Automated Testing of Configuration Management Libraries

Marc Lundwall
mlundwall@student.ethz.ch
ETH Zürich
Switzerland

Karim Umar
umarka@student.ethz.ch
ETH Zürich
Switzerland

ABSTRACT

This report documents an automated testing approach for Ansible and Puppet, two open-source automation tools used for configuration management, application deployment, and task automation. The motivation for this testing is to identify bugs in the modules' configuration options, which can lead to crashes, security vulnerabilities, or unexpected behaviour of the target system.

We describe our approach, which involves mutating existing programs with transformations that are likely to cause the discovery of a bug. These transformations are designed in a way such that if these automation tools were bug-free, the mutated programs would always result in an equivalent execution.

The exploration of appropriate test oracles led us to adding a snapshot-based system, where for every step of the execution, a subset of the state of the system is compared between the executions of the unmodified and modified programs.

By leveraging pre-existent integration tests, we have a large collection of correct sample input programs to mutate. These transformations are automatically chosen and customized for each module under test by parsing the relevant information from that module's documentation.

We demonstrate the effectiveness of our approach by showing it can detect previously reported bugs in widely used Ansible modules.

The source code for this project is available on GitHub [8]

ACM Reference Format:
Marc Lundwall and Karim Umar. 2023. Automated Testing of Configuration Management Libraries. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ansible[1] and Puppet[5] are open-source automation tools used for configuration management, application deployment, and task automation. Both frameworks use modules, which are designed to perform specific tasks on target systems.

Finding bugs in these libraries is an interesting task because Ansible and Puppet are widely used by system administrators and DevOps teams. By automating the bug detection process, we can minimize the impact of incorrect configuration parameters, reduce time spent on manual testing, and improve the overall quality of the modules.

Our approach of testing these modules is motivated by the success of approaches like Hephaestus[3] in compiler testing. We take programs known to be correct and modify them in a way that should not alter their execution, and check to see if this uncovers bugs.

We show that our method can be used to reproduce reported bugs for widely used modules, by applying our mutations to the integration tests rolled back to the time the bugs were reported.

This is, to our knowledge, the first exploration of metamorphic testing (through comparisons of the system state) applied to configuration management libraries.

2 APPROACH

At a first glance, a Configuration Management Library shares many characteristics with a compiler: both take as input a program, parse it and produce an output (either in the form of a binary executable or a sequence of actions performed on a target system). But the methods used for uncovering bugs in compilers are not naïvely transferable to testing Configuration Management Libraries. This stems from two main differences:

(1) The input format to Configuration Management Libraries lacks the rigorous semantic constraints of a programming language. Furthermore, sample configurations are too module-specific to be generalizable to all modules. This makes it infeasible to generate candidate programs from scratch that adequately cover all the module's functionality, since it would need to be done manually.
(2) The outcome of running a Configuration Management Library on a configuration file depends on the state of the target system, so whether a program can be considered to be correct therefore depends on more than just the program output, but also on how the target system is configured.

Nevertheless, we follow a scaled back approach of this method that is successfully applied to finding compiler bugs. We mutate programs known to be correct, in a way that should not alter their outcome. Then, after running these programs, we check for configuration failures, as well as inspect the configuration target for any differences in configuration. If any failures or differences in configurations are present, a bug has been uncovered.

We initially consult the GitHub issues pages of both Ansible and Puppet. Most bugs of interest cause one of 2 outcomes:

- Bugs that cause a failure: Example: The Ansible module rhsm_repository expects that output of the Red Hat subscription-manager program that it interacts with to be in English, so when this is not the case (i.e. when the target's locale is changed) the module fails unexpectedly because it cannot parse the program's output in different languages.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Bugs that cause wrong configurations: Example: The interfaces_file module from Ansible community has a bug in its regex formula to modify the interfaces file. This causes it to ignore some lines, which results in a different interfaces file than expected, with duplicated lines. Note that in this case, the execution wouldn't fail, so we need a way of catching this misconfiguration.

Our approach is based on the insight that both of these types of bugs can be triggered by taking an existing program that executes correctly, and modifying either the program itself or modifying the environment of the host and target system.

For example, this bug in lineinfile can be triggered by taking a working config task:

```
- lineinfile:
    name: ./myFile.txt
    line: "myLine"
    create: yes
```

And removing a "./" from the beginning of a file's path:

```
- lineinfile:
    name: myFile.txt
    line: "myLine"
    create: yes
```

Removing "./" should result in an identical execution as these paths are equivalent in Linux, but the incorrect handling of paths within the lineinfile module causes the bug to appear. Similarly, the rhsm_repository bug can be triggered by setting the target system's locale to French ("fr_FR.UTF-8") and then running any command from the rhsm_repository module.

We aggregate a repository of transformations that trigger bugs, and apply them automatically to existing correct programs to then detect either a failure or a wrong configuration.

2.1 Finding correct programs

As mentioned above, since the input format to Puppet and Ansible lacks strict semantic rules, it is infeasible to generate correct programs from scratch that cover enough of the module's functionality. Luckily, for both Ansible and Puppet, most modules already provide a host of integration tests. This repository of module-specific programs can be assumed to be correctly written, and covers a wide range of each module's functionality, so is an ideal starting point for modifications.

2.2 Transformations

We perform two broad types of Transformations:

- **Environment Mutations:** Here we apply modifications to the host and target systems before the tests are performed. These modifications include changing the file names, and changing environment variables.
- **Program Mutations:** Here we apply mutations to the actual configuration programs. These are harder to implement as they must guarantee an equivalent program execution, and are therefore limited in their scope and effect. An example of such a mutation is modifying the name of a file to include potentially problematic characters.

We also subdivide our transformations by their applicability:

- **General transformations:** these can be applied to all modules.
- **Option-specific transformations:** these transformations need to be configured on a per-module basis. This allows more advanced transformations to be applied. For example, a module's option which expects a path can be modified to anything with Unicode characters, which may not be the case for a module that expects URLs. Which option-specific transformations can be applied to a module's options is determined at the beginning of a testing run by exploiting the module's documentation, in the case of Ansible. This is sufficiently standardized across all modules to reliably know the type of each of the module's options, such as 'path'. We perform this check automatically when deciding which transformations can be run when testing a module, and parametrize them depending on the module's options.

Some transformations are randomized, and can benefit from being run multiple times. Deterministic transformations, however, only need to be tried once.

2.2.1 Example transformations. Here we present a few transformations and motivate their importance:

Add/Remove "./" This simple transformation illustrates our approach. As `./<path>` and `<path>` are equivalent on Linux, these two paths should be handled identically. We therefore introduce a transformation that appends or removes this from paths to induce bad handling of paths in the modules.

Modify Filenames This transformation aims to detect bad handling of potentially problematic filenames. This transformation scans all the relevant test cases, records the filenames referenced in the module under test's options that we know refer to files, and mutates them everywhere to a random Unicode string. As equivalent filenames are mapped to the same random string, the execution should be identical apart from the different names of files produced on the target system. The existing filenames are also modified. This transformation is only applicable to the options of a module that expect a path, which is automatically determined by leveraging the module's documentation.

Modify Field This transformation is identical to Modify Filenames, but it applies more generally to any field, not only representing a path. For example, this could be used for an option to select a name or a password. The difference here is that since the field isn't a path, we don't change the names of any files in the filesystem. Since the documentation often does not give a more precise type than 'str' for such fields, this transformation requires some information about the options of the module that it is allowed to modify.

Change Default Locale This transformation changes the locale environment variable before performing tests. Some Ansible modules depend on third-party software (a good example is the RHSM module). The output these third-party modules provide can depend on the system's environment, such as the `locale`. For example, if this output is expected to be in English but appears in the configured system language, this can cause an error. Such a bug was reported in the RHSM module, which expected the output of RHSM to be English and failed to parse the output correctly if the language

was not English.

Check Idempotency This transformation duplicates all instances of the calls to the module, to check if the module is idempotent. This is particularly important in configuration management systems, where the target could already be configured correctly, in which case nothing should happen when repeating the configuration steps. However, not all integration tests would succeed with such a transformation. For example, lineinfile is capable of removing the first line of a file, and running this twice would remove the first two lines, instead of only the first, which fails the test. So, this transformation should only be set when the module is guaranteed to be idempotent, something that is impossible to know automatically, even with the documentation.

Therefore, for a given module, we can automatically detect which options are filenames and select both the 'Change Filenames' and 'Add "./"' transformations with the relevant option names. Another transformation that can be always used, and so is part of the default transformations for all modules, is the 'Change Default Locale' transformation. Other transformations need to be selected and configured on a case-by-case basis, depending on the module.

2.3 Testing Life cycle

When testing a module, the sequence of events is as follows:

(1) Examine the module's options to determine which options can be subjected to more specific transformations. For options that expect the input to be in a certain format like paths, this allows us to subject this option to more specific transformations.
(2) Run the unmodified test and record system artefacts.
(3) Apply the transformations to the test configuration file or to the host/target system, run the test and collect artefacts same.
(4) If a crash is detected or there is a difference between artefacts of the unmodified tests and the modified tests, Record the output
(5) GOTO 3

This is illustrated in Figure 1

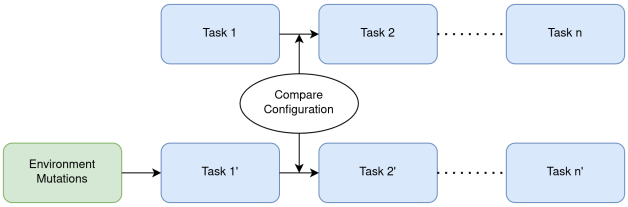

Figure 1: Illustration of our testing workflow

2.4 Test Oracles to Detect Bugs

We monitor two signals that indicate a bug is present:

(1) Ansible or Puppet reports a failure. This is done by monitoring the logs.
(2) A difference in the resulting configuration between modified and unmodified tests. We collect a range of information about the intermediate- and end-states of the target system, which is compared at the end of a run. This information includes:
- The structure of parts of the file system
- The hashes of config files
- The currently set environment variables and their values
- etc.

Some elements of the system are expected to change between runs, and so we exclude them from the data collection.

3 IMPLEMENTATION & RESULTS

3.1 Implementation

In this section, we provide a brief technical overview of our testing infrastructure. As Ansible and Puppet are built slightly differently, their infrastructure for testing is also different:

3.1.1 ***Ansible.*** To simulate actual host/target systems we use Docker[4], this lets us easily create a consistent environment for our tests. A single testing run requires two running Docker containers, one acts as a host, the other as a target. The host container executes the tests with the target container as its configuration target. As these operations take place via SSH, this only requires passing the target's IP address to the host at launch time, then the host performs the configuration as if the target container were a remote machine.2.

During a single test run, the docker containers are launched, any modifications to the environment are performed by executing a bash script, the modified tests are mounted into the host's file system and then the tests are performed as usual. During the test, we take snapshots of the target system every time the module under test is called.

3.1.2 ***Puppet.*** The most common Puppet testing framework is beaker-rspec[2]. Instead of integration tests being written through configuration files directly, such as .yaml playbooks or roles for Ansible, or the corresponding .pp manifest in Puppet, beaker-rspec allows the tests to be written in Ruby, with various helper tools. Its most important features are the integration with Serverspec[6] to be certain that the relevant aspects of the server are configured correctly, and the integration with Docker[4] to automatically provision a container with the necessary software. This gives the tester more freedom in designing comprehensive integration test suites, but requires some changes in the architecture of thefuzz compared to Ansible.

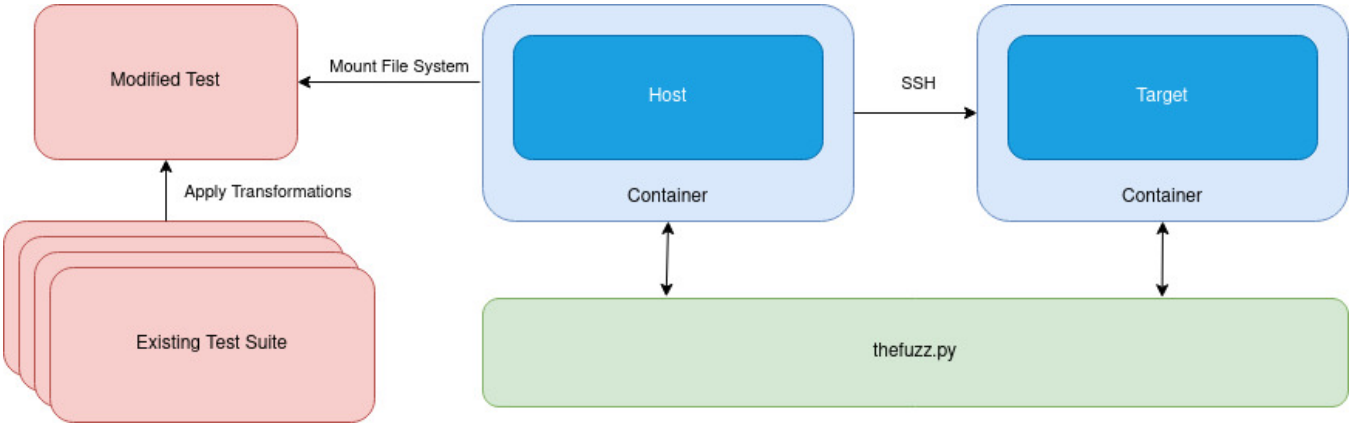

Figure 2: Ansible testing infrastructure overview.

The target Docker container is no longer created and managed by us, but by beaker-rspec directly. Transformations which rely on locating the calls to the module under test are also engineered to work with beaker-rspec's apply_manifest() function. We have created a modular interface to apply changes to Ansible and Puppet: lower-level manipulations, such as adding an option to the module call, or running a script after every task, or duplicating module executions, are all implemented for both Puppet and Ansible through the AnsibleModuleTest and PuppetModuleTest objects, inheriting from a BaseModuleTest object. Due to polymorphism, transformations interact with the parent class BaseModuleTest in a modular way, independent of the framework being used. Extending this system to other configuration management libraries is possible by simply implementing the abstract low-level manipulation methods of BaseModuleTest in another class.

3.2 Results

Although we did not manage to discover any new bugs in Ansible or Puppet, we provide instructions to reproduce two bugs in Ansible that inspired some of our transformations. This is done by reverting the Ansible code to a pre-bug-fix state and removing testcases that were introduced to cover the reported bug. Our transformations can take existing testcases and trigger the reported bugs. To achieve this, simply run the https://github.com/lundwall/thefuzz with the environment variable `REPRODUCE` set to either `lineinfile` or `rhsm`.

4 RELATED WORK

Academic research into automated testing of configuration management systems is limited. An automatic reliability testing system [7] has been successful in finding bugs in Kubernetes controllers, but to our knowledge, no similar work for Ansible or Puppet has been done.

5 CONCLUSION

5.1 Summary

To conclude, we have designed a novel testing framework, thefuzz, for configuration management libraries. Its extensibility makes it generalizable with little engineering effort to any configuration management library, not only Puppet and Ansible. Through the reuse of existing integration tests and by leveraging the documentation of tested modules, thefuzz can be used in an automated way with both official and community modules. Some manual feedback on the transformations for the tested module can greatly increase their pertinence, such as by specifying the expected type of an option. This framework would have caught the previously mentioned bugs, had it been available at that time. This tool can thus be invaluable for module developers to increase their confidence in the correctness of their developed modules.

5.2 Limitations and future work

A major limitation of our approach lies in the restricted scope of our transformations. We currently limit the transformations to transformations that are guaranteed to result in the same program execution and target state if no bugs are present.

A future path worth exploring would be to relax this condition. This would allow more extensive transformations to be applied, but would also require a mechanism for determining whether the target's end-state really is in the desired configuration.

Alternatively, we could envision transformations that should make the execution fail, and detect a bug if it does not fail. This would make our framework further resemble Hephaestus[3], which has both a transformation guaranteed to fail, and another guaranteed to succeed, thus finding bugs in both ways. What makes this complicated with Configuration Management Libraries is that their rules are more relaxed than a compiler's, and any such transformation would need to be module-specific, which goes against our aim of an automated approach.

Another avenue would be an approach similar to differential testing, where two configuration management libraries perform the same configuration and the resulting system state is compared. This is not feasible with Ansible and Puppet, as their input programs under test are structured differently and cannot easily be translated to each other's format.

REFERENCES

[1] *Ansible*. https://docs.ansible.com/. Accessed: 2023-03-22.
[2] *beaker-rspec*. https://github.com/voxpupuli/beaker-rspec. Accessed: 2023-06-06.
[3] Stefanos Chaliasos et al. "Finding typing compiler bugs". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 183–198.
[4] *Docker*. https://www.docker.com/. Accessed: 2023-06-06.
[5] *puppet*. https://www.puppet.com/. Accessed: 2023-06-06.
[6] *ServerSpec*. https://serverspec.org/. Accessed: 2023-03-22.
[7] Xudong Sun et al. "Automatic Reliability Testing for Cluster Management Controllers". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. July 2022.
[8] *thefuzz*. https://github.com/lundwall/thefuzz. Accessed: 2023-06-06.