Practical Fault Detection in Puppet Programs

Thodoris Sotiropoulos,* Dimitris Mitropoulos*† and Diomidis Spinellis*
*Athens University of Economics and Business
† National Infrastructures for Research and Technology – GRNET
{theosotr,dimitro,dds}@aueb.gr

ABSTRACT

Puppet is a popular computer system configuration management tool. By providing abstractions that model system resources it allows administrators to set up computer systems in a reliable, predictable, and documented fashion. Its use suffers from two potential pitfalls. First, if ordering constraints are not correctly specified whenever a Puppet resource depends on another, the non-deterministic application of resources can lead to race conditions and consequent failures. Second, if a service is not tied to its resources (through the notification construct), 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, a formal model for traces allows us to capture the interactions of Puppet resources with the file system. By analyzing these interactions we identify (1) resources that are related to each other (e.g., operate on the same file), and (2) resources 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 popular Puppet modules, and discovered 92 previously unknown issues in 33 modules. Performance benchmarking shows that our approach can analyze in seconds 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 Relationships, Notifiers, Program Analysis, System Calls

ACM Reference Format:
Thodoris Sotiropoulos,* Dimitris Mitropoulos*† and Diomidis Spinellis*. 2020. Practical Fault Detection in Puppet Programs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '20, May 23–29, 2020, Los Alamitos, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [8, 23, 25, 34]. *Infrastructure as Code* (IaC) methods and tools automate the setup and provision of these environments, promoting reliability, documentation, and reuse [34]. Specifically, IaC (1) boosts the reliability of an infrastructure, because it minimizes the human intervention which is both laborious 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 [15, 23, 34, 38, 39].

Puppet [21] is one of the most popular system configuration tools used to manage infrastructures [28, 32]. It abstracts the state of different system entities such as files, users, software packages, or running processes, in a declarative manner using built-in primitives called *resources*. A Puppet program consists of a collection of resources that the underlying execution engine applies one-by-one so that the system eventually reaches the desired state.

By default, any execution sequence of resources is valid, unless there are specific ordering constraints imposed by their interdependencies, e.g., an Apache service should be run only after the installation of the corresponding package. Developers need to declare these ordering constraints in the program to avoid erroneous execution sequences, such as 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 resource in topological ordering. In this context, all the unrelated resources are processed *non-deterministically*. Furthermore, Puppet allows programmers to apply certain resources 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 with the underlying operating system. Notably, such errors can have a negative impact on the reliability of an organization's infrastructure leading to inconsistencies [32] and outages [10]. For example, Github's services became unavailable when a missing notifier in their Puppet codebase caused a chain of failures, such as DNS timeouts [10].

Approaches that automatically detect these mistakes in production code [13, 32] have significant room for improvement, facing limitations that prevent them from being practical. *Rehearsal* [32]

Preprint

employs static code verification to detect faults in Puppet programs. Nevertheless, it cannot manage realistic cases because it is unable to handle Puppet resources that abstract arbitrary shell commands. Notably, such resources (e.g., `exec`) are highly pervasive as they appear in the *56%* of the top-1000 most widely used Puppet modules found in the Forge API [27]. Additionally, the model-based testing approach adopted by *Citac* [13] imposes a significant overhead (analyzing around 100 modules with Citac takes roughly 9 days) and restrictions on the supported Puppet programs under test (they must be able to run in Docker containers). It also requires extra instrumentation to be added in the execution engine of Puppet. Finally, none of those tools addresses missing notification faults.

We have developed a *practical* and *effective* approach to identify faults involving ordering violations and notifiers in Puppet programs. To do so, we record the system call trace produced by a single Puppet execution. Then, we operate as follows. First, we *model* the system call trace of a Puppet execution in a representation (namely, FStrace) that allows us to precisely capture the interactions of higher-level programming constructs (Puppet resources) with the file system. By examining their interplays, we *infer* the set of the expected relationships of every Puppet resource with each other. These relationships correspond to either notifications (e.g., resource x should notify resource y) or ordering constraints (e.g., x should run before y). Then, for the given Puppet program, we statically *build* the *dependency graph* that 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 [13, 32], our approach (1) can reason about which system resources are affected by the program's execution and how, and (2) requires only a single Puppet run for discovering issues.

Contributions. Our work makes the following contributions:

- We introduce FStrace, a representation for system call traces that models the intricate semantics of system calls and their effects on the file system. Building upon FStrace, we propose a novel trace analysis that allows us to infer the inter-relationships among Puppet resources (Section 3).
- We provide a framework and its open-source implementation for detecting faults regarding ordering violations and notifiers in Puppet programs. To the best of our knowledge, it is the first to deal with issues involving notifiers (Sections 4, 5).
- We demonstrate the effectiveness and performance of our tool on 354 Puppet modules. Specifically, our tool was able to detect 92 previously unknown faults in 33 modules, including well-established ones. More than a half of the issues (62 out of 92) were confirmed and fixed by the developers. This implies that our tool is capable of discovering faults that are useful to the Puppet community (Section 6).

Availability. The source code of our implementation is available as open-source software under the GNU General Public License v3.0 at https://github.com/AUEB-BALab/fsmove/.

2 OVERVIEW

We provide a brief overview of Puppet, two motivating examples that demonstrate the types of errors our approach detects, and how our approach is structured.

```
1  package {"mysql-server": ensure => "installed" }
2  file {"/etc/mysql/my.cnf":
3    ensure => "file",
4    content => "user db settings..."
5    require => Package["mysql-server"]
6  }
7  exec {"Initialize MySQL DB":
8    command => "mysqld --initialize",
9    require => Package["mysql-server"]
10 }
```

Figure 1: A program missing an ordering relationship.

Puppet. Puppet allows developers to describe the desired state of a system through a declarative specification language. For example:

```
1  $service_name = "apache2"
2  $conf_file = "/etc/apache2/apache2.conf"
3  package {$service_name: ensure => "installed"}
4  file {$conf_file: ensure => "file"}
5  service {$service_name: 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 diverse types for abstracting system resources, including `file`, `package`, `service`, `exec`. Also, the Puppet language provides variable declarations that begin with the "$" symbol (e.g., `$service_name`, `$conf_file`), conditionals, loops, and—for reusability—supports the creation of custom resources and classes.

Puppet code is stored in files called *manifests*. Puppet compiles manifests into executables called *catalogs*. Catalogs are JSON documents that specify all the resources that Puppet needs to apply in a particular system to reach the desired state [19]. The following JSON snippet shows a part of the catalog derived from the previous Puppet code:

```
1  "resources": [{
2      "title": "/etc/apache2/apache2.conf",
3      "type": "File",
4      "parameters": { "ensure": "file" }
5  }, { /* another resource... */ } ]
```

The field `resources` contains the Puppet resources declared in the initial program along with their parameters. During catalog compilation, every variable defined in the manifests resolves to its value, e.g., the variable `$conf_file` resolves to `/etc/apache/apache2.conf` (line 2 of the earlier Puppet code).

Puppet 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.

Motivating Examples. We present two examples of faulty programs that demonstrate the issues that our approach addresses.

Missing Ordering Relationships (MOR) occur when a developer fails to define a *happens-before* relation between two Puppet resources 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 in which Puppet processes resources.

Consider the real-world Puppet program shown in Figure 1 that sets up the MySQL database in a server. First, the code declares the installation of the `mysql-server` package (line 1), which—among other things—creates the `/etc/mysql/my.cnf` file that contains the default database settings. Then, it configures this file (lines 2–6) whose contents are specified by the `content` parameter at line 4. Note that Puppet evaluates the `file` resource after `package`. This

Preprint

```
1  package {["libssl", "apache2"]: ensure => "latest" }
2  file {"/etc/apache2/apache2.conf":
3    ensure => "file",
4    require => Package["apache2"]
5  }
6  service {"apache2":
7    ensure => "running",
8    subscribe => [ Package["apache2"],
9                   File["/etc/apache2/apache2.conf"] ]
10 }
```

Figure 2: A program missing a notifier.

is expressed through the require property at line 5. In lines 7 to 10, the program declares the execution of a shell script (mysqld --initialize) that prepares the database according to the settings specified in the /etc/mysql/my.cnf file. Although the shell command needs to be invoked only after the file /etc/mysql/my.cnf is configured (lines 2–6), the require parameter at line 9 omits this dependency. Therefore, applying the exec resource before file makes Puppet set up the database with incorrect settings. A static analyzer (such as Rehearsal) cannot extract this dependency because it is unable to infer that the underlying shell command (mysqld --initialize) consumes the database configuration file.

Missing Notifiers (MN). Notifiers are necessary for services. An update to a resource (e.g., configuration file) could directly affect the state of a service. To ensure that all services are running on a fresh environment, Puppet triggers the restart of a service whenever there is a change to one of the service's dependent resources via notifications declared by the programmers.

A missing notifier issue is illustrated in Figure 2. The code first installs the latest version of the libssl and apache2 packages (line 1), configures the file located at /etc/apache2/apache.conf (lines 2–5), and then, boots the Apache server (lines 6–10). The subscribe primitive (line 8) creates a notifier that restarts Apache whenever there is a change to the corresponding configuration file or an update to the apache2 package (e.g., a newer version is installed in the system). However, the code lacks a notifier from the libssl package to the Apache service. A change to libssl requires the restart of Apache so that the server maps the updated version of the library to its memory (i.e., Apache maps the /usr/lib/libssl.so file created during the installation of the libssl package). Failing to do so makes the server not get the latest updates or security patches of the library. Once again, a static analyzer is not capable of inferring such dependencies (i.e., the fact that Apache depends on /usr/lib/libssl.so), because they are hidden from the corresponding manifests.

Framework. To address these issues, we propose a framework—illustrated in Figure 3—that consists of the following components. The *executor* applies the Puppet manifests given as input by invoking the actual Puppet execution engine. Also, the executor intercepts Puppet as follows. First, it stores the compiled catalog of the program before it is applied to the system, and second, it monitors the system calls of the Puppet process and its descendants, generating the system call trace that stems from the catalog application.

The *analyzer* (Section 3) operates on the system call trace produced by the executor, and performs two steps. In the first step (*parser*), the analyzer splits system calls into different blocks corresponding to the execution of every Puppet resource defined in the initial program. Then, it parses the system call trace and transforms it into an FStrace representation, which is used to model the semantics of every system call and their effects on the file system. In the next phase (*interpreter*), the analyzer evaluates the resulting FStrace program and infers the set of relationships (i.e., ordering constraints and notifications) between the declared Puppet resources. To do so, it inspects their interactions with the file system in terms of the files consumed, produced, expunged by their execution.

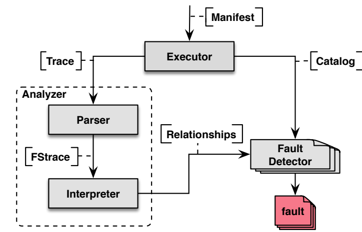

Figure 3: The architecture of our framework.

Finally, the *fault detector* (Section 4) first builds the dependency graph by examining the parameters of every Puppet resource specified in the compiled catalog given as input. The dependency graph is a directed acyclic graph that contains all the actual ordering relationships and notifications declared in the original Puppet program by the programmer. Then, the fault detector compares the generated graph against the expected relationships inferred from the output of the analyzer. If a mismatch is identified, the fault detector reports a potential fault.

Trace Example. In order to generate traces, the executor employs a system call tracing program [22, 30], namely, strace. Figure 4 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 4, we observe that Puppet initially processes the Exec[Initialize MySQL DB] resource (lines 1–6), and then the File[/etc/mysql/my.cnf] resource (lines 7–12). Observe the calls of write at lines 1, 6–7, 12 that correspond to messages printed to the standard output by Puppet engine for debugging purposes. These messages indicate the points where the application of each resource starts and ends respectively. The analyzer exploits these points to classify system calls according to the Puppet resource they come from (Section 3.2).

3 ANALYZING SYSTEM CALL TRACES

To tackle the complexity and interactions of realistic system call traces (Section 3.1), we design our trace analysis as an interpretation over a trace language used to abstract each system call and map it to the Puppet resource it comes from (Section 3.2). The output of the analysis is the set of the expected relationships (ordering constraints and notifications) among declared resources. To produce this output, the analysis examines the file system interactions stemming from Puppet execution in terms of the files that are consumed, produced, or expunged (Section 3.3). The analysis output is later used by our fault detector (Section 4).

Preprint

```
1   103 write(1, "Info: /Stage[main]/Exec[Initialize MySQL DB]: Starting to evaluate the resource", 80) = 80
2   103 execve("/usr/sbin/mysqld", ["/usr/sbin/mysqld", "--initialize"], ...) = 0
3   650 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f70159c39d0) = 660
4   660 open("/etc/mysql/my.cnf", O_RDONLY) = 3
5   660 read(3, "default content..."..., 44) = 44
6   103 write(1, "Info: /Stage[main]/Exec[Initialize MySQL DB]: Evaluated in 1.85 seconds", 72) = 72
7   103 write(1, "Info: /Stage[main]/File[/etc/mysql/my.cnf]: Starting to evaluate the resource", 78) = 78
8   103 open("/etc/mysql/my.cnf20190128-32-15kba2r", O_RDWR|O_CREAT, 0600) = 7
9   103 write(7, "db settings..."..., 48) = 48
10  103 close(7) = 0
11  103 rename("/etc/mysql/my.cnf20190128-32-15kba2r", "/etc/mysql/my.cnf") = 0
12  103 write(1, "Info: /Stage[main]/File[/etc/mysql/my.cnf]: Evaluated in 0.06 seconds", 70) = 70
```

Figure 4: An example of trace produced by the `strace` utility.

3.1 Motivation of Design Choices

Previous work on trace analysis has focused on extracting dependencies from build scripts (e.g., Make for C/C++ programs) for testing and refactoring purposes [11, 20], or identifying license inconsistencies in software projects [37]. These approaches cannot apply to the domain of Puppet for the following two reasons.

Low Fidelity. Typically, the system call traces that come from the execution of Puppet manifests are far more complex than those of build scripts. Indeed, Puppet traces involve the application of diverse entities—such as, execution of arbitrary scripts, configuration and management of different services, installation of packages, and more—that apart from simply reading and writing files, they perform many operations on the transient OS structures (e.g., file descriptor table, process table, etc.).

For a precise trace analysis (e.g., correctly resolving the absolute file paths that a system call works on), we need to accurately track all the operations performed on these structures. This requires a careful design to deal with the complexities and semantics of the system calls as well as the underlying structure of the file system. For instance, many processes spawned by Puppet share their file descriptor table or working directory with their child processes.[1] Hence, any update to any of those entities performed by one process, also affects the other one. By ignoring this behavior, processes will hold stale information about their file descriptor table and working directory.

As another example, consider the following trace:

```
1   100 open("/usr/lib/perl5", O_RDONLY) = 3
2   100 rename("/usr/lib/perl5", "/usr/lib/perl") = 0
3   100 openat(3, "5.26/Socket6.pm", O_RDONLY) = 4
```

To determine the absolute path that `openat` operates on (`/usr/lib/perl/5.26/Socket6.pm`), we have to interpret the file `5.26/Socket6.pm` relative to the directory related to the file descriptor 3. To do so, we need to consider that in a Unix-like file system, every file is associated with an `inode` rather than a path. Thus, the file-related OS structures (e.g., file descriptor table) have to refer to `inodes` instead of path names. Existing approaches that ignore the organization of the file system, either cannot resolve the absolute path of the `openat` system call [37] (they do not support file descriptors), or they describe files through their paths [20]. The latter makes the corresponding file descriptor table hold the stale entry (3, `/usr/lib/perl5`), after the `rename` at line 2.

[1] For example, in Linux, this can be achieved through the `CLONE_FD` and `CLONE_FS` flags passed in the `clone` system call.

Granularity. Given the system call trace of a build script, such as Make, previous approaches assume that every step of a project build is performed through a separate process [20, 37]. In this context, the analysis estimates the dependencies among source files (e.g., the object file `foo.o` depends on the header file `bar.h`) by computing the input and output files of each process (i.e., it presumes that the output depends on the input). To verify the inferred dependencies, the existing work [20] triggers incremental builds by touching the input files, and then checks whether the output files are indeed changed in response to the updates of their dependencies.

However, the granularity of processes is not fruitful for domains, such as Puppet, where the same process may handle different execution phases. In such cases, the existing work [20] is not able to distinguish which file system resources are affected by which execution phase—and therefore, it is not able to infer the inter-relationships among execution steps.

We design an approach on analyzing system call traces that overcomes the limitations of the previous approaches as follows. To tackle low fidelity, we introduce FStrace, a representation that enables us to: (1) translate every system call into primitives that conquer the large number and complexity of POSIX (Unix/Linux) system calls by decomposing them into simpler building blocks (Section 3.2), and (2) formally model the effects of system calls on the file system and the transient OS structures (Section 3.3.1).

To address granularity, we split the main process that governs the execution (Puppet process) into different blocks that indicate the boundaries of every execution phase (Puppet resource). This allows us to infer the inter-relationships between all Puppet resources by examining their interactions with the underlying operating system (Section 3.3.2). In turn, this enables us to combine (as explained in Section 4) the trace analysis output (low-level analysis) with the program's relationships inferred statically by analyzing compiled Puppet catalogs (high-level analysis). This makes our approach efficient, as we are able to detect faults by monitoring *only* one Puppet execution, i.e., we do not need to apply Puppet manifests multiple times (as in the case of incremental builds) to verify the relationships extracted by the trace analysis. Note that our treatment of system call traces is generic and can be applied to other domains such as Make or Java Maven. In this case, the boundaries of every execution phase correspond to the application of every build rule.

3.2 Modeling System Call Traces

The first step of our analyzer is to parse a given system call trace and transform it into an FStrace representation. FStrace primitives are designed to model system calls that operate on file system

Preprint

$$
\begin{aligned}
& f \in F = \mathbb{Z}, \ \ z \in Proc = \mathbb{Z}^*, \ \ b \in BlockID, \ \ v \in File, \ \ p \in Path = v^* \\
& e \in Trace ::= x^* \\
& x \in Block ::= \textbf{begin}\ b\ (z, s)^*\ \textbf{end} \\
& s \in Sys ::= \textbf{delfd}\ f \mid \textbf{dupfd}\ f_1\ f_2 \mid \textbf{hpath}\ d\ p\ m \mid \textbf{hpathsym}\ d\ p\ m \mid \\
& \qquad \textbf{link}\ d_1\ p_1\ d_2\ p_2 \mid \textbf{newfd}\ d\ p\ f \mid \textbf{newproc}\ c^*\ f \mid \\
& \qquad \textbf{rename}\ d_1\ p_1\ d_2\ p_2 \mid \textbf{setcwd}\ p \mid \textbf{symlink}\ d\ p_1\ p_2 \mid \textbf{nop} \\
& m \in Eff ::= \textbf{consumed} \mid \textbf{produced} \mid \textbf{expunged} \\
& c \in Flags ::= \textbf{fd} \mid \textbf{cwd} \\
& d \in DirFd ::= f \mid \textbf{at_fdcwd}
\end{aligned}
$$

Figure 5: The syntax of FStrace.

resources. Some of the constructs are generic enough so that they can represent a family of system calls. Complex system calls are represented with a number of FStrace primitives—something that *decouples* their intricacies from each other. We group system calls into execution blocks, and we assign a unique ID to each of them.

The syntax of FStrace is shown in Figure 5. It consists 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) the constructs **fd** and **cwd** that indicate what kind of entities a spawned process shares with its parent, (2) primitives (**consumed**, **produced**, **expunged**) that stand for the types of the effect that a system call has on a file, and (3) 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 is 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.

FStrace models every system call $s \in Sys$ using eleven constructs. We have **setcwd** that changes the working directory of the current process, and three primitives for performing operations on file descriptors: (1) **newfd** creates a new file descriptor and relates it to the given path p, (2) **delfd** deletes the provided file descriptor from the corresponding table of the process, and (3) **dupfd** copies a given file descriptor and is used to model a number of system calls, such as `dup`-like system calls or `fcntl(fd, F_DUPFD)`. FStrace supports hard and symbolic links through the **link** and **symlink** constructs respectively, while it offers **newproc** for spawning new processes. FStrace models operations on file paths explicitly through the **hpath** primitive. **hpath** $d\ p\ m$ captures the effect m that a system call has on the path p. We consider p relative to the file descriptor d, when p is not an absolute path. When d is **at_fdcwd**, we interpret p as relative to the current working directory. **hpath** models the system calls that work on file paths. For instance, we represent the system call `mkdir("foo/bar")`—which creates a new directory at path `foo/bar`—as **hpath at_fdcwd** ("foo", "bar") **produced**. The **hpathsym** primitive operates in a way similar to **hpath**. In **hpathsym** though, if the given path 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`. The `rename` primitive arranges that an existing path is accessed through a new file path. Finally, FStrace has a dedicated construct (**nop**) to model all system calls that we do not need to take into account, e.g., `write`, `read`, `sync`.

To leverage FStrace, the analyzer classifies system calls according to the applied Puppet resource that triggered them. Our analyzer presumes that an execution block begins or ends when the evaluation of the corresponding resource starts or terminates, because Puppet processes every resource atomically. In this context, the

```
1   begin Exec[Initialize MySQL DB]
2     103 hpath (usr, sbin, mysqld) consumed # execve
3     650 newproc {} 660 # clone
4     660 hpath at_fdcwd (etc, mysql, my.cnf) consumed # open
5     660 newfd at_fdcwd (etc, msql, my.cnf) 3 # open
6     660 nop # read
7   end
8   begin File[/etc/mysql/my.cnf]
9     103 hpath at_fdcwd (etc, mysql, my.cnf20190128-32-15kba2r)
          produced # open
10    103 newfd at_fdcwd (etc, mysql, my.cnf20190128-32-15kba2r)
          7 # open
11    103 nop # write
12    103 delfd 7 # close
13    103 hpath at_fdcwd (etc, mysql, my.cnf20190128-32-15kba2r)
          expunged # rename
14    103 hpath at_fdcwd (etc, mysql, my.cnf) produced # rename
15    103 rename at_fdcwd (etc, mysql, my.cnf20190128-32-15kba2r)
          at_fdcwd (etc, mysql, my.cnf) # rename
16  end
```

Figure 6: The FStrace representation of the trace of Figure 4.

name of the execution block corresponds to the name of the Puppet resource. It is easy to identify the points where the evaluation of a Puppet resource starts/finishes by decoding the Puppet's debug messages. Recall from Figure 4 that those messages appear in the execution traces as writes to the standard output. During trace parsing, the analyzer detects those debug messages and marks them as the entry and exit points of execution blocks.

For example, consider again the trace in Figure 4. We can model the trace entry at line 1 as the entry point of an execution block whose name is "`Exec[Initialize MySQL DB]`", whereas the system call at line 6 signals the ending of that execution block. Hence, all system calls that appear between lines 1 and 6, are included in this block. Figure 6 shows the complete FStrace representation of the trace shown in Figure 4. Notice that some system calls are represented through a number of FStrace primitives. For instance, we model `open("/etc", O_RDONLY)=3` with **hpath** to indicate that we consume the file `/etc`, and **newfd** to associate the provided path with the file descriptor 3 returned by `open`.

3.3 Interpreting FStrace Programs

To infer the ordering and notification relationships among Puppet resources, the analyzer enumerates the set of files consumed, produced or expunged in every execution block. This is done by interpreting the FStrace program produced by the parser (Section 3.2).

$$
\begin{aligned}
& \iota \in INode = \{\iota_i \mid i \in \mathbb{Z}^*\} \cup \{\iota_r\} \quad \alpha \in Ident = \{\alpha_i \mid i \in \mathbb{Z}^*\} \\
& \tau \in INodeT = (INode \times Filename) \to INode \\
& \pi \in FdT = Ident \hookrightarrow (F \hookrightarrow INode) \\
& \nu \in ProcT = Proc \to (Ident \times Ident) \\
& \phi \in CwdT = Ident \hookrightarrow INode \\
& \kappa \in SymT = INode \hookrightarrow Path \\
& \rho \in FSAcc = Path \hookrightarrow \mathcal{P}(Eff \times BlockID)
\end{aligned}
$$

Figure 7: Semantic domains for FStrace.

3.3.1 Domains and Semantics. We define a semantics for FStrace that we use as a base for our interpretation. Figure 7 illustrates the semantic domains of FStrace. The FStrace state consists of

Preprint

six components: An `inode` table $\tau \in INodeT$ is a map of a pair, consisting of an `inode` and a file name to another `inode`. The first element of the pair is the `inode` of the directory where the file name exists. 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 "/". A file descriptor table $\pi \in FdT$ maps an identifier and a file descriptor to an `inode`. We use this component to map the open file descriptors of a process to the resource they handle. The $CwdT$ element maps a unique identifier to an `inode`. That `inode` stands for the current working directory of a process.

Each process points to a pair of unique identifiers (see the domain *ProcT*). The first element of the pair is the identifier corresponding to the file descriptor table of the process. The second element of the pair reflects the identifier that stands for the current working directory of the process. This part of the state allows us to model the case where 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 (α_1). Similarly, since their second identifiers do not match ($\alpha_2 \neq \alpha_3$), we presume that they do not share the same working directory; thus, any change imposed by one process does not affect the other one.

We use the table $\kappa \in SymT$ to store symbolic links. Each symbolic link is an `inode` that points to a file path. The last component of FStrace ($\rho \in FSAcc$) 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. We exploit this component later on to extract the ordering and notification relationships of every block with each other. The state $\langle \tau, \pi, \phi, \nu, \kappa, \rho \rangle$ is a tuple consisting of the six components described above.

Figure 8 shows a small subset of the interpretation rules of FStrace.[2] Each rule defines state transitions as follows:

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

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 $\langle \tau', \pi', \phi', \nu', \kappa', \rho' \rangle$. The binary operator :: denotes the addition of an element to a set, while $\downarrow_i$ manifests the projection of the ith element. The function $\text{Ab}(d, p, \dots)$ gives the absolute path of the path p relatively to the given file descriptor d. The function $I(p, \tau)$ computes the `inode` that the path p points to based on the `inode` table τ. For example, the [HPATH] rule records the effect m that a system call has in the current execution block b by updating the ρ component of the state.

3.3.2 Inferring Relationships. Based on the state derived from the interpretation of an FStrace program, we now formally define the ordering and notification relationships between two Puppet resources. To achieve this, we exploit the computed file accesses performed in every block as defined in the resulting state ($\rho \in FSAcc$). Recall that the component ρ shows the set of files that are consumed, produced and expunged inside every execution block.

[2] The rest rules are described in the long version of the paper.

NEWPROC-SHARE

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

DUPFD

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

HPATH

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

Figure 8: A subset of the interpretation rules of FStrace.

Ordering Relationship. For ordering relationships we consider that a block b_1 producing a certain file p must precede a block b_2 that consumes or expunges the same file p. If this is not the case—and assuming that b_2 does not create p—there is an ordering violation. Formally, given the ρ instance found in the FStrace state we define the ordering relationship $\prec_\rho$, which states that the block b_1 comes *before* b_2, as follows:

$$\begin{aligned} & b_1 \prec_\rho b_2 \Rightarrow \\ & \quad \exists p \in Path, m \in Eff.\ m \neq \textbf{produced} \land \\ & \quad (\textbf{produced}, b_1) \in \rho(p) \land (m, b_2) \in \rho(p) \land (\textbf{produced}, b_2) \notin \rho(p) \end{aligned} \tag{1}$$

Consider again the program of Figure 6. After interpreting it, the analyzer outputs the following relationship: `file` $\prec_\rho$ `exec`, where `exec` stands for the block `Exec[Initialize MySQL DB]` (lines 1–7), while `file` corresponds to `File[/etc/mysql/my.cnf]` (lines 8–16). In particular, `exec` consumes the file `/etc/mysql/my.cnf` at line 4, while `file` produces the same file at line 14. According to the Definition 1, the creation of the file must be processed fist, so the analyzer presumes that the block `file` must precede `exec`.

Notification Relationship. In order to define the notification relationship we first need to identify pairs of execution blocks where the application of the first element should trigger the application of the second one. In the context of Puppet, such relationships involve `service` resources. Specifically, we look for blocks that produce a particular resource p. These blocks must have notification relationships with service-oriented blocks consuming the resource p. Formally, based on the component $\rho \in FSAcc$ of the state, we introduce the notification relationship $\rightsquigarrow_\rho$ that represents that the block b_1 *notifies* b_2:

$$\begin{aligned} & b_1 \rightsquigarrow_\rho b_2 \Rightarrow \\ & \quad \exists\, p \in Path, m \in Eff.\ \text{isService}(\text{b}_2) \land \\ & \quad (b_1, \textbf{produced}) \in \rho(p) \land (b_2, \textbf{consumed}) \in \rho(p) \end{aligned} \tag{2}$$

4 DETECTING FAULTS

Having introduced our approach for analyzing traces, we locate faults in Puppet manifests by combining the analysis output with the compiled catalog of a program. Our fault detection method performs two tasks: (1) given a catalog, it builds the dependency graph,

Preprint

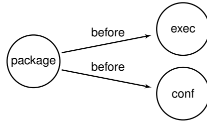

Figure 9: The dependency graph of the program of Figure 1.

a directed acyclic graph that captures all the ordering relationships and notifications declared by the developer (Section 4.1), and then, (2) it verifies that the relationships inferred by the trace analysis step appear in the dependency graph (Section 4.2).

4.1 Dependency Graph Construction

We define the dependency graph as $DG = (V, E)$, where V is the set of Puppet resources, $E \subseteq V \times V \times L$ is the set of edges, and $L = \{\textbf{before}, \textbf{notify}\}$, a set of labels that we assign to every edge. An edge $\overset{\text{before}}{\to}$ from a source node s to a target t indicates that Puppet applies s *before* t. An edge $\overset{\text{notify}}{\to}$ denotes that, apart from preceding t, s also *notifies* the target whenever there is an update to itself. When a node t is reachable from s, we presume that the application of s happens before that of t. On the other hand, a node notifies a target when they are connected with a path where all edges are $\overset{\text{notify}}{\to}$. This is explained by the fact that the $\overset{\text{notify}}{\to}$ edges transitively trigger updates to all the intermediate nodes between the source and the target resource.

To construct the graph, we parse a program's catalog, and we examine the parameters of every Puppet resource (recall Section 2). Specifically, given the parameters of a resource p, we create edges as follows.

- p has the parameter `"before":v`. This indicates that the resource p is applied before every resource included in the value of `"before"`. In this case, we add a $\overset{\text{before}}{\to}$ edge from p to every element of the list `v`.
- p has the parameter `"require":v`. This indicates that Puppet processes p after every element included in the value of `"require"`. Thus, we create a $\overset{\text{before}}{\to}$ edge from every element of `v` to p.
- p has the parameter `"notify":v`. The same as the `"before"` parameter, but this time, we create $\overset{\text{notify}}{\to}$ edges.
- p has the parameter `"subscribe":v`. The same as the `"require"` parameter, but this time, we create $\overset{\text{notify}}{\to}$ edges.

Figure 9 depicts the dependency graph of the program of Figure 1. We observe that the configuration file (i.e., the node `conf`) has neither an ordering nor a notification relationship with the `service`, because the corresponding nodes are not connected to each other.

4.2 Fault Detection

Algorithm 1 summarizes our fault detection approach. The algorithm expects as input the relationships of every Puppet resource as specified by the analysis of traces, and a dependency graph $g \in DG$ generated by the previous step. For every ordering relationship between two resources ($b_1 \prec_\rho b_2$), the algorithm consults the dependency graph to determine whether there is path from b_1 to b_2, i.e., it checks whether this ordering relationship actually appears in the program. If this is not the case, the algorithm reports a missing ordering relationship, that is, b_1 must be applied before b_2. To identify missing notifiers, the algorithm operates in a similar manner. In this case though, the algorithm is interested in paths that contain only $\overset{\text{notify}}{\to}$ edges (i.e., the function hasNotificationPath, line 8).

Algorithm 1 Detecting Faults

Input: $\rho \in FSAcc$, $g \in DG$
1: **for all** $(b_1, b_2) \in V \times V$ **do**
2: **if** $b_1 \prec_\rho b_2$ **and not** hasPath(g, b_1, b_2) **then**
3: report MOR between b_1, b_2
4: **end if**
5: **if** $b_1 \rightsquigarrow_\rho b_2$ **and not** hasNotificationPath(g, b_1, b_2) **then**
6: report MN from b_1 to b_2
7: **end if**
8: **end for**

As as an example, recall that the analyzer yielded the `file` $\prec_\rho$ `exec` relationship when it examined the trace file of Figure 4. The algorithm verifies this relationship by viewing the dependency graph of Figure 9. It then reports a missing ordering relationship because there is not a path from `file` to `exec`.

Remark: Observe that our approach does not make any assumption about the execution order imposed by Puppet, i.e., it does not require the resources with missing ordering relationships to be applied in the right order so that it infers their inter-dependencies. It is clear from the example trace of Figure 4 (where Puppet executes resources in the erroneous order) that our method is still able to observe that `exec` depends on `file` and eventually report the fault.

5 IMPLEMENTATION

We have developed a prototype that implements our approach in OCaml. The tool provides a command-line interface, and takes as input the path where the main Puppet manifest is located. It then stores the compiled catalog of the program, and executes it using `strace` to collect the system call trace. In turn, the tools employs the trace analyzer and fault detector as described in Sections 3, and 4.

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.

Currently, our tool has only been tested on Linux distributions. Also, as we discuss this in Section 6, our tool may produce false positives when two Puppet resources operate on the same file, but they are commutative to each other, i.e., the application order does not matter. Even though commutative pairs are not so common (see Section 6.5), we plan to address this issue in future work by examining Puppet catalogs to identify such pairs. For instance, we could identify resources whose execution is conditional, and depends on the presence of conflicting resources.

6 EVALUATION

We evaluate 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)

Preprint

RQ2 What are the main patterns of the detected faults? (Section 6.3)

RQ3 What is the performance of our approach? (Section 6.4)

6.1 Experimental Setup

We collected a large number of Puppet modules taken from Forge API and Github. We were particularly interested in non-deprecated modules that support Debian Stretch, because Debian is one of the most popular Linux distributions [1]. We inspected the top-1000 modules returned by the Forge API that satisfied our search criteria. We used Docker to spawn a clean Debian environment efficiently. Then, we automatically ran every module separately through the `include <module-name>` statement.[3] We monitored the Puppet process and collected the system call trace of every program via `strace`. Through this process, we successfully applied 354 Puppet modules in total. The remaining modules failed because they required extra arguments or further setup before their invocation. For example, many of the failing modules required multiple pre-installed packages, or in other cases we needed to infer specific values for the modules' arguments including IPs of DNS servers and URLs of specific upstream directories. Note that the failing modules and the successful ones were pretty similar in terms code size, popularity, and age in Puppet Forge. Notably, the list of analyzed modules contains well-established ones, including modules developed by popular organizations, such as Puppet Inc., and Vox Pupuli.[4] Finally, for every Puppet module that succeeded, we ran each step of our approach (trace analysis and fault detection) and logged the reports generated by our tool.

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 machine with an Intel i7 2.2GHZ processor with 12 logical cores and 16GB of RAM.

6.2 Fault Detection Results

Our framework detected 92 previously unknown faults in 33 Puppet modules. Table 1 presents the analysis results for each module. To the best of our knowledge, this is the first study that led to the disclosure of such a large number of faults in Puppet repositories. Our tool marks 70 out of 92 faults as missing ordering relationships (column MOR). The remaining faults are related to missing notifiers (column MN). Remarkably, our tool found faults in modules that are widely used by the Puppet community e.g., `puppetlabs-apache` (> 9000k downloads), and `deric-zookeeper` (> 4500k downloads),

Based on the reports of our tool, we manually verified that each reported fault can lead to a problematic situation by reproducing each case. Specifically, we repeatedly applied every manifest in a clean container until Puppet applied resources in the wrong order, leading to a failure or an inconsistent state. Only few trials were needed (1–3) for that. In turn, we submitted fixes to the developers. The development teams of 24 projects confirmed and fixed 62/92 issues in total. The developers welcomed our patches. Notably, in some projects (e.g., `deric-zookeeper`, `Slashbunny-phpfpm`, `cirrax-dovecot`, and more), the developers provided instant bug-fix releases after the approval of our patches. This indicates that our tool detects faults that are meaningful to developers. At the time of the submission, none of our patches have been rejected.

[3] `include` applies all the resources defined in the module using the default settings.
[4] Vox Pupuli is a big community that is currently managing and maintaining more than one hundrend Puppet modules. https://voxpupuli.org/

Table 1: Faults found in Puppet modules. Each table entry consists of the name of the module, the number of faults detected by our tool and a check mark indicating whether our fixes were accepted by the module's developers.

#	Module	Number of Faults			Fix
		Total	MOR	MN	Accepted
1	puppet-proxysql	10	10	0	✔
2	istlab-stereo	9	9	0	✔
3	olivierHa-influxdb	7	6	1	-
4	hetzner-filebeats	6	5	1	✔
5	geoffwilliams-auditd	5	5	0	✔
6	coreyh-metricbeat	4	4	0	✔
7	coreyh-packetbeat	4	4	0	✔
8	norisnetwork-packetbeat	4	4	0	✔
9	Slashbunny-phpfpm	4	2	2	✔
10	nogueirawash-mysqlserver	3	3	0	-
11	cirrax-dovecot	3	1	2	✔
12	nextrevision-flowtools	3	1	2	✔
13	deric-zookeeper	3	0	3	✔
14	albatrossflavour-os_patching	2	2	0	✔
15	hardening-os_hardening	2	2	0	✔
16	vpgrp-influxdbrelay	2	2	0	✔
17	puppet-collectd	2	1	1	✔
18	sgnl05-sssd	2	1	1	-
19	jgazeley-freeradius	2	0	2	✔
20	saz-ntp	2	0	2	-
21	walkamongus-codedeploy	1	1	0	✔
22	spynappels-support_sysstat	1	1	0	-
23	roshan-mysqlzrm	1	1	0	-
24	puppetfinland-nano	1	1	0	✔
25	noerdisch-codeception	1	1	0	-
26	baldurmen-plymouth	1	1	0	✔
27	saz-locales	1	1	0	-
28	alertlogic-al_agents	1	1	0	-
29	puppet-telegraf	1	0	1	✔
30	puppetlabs-apache	1	0	1	✔
31	example42-apache	1	0	1	✔
32	alexharvey-disable_transparent_hugepage	1	0	1	✔
33	camptocamp-ssh	1	0	1	✔
	Total	**92**	**70**	**22**	**62/92**

6.3 Fault Patterns

Below, we categorize and discuss some of the faults identified by our tool. Most represent previously unknown to us fault patterns which we learned through our tool. Notably, these detected faults can lead to crashes, inconsistent states, and data loss.

6.3.1 Missing Ordering Relationships. We have observed three 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. Consequently, the execution of Puppet may complete with failures, when resources are applied in an erroneous order. We observed this violation in 16 Puppet modules such as `alertlogic-al_agents`, `hardening-os_hardening`, and more.

Figure 10 shows a fragment from `alertlogic-al_agents` [17]. The code first fetches a `.deb` package (a Debian archive) using the

Preprint

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

Figure 10: An ordering violation between `package` **and** `exec`.

`wget` command (lines 2–4). The package is stored in the path specified by the `$package_path` variable whose value is `/tmp/al-agent`. Then, the code installs the Debian archive on the system (lines 5–9) through the `dpkg` package management system. It is easy to see that the package depends on the `exec`, because it requires `$package_path` (the `.deb` file) to exist in the system (line 8) so that it can install the package successfully. Otherwise, when Puppet 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 resource 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 five modules, including `saz-ntp`, `vpgrp-influxdbrelay`, and `jgazeley-freeradius`.

Figure 1 illustrates a program with an issue related to this category. When the shell script is invoked, the configuration file is guaranteed to be there because `package` creates it during installation. However, it is possible that `exec` does not read the desired contents of the `/etc/mysql/my.cnf` file specified by `content => "user db settings.."` (line 4), because there is a missing ordering relationship between `file` and `exec`. Note that this category—unlike the previous one—may lead to errors that are difficult to debug as the application of the catalog does not produce any error messages.

API Misuse. Many Puppet modules expose an API that other modules rely on to build their functionality. These APIs may establish some constraints that the dependent modules need to respect to achieve the intended functionality. As with traditional software [2, 18], failing to do so can have a negative impact on the reliability of applications. In particular, in Puppet, API misuses can lead to missing dependencies and race conditions. Eight modules (such as `puppet-proxysql`) do not properly use the API of their dependencies, causing the ordering violations reported by our tool.

For example, the `puppetlabs-apt` module provides an interface for managing `apt`[5] sources and keys. The API of this module includes—among other things—the `apt::source` resource and the `apt::update` class. The former is used for adding new repositories to the list of `apt` sources, while the latter retrieves all the essential information about the newly-added repositories by executing the `apt update` command. The `puppet-proxysql` module employs the `apt::source` resource to add the `http://repo.proxysql.com/` repository from which it installs `proxysql` (via the `package` resource). The documentation of the `puppetlabs-apt`'s API [26] states: *"If you are adding a new source and trying to install packages from the new source on the same Puppet run, your package resource should depend on Class['apt::update'], as well as depending on the Apt::Source resource"*. However, the developers of `puppet-proxysql` consider only the `Apt::Source` dependency in their code, i.e., they omit the `Class[apt::update]` dependency. As a result, the code may crash with an *"Unable to locate package proxysql"* message, when Puppet tries to install `proxysql`, before invoking the `apt update` command first. The developers of `puppet-proxysql` immediately confirmed and fixed this fault.

6.3.2 Missing Notifiers. We have identified three 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 12, 19, 20, 32 of Table 1) this is not the case.

Log Files. Typically, services log various events in dedicated files. For instance, the log file of an Apache server records every incoming HTTP request. Log files are essential for debugging and monitoring purposes [35, Item 56]. 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`, and `deric-zookeeper`). 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 data loss. Consider the case where the log file of a service is removed or renamed. When we remove or rename an open file, the underlying system call (`unlink` or `rename`) only changes the file entry, not the `inode`. This means that although the filename disappears from the file system and Puppet creates a new one, the service still uses a file descriptor that points to the `inode` of the original file. The issue is that after removal, the `inode` typically 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. The developers of both projects confirmed this kind of fault.

Packages. When Puppet applies a `package` resource, the service that depends on that package should restart. This ensures that a service gets all the necessary updates, including, security patches, new features, and more. Our tool identified this kind of issue in twelve modules, including `example42-apache`, and `puppet-telegraf`. Specifically, the `package` resources that are responsible for installing Apache, and telegraf do not notify the running instances whenever there is a new version of those packages.

6.4 Performance

Figure 11 shows the running times (in seconds) of the trace analyzer and fault detector 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. Notably, our framework is able to

[5] https://salsa.debian.org/apt-team/apt

Preprint

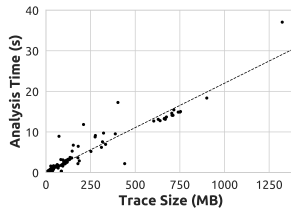

Figure 11: The trace analysis and fault detection time as a function of the trace size. Each spot shows the average time spent on both the trace analysis and fault detection phases for a given trace obtained by the execution of a module.

handle a large volume of traces (more than 1.2GB) in a reasonable amount of time (< 40 seconds). The average trace size and analysis time of the inspected modules is 72 MB and 2 seconds respectively.

There are 4 cases out of 354 where the execution times were relatively high compared to the remaining modules. However, they all remain in acceptable limits. By examining the characteristics of the traces coming from these modules, we observed that they contain more `unlink` system calls than the rest of the modules. Such calls involve more expensive operations on the analysis state.

For collecting traces using `strace`, our tool imposes a 1.68 times slowdown on Puppet, on average. Unlike existing work [13], though, our approach requires only a single Puppet execution to locate faults. Overall, we argue that the overhead of our tool is relatively small, and our approach can be used as part of the testing process for Puppet manifests.

6.5 False Positives

Beyond the actual faults listed in Table 1, we have identified 17 false positives in 9 out of 354 Puppet modules Fourteen false alarms are related to commutative resources reported as missing ordering relationships. For example, in the `claranet-varnish` module [5], the developers use two different resources 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 resources 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. However, as observed in the inspected modules, configuring a file through the combination of resources is not particularly common.

Only three false positives are associated with missing notifiers. The developers of `bodgit-dbus` [7] use a custom command (expressed via `exec`) to reload the configuration of the service. Consequently, the configuration files notify the `exec` resource instead of `service`. We did not observe this case elsewhere, because Puppet programmers typically employ 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. [33] 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. [36] introduce a quality model for Puppet programs which is empirically evaluated by interviewing practitioners from industry. Schwarz et al. [31] do similar work focusing on Chef recipes. Endeavors have recently moved to the identification of security issues. Rahman et al. [28] define and classify security smells into seven categories (such as hard-coded passwords, 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. [29] 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. [4] 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 [12].

There are few automated techniques proposed for improving the reliability of configuration management programs. Rehearsal [32] 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. Then, 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` resources because it is unable to reason about the file system resources that shell commands process. Unlike Rehearsal, our approach works by reasoning actual system calls rather than Puppet manifests; thus, it can effectively determine which files are affected by a Puppet run and how.

Other advances [13, 16] adopt a model-based testing approach for checking whether configuration scripts meet certain properties. Hummer et al. [16] 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 trigged by a Chef script, they determine if idempotence holds for the given program. Hanappi et al. [13] 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 resources. 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 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 in the context of dependency inference

Preprint

for builds scripts. Many existing approaches are used for refactoring and testing [11], boosting performance [3, 6] of builds, or detecting license inconsistencies in open-source projects [37]. As discussed in Section 3.1, most of them do not deal with the intricacies of system calls. This degrades the precision and effectiveness of the analysis. Also, unlike existing work, our trace analysis enables us to compute relationships in higher-level of abstraction, i.e., Puppet resources.

Modeling File System Operations. Several researchers have designed specifications for the POSIX file system [9, 14, 24]. The specifications mainly focus on program reasoning and verification. Shambaugh et al. [32] have introduced FS; a small language used to model the effects of Puppet resources on the file system. In this work, we model system calls rather than Puppet resources.

8 CONCLUSION

We have introduced an effective and practical approach for identifying missing dependencies and notifiers in Puppet manifests. Our method collects the system calls invoked by a Puppet program and models them in FStrace. Through FStrace, we design a trace analysis that captures how higher-level programming constructs, such as Puppet resources, interact with the operating system. This enables us to infer their inter-relationships, and check these relationships against the program's dependency graph for potential mismatches.

The effectiveness of our approach is exemplified by the uncovering of 92 previously unknown issues in 33 projects, including well-established ones, such as `puppetlabs-apache`. Notably, 62 of them were confirmed and fixed by the developers. We have further showed that our tool can handle realistic traces in a couple of seconds. 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. Consequently, future studies can build on our work to detect concurrency faults in many other areas.

Preprint

REFERENCES

[1] Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. German. 2016. An empirical study of integration activities in distributions of open source software. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 960–1001.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* (2018), 1–1.

[3] Glenn Ammons. 2006. Grexmk: Speeding Up Scripted Builds. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis* (Shanghai, China) *(WODA '06)*. ACM, New York, NY, USA, 81–87.

[4] W. Chen, G. Wu, and J. Wei. 2018. An Approach to Identifying Error Patterns for Infrastructure as Code. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 124–129.

[5] Claranet. 2019. Install and configure Varnish Cache. https://forge.puppet.com/claranet/varnish.

[6] Derrick Coetzee, Anand Bhaskar, and George Necula. 2011. apmake: A reliable parallel build manager. In *2011 USENIX Annual Technical Conference (USENIX)*.

[7] Matt Dainty. 2019. Puppet Module for managing D-Bus. https://forge.puppet.com/bodgit/dbus.

[8] Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. 2010. A Survey of System Configuration Tools. In *Proceedings of the 24th International Conference on Large Installation System Administration* (San Jose, CA) *(LISA'10)*. USENIX Association, Berkeley, CA, USA, 1–8.

[9] L. Freitas, Z. Fu, and J. Woodcock. 2007. 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)*. 3–14.

[10] Github. 2014. DNS Outage Post Mortem - The GitHub Blog. https://github.blog/2014-01-18-dns-outage-post-mortem/. [Online; accessed 28-January-2019].

[11] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014. Automated Migration of Build Scripts Using Dynamic Analysis and Search-based Refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 599–616.

[12] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 144–156.

[13] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting Reliable Convergence for Configuration Management Scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 328–343.

[14] Wim H. Hesselink and Muhammad Ikram Lali. 2012. Formalizing a hierarchical file system. *Formal Aspects of Computing* 24, 1 (01 Jan 2012), 27–44.

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

[16] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *Middleware 2013*, David Eyers and Karsten Schwan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–388.

[17] Alert Logic Inc. 2019. Alert Logic Agent Puppet Module. https://forge.puppet.com/alertlogic/agents.

[18] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and Efficient API Misuse Detection via Exception Propagation and Search-based Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 192–203.

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

[20] Nándor Licker and Andrew Rice. 2019. Detecting Incorrect Build Rules. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1234–1244.

[21] James Loope. 2011. *Managing Infrastructure with Puppet: Configuration Management at Scale.* O'Reilly Media.

[22] Richard McDougall, Jim Mauro, and Brendan Gregg. 2006. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris.* Prentice Hall PTR, Upper Saddle River.

[23] Kief Morris. 2016. *Infrastructure As Code: Managing Servers in the Cloud* (1st ed.). O'Reilly Media, Inc.

[24] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. 2018. A Concurrent Specification of POSIX File Systems. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.4

[25] Shawn Plummer and David Warden. 2016. Puppet: Introduction, Implementation, & the Inevitable Refactoring. In *Proceedings of the 2016 ACM on SIGUCCS Annual Conference* (Denver, Colorado, USA) *(SIGUCCS '16)*. ACM, New York, NY, USA, 131–134.

[26] Puppet. 2019. Provides an interface for managing Apt source, key, and definitions with Puppet. https://forge.puppet.com/puppetlabs/apt.

[27] Puppet. 2019. Puppet Forge. https://forge.puppet.com/.

[28] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure As Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 164–175.

[29] A. Rahman and L. Williams. 2018. Characterizing Defective Configuration Scripts Used for Continuous Deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 34–45. https://doi.org/10.1109/ICST.2018.00014

[30] R. Rodriguez. 1986. A System Call Tracer for UNIX. In *USENIX Conference Proceedings* (Atlanta, GA). USENIX Association, Berkeley, CA, USA, 72–80.

[31] Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 220–228.

[32] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 416–430.

[33] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 189–200.

[34] D. Spinellis. 2012. Don't Install Software by Hand. *IEEE Software* 29, 4 (July 2012), 86–87.

[35] Diomidis Spinellis. 2016. *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley Professional, Boston, MA.

[36] Edward van der Bent, Juriaan Hage, Joost Visser, and Georgios Gousios. 2018. 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)* (Campobasso, Italy) *(SANER 2018)*. 164–174.

[37] Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. 2014. Tracing Software Build Processes to Uncover License Compliance Inconsistencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 731–742.

[38] Joost Visser, Sylvan Rigal, Gijs Wijnholds, and Zeeger Lubsen. 2016. *Building Software Teams: Ten Best Practices for Effective Software Development.* " O'Reilly Media, Inc.".

[39] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 244–259.

Preprint