (\textbf{consumed}, b_2)\}$ indicates that the path `/foo` is produced in the block b_1 and consumed in b_2. The state $\langle \tau, \pi, \phi, \nu, \kappa, \rho \rangle$ is a tuple consisting of the six components described above.

3.1.2 Preliminary Definitions. A number of specific operations apply to FStrace's domains. The binary operator :: denotes the

addition of an element to a set, while $\downarrow_i$ manifests the projection of the i^{th} element. Also, we define the following auxilliary functions:

- $I(p, \tau)$: returns the `inode` to which the path p points based on the inode table τ.
- $P(\iota, \tau)$: returns the paths that point to the `inode` ι according to the inode table τ.
- $join(p_1, p_2)$: joins the two paths p_1 and p_2.
- $dir(p)$ returns the parent directory of the path p.
- $base(p)$ returns the base name of the path p.

We also define the function $\text{Ab}(d, p, l, r, \tau)$ which gives the absolute path for a given path name p and a file descriptor d with regards to the provided open file descriptors l and the current working directory r of a process.

$$\text{Ab}(d, p, l, r, \tau) = \begin{cases} p, & \text{if } p \downarrow_1 = \text{"/"} \\ join(P(r, \tau) \downarrow_1,\ p) & \text{if } d = \textbf{at\_fdcwd} \\ join(P(l(d), \tau) \downarrow_1,\ p), & \text{otherwise} \end{cases}$$

Finally, the function $Op(m)$ gives the effect that the `open` system call has on a file based on the flags m. Op is defined as:

$$\text{Op}(m) = \begin{cases} \textbf{produced}, & \text{if } (\textbf{trunc} \in m \wedge \textbf{write} \in m) \vee \textbf{creat} \in m \\ \textbf{consumed}, & \text{otherwise} \end{cases}$$

3.1.3 Semantics. Figure 6 shows the semantics of FStrace. We present a subset of our rules for brevity. Each rule follows the form below:

$$\langle \tau, \pi, \phi, \nu, \kappa, \rho \rangle \xrightarrow{b,e} \langle \tau', \pi', \phi', \nu', \kappa', \rho' \rangle$$

The relation $\xrightarrow{b,e}$ indicates that given a trace entry e (a pair of a PID and a system call) in execution block b, the initial state $\langle \tau, \pi, \phi, \nu, \kappa, \rho \rangle$ transitions to a new state $\langle \tau', \pi', \phi', \nu', \kappa', \rho' \rangle$.

`[CHDIR]` changes the working directory of the current process z. First, it inspects the process table ν to get the address that holds the value of the current process's working directory. Then, it updates ϕ so that the address α points to the `inode` of the path p.

`[CLONE-COPY]` demonstrates the case where we spawn a new process f by passing the empty sequence $c = \emptyset$. In this case, f shares neither the file descriptor table nor the working directory with the parent process z. So, we make copies of those values by creating two fresh addresses α_1, α_2. Then, we update the process table ν so that the new process f points to those new addresses.

`[CLONE-SHARE]` behaves in a similar way with `[CLONE-COPY]`. However, this time, the new process f shares the open file descriptors (flag **fd**) and the working directory with z (flag **cwd**). Therefore, the freshly-created process f points to the same addresses as z.

`[DUPFD]` involves the scenario where we duplicate a provided file descriptor. Specifically, we lookup the address α of the current process's file descriptors table. Then, we retrieve the `inode` ι pointed by the file descriptor f_1. Finally, we add the file descriptor f_2, whose `inode` value is ι, to the file descriptor table of z.

`[OPEN]` opens a file and returns a new file descriptor. First, it inspects the addresses α_1, α_2 where the file descriptor table and the working directory of the process z are located. Through the Ab function, it computes the absolute path p' using the file descriptor d, and the path p. This computation is boilerplate, so we abbreviate it as $\text{Ab}(d, p, \dots)$ in the next rules. Given the flags o^*, it estimates the effect m that `open` has on the path p' (via the function Op). In turn, it performs two updates. First, it adds f to the file descriptor table of the process z using the address α_1. Notice that f points to the `inode` of the path p' ($f \rightarrow I(p')$). Second, it updates the ρ element: the path p' receives the effect m in the the block b.

`[HPATH]` records the effect m that a system call has in the current execution block b. It handles the case when the given effect m is not **expunged**. Specifically, it determines the absolute path p' through $\text{Ab}(d, p, \dots)$. It then inspects the symbolic link table to check whether the path p' points to another path or not. If this is the case (i.e., $\kappa(p') \neq \textbf{undef}$), we associate the resulting resource p'' with the effect m in the current execution block b. Note that the **hpathsym** operates similarly, but it does not check whether p' is a symbolic link or not; it just considers the path p'.

`[HPATH-EXPNG]` illustrates the case where we expunge the provided file. As before, we first compute the absolute path p' associated with that resource. Subsequently, we remove all the effects associated with p' in the current execution block b, leading to the set l' (i.e., $l = \{m \mid \forall m \in \rho(p') :\ m \downarrow_2 \neq b\}$). We add the **expunged** effect to l', and finally, we *unlink* the path p' from the `inode` table. For unlinking, the pair $(I(p_1), p_2)$ refers to **undef**. Notice that p_1 is the parent directory of p', and p_2 is the base name of p'.

`[LINK]` creates a hard link between two files. As a starting point, we take the absolute paths p'_1 and p'_2, where p'_1 corresponds to the existing file, while p'_2 stands for the path where we create the hard link. Then, the `inode` of the new path p'_2 is identical to that of p'_1 ($\tau' = \tau[(I(w), t) \rightarrow I(p'_1)]$). We also change the table ρ so that the path p'_2 is produced in the current execution block b.

`[SYMLINK]` creates a new symbolic link that points to the path p_1. It first estimates the absolute path p'_2 of the fresh symbolic link. Then, it creates a new `inode` ι which the symbolic link points to by updating the `inode` table τ. It also changes the table κ so that the new `inode` ι targets the path p_1. Finally, the path p'_2 is produced in the current execution block b leading to the new table ρ'.

`[RENAME]` renames the name of a given file. First, it retrieves the absolute paths corresponding to the old and the new path names (i.e, p'_1 and p'_2). Then, it updates the `inode` table τ: the `inode` of p'_2 is the same with that of p'_1. In turn, it removes p'_1 from the inode table (i.e., p'_1 points to **undef**). For these updates, it is necessary to estimate: (1) the `inode` of their parent directories, and (2) their base names. Finally, it updates the component ρ. In particular, it removes any effects on path p'_1 that took place within the block b, and it marks p'_1 as **expunged** and p'_2 as **produced** in b.

3.2 Modeling Puppet Traces

To leverage FStrace we need to group system calls into blocks corresponding to higher-level programming constructs. Specifically for Puppet artifacts, it makes sense to classify system calls according to the Puppet abstraction where they come from. In this context, we presume that an execution block begins or ends when the evaluation of a Puppet abstraction starts or terminates. Thus, the name of the execution block corresponds to the name of the Puppet abstraction.

It is easy to identify the points where the evaluation of a Puppet abstraction starts/finishes by decoding the Puppet's debug messages. Recall from Figure 3 that those messages appear in the execution traces as writes to the standard output. We have developed a block tagger for Puppet that detects those debug messages and marks them as the entry and exit points of execution blocks. For example, consider again the traces in Figure 3. We can model the

CHDIR

$$\frac{e = z,\ \textbf{chdir}\ p \qquad \alpha = \nu(z)\downarrow_2 \qquad \phi' = \phi[\alpha \to I(p,\tau)]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau, \pi, \phi', \nu, \kappa, \rho\rangle}$$

CLONE-COPY

$$\frac{e = z,\ \textbf{clone}\ \emptyset\ f \qquad \text{fresh}\ \alpha_1 \qquad \text{fresh}\ \alpha_2 \qquad \nu' = \nu[f \to (\alpha_1, \alpha_2)] \qquad \pi' = \pi'[\alpha_1 \to \pi(z)] \qquad \phi' = \phi[\alpha_2 \to \phi(z)]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau, \pi', \phi', \nu', \kappa, \rho\rangle}$$

CLONE-SHARE

$$\frac{e = z,\ \textbf{clone}\ (\textbf{fd}, \textbf{cwd})\ f \qquad (\alpha_1, \alpha_2) = \nu(z) \qquad \nu' = \nu[f \to (\alpha_1, \alpha_2)]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau, \pi, \phi, \nu', \kappa, \rho\rangle}$$

DUPFD

$$\frac{e = z,\ \textbf{dupfd}\ f_1\ f_2 \qquad \alpha = \nu(z)\downarrow_1 \qquad \iota = \pi(\alpha)(f_1) \qquad \pi' = \pi[(\alpha \to (\pi(\alpha)[f_2 \to \iota])]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau, \pi', \phi, \nu, \kappa, \rho\rangle}$$

OPEN

$$\frac{e = z,\ \textbf{open}\ d\ p\ o\ f \qquad (\alpha_1, \alpha_2) = \nu(z) \qquad p' = \text{Ab}(d, p, \pi(\alpha_1), \phi(\alpha_2), \tau) \qquad m = Op(o) \qquad \pi' = \pi[\alpha_1 \to \pi(\alpha_1)[f \to I(p')]] \qquad \rho' = \rho[p' \to (m, b) :: \rho(p')]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau, \pi', \phi, \nu, \kappa, \rho'\rangle}$$

HPATH

$$\frac{e = z,\ \textbf{hpath}\ d\ p\ m \qquad m \neq \textbf{expunged} \qquad p' = \text{Ab}(d, p, \dots) \qquad p'' = \kappa(I(p')) \qquad p'' \neq \textbf{undef} \qquad \rho' = \rho[p'' \to (m, b) :: \rho(p'')]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau, \pi, \phi, \nu, \kappa, \rho'\rangle}$$

HPATH-EXPNG

$$\frac{e = z,\ \textbf{hpath}\ d\ p\ \textbf{expunged} \qquad p' = \text{Ab}(d, p, \dots) \qquad l = \{m \mid \forall m \in \rho(p') :\ m \downarrow_2 \neq b\} \qquad \rho' = \rho'[p' \to (\textbf{expunged}, b) :: l] \qquad p_1 = dir(p) \qquad p_2 = base(p) \qquad \tau' = \tau[(I(p_1), p_2) \to \textbf{undef}]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau', \pi, \phi, \nu, \kappa, \rho'\rangle}$$

LINK

$$\frac{e = z,\ \textbf{link}\ d_1\ p_1\ d_2\ p_2 \qquad p_1' = \text{Ab}(d_1, p_1, \dots) \qquad p_2' = \text{Ab}(d_2, p_2, \dots) \qquad w = dir(p_2') \qquad t = base(p_2') \qquad \tau' = \tau[(I(w), t) \to I(p_1')] \qquad \rho' = \rho[p_2' \to (\textbf{produced}, b) :: \rho(p_2')]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau', \pi, \phi, \nu, \kappa, \rho'\rangle}$$

SYMLINK

$$\frac{e = z,\ \textbf{symlink}\ p_1\ d\ p_2 \qquad p_2' = \text{Ab}(d_2, p_2, \dots) \qquad \text{fresh}\ \iota \qquad w = dir(p_2') \qquad t = base(p_2') \qquad \tau' = \tau[(I(w), t) \to \iota] \qquad \kappa' = \kappa[\iota \to p_1] \qquad \rho' = \rho[p_2' \to (\textbf{produced}, b) :: \rho(p_2')]}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau', \pi, \phi, \nu, \kappa', \rho'\rangle}$$

RENAME

$$\frac{\begin{array}{c} e = z,\ \textbf{rename}\ d_1\ p_1\ d_2\ p_2 \\ p_1' = \text{Ab}(d_1, p_1, \dots) \qquad p_2' = \text{Ab}(d_2, p_2, \dots) \qquad w_1 = dir(p_1') \\ t_1 = base(p_1') \qquad w_2 = dir(p_2') \qquad t_2 = base(p_2') \qquad l = \rho(p_1') \qquad l' = \{e \mid \forall e \in l :\ e \downarrow_2 \neq \alpha\} \\ \tau' = \tau[(I(w_2), t_2) \to I(p_1')][(I(w_1), t_1) \to \textbf{undef}] \qquad \rho' = \rho[p_1' \to (\textbf{expunged}, b) :: l'][p_2' \to (\textbf{produced}, b) :: \rho(p_2')] \end{array}}{\langle \tau, \pi, \phi, \nu, \kappa, \rho\rangle \xrightarrow{b,e} \langle \tau', \pi, \phi, \nu, \kappa, \rho'\rangle}$$

Figure 6: The interpretation rules of FStrace.

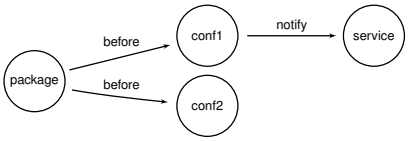

Figure 7: The dependency graph of the program of Figure 1.

trace entry at line 2 as the entry point of an execution block whose name is "`File[/etc/default/ntp]`", whereas the system call at line 8 signals the ending of that execution block. Hence, all system calls that appear between lines 2 and 8, are included in the aforementioned block.

4 DETECTING FAULTS

We locate faults in Puppet programs by combining the trace analysis output with the dependency graph: a directed acyclic graph used to capture all the ordering and notification relationships between the abstractions of a given Puppet program.

4.1 The Dependency Graph

We consider the dependency graph as an element of the following power set:

$$g \in DG = \mathcal{P}(P \times P \times L)$$

where P is the set of Puppet abstractions, and $L = \{\textbf{notify}, \textbf{before}\}$. An entry $(p_1, p_2, l) \in g$ means that the Puppet abstraction p_2 is dependent on p_1. The label l shows the relationship's type between p_1 and b_2: if $l = \textbf{before}$, p_1 is processed before p_2, whereas if $l = \textbf{notify}$, apart from preceding p_2, p_1 also sends notifications to p_2.

Let the binary relation $\prec_g$ on nodes of a dependency graph $g \in CG$ defined as follows

$$p_1 \prec_g p_2 \Rightarrow (p_1, p_2, l) \in g,\ \text{where}\ p_1, p_2 \in P, l \in L$$

This relation is transitive so

$$p_1 \prec_g p_2 \wedge p_2 \prec_g p_3 \Rightarrow p_1 \prec_g p_3,\ \text{where}\ , p_1, p_2, p_3 \in P$$

The relation $\prec_g$ forms a *happens-before* relation between two Puppet abstractions i.e., if there is path (of any length) from p_1 to p_2, then the former is executed before the latter.

Similarly, we define the transitive relation $\rightarrow_g$ on nodes of a dependency graph $g \in CG$ as

$$p_1 \rightarrow_g p_2 \Rightarrow (p_1, p_2, \textbf{notify}) \in g$$

$$p_1 \rightarrow_g p_2 \wedge p_2 \rightarrow_g p_3 \Rightarrow p_1 \rightarrow_g p_3,\ \text{where}\ p_1, p_2, p_3 \in P$$

The relation $p_1 \rightarrow_g p_2$ indicates that the abstraction p_1 notifies p_2. For that purpose, it only considers paths in g with **notify** edges.

Figure 7 depicts the dependency graph of the program of Figure 1. For brevity, the node `conf1` stands for the `/etc/ntp.conf`, and `conf2` is the `/etc/default/ntp` file. We observe that the resource `/etc/default/ntp` has neither an ordering nor a notification relationship with the `service`, because the corresponding nodes are not connected to each other. Also, `package` does not send notifications to `service` because there is no path from the former to the latter where all edges have **notify** labels.

4.2 Combining FStrace with the Dependency Graph

The dependency graph is a key element in our fault detection approach. Recall that the execution blocks in FStrace are not totally ordered. For example, consider two blocks b_1, b_2 that affect the same file: b_1 produces it and b_2 reads its contents. In this case, b_2 can be processed first, because FStrace does not define a temporal relation between the two blocks. As a result, there will be a failure because b_2 will attempt to consume a file that does not exist.

Thankfully, the dependency graph of a Puppet program can be employed to define the ordering relationships of two execution blocks (expressed through the $\prec_g$ relation). Specifically, we need to check whether the $\prec_g$ relation is defined for b_1, b_2 to identify missing ordering relationships.

Algorithm 1 Detecting Faults

Input: $\rho \in Res, g \in CG$

```
1:  for all p, l in ρ do
2:      get consumed c = {b | ∀(m, b) ∈ l. m = consumed}
3:      get produced t = {b | ∀(m, b) ∈ l. m = produced}
4:      for all (b1, b2) ∈ t × c do
5:          if b1 ⊀g b2 ∧ b2 ⊀g b1 ∧ b1 ≠ b2 then
6:              report MOR between b1, b2 on path p
7:          end if
8:          if isService(b2) then
9:              if b1 ↛g b2 then
10:                 report MN from b1 to b2
11:             end if
12:         end if
13:     end for
14: end for
```

For missing notifiers, we first need to identify pairs of Puppet abstractions where the application of the first element should trigger the application of the second one. To this end, we look for blocks that produce a particular resource p. If the same resource p is consumed by a block that maps to a `service`, presumably, the blocks, which produced p, should have notification relationships with the `service` block. That is, if they produce an update to p, `service` should be refreshed to consume the new version of p.

Algorithm 1 summarizes our fault detection approach. The algorithm expects as input the map $\rho \in Res$—as specified from the analysis of traces—and a dependency graph $g \in CG$. Then it iterates over every key-value pair of ρ. Recall that ρ is a map of a path p to a set of pairs l; each pair $(m, b) \in l$ stands for the effect m that took place in the block b. For a certain path p, we retrieve the set of blocks c that consumed p (line 2). Then, at line 3, we do the same in order to compute the set of blocks t that produced p. In turn, for every block pair (b_1, b_2) of the Cartesian product $t \times c$, we check whether there is a happens-before relation between b_1 and b_2. If b_2 is not dependent on b_1 ($b_1 \not\prec_g b_2$) and vice versa, we report a missing ordering relationship (line 6).

As a next step, the algorithm checks for missing notifiers. If the block b_2, which consumed p, corresponds to a `service` (line 8), the algorithm examines whether the relation $b_1 \rightarrow_g b_2$ holds (line 9). If the block b_1, which produced p, does not send notifications to the `service` b_2 the algorithm reports a missing notifier (line 10).

5 IMPLEMENTATION

Here are our method's implementation details and their current limitations.

5.1 Details

We have developed a prototype that implements our approach in the OCaml programming language[2]. Our tool consists of three different components: (1) an executor that is responsible for tracing Puppet programs by taking a Puppet manifest as input and executes it using `strace` to collect traces; (2) an analyzer that receives a sequence of system calls, models them in FStrace, and implements the interpretation rules presented in Figure 6; (3) a fault detector for Puppet, which takes the analyzer's output and follows the steps of Algorithm 1. Note that, we build the dependency graph of a Puppet program through a simple analysis of the catalogs produced by Puppet after the compilation of the manifests. (Catalogs are JSON documents that list all Puppet abstractions that are going to be applied along with their dependencies.)

We have implemented our method with efficiency in mind. Our tool is able to handle GB-sized traces with reasonable time and space requirements (see Section 6.4). This was made possible through a number of optimizations, such as the use of streams to process and analyze traces, a reversed `inode` table to lookup paths based on their `inodes`, and function memoization.

5.2 Current Limitations

Currently, our tool can only support Linux distributions because `strace` is a utility for Linux-based operating systems. However, we can easily extend it to support other POSIX-compliant environments such as FreeBSD or Solaris. Also, as we will observe in Section 6, our tool might produce false positives when two Puppet abstractions operate on the same file, but they are commutative to each other, i.e., the application order does not matter. Even though commutative pairs of abstractions are not so common (see Section 6), we plan to address this issue in future work by examining Puppet catalogs to identify such pairs.

6 EVALUATION

We have evaluated our framework by examining a large number of Puppet modules in order to answer the following research questions.

RQ1 Is the proposed approach effective for finding faults in Puppet manifests? (Section 6.2)
RQ2 How can we categorize the detected faults? (Section 6.3)
RQ3 What is the performance of our analysis? (Section 6.4)

6.1 Experimental Setup

We collected a large number of Puppet modules taken from Forge API[3] and Github. We were particularly interested in modules that support Debian Stretch, because Debian is one of the most popular Linux distributions [1]. We used Docker to spawn a clean Debian environment efficiently. Then, we ran our framework on every module separately. We monitored the Puppet process and collected the system call trace of every program through `strace`. Finally, we ran each step (trace analysis and fault detection) and logged the reports generated by our framework. Through this process, we successfully ran and analyzed 351 Puppet modules in total.

To compute the performance of our approach we ran the trace analysis and fault detection steps ten times to get reliable measurements. By examining the standard deviation, we observed that the running times did not vary significantly among different executions. All the experiments were run on a Virtual Machine with an 2.1GHz 8-core processor and 8GB of RAM.

6.2 Fault Detection Results

Our framework detected 57 previously unknown issues in 30 Puppet modules. Table 1 presents the analysis results for each module. Notably, this is the first study that led to the disclosure of such a large number of faults in Puppet repositories. Our framework marks 43 out of 57 faults as missing ordering relationships (column

[2]We plan to release it as an open-source software.

[3]https://forgeapi.puppetlabs.com/

Table 1: Faults found in Puppet modules

#	Module	# Faults	MOR	MN	Fix Accepted
1	istlab-stereo	9	9	0	✔
2	geoffwilliams-auditd	4	4	0	-
3	wiltonms-webserver	4	4	0	-
4	nogueirawash-mysqlserver	3	3	0	-
5	puppet-proxysql	3	3	0	✔
6	saz-ntp	3	1	2	-
7	deric-mesos	3	0	3	✔
8	hardening-os_hardening	2	2	0	✔
9	saz-locales	2	2	0	-
10	vpgrp-influxdbrelay	2	2	0	✔
11	jgazeley-freeradius	2	1	1	✔
12	ploperations-puppet	2	1	1	-
13	walkamongus-codedeploy	1	1	0	✔
14	spynappels-support_sysstat	1	1	0	-
15	cirrax-dovecot	1	1	0	✔
16	sgnl05-sssd	1	1	0	-
17	roshan-mysqlzrm	1	1	0	-
18	puppetfinland-nano	1	1	0	✔
19	olivierHa-influxdb	1	1	0	-
20	noerdisch-codeception	1	1	0	-
21	nextrevision-flowtools	1	1	0	✔
22	baldurmen-plymouth	1	1	0	✔
23	alertlogic-al_agents	1	1	0	-
24	puppet-telegraf	1	0	1	✔
25	puppetlabs-apache	1	0	1	✔
26	example42-apache	1	0	1	✔
27	deric-zookeeper	1	0	1	✔
28	camptocamp-tomcat	1	0	1	-
29	alexharvey-disable_transparent_hugepage	1	0	1	-
30	camptocamp-ssh	1	0	1	✔
	Total	**57**	**43**	**14**	-

MOR). We observe that ordering violations are the most prevalent issue in the inspected Puppet manifests. The rest of the faults are related to missing notifiers (column MN).

Based on the reports of our tool, we manually verified that each reported fault can lead to a problematic situation by reproducing each case. We provided fixes for 21 projects, and 16 of them were accepted by their development teams and integrated into their code. This indicates that our tool produces reports that are meaningful to developers. At the time of the submission, none of our patches have been rejected.

6.3 Fault Patterns

Below, we categorize and discuss some of the faults identified by our framework. Most represent previously unknown to us fault patterns which we learned through our tool.

6.3.1 Missing Ordering Relationships. We have observed two types of missing ordering relationships issues.

Generate-Use Violation. The use of a resource must always succeed its creation. Many modules fail to preserve that ordering relationship. We observed this violation in 16 Puppet modules such as `alertlogic-al_agents`, `hardening-os_hardening`, etc. Figure 8 shows a fragment from `alertlogic-al_agents` [7]. The code first fetches a `.deb` package (a Debian archive) using the `wget` command (lines 1–5). The package is stored at the path specified by the `$package_path` variable whose value is `/tmp/al-agent`. Then, the code installs the Debian archive on the system (lines 6–10) through `dpkg`. [4] It is easy to see that the `package` depends on `exec` because

```
1  $package_path = "/tmp/al-agent"
2  exec {"download":
3    command => "/usr/bin/wget -O ${package_path} ${pkg_url}",
4    creates => $package_path
5  }
6  package {"al-agent":
7    ensure => "installed",
8    provider => "dpkg",
9    source => $package_path,
10 }
```

Figure 8: A Missing Ordering Relationship between `package` and `exec`.

it requires `$package_path` (the `.deb` file) to exist in the system (line 9) so that it can install the package successfully.

The *Generate-Use* category produces observable errors (errors that manifest during the catalog's application), when Puppet applies abstractions in the erroneous order. For example, when it processes `package` before `exec` the application of the catalog fails with the following error: "`dpkg: error: cannot access archive "/tmp/al-agent": No such file or directory`"

Configure-Use Violation. The configuration of a file must precede its use. For example, when a service starts, all the files consumed by that service have to be properly configured. This category differs from the previous one because when a Puppet abstraction attempts to use the file, the latter exists in the system. However, this is not in the expected state (e.g., the file does not have the right contents, permissions, etc). This error pattern appears in four modules: `saz-ntp`, `vpgrp-influxdbrelay`, `olivierHa-influxdb`, and `ploperations-puppet`.

Figure 1 illustrates a program with an issue related to this category. When the NTP service starts, the configuration files are guaranteed to be there because the abstraction `package` creates them during installation. However,, it is possible that the NTP service does not read the desired contents of the `/etc/default/ntp` file specified by `content => "conf content..."` (line 11), because there is a missing ordering relationship between `file` and `service`. Note that this category—unlike the previous one—might lead to unexpected behaviors silently, i.e., the application of the catalog does not produce any error messages.

6.3.2 Missing Notifiers. We have identified four different categories of issues related to notifiers.

Configuration Files. A configuration file must always send notifications to a service so that any change to that file triggers the restart of the corresponding service. Although this is a standard pattern, we observed that in four modules (shown in rows 6, 7, 11, 12, 30 of Table 1) this is not the case. As an example recall the program discussed in Section 2.

Log Files. Typically, services log various events in dedicated files. For instance, the log file of an Apache server records—among other things—every incoming HTTP request. Log files are very beneficial for debugging and monitoring purposes. When a service starts, it opens a corresponding log file, which remains open, while the service is up, to write any events that take place.

We discovered issues related to logging in two popular Puppet modules (`puppetlabs-apache` [11], and `deric-zookeeper` [13]). These modules declare the log files for the `apache` and `zookeeper` services in their manifests. However, the log files do not have a notifier for their associated services. This may lead to a problematic

[4] `dpkg` is a package management system for Debian-based operating systems

```
1  # config.pp manifest
2  define tomcat::instance::config (..., $basedir, $javahome) {
3    file {"/etc/init.d/tomcat-${name}":
4      ensure => $present,
5      content => template("tomcat/tomcat.init.erb"),
6      require => Concat["${catalina_base}/bin/setenv.sh"],
7    }
8  }
9  # tomcat/tomcat.init.erb template
10 export JAVA_HOME=<%= @javahome %>
11 export CATALINA_BASE=<%= @basedir %>
12 export CATALINA_PID=<%= @basedir %>/temp/tomcat.pid
```

Figure 9: Manifest and its template taken from `camptocamp-tomcat`. We omit irrelevant code for brevity.

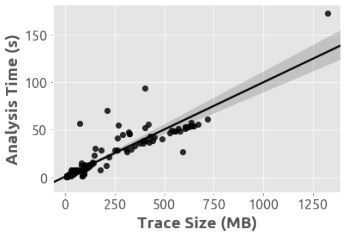

Figure 10: The trace analysis and fault detection time as a function of the trace size. Each spot shows the average time spent on both the trace analysis and fault detection phases for a given trace obtained by the execution of a module.

situation. Consider the case where the log file of a service is removed from the host. When we remove an open file, the underlying system call (`unlink`) does not update the file descriptors associated with the removed file, even though Puppet will create a new one. This means that although the file disappears from the file system, the service still handles a file descriptor that points to the `inode` of the original file. The issue is that after removal, the `inode` becomes an *orphan* (i.e., it is not linked with any file), which means that it is no longer accessible through a file path. Therefore, in the case of a missing notifier, the log history of the upcoming events is lost because the service writes to an orphan `inode`. To fix that issue, the log file should notify the service so that the service opens the newly-created log file.

Init Scripts. Init scripts specify how a service starts, stops or restarts. In practice, they are wrapper shell scripts which setup the required environment and invoke the actual executables of the services with the appropriate parameters.

Puppet manifests, that manage init scripts should notify the corresponding service whenever there is a change to those scripts. The `camptocamp-tomcat` [16] and `alexharvey-disable_transparent_hugepage` [9] modules fail to follow that pattern. Consider the code listed in Figure 9, coming from the `camptocamp-tomcat` module. The Figure shows fragments, coming from two different files. First, the `config.pp` Puppet manifest, defines a custom abstraction named `tomcat::instance::config`, which takes the variables `$basedir` and `$javahome` as parameters (line 1). This abstraction configures the init script of the tomcat service (lines 2–8) whose contents are determined by the `tomcat/tomcat.init.erb` template (lines 9–12). By examining this template we see that, before the init script starts tomcat, it sets some environment variables based on the values of the Puppet parameters `$basedir` and `javahome` (lines 10–12). When there is an update to the init script (e.g., `$javahome` variable has a different value), Tomcat should restart in order to operate on the new environment, e.g., to use a different version of Java.

Packages. When Puppet applies a `package` abstraction, the service that depends on that package should restart. In this way, we ensure that a service gets all the necessary updates, including, security patches, new features, etc. Our tool identified this kind of issue in `example42-apache` [14], `saz-ntp` [12], and `puppet-telegraf` [8]. Specifically, the `package` abstractions that were responsible for installing the Apache, NTP, and telegraf did not notify the running instances whenever there was a new version of those packages.

6.4 Performance Evaluation

Figure 10 shows the running times (in seconds) of the trace analysis and fault detection phases relatively to the size of the provided traces (in MB). We observe that the correlation between the trace size and analysis time is almost linear. Notice that our framework is able to handle a large volume of traces (more than 1.2GB) in a reasonable amount of time (< 3 minutes). The average trace size and analysis time of the inspected modules is 84MB and 9 seconds respectively.

There are 4 cases out of 351 (`ceritsc-dkms`, `datadog-datadog_agent`, `nexcess-ksplice`, `puppet-rabbitmq`) where the execution times were relatively high compared to the rest of the modules. Nevertheless, they all remain in acceptable limits. By examining the characteristics of the traces obtained by the execution of these modules, we observed that they contain more `unlink` system calls than the rest of the modules. Notably, such calls involve more expensive operations on the analysis state (they are modeled as **hpath** d, p **expunged**, recall Figure 6).

Overall, the overhead of the analysis is relatively small. We argue that our approach is practical and can be used as part of the testing process for Puppet manifests.

6.5 False Positives

We have manually inspected the reported issues and identified a potential source of false positives. Consider two abstractions that are commutative to each other. For example, in the `claranet-varnish` module [10], the developers use two different abstractions to partially configure a certain file. On the one hand they use `file` to set the permissions and ownership of the file, and on the other they use `exec` to initialize its contents. In this case the execution order in which Puppet processes abstractions does not matter. Specifically, Puppet can first use `exec` to create the file with the desired contents, and then apply `file` to set the appropriate file's attributes or vice versa. Our approach reported false positives *only* in 7 out of 351 cases. Therefore, we argue that this pattern (i.e, configuring a file through the combination of abstractions) is not particularly common.

We noticed one more false positive which was related to missing notifiers. The developers of `bodgit-dbus` [15] use a custom command (expressed via `exec`) to reload the configuration of the service. Consequently, the configuration files notify the `exec` abstraction instead of `service`. We did not observe this case elsewhere, because Puppet programmers typically exploit the `restart` parameter of the

service type to define a custom restart command in the following manner: "service { restart => "/custom/cmd", . . . }".

7 RELATED WORK

Our work is related to three research areas, namely quality in IaC, trace analysis, and modeling of file systems operations.

Quality in IaC. With the proliferation of the IaC process, there have been numerous attempts to identify defects and quality concerns in configuration code.

A number of studies focus on maintainability issues. Sharma et al. [38] design and implement a code-smell detection scheme for Puppet, which searches for issues related to naming conventions, code design, indentation, etc. Their findings suggest that such anti-patterns—as in the traditional programs—exist in many IaC repositories. Van der Bent et al. [40] introduce a quality model for Puppet programs which is empirically evaluated by interviewing practitioners from industry. Schwarz et al. [36] do similar work focusing on Chef recipes. Endeavors have recently moved to the identification of security issues. Rahman et al. [33] define and classify security smells into seven categories, such as hard-coded passwords and the use of weak cryptographic algorithms, and then build a tool for statically detecting these smells in Puppet repositories.

Other studies attempt to extract error patterns and source code properties from the analysis of defective IaC programs. Rahman et al. [34] employ machine learning and text processing techniques to identify properties that faulty Puppet programs hold. Then, they build a prediction model for asserting whether IaC scripts manifest faults or not. Chen et al. [3] identify error patterns in Puppet manifests by following a different approach. First, they inspect the code changes from repositories' commits. Second, they construct an unsupervised learning model to detect error patterns based on the clustering of the proposed fixes. Their approach is based on the assumption that similar faults are fixed with similar patches [19].

There are few automated techniques proposed for improving the reliability of configuration management programs. Rehearsal [37] statically verifies that a given Puppet configuration is deterministic and idempotent. Rehearsal models a given Puppet manifest in a small language called FS and then it constructs logical formulas based on the semantics of each language's primitive. Finally, an SMT solver decides whether the initial program is non-deterministic or not. Compared to our approach, Rehearsal is less effective and practical. Specifically, Rehearsal employs a form of static analysis that can only handle a subset of Puppet programs. For example, the analysis does not support exec abstractions because it cannot reason about the file system resources that exec processes. Unlike Rehearsal, our approach operates on the actual system calls rather than Puppet manifests; thus, it can effectively determine which files are affected by a Puppet run and how.

Other advances [20, 23] adopt a model-based testing approach for checking whether configuration scripts meet certain properties. Hummer et al. [23] focus on testing the idempotence of Chef scripts. Their proposed framework generates multiple test cases that explore different task schedules. By tracking the changes in the system (they compare the system state before and after execution), they determine if idempotence holds for a given program. Hanappi et al. [20] extend the work of Hummer et al. and introduce Citac; a framework that can be applied to Puppet manifests to examine the convergence of programs. Convergence states that the system reaches a desired state even at the presence of failed Puppet abstractions. They formally express the properties of idempotence and convergence, and through test case generation, they verify if the provided manifests violate those properties. Contrary to Citac, we adopt a more lightweight and practical approach applying manifests only once. Finally, neither Rehearsal nor Citac detect issues involving missing notifiers.

Trace Analysis. Analysis of system call traces has been widely used in the past, especially for intrusion and malware detection [2, 6, 24, 42]. Mutlu et al. [30] collect execution traces from JavaScript applications. Their traces do not track system calls, but they capture memory and storage (e.g., cookies) accesses in the context of the browser. They split traces into blocks, where each block describes the execution of an asynchronous callback (e.g., AJAX handler). As the execution of each handler is partially ordered, they apply a simple data-flow analysis over traces to join the states coming from different handlers. In this manner, they effectively detect data races by identifying handler pairs where the merges of their corresponding states result in different values of the same variable. In our work, we also separate the trace sequences into blocks. However, we are interested in file system operations instead of reads and writes to memory locations. Also, we apply a different methodology for discovering execution blocks that might lead to harmful scenarios.

Modeling File System Operations. Several researchers have designed specifications for the POSIX file system [17, 21, 31]. The specifications mainly focus on program reasoning and verification. Furthermore, Shambaugh et al. [37] have introduced FS; a small language used to model the effects of Puppet abstractions on the file system. In this work, we model system calls rather than Puppet abstractions.

8 CONCLUSION

We have introduced a novel and practical approach for identifying missing dependencies and notifiers in Puppet programs. Our method collects the system calls invoked by a Puppet program and models them in FStrace. Through FStrace, we capture how higher-level programming constructs, such as Puppet abstractions, interact with the operating system and derive their relationships. Then, our method checks the inferred relationships against the program's dependency graph and reports potential mismatches.

The effectiveness of our approach is exemplified by the uncovering of 57 previously unknown issues in 30 projects. Notably, we provided fixes for 21 modules and 16 of them were accepted by the developers. We have further showed that our tool can handle realistic traces in a reasonable time. Our results indicate that our tool can be used as part of the testing process for Puppet programs.

FStrace is a generic model that can be applied to other domains with partially ordered constructs such as the asynchronous callbacks of JavaScript. Recent work [4, 43] has showed that many concurrency faults in Node.js applications are caused by data races that appear in files instead of memory locations. As future work, we are planning to leverage our method to detect such concurrency faults in JavaScript server-side applications.

REFERENCES

[1] B. Adams, R. Kavanagh, A. E. Hassan, and D. M. German. An empirical study of integration activities in distributions of open source software. *Empirical Software Engineering*, 21(3):960–1001, Jun 2016.

[2] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[3] W. Chen, G. Wu, and J. Wei. An approach to identifying error patterns for Infrastructure as Code. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 124–129, Oct 2018.

[4] J. Davis, A. Thekumparampil, and D. Lee. Node.Fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 145–160, New York, NY, USA, 2017. ACM.

[5] T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[7] P. Forge. Alert Logic agent Puppet module. https://forge.puppet.com/alertlogic/agents, 2019.

[8] P. Forge. Configuration and management of InfluxData's Telegraf metrics collection agent. https://forge.puppet.com/puppet/telegraf, 2019.

[9] P. Forge. Disables transparent hugepages (THP). https://forge.puppet.com/alexharvey/disable_transparent_hugepage, 2019.

[10] P. Forge. Install and configure Varnish cache. https://forge.puppet.com/claranet/varnish, 2019.

[11] P. Forge. Installs, configures, and manages Apache virtual hosts, web services, and modules. https://forge.puppet.com/puppetlabs/apache, 2019.

[12] P. Forge. Manage NTP client and server via Puppet. https://forge.puppet.com/saz/ntp, 2019.

[13] P. Forge. Module for managing Apache Zookeeper. https://forge.puppet.com/deric/zookeeper, 2019.

[14] P. Forge. Puppet module for Apache. https://forge.puppet.com/example42/apache, 2019.

[15] P. Forge. Puppet module for managing D-Bus. https://forge.puppet.com/bodgit/dbus, 2019.

[16] P. Forge. Puppet Tomcat module. https://forge.puppet.com/camptocamp/tomcat, 2019.

[17] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 3–14, July 2007.

[18] Github. DNS outage post mortem - the GitHub blog. https://github.blog/2014-01-18-dns-outage-post-mortem/, 2014. [Online; accessed 28-January-2019].

[19] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 144–156, New York, NY, USA, 2016. ACM.

[20] O. Hanappi, W. Hummer, and S. Dustdar. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 328–343, New York, NY, USA, 2016. ACM.

[21] W. H. Hesselink and M. I. Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, Jan 2012.

[22] J. Humble, C. Read, and D. North. The deployment production line. In *AGILE 2006 (AGILE'06)*, pages 6 pp.–118, July 2006.

[23] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. Testing idempotence for infrastructure as code. In D. Eyers and K. Schwan, editors, *Middleware 2013*, pages 368–388, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[24] A. P. Kosoresow and S. A. Hofmeyer. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, Sep. 1997.

[25] P. Labs. Idempotence: not just a big and scary word. https://puppet.com/blog/idempotence-not-just-a-big-and-scary-word, 2016. [Online; accessed 28-January-2019].

[26] P. Labs. Catalog compilation - Puppet (PE and open source) 5.5. https://puppet.com/docs/puppet/5.5/subsystem_catalog_compilation.html, 2018. [Online; accessed 28-January-2019].

[27] J. Loope. *Managing Infrastructure with Puppet: Configuration Management at Scale*. O'Reilly Media, 2011.

[28] R. McDougall, J. Mauro, and B. Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall PTR, Upper Saddle River, 2006.

[29] K. Morris. *Infrastructure As Code: Managing Servers in the Cloud*. O'Reilly Media, Inc., 1st edition, 2016.

[30] E. Mutlu, S. Tasiran, and B. Livshits. Detecting JavaScript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 381–392, New York, NY, USA, 2015. ACM.

[31] G. Ntzik, P. da Rocha Pinto, J. Sutherland, and P. Gardner. A concurrent specification of POSIX file systems. In T. Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[32] S. Plummer and D. Warden. Puppet: Introduction, implementation, & the inevitable refactoring. In *Proceedings of the 2016 ACM on SIGUCCS Annual Conference*, SIGUCCS '16, pages 131–134, New York, NY, USA, 2016. ACM.

[33] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in Infrastructure as Code scripts.

[34] A. Rahman and L. Williams. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 34–45, April 2018.

[35] R. Rodriguez. A system call tracer for UNIX. In *USENIX Conference Proceedings*, pages 72–80, Berkeley, CA, Summer 1986. USENIX Association.

[36] J. Schwarz, A. Steffens, and H. Lichter. Code smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228. IEEE, 2018.

[37] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: A configuration verification tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 416–430, New York, NY, USA, 2016. ACM.

[38] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 189–200, New York, NY, USA, 2016. ACM.

[39] D. Spinellis. Don't install software by hand. *IEEE Software*, 29(4):86–87, July 2012.

[40] E. van der Bent, J. Hage, J. Visser, and G. Gousios. How good is your puppet? an empirically defined and validated quality model for Puppet. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, SANER 2018, pages 164–174, Mar. 2018.

[41] J. Visser, S. Rigal, G. Wijnholds, and Z. Lubsen. *Building Software Teams: Ten Best Practices for Effective Software Development.* " O'Reilly Media, Inc.", 2016.

[42] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 156–168, May 2001.

[43] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 520–531, Oct 2017.