Andreas Hortlund

Security smells in open-source infrastructure as code scripts

A replication study

Information Systems

Bachelor Thesis

Term:	VT-21
Supervisor:	Sana Rouis Skandrani

Karlstad Business School
Karlstad University SE-651 88 Karlstad
Phone: +46 54 700 10 00
E-mail: handels@kau.se kau.se/en/hhk

Abstract

With the rising number of servers used in productions, virtualization technology engineers needed a new a tool to help them manage the rising configuration workload. Infrastructure as code(IaC), a term that consists mainly of techniques and tools to define wanted configuration states of servers in machine readable code files, which aims at solving the high workload induced by the configuration of several servers. With new tools, new challenges rise regarding the security of creating the infrastructure as code scripts that will take over the processing load. This study is about finding out how open-source developers perform when creating IaC scripts in regard to how many security smells they insert into their scripts in comparison to previous studies and such how developers can mitigate these risks. Security smells are code patterns that show vulnerability and can lead to exploitation.

Using data gathered from GitHub with a web scraper tool created for this study, the author analyzed 400 repositories from Ansible and Puppet with a second tool created, tested and validated from previous study. The Security Linter for Infrastructure as Code uses static code analysis on these repositories and tested these against a certain ruleset for weaknesses in code such as default admin and hard-coded password among others. The present study used both qualitative and quantitative methods to analyze the data.

The results show that developers that actively participated in developing these repositories with a creation date of at latest 2019-01-01 produced less security smells than Rahman et al (2019b, 2020c) with a data source ranging to November 2018. While Ansible produced 9,2 compared to 28,8 security smells per thousand lines of code and Puppet 13,6 compared to 31,1. Main limitation of the study come mainly in looking only at the most popular and used tools of the time of writing, being Ansible and Puppet. Further mitigation on results from both studies can be achieved through training and education. As well as the use of tools such as SonarQube for static code analysis against custom rulesets before the scripts are being pushed to public repositories.

Keywords: infrastructure as code, security, Ansible, Puppet, static code analysis, security smells

Preface

Thank you to my supervisor Sana Rouis Skandrani and my thesis group for the feedback and help with writing this thesis. Thank you to my friends and family for the support during three years of studies.

Table of Contents

Dictionary *1*

List of Abbreviations *2*

1 Introduction *3*

1.1 Background **3**

1.2 Purpose **5**

1.3 Target group **5**

1.4 Research questions **6**

1.5 Demarcation **6**

1.6 Ethical considerations **6**

2 Literature overview *7*

2.1 Infrastructure as code **7**

2.1.1 Defining infrastructure as code 7
2.1.2 What problems does it solve? 7
2.1.3 Immutable vs Mutable 8
2.1.4 Configuration management vs provisioning 9
2.1.5 Challenges with infrastructure as code 10

2.2 Earlier studies **12**

2.2.1 Studies within implementing IaC and DevOps 12
2.2.2 Studies pioneering in security within IaC scripts 13

2.3 Security in infrastructure scripts **13**

2.3.1 Design patterns 14
2.3.2 Security smells 14
2.3.2.1 Admin by default 15
2.3.2.2 Empty password 16
2.3.2.3 Hard-coded secret 16
2.3.2.4 Invalid IP address binding 17
2.3.2.5 Suspicious comment 17
2.3.2.6 Use of HTTP without TLS 17
2.3.2.7 Use of weak cryptography algorithms 18
2.3.2.8 No integrity checks 18

2.4 Defect analysis **19**

2.4.1 Static code analysis 19

2.4.2 Dynamic code analysis 19
2.4.3 Code reviews 20
2.4.4 Syntax validation 20
2.4.5 Configuration simulation 20

2.5 Analysis model 21

3 Method 22

3.1 Data gathering 22

3.2 Static Code Analysis 23

3.3 Quantitative analysis 26

4 Implementation and results 26

4.1 Environment 26

4.2 Clarification regarding categories 27

4.3 Puppet results 27

4.4 Ansible results 29

5 Analysis 30

5.1 Reliability 30

5.2 Internal validity 30

5.3 External validity 31

5.4 Puppet period 1 vs period 2 31

5.5 Ansible period 1 vs period 2 33

5.6 Trends 35

5.7 Possible solutions 35

6 Conclusion 38

Bibliography 41

Appendices 45

Appendix 1: Scraper code snippet 45

Appendix 2: List of analyzed Puppet repositories (Precede URL with github.com) 46

Appendix 3: List of analyzed Ansible repositories (Precede URL with github.com) 54

List of Tables

Table 1. Key terms and their brief descriptions..1

Table 2. Abbreviations. 2

Table 3. Differences from Iron age to Cloud age. From Morris (2020, p. 28) used with permission..........8

Table 4. String patterns for SLIC tool. From Rahman et al (2020c, p. 15). 24

Table 5. Rulesets for SLIC Puppet tool. From Rahman et al (2019b, p. 6)...25

Table 6. Rulesets for SLIC Ansible tool. From Rahman et al (2020c, p. 14). 25

Table 7. Security smells for Puppet and their occurrences. 27

Table 8. Security smells for Ansible and their occurrences. 29

List of Figures

Figure 1. Visualization of consolidation. From Portnoy (2016, p. 10) used with permission......................3

Figure 2. Illustration of use cases for IaC tools. From Gattani (2020). 9

Figure 3. Illustration of configuration drift. From Morris (2020, p. 53) used with permission. 10

Figure 4. Illustration of fear spiral. From Morris (2020, p. 56) used with permission. 11

Figure 5. Screenshot of security smell example. From Rahman (2020a). 14

Figure 6. Pie chart of security smells categories for Puppet. 28

Figure 7. Pie chart of security smells categories for Ansible. 29

Figure 8. Bar chart of security smells comparing occurrences for Puppet...32

Figure 9. Bar chart of security smell category hard-coded secret comparing occurrences for Puppet....32

Figure 10. Bar chart of security smells per thousand lines of code for Puppet on previous study vs this study...33

Figure 11. Bar chart of security smells comparing occurrences for Ansible. 34

Figure 12. Bar chart of security smell category hard-coded secret comparing occurrences for Ansible. 34

Figure 13. Bar chart of security smells per thousand lines of code for Ansible on previous study vs this study...35

Dictionary

Table 1. Key terms and their brief descriptions.

Term	Description
DevOps	A set of practices that works to automate and integrate the processes between software development and IT teams
Infrastructure as code	The process of managing and provisioning computer data centers through machine-readable definition files.
Manifest	Another word used to reference to Puppet script
Playbook	Another word used to reference to Ansible script
Security smell	Recurring code patterns that are indicative of security weakness

List of Abbreviations

Table 2. Abbreviations.

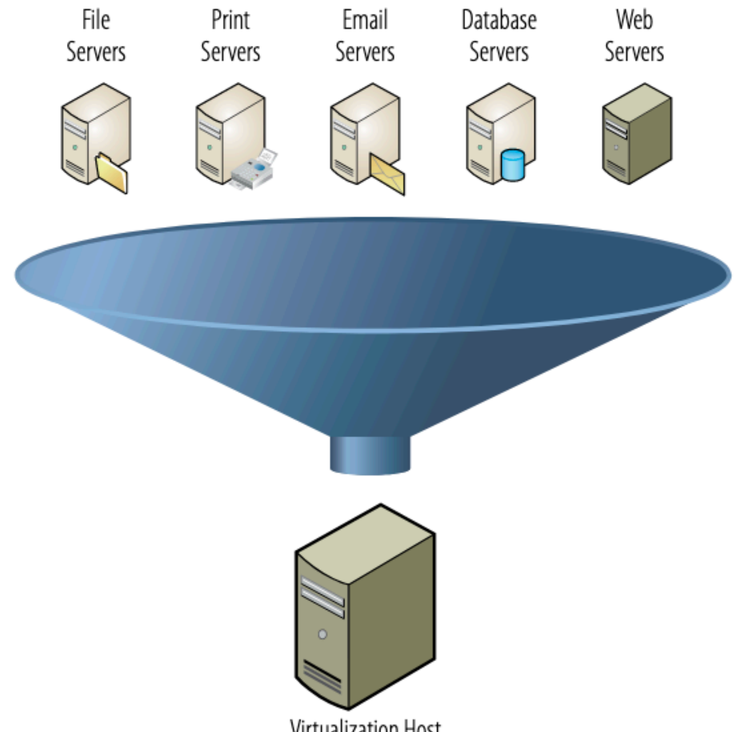
Term	Abbreviation
Application Programming Interface	API
Central Processing Unit	CPU
Comma-Separated Values	CSV
Common Weakness Enumeration	CWE
Domain Name System	DNS
Extensible Markup Language	XML
Hypertext Transfer Protocol	HTTP
Hypertext Transfer Protocol Secure	HTTPS
Information Technology	IT
Infrastructure as Code	IaC
Integrated Development Environment	IDE
Internet Protocol	IP
JavaScript Object Notation	JSON
Secure Shell	SSH
Secure Sockets Layer	SSL
Security Linter for Infrastructure as Code	SLIC
Thousand Lines of Code	KLOC
Transport Layer Security	TLS
YAML Ain't Markup Language	YAML

1 Introduction

1.1 Background

Today's servers and cloud infrastructure no longer run only on bare metal machines. Within a short space of time, engineers have come from working in the iron age of IT to the cloud age of IT as Morris (2020, p. 26) calls it. Back in the iron age of IT when machines were running only on physical hardware and every server instance was found in a rack somewhere in the datacenter where it resided. To the cloud age of IT with technology such as hypervisors allowing hardware resources such as CPU, memory and storage to transcend physical hardware and float across it. Yet engineers still need the physical hardware in place, but instead of running one server instance on each machine it is now possible to runs hundreds of servers on one physical machine. It is even possible to have dedicated machines solely to storage or CPU and have them act as together as one server. This provides huge benefits regarding costs and server space in datacenters, engineers can now consolidate servers to different ratios, as Portnoy (2016, p. 10) writes. A consolidation rate of 5:1, meaning that five servers are running on one physical host, as illustrated in figure 1.

Figure 1. Visualization of consolidation. From Portnoy (2016, p. 10) used with permission.



We will not go into detail on how hypervisors or virtualization work in this study. We can read a simple, yet through description of how hypervisors and virtualization work that Portnoy (2016) writes about. With the virtualization technology being more available, the cloud operation could expand when costs and space requirements decreased. This has led to that the workload that the administrators and engineers are expected to handle is unreasonable and unfeasible. Specifically, Morris (2020, pp. 24-25) means that the digitalization of business has put pressure to work faster. In combination with the consolidation rates of server that one physical server can hold several virtual ones meant that one person no longer could manage the number of servers needed to be configured or maintained. The cloud scale of things also means that engineers had to rethink designing distributed cloud systems and change the process of provisioning them.

In comes the concept of infrastructure as code to help administrators and engineers cope with the new explosion of running servers some tools were needed to automate tasks. Administrators have always used scripts to automate tasks and have them run on schedules, but this was on another scale. CFEnginge was the first tool the emerge back in 1993, even before the term IaC - Infrastructure as Code (Morris, 2020, pp. 12-13).

Today many tools in IaC and specifically configuration management exists. This brings new opportunities but new challenges as well. Since this is a relatively new field and the fact that it was first popularized in the middle of the 2000s with the rise of amazon web services, this may explain that the academic field have not had the opportunity to research IaC as much as other fields.

Most of the research done in the IaC field is related to how to adopt it into the DevOps methodology and handle implementation challenges. This is in part what Guerriero et al (2019) write mainly about in their research work on how to adopt and use the tools available within IaC, best practices, the practitioners need and what support is already available. Other studies focused on how to implement these tools practically with cloud service providers such as what Shvetcova et al (2019) work on how to extend the support for IaC and how to implement them to different cloud service providers and increase interoperability.

Merely a few studies are done on mapping the area around IaC and DevOps to reveal knowledge gaps and encourage researchers to investigate further in different areas. This is in part what Rahman et al (2019a, p. 74) analyzed in their study. Some sub-fields in IaC have got more research focus and more research done on these than others. The fields that are lacking research the most and are of key concern in IaC are defect analysis and security within IaC scripts. Rahman et al (2019a, p. 74) encourages more research to be done in these areas because of the current knowledge gap in these areas.

When things go wrong when using IaC scripts like Fryman (2014) writes about in postmortem, things go wrong fast. Due to the scale of the production environments when configuring many servers or because of the abstraction level of the IaC scripts, errors easily occur. When engineers run and apply configurations scripts they apply instantly. More often, administrators have had simple syntax checks that are in place. But even though valid syntax is there this does not always mean the code is considered correct. There are several ways scripts can be configured badly, from bad configurations to anti-patterns in code and different security smells such as those defined and categorized in Rahman et al (2019b). Security smells will be discussed later but can be defined as "[…] recurring coding patterns that are indicative of security weakness" (Rahman et al, 2019b, p. 3). These will be discussed extensively in the literature review chapter.

The ever-evolving nature of the IT-field means that things are changing faster than the researchers can keep up. That is why there is a need to keep doing the kind of studies that Rahman et al (2019a) did to keep mapping the field for areas that are in need of research, but also to do studies like (Rahman et al, 2019b, 2020c) did to keep up to date with the changes in security weaknesses, smells and other defects in IaC.

1.2 Purpose

The purpose of this study is to identify how many security smells that are contributed to open-source IaC repositories on GitHub by independent developers. The timeframe for this is the last two years of contributions to observe if there is a trend for these security smells. To also check if collaboration via GitHub may have helped to decrease the number of security smell occurrences. So, the aim is to create knowledge to the field regarding how developers perform, best practices and to mitigate previous results. With that an insight of where the trend is heading in terms of security smells within IaC scripts.

1.3 Target group

The target group for this study is practitioners, managers and security researchers within the DevOps field and others that uses IaC tools within their corporation for production use. In the hopes of providing information and recommendation of tools to use and best practices to help mitigate risks with IaC tools and security smells. Future researchers may also benefit from the results by the analyze of newer data sources to build research on.

1.4 Research questions

When looking at the need to check these security smells within short intervals to keep up with the changes in the IT-field. Then a look at how developers perform in identifying these smells is needed. Therefore, the first research question is:

R1: How did open-source practitioners perform in identifying security smells in IaC development during the period 2019-01-01 to 2021-03-29?

The answer to R1 will be complemented by a discussion on how to mitigate security question with the question being formulated as:

How can developers better mitigate security smells?

This question is not the focus of this study and are therefore not a research question in itself, but a complement to R1 as stated.

1.5 Demarcation

The focus of this study has been to evaluate only the top two most used open-source configuration management tools, which is Ansible and Puppet according to (Datanyze Team, u.d.). Microsoft System Center Configuration Manager is most popular but are proprietary and therefore not eligible for this study. Many other articles and sources also state that Ansible and Puppet is the most used (UpGuard Team, 2020). These two tools were also among the top five in Torberntsson and Rydin (2014, p. 9) study. This demarcation was done in respect of the timeframe of the study, and also to be able to produce quality result in both implementation, analysis and conclusion.

The demarcation of IaC repositories is made to 200 repositories for each tool. A total of 400 repositories due to the time frame of the study and physical capacity of the storage on the computer is done on. The computer used in this study is my personal daily driver, a MacBook pro 2017 with 256GB of storage of which roughly 200GB already are allocated.

1.6 Ethical considerations

The tests regarding security and defect analysis are done in a complete virtual environment intended for the tests and without any information or revenue at risk. The repositories that are examined are open to the public with a permissive license that allows use, modification and redistribution. Any fault or

defect that is of security character that may be found will be reported to the owner of the repository as soon as it is discovered.

2 Literature overview

Here the author will go through main theoretical background and earlier studies about infrastructure as code, explain what it is, why it is used. The author covers what challenges and risks exists and also security smells and best practices when working with IaC as other experts have concluded and stated in previous reports.

2.1 Infrastructure as code

2.1.1 Defining infrastructure as code

Infrastructure as code is a term that is used mostly within the DevOps movement where the practice and technology find most use, it is used allover though not just within this environment. DevOps is a movement that Morris (2020, p. 25) defines as "[…] a movement to reduce barriers and friction between organizational silos—development, operations, and other stakeholders involved in planning, building, and running software. […]". The infrastructure as code term is most easily defined as "Infrastructure as code (IaC) means to manage your IT infrastructure using configuration files." as Schults (2019, Defining Infrastructure as Code) so concretely puts it, because that is what it is. Putting a description on how the state of the cloud environment to be and automate the task of configure them as such.

2.1.2 What problems does it solve?

The problem that IaC automation was designed to solve was the number of servers that would need to be provisioned and maintained in the cloud age was too large to handle. The cost to hire more personnel would not be viable. IaC lets the same amount of personnel handle many times the number of servers.

Morris (2020) established a simple, yet precise comparison that illustrates the differences between in the cloud age vs the iron age as can be seen in Table 3.

Table 3. Differences from Iron age to Cloud age. From Morris (2020, p. 28) used with permission.

Iron Age	**Cloud Age**
Physical hardware	Virtualized resources
Provisioning takes weeks	Provisioning takes minutes
Manual processes	Automated processes
Cost of change is high	Cost of change is low
Changes represent failures (changes must be “managed”, “controlled”	Changes represents learning and improvement
Reduced opportunities to fail	Maximize speed of improvement
Deliver in large batches, test at the end	Deliver small changes, test continuously
Long release cycles	Short release cycles
Monolithic architectures (fewer, larger moving parts)	Microservices architectures (more, smaller parts)
GUI-driven or physical configuration	Configuration as code

IaC is a necessity for working with servers within the cloud age. Doing so also cuts costs and lets the services and servers run in a more reliable way, which in turn have pushed the growth of cloud operations and the scale of server consolidation further.

2.1.3 Immutable vs Mutable

There is a difference in how certain companies would choose to work with IaC. The two main ways to go about maintaining the infrastructure with code is either configure an existing server to a wanted state and keep changing that existing server whenever the need arises or to destroy the server in need of change and provision a new server with new configuration. The first option is called working with a mutable environment, the environment can change after the needs. This is a fast way of adapting to change and works well for updating servers. This has the drawback of potentially causing something called configuration drift, well go through this and more in section 2.1.5.

When choosing to work with the second option, i.e., immutable environment the process looks a bit different. When a need of change or a new configuration is needed, instead of changing an existing server, a change of an image or the configuration file for the server is done. This will in turn destroy the existing one and then start up a new server with a different configuration altogether. This is possible in

the cloud age because of the fast-provisioning times, doing this without virtualization would be counterproductive. This has the benefit of not causing the same configuration drift because there is never a risk that someone might change something manually on the server, since it gets destroyed every time a new change was made (IBM Cloud Education, 2019).

2.1.4 Configuration management vs provisioning

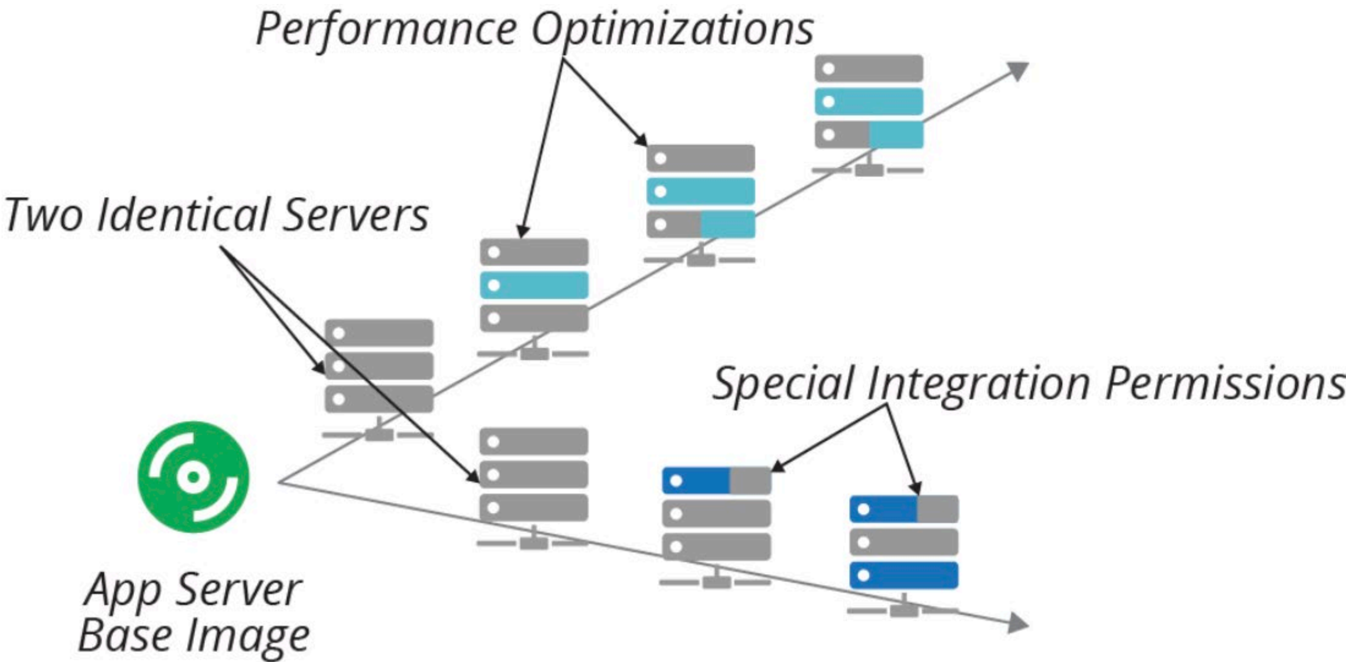
As stated before, IaC is a broad field. Many different tools exist for different purposes. As seen in the Figure 2 below, there are certain situations where one tool should be used over another. For example, terraform should preferably be the tool used create and provision servers. While ideally Ansible, Puppet or any other tool in the same bracket as Figure 2 shows should be used configure the servers. Ansible and Puppet could be used to provision infrastructure and have limited functionality to do so. However, as Nuñez (2017) writes in his article doing this can lead to unnecessary complex code because the DSL (Domain Specific Language) was not necessarily designed to work this way. This further increases the risk for errors and misconfigurations if not properly handled or divided up in different modules. This should be avoided since it is considered an anti-pattern. Anti-pattern are certain design choices that are considered bad to do.

Figure 2. Illustration of use cases for IaC tools. From Gattani (2020).

2.1.5 Challenges with infrastructure as code

The most prevalent challenge with infrastructure as code is to only make configuration changes within the code files where it should be defined. Doing configuration changes outside these files directly on the server will result in something called configuration drift. “Configuration drift is when instances of the same thing become different over time” (Morris, 2020, p. 53). This can happen when doing manual changes as said, but also when using these IaC tools to only make changes on some of the servers and not all of them. This can cause configuration differences on the servers that will cause problems later on when trying to make configuration changes on all of them. The figure below (Figure 3) illustrates configuration drift with two identical servers becoming different over time, shown by the colors.

Figure 3. Illustration of configuration drift. From Morris (2020, p. 53) used with permission.



Another challenge that Morris (2020, p. 54) mentions is what he calls “The automation fear spiral”. As illustrated in Figure 4 below:

Figure 4. Illustration of fear spiral. From Morris (2020, p. 56) used with permission.

Morris (2020) had asked the question of how many people were using some kind of IaC tool in a DevOps conference. Everybody used it at some point, but they did not use it such as the principle of IaC suggests. They used it manually and applied configuration little by little. As such the fear of breaking the system by letting it run automatically at pre-determined schedules, that is how IaC is actually supposed to be used. But the fear of letting something run automatically was the cause of configuration drift itself.

By using the preferred and proven principles and design patterns, such as using IaC only to define configuration changes within the files that run and not manually outside of this tool. A lot of problem and challenges can be mitigated. This requires new thinking when designing systems as engineer was used to do from the iron age of IT.

2.2 Earlier studies

Since the IaC field is relatively new, there is also few scientific research papers produced in this area. This is a status that Torberntsson and Rydin (2014, p. 5) also observed when doing their comparison study on configuration management tools. This study builds partly on their work by choosing Ansible and Puppet for comparison, two of the top five tools that they got as a result. Their study was more focused on comparing many configuration management tools based on many parameters that would suite a certain company when choosing tools.

This study is more focused on defect analysis and security within IaC scripts using Ansible and Puppet to find out if there are differences in those two tools within two time periods and studies. In that field there is also not much research done. Rahman et al (2019a) did a study where they mapped out how the research landscape looked like for IaC. They identified five different areas where they found a lack of research and encouraged more studies to be made. These were:

- Anti-Patterns
- Defect analysis
- Security
- Knowledge and training
- Industry best practices

2.2.1 Studies within implementing IaC and DevOps

Rahman et al (2019a) clearly highlighted gaps in research that have not been investigated since their research work got published. We can also read about previous research at the intersection of IaC implementation and DevOps environment. Looking into reports from the industry and about best practices, practical knowledge and training documents, as well as scientific articles, there are a few research papers about adopting and implementing DevOps and IaC like Guerriero et al (2019) and (Shvetcova et al, 2019). This study focuses more on the security aspect of IaC, as this came as a key concern and yet a subject that requires continuous investigation, but to have a better understanding of this specific issue, first an understanding how it is used within DevOps is needed. Only then can knowledge of how and why certain security smells occur and how practitioners best can mitigate them with better practices be gathered.

Guerriero et al (2019) write a research paper about this topic about implementing and supporting IaC within a company. They interviewed 44 developers of their opinions and experience and came to the

conclusion that the support for these tools is limited. More research is needed in this area to better help companies to adopt these DevOps and IaC tools and the way of working with this specific environment.

2.2.2 Studies pioneering in security within IaC scripts

This study focuses on the areas of defect analysis and security. One of the authors, Akond Rahman have been very active in IaC research. He carried out further research in these fields and got published in well-known journals and conferences as IEEE. Much of what the author base research on comes from articles and papers that he co-wrote with other authors in the studies Rahman et al (2019b, 2020c).

Rahman together with his colleagues in the field does a study on open-source repositories containing Puppet scripts and analyzing security smells and defects within those scripts (Rahman et al, 2019b). Furthermore, in Rahman et al (2020c) they replicate the study and apply it to Ansible and Chef scripts. They identify big problems with hard-coded secrets such as passwords. They also found bugs throughout the process and submitted as much as 1000 bug reports for some of the defects they found. To help them with this they designed and implemented something they call “Security Linter for Infrastructure as Code scripts” or SLIC. A static analysis tool that scans through code files and finds different defects and security smells.

In studies earlier such as Rahman et al (2020b) they also did this type of analyzing and qualitative work, however without the help of the SLIC software and the result are much the same. Despite there being some time in between. This might be due to the effectiveness of the SLIC tool to help find these defects automatically with more accuracy or that there are still the same number of defects being produced in IaC scripts. A research area for knowledge and training to be applied.

2.3 Security in infrastructure scripts

Security in IaC scripts is a lot like other programming languages. Since most of the tools like Puppet and Ansible use some well know established language to define the code. Puppet for instance uses Ruby, a popular general purpose programming language. Ansible uses YAML, a popular data serialization language, mostly used to define configuration files or used where data needs to be transmitted and is similar to JSON and XML.

Therefore, it is easy to see that the same mistakes and errors that can be made in the programming languages can be made here in IaC as well. Additionally, because IaC scripts is an abstract level above

normal programming languages even more errors and defects can occur. Somethings like hardcoded secrets is a universal problem and not unique to IaC, but since IaC handles critical infrastructure big problems can occur. An example of this is the GitHub outage of DNS because of a defect Puppet manifest (Fryman, 2014).

2.3.1 Design patterns

Design patterns are a term that refers to recurring design choices within software engineering. This can be best practices or anti patterns, that means patterns that are generally seen as bad to do. There also a pattern called security smells, that Rahman et al (2019b, p. 3) defines as “Security smells are recurring coding patterns that are indicative of security weakness” as mentioned before.

2.3.2 Security smells

Below is an example of bad code, a “security smell” that assign the IP address 0.0.0.0 as listen address. This is bad because 0.0.0.0 is an unrestricted address and will listen to all addresses. There is also a good example of how this should be defined.

Figure 5. Screenshot of security smell example. From Rahman (2020a).

There is a big number of different security smells that can occur and in the study Rahman et al (2019b, 2020c) they identified and categorized seven types of security smells, these are listed below and marked with the icon of each technology it appears in. ( ) for Ansible and ( ) for Puppet:

- Admin by default
- Empty password
- Hard-coded secret
- Invalid IP address binding
- Suspicious comment

- Use of http without TLS
- Use of weak cryptography algorithms
- No integrity check

The reason for why some of the security smells, in this case only two differ from each other is because of how these tools differ in their use and how their architecture was designed. For Puppet, there were no weaknesses found from CWE that regarded integrity check of downloads, and respectively no weaknesses connected to use weak cryptographic algorithms within Ansible were found. That is why their respective SLIC tool does not check for these smells in each respective tool (Rahman et al, 2019b, 2020c). Below follows a more detailed description of each category.

2.3.2.1 Admin by default

This security smell is distinctive of using the admin user by default and breaking the principle of least privilege. The principle of least privilege is the idea to only allow access to the minimum resources a person need to accomplish their task, no more, no less. This is described in a paper by the National Institute of Standards and Technology (NIST) (Ross, 2020, p. 36).

Using an admin user by default opens up for immediate security weaknesses and would let and exploiter get full control of the machine the code runs on. This issue has several posts in the Common Weakness Enumeration database (CWE) particularly relevant are CWE-798 and CWE-250 that describes the issues that can occur (Rahman et al, 2019b, p. 3), (Rahman et al 2020c, p. 10).

To describe the relevant weaknesses mentioned and to get a better idea about what it is about the author of this study quote Kingdoms on CWE-250 “The software performs an operation at a privilege level that is higher than the minimum level required, which creates new weaknesses or amplifies the consequences of other weaknesses.” (Kingdoms, 2006a). CWE-798 “The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.” (MITRE, 2010).

MITRE (2010) also suggests one of the detection methods to be automatic static analysis which will be done in this study; however, they also debate the effectiveness of this method.

2.3.2.2 Empty password

This security smell is described by Rahman et al (2019b, 2020c) as declaring a password of a zero-length string. Which leads to the ability to better guess the password, as there is nothing specified. A recommendation is made to, when possible, use the SSH-protocol as authentication with cryptographic keys.

The empty password smell is indicative of CWE-258. A simple description is used here “Using an empty string as a password is insecure.” (Kingdoms, 2006b). Kingdoms (2006b) also mentions possible consequences and mitigations. The most prominent and maybe even the only and direct consequence is to gain access and assume identity of the user connected with the empty password.

The potential mitigation that is mentioned is during the system configuration phase, a phase where IaC scripts are heavily involved. So, a mistake here when mitigation is possible will lead to a definitive weakness. The recommendation to always use at least eight characters long passwords, preferably longer.

2.3.2.3 Hard-coded secret

The hard-coded secret smell is divided into three subsets, this includes usernames, passwords and different keys, such as API-keys or SSH-keys (Rahman et al, 2019b, 2020c). Because IaC scripts are used to set up entire systems the need for some information is bound to be present and therefore the likelihood of exploitation by these weaknesses is classed as high. The best practice of this is not too hard-code them into the script that may lead to several security weaknesses such as CWE-798 (MITRE , 2010). CWE-259 is described as “The software contains a hard-coded password, which it uses for its own inbound authentication or for outbound communication to external components.” (Kingdoms, 2006c). Both of these weaknesses can be mitigated by using secured configuration files and import the values from these files instead.

The consequences of these weaknesses are access for malicious users as well as confidentiality, integrity and availability breaches. This will let malicious users to gain the ability to execute arbitrary code on target machines.

The same method for detecting these security smells is also mentioned in the CWE-posts and is the same for the rest of the security smell, it is possible to detect them with automatic static analysis but also manual review in form of code reviews.

2.3.2.4 Invalid IP address binding

This security smell is the same as the example given in the beginning of this section. It is regarding the binding of invalid IP addresses that might cause unexpected behaviors and listen to the wrong intended address. This is connected to the weakness CWE-284 described as "The software does not restrict or incorrectly restricts access to a resource from an unauthorized actor." (PLOVER, 2006a). Rahman (2020a) described this as well in the CWE-1327 post, that was a result from the studies done.

The consequences from this weakness when exploited can be connectivity problems from denial-of-service attacks and/or access breaches to servers not intended to be access from certain areas or IP addresses.

Mitigation and detection can be done via automatic static analysis tools and via reviews and are done in the system configuration phase of implementation. Again, where IaC scripts are very relevant. Assign the correct IP address other than 0.0.0.0.

2.3.2.5 Suspicious comment

Comments can be a helpful tool in development to help yourself and others with problems and remember code and what the functions are doing. But leaving too much information in comments about the problems and security weaknesses, is a weakness in itself especially with open-source software which are available to the public.

CWE-546 describes the weakness as "The code contains comments that suggest the presence of bugs, incomplete functionality, or weaknesses." (MITRE, 2006). The best option for mitigation is to remove all comments regarding security issues or bugs before deploying the application or script in question to any platform where it might be shared. The consequence of letting such comments be leads to any weaknesses in the script or application easier to find and exploit for malicious intents.

2.3.2.6 Use of HTTP without TLS

This security smell refers to the antipattern of using the HTTP protocol without a security layer. This is problematic because any authentication that may be occurring can be seen in cleartext if the contents of these packages are not properly encrypted. If these credentials are encrypted with a strong encryption

algorithm it may not pose as a security weakness, sometimes credentials are sent as cleartext between parties and then HTTPS is a must (Rahman et al, 2019b).

This is described in the CWE-319 post "The software transmits sensitive or security-critical data in cleartext in a communication channel that can be sniffed by unauthorized actors." (PLOVER, 2006b). Howard et al (2009) also brings up this problem in their book about 24 sins in computer security.

2.3.2.7 Use of weak cryptography algorithms

This security smell refers to the antipattern of using weak cryptographic algorithms, there are several old and weak algorithms that are seen as unsecure. This means that they have been solved and can be broken within seconds for the average computer. One such algorithm is MD5 that still is used a lot but not seen as secure anymore, it is mostly used as an easy obscuration tool (Rahman et al, 2020c).

CWE-326 described as "The software stores or transmits sensitive data using an encryption scheme that is theoretically sound but is not strong enough for the level of protection required." (PLOVER, 2006c). CWE-327 described as "The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the exposure of sensitive information." (CLASP, 2006a). They are both connected to using weak cryptographic algorithms.

This opens up for brute force attacks that can breach and decrypt data revealing information that might be intended to be confidential. Also violating the integrity and confidentiality of the data. Mitigating this is simple, switch out old and insecure algorithms for new and secure ones. OpenStack provides some good guidelines regarding cryptographic algorithms. MD5 and SHA1 are industry known weak algorithms and should be avoided for purposes other than checksums for file integrity (OpenStack, u.d.)

2.3.2.8 No integrity checks

This security smell is referring to downloading assets from the internet and not checking if it has been tampered with. An integrity check can be done by checking a cryptographic checksum or signature against what the provider has on their website or other site to what the result is from the downloaded file. If the two differs, then the file downloaded is not what it first seemed to be and should be discarded immediately (Rahman et al, 2020c).

CWE-353 is connected to this security smell. “The software uses a transmission protocol that does not include a mechanism for verifying the integrity of the data during transmission, such as a checksum.” (CLASP, 2006b).

The most apparent consequence of this security smell is that the integrity of the data can never guaranteed without it. Which opens the door for many more exploits if the data have been tampered with. Mitigations for this is best done to implement an integrity check before proceeding working with data received. For this Ansible have inbuilt functions to handle such checks, so there are no excuses not to use them (Ansible, 2021).

2.4 Defect analysis

Defect analysis is a broad field in itself regarding finding defects and bugs in code bases. There can be generic defects occurring such as the previously mentioned IP address binding example and context-specific defects for IaC and more precisely for Ansible and Puppet. There are many ways of mitigating risks and defects with some methods described below. For the best results a combination of the methods the author describe in this section below would give the best result of mitigating.

2.4.1 Static code analysis

Static code analysis is probably the most common method for defect analysis within software engineering. This analysis is done commonly within an IDE or when the program is compiled, for scripting language when it is interpreted with the help of something called a linter. The linter is a piece of software that checks the syntax of a programming language for errors and some bugs. In their book Secure Programming with Static Analysis, Brian Chess and Jacob West (2007) goes thoroughly through the whole process of programming securely with the help of static code analysis. What the internals of a linter is and how its lexical analysis works (Chess & West, 2007, p. 71).

2.4.2 Dynamic code analysis

Dynamic code analysis is when the defect analysis is done with the program already running. This takes more work to apply but gives a more advanced defect detection and allows findings of edge cases. Dynamic code analysis is mostly used for systems development for complex system and network applications. Right now, this method is not used in IaC because of the nature on how these configuration

files work. The author mention this because of its relation to defect analysis and maybe its future use within IaC security but right now it is not used within security in IaC (Lalithraj, 2020).

2.4.3 Code reviews

Code reviews is a common practice within IT companies who develops software. It is very effective for detecting anti-patterns and security smells on one condition. The knowledge and training need to be there, if nobody have the knowledge or experience then the code reviews would not give the same effect.

Code reviews works best when the whole team is present and engaged. Howard et al (2009) presents 24 sins in their book and for every sin they also present tips and tricks on how to find these kinds of sins as they call them during code reviews. Some of the security smells that have been described is present as sins in their book and could be eradicated with a carefully executed code review.

2.4.4 Syntax validation

Before running a program, such as an Ansible and Puppet with a code file. The Ansible and Puppet software will go through the syntax written and deem it fit or not to execute. This will eliminate some of the more obvious syntax errors, not in any case will this eliminate security smells defined by Rahman et al (2019b, 2020c). The syntax can be completely correct but may still contain security smells such as hardcoded secrets. Syntax validation is really only there to make sure the program has the correct grammar that the language in question was designed to work with. Bugs and security smells can still be entered and executed.

2.4.5 Configuration simulation

When working with IaC and configuration management that affects thousands of servers it is good to have some functionality that lets practitioners preview the changes that is going to be made. A precaution so that nothing unexpected breaks and checks so that the configuration is really what is intended to be applied. This can be done by having some testing environment that the scripts are first tested on. This however can be a slow process and more work is needed to keep a testing environment running.

The better alternative would instead use the inbuilt tools that Ansible and Puppet provide to validate and simulate changes and configurations. For Ansible this would look like this "*ansible-playbook foo.yml --check –diff* " (Ansible, 2021). The check option would run the script and simulate the changes that would be made without applying them. The diff option would show how the changes would differ and output them to the screen, such as the changes made to a configuration file.

Puppet has similar functions. For running a manifest without applying changes one can run "*puppet apply –noop foo.pp*" (Puppet, u.d.). This has similar effects as Ansibles –check option. Running a manifest and showing the differences that would have been made and get it outputted on the screen as Ansibles –diff command would become the option –test. That option will enable the show_diff flag and show us hat information in a verbose way. This can be enabled in configuration files for Puppet as well.

2.5 Analysis model

Referring to the literature and information discussed in earlier section and the fact that this is a replication study it is clear that author Akond Rahman with co-authors means a lot to this study. The foundation of the work Rahman et al (2019b, 2020c) did and their respective resources and references is where this study get a big part of my information and methodology from. The other literature is there for defining and help us get a better overview of the field in a whole. What different techniques that these tools use to be effective and how the work overall. This will give us an understanding later when the results come in from the analysis of the script and help us better decode them and understand them. The work that came from Rahman et al (2019b, 2020c) also produced the main analysis linter that will be used to go through the different manifests and playbooks.

The linter software will be used to analyze the repositories and its associated scripts to check rules specified in the method section. This will give us a result based on what security smells are present and how many of them, giving us an idea of the extent of the issue.

Using the knowledge and methodology that previous studies and articles gives, the author will then be able to answer if there is a difference between the occurrences of security smells in repositories. The author compares repositories in different time periods against the time periods from Rahman et al (2019b, 2020c) studies and check if their category number differs. More, the result will be able to show if one tool is more prone to errors by the numbers generated. How do developers perform regarding producing security smells in IaC scripts, and can they be mitigated further?

3 Method

Here the author will go through the method the author has used to collect and review the data from repositories containing Ansible and Puppet scripts using quantitative method. Later to use qualitative method to interpret the data that was collected.

3.1 Data gathering

For the data gathering the author search open-source repositories containing Ansible and Puppet scripts. This is done from individual repositories listed on GitHub.

This is done via GitHub advanced search function that lets one specify precisely what type of file and code that are interesting for this study. This would still result in a massive result back from a search term for example Ansible and Yaml. That is why a python script was made that scrapes the result section for the 200 best matches for Ansible and Puppet respectively from that search and clones the repositories, a total of 400 repositories. A search term for ansible repositories is in this case is arranged like this in the scraper script:

```
https://github.com/search?l=&p='+str(x+1)+'&q=ansible+created%3A
%3E%3D2019-01-01+extension%3A.yml&type=Repositories
```

This is formatted in such a way that the script can change pages as GitHub shows only 10 search results per page. Also, in the search string the date of creation is at the latest 2019-01-01. This is done as earlier explained to gather data newer than the study Rahman et al (2019b, 2020c) did. This way a difference over time can be seen regarding security smells in IaC scripts. See **Appendix 1** for code snippet.

The author will use different time periods of Rahman et al (2019b, 2020c) studies compared to the present study. For Puppet this period, consist of one date where the repositories used were downloaded, this was 2017-07-30 (Rahman et al 2019b, p. 3). Period 2 for Puppet will be between 2019-01-01 to 2021-03-29, that is when the repositories in this study could be created and accessed.

The same goes for Ansible. In Rahman et al (2020c) the time period for period 1 is 2014-02 to 2018-11 that is the time from where Rahman et al (2020c) access the repositories used. Period 2 for Ansible in this study is also 2019-01-01 to 2021-03-29. These periods will later be used to show differences in the analysis part.

The data gathering is seen as the qualitative part of the method. There is of course the question if these top matches that GitHub recommends is the most viable. Because of the time frame of this study, it is impossible to determine. However, random samples of repositories are manually reviewed to see if these somewhat relevant and in line with this study.

This type of data gathering would as Tracy (2019, s. 86) writes be classified as somewhere in the extreme instance of qualitative data collection. This is because of using the best matches for Ansible and Puppet and their respective file extensions. This extreme instance of collection leads to only analyze the top matches with little room for equal representation of repositories with lower search match. This data collection is fine for this study regarding scope and time frame and for the intent to review defects and best practices. Though a more inclusive analysis would be favorable for a more "real world" result.

This type of data gathering gives a lot of data that can be used, but the quality of repositories is lost when doing large scans like this one. For this study a lot of data was needed and manual review and collection of 400 repositories would not be viable. Therefore, an automatic scraper was needed and only some repositories was reviewed.

The benefit of these collaborative platform such as GitHub is that there are a lot of publicly available data that is used in production in many environments. When the community shares, code errors and problems get found out and solved faster than if it were to be proprietary, since many more sees the code and can review it transparently.

3.2 Static Code Analysis

The second part of this method is the analysis. This is done with a specially developed tool by Akond Rahman and his collogues for the studies Rahman et al (2019b, 2020c) did. This tool works as a normal compiler linter, it goes through the code and checks for suspicious syntax constructs and other signs of security smells. The authors of the study have established a set of syntax rules presented in ruleset section below.

The SLIC tool has a couple of general rules it follows to detect these security smells. These rules can be described in an abstract pseudo code. Below an illustration these rules for both Puppet and Ansible as well the string patterns for them will be shown. When these rules evaluate to true in the static code analysis part of the process, a hit on a security smell with the respective rule that was true is marked.

The analysis of these repositories will then result in a CSV file with a column for each security smell that is determined and can be analyzed. Information such as date, file path is also included. This CSV file can then be used to calculate other information such as security smell per line of code and other trends that can be useful.

Table 4 shows string patterns for ruelsets. The rules isVariable(), isAttribute(), isFunction() or isComment() uses tokens generated from the SLIC tool to determine if they are true (Rahman et al, 2019b, pp. 5-6). Table 5 shows rulesets for Puppets, and table 6 shows rulesets for Ansible.

Table 4. String patterns for SLIC tool. From Rahman et al (2020c, p. 15).

Function	String pattern
hasBugInfo()	$‘bug[\#\backslash t]*[0-9]+’, ‘show_bug\backslash.cgi?id=[0-9]+’$
hasWrongWord()	$‘bug’, ‘hack’, ‘fixme’, ‘later’, ‘later2’, ‘todo’$
isAdmin()	$‘admin’$
isDownload()	$‘http[s]?://(?:[a-zA-Z]\|[0-9]\|[\$-_@.\&+]\|[!*,]\|(?:\%[0-9a-fA-F][0-9a-fA-F]))+.[dmg\|rpm\|tar.gz\|tgz\|zip\|tar]’$
isHTTP()	$‘http:’$
isInvalidBlind()	$‘0.0.0.0’$
isIntegrityCheck()	$‘gpgcheck’, ‘check_sha’, ‘checksum’, ‘checksha’$
isPassword()	$‘pwd’, ‘pass’, ‘password’$
isPvtKey()	$‘[pvt\|priv]+*[cert\|key\|rsa\|secret\|ssl]+’$
isRole()	$‘role’$
isUser()	$‘user’$
usesWeakAlgo()	$‘md5’, ‘sha1’$

Table 5. Rulesets for SLIC Puppet tool. From Rahman et al (2019b, p. 6).

Smell name	Rule
Admin by default	$(isParameter(x)) \wedge (isAdmin(x{:}\,name) \wedge isUser(x{:}\,name))$
Empty password	$(isAttribute(x) \vee isVariable(x)) \wedge ((length(x{:}\,value) == 0 \wedge isPassword(x{:}\,name))$
Hard-coded secret	$(isAttribute(x) \vee isVariable(x)) \wedge (isUser(x{:}\,name) \vee isPassword(x{:}\,name) \vee isPvtKey(x{:}\,name)) \wedge (length(x{:}\,value) > 0)$
Invalid IP address binding	$((isV\ ariable(x) \vee isAttribute(x)) \wedge (isInvalidBind(x{:}\,value))$
Suspicious comment	$(isComment(x)) \wedge (hasWrongWord(x) \vee hasBugInfo(x))$
Use of HTTP without TLS	$(isAttribute(x) \vee isVariable(x)) \wedge (isHTTP(x{:}\,value))$
Use of weak crypto. algo.	$(isFunction(x) \wedge usesWeakAlgo(x{:}\,name))$

Table 6. Rulesets for SLIC Ansible tool. From Rahman et al (2020c, p. 14).

Smell name	Rule
Empty password	$(isKey(k) \wedge length(k.value) == 0 \wedge isPassword(k))$
Hard-coded secret	$(isKey(k) \wedge length(k.value) > 0) \wedge (isUser(k) \vee isPassword(k) \vee isPrivateKey(k))$
No integrity check	$(isKey(k) \wedge (isIntegrityCheck(x) == False \wedge isDownload(x.value)))$
Invalid IP address binding	$((isV\ ariable(x)\ _\ isAttribute(x)) \wedge (isInvalidBind(x{:}\,value))$
Suspicious comment	$(isComment(k) \wedge hasWrongWord(k) \vee hasBugInfo(k))$
Unrestricted IP address	$(isKey(k) \wedge isInvalidBind(k.value))$
Use of HTTP without SSL/TLS	$(isKey(k) \wedge isHTTP(k.value))$

3.3 Quantitative analysis

The output from the SLIC analysis explained in the previous section gives us good numbers on the amount of security smells per line of code. This can be compared to the results that Rahman et al (2019b, 2020c) got from their studies to see how these numbers have changed and get a glimpse of a trend for security smells in IaC scripts. This is done by calculating the number of security smells that get matched out of the manifests and playbooks analyzed. Then divide it by the number of lines of code divided by 1000 to get a security smell density out of that. The formula for this looks like the following:

$$Density = \frac{Security\ smells(total\ or\ specific)}{KLOC(per\ 1000\ lines\ of\ code)}$$

Using this can give a good idea of how dense the security smells is within a population of repositories.

4 Implementation and results

In this section the implementation of the method, what tools that was used and the result of these analyzes will be presented.

4.1 Environment

The environment for which the analyzes were conducted in was a docker container created from images that Akond Rahman created for his original study and replication study. The images hold the code for analyzing Puppet and Ansible respectively. The image for puppet SLIC tool can be found here Rahman (akondrahman/ruby_for_sp, u.d.) and Ansible SLIC tool can be found here (Rahman, akondrahman/slic_ansible, u.d.).

To run the docker images, first issue the docker command 'docker run -it --name puppet' to keep the container running and give it the name "puppet" to have some sort of reference to that specific container. To this command also add '-v ~/Documents/GitHub/examensarbete/puppetRepos:/var/puppetRepos' this is to mount a volume inside the container that lets us access the repositories collected earlier in the data collection step. Last, add 'akondrahman/ruby_for_sp' this specifies what image to use to start the container. The same steps are made for the Ansible container, just switch out the name tag and image name to 'akondrahman/slic_ansible'.

This way of working with container easily creates an isolated virtual environment for which in developers can create and distribute tools and use them in a disposable way. Many practitioners use this technology to create fast and easy ways to start proof of concept tools or virtual containers for separate applications.

4.2 Clarification regarding categories

Below the author present the result from the analysis in the categories that were presented earlier in section 2.3.2 security smells. The category of hard-coded secrets is a broad one and in the results this category will be split up to also show hard-coded usernames and hard-coded passwords to better illustrate what these "secrets" are. The category also contains other attributes collected from the parser. Please refer to section 3.2.1 section for the string patterns for more clarification.

That is why the categories plus hard-coded usernames and hard-coded passwords below and bear in mind that those categories does not add to the total of the hard-coded secrets.

4.3 Puppet results

The result for Puppet analyzes is based on 200 repositories based on the search query mentioned in the method section. The 200 repositories hold far more scripts of Puppet, and a bit more than 5000 scripts were analyzed.

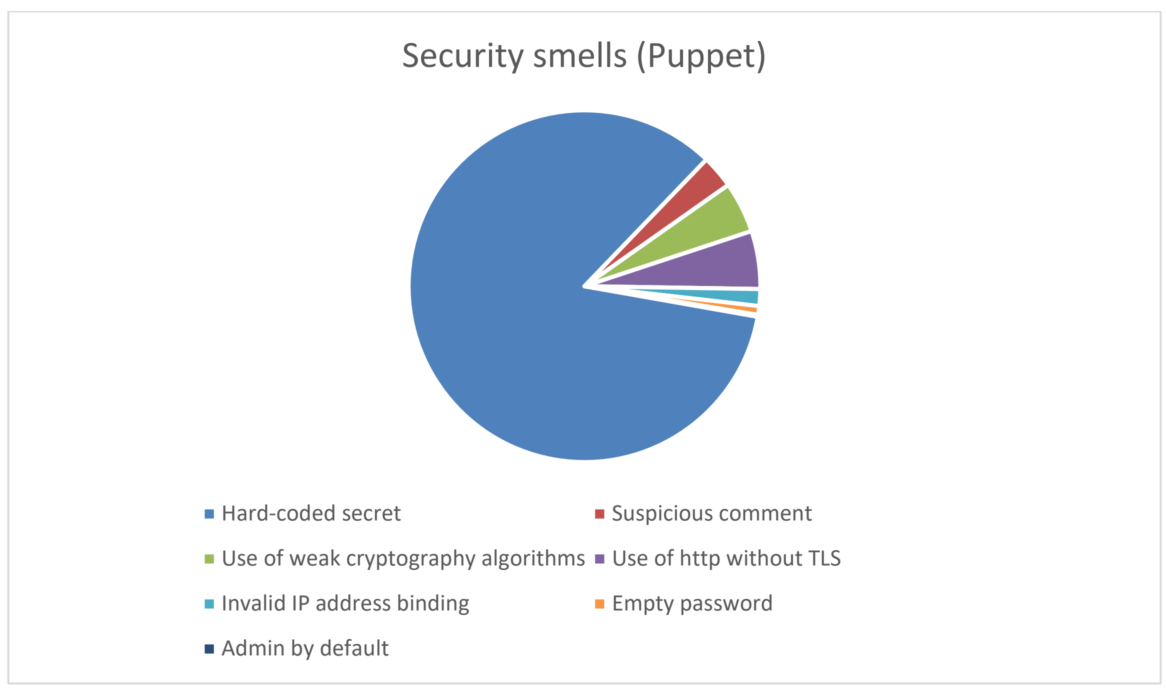
Below a table from the result spreadsheet can be seen. Only the total is shown here because of the large size of the whole document, in which there are more detail.

Table 7. Security smells for Puppet and their occurrences.

Security smell	Number of smells
Hard-coded secret	4084
Suspicious comment	147
Use of weak cryptography algorithms	227
Use of http without TLS	258
Invalid IP address binding	75
Empty password	39
Admin by default	8
Hard-coded username	983
Hard-coded password	507

As can be seen, the header row is referring to the category of security smell defined earlier and the number of occurrences. The result show that from a selection of 5215 Puppet manifests, a lot of them contains some sort of security smell. Below is a pie chart to better visualize the categories of security smells from the total occurrences.

Figure 6. Pie chart of security smells categories for Puppet.



When using the formula from section 3.3 one can calculate the KLOC value for Puppet that will be used later in the analysis section. The result for Puppet looks like following:

$$13{,}6 \approx \frac{4838}{355125/1000}$$

This is based on the total of 4838 security smells and the 355125 number of code lines that were analyzed. 355125 divided by 1000 to get the KLOC value. Puppet get an average of 13,6 security smells per 1000 lines of code compared to the value from Rahman et al (2019b, p. 9) which was 31,1.

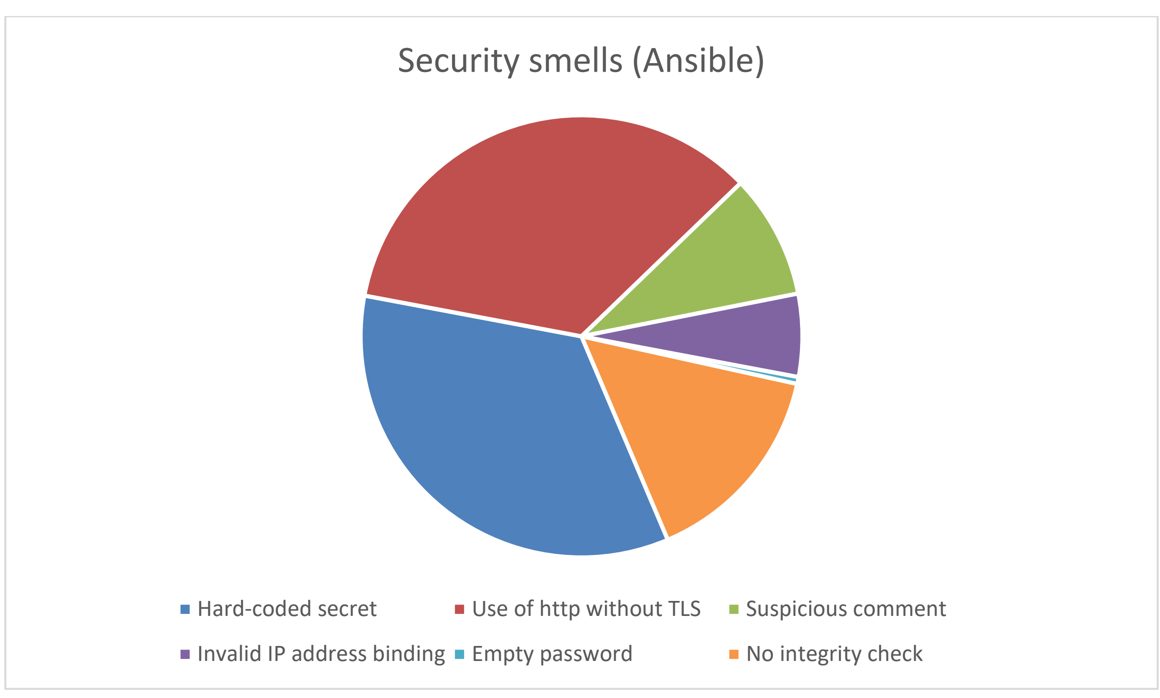
4.4 Ansible results

The result of the Ansible analyzes is also based on 200 repositories from the Ansible search query mentioned in the method section. Below is the total of security smells calculated for the Ansible scripts.

Table 8. Security smells for Ansible and their occurrences.

Security smell	Number of smells
Hard-coded secret	68
Empty password	1
Use of http without TLS	69
Invalid IP address binding	12
Suspicious comment	18
No integrity check	30
Hard-coded username	36
Hard-coded password	18

Figure 7. Pie chart of security smells categories for Ansible.



One reason of why there is a lot less security smells here is because there was less Ansible scripts per repository, a total of 334 Ansible scripts were analyzed.

Using the same equation as previous the result for Ansible will look like this:

$$9{,}2 \approx \frac{171}{18584/1000}$$

This is based on the total of 171 security smells and the 18584 number of code lines that were analyzed, but 18584 divided by 1000 to get the KLOC value. Ansible got 9,2 security smells per 1000 lines of code compared to the value from Rahman et al (2020c) which was 28,8.

5 Analysis

In this section the author will discuss and analyze the results.

5.1 Reliability

The accuracy and precision of the SLIC tool have been assessed and verified on previous studies by a large group of tests (Rahman et al, 2019b, 2020c). For the tool that analyses Puppet the accuracy of finding security smells defined by the rulesets the accuracy is 99% and for the tool that analyses Ansible the accuracy is 99.9% (Rahman et al, 2019b, 2020c).

The repositories collected represents only a small sample of the total population of repositories that contains such scripts. They were also gathered trough a search that show the best matches after a certain date. This is not by any means a certainty that these repositories are the best representatives for the larger mass and by such the result will not mirror the reality exactly. However, it is a good pointer to where it is headed, but a larger scale research and analysis would be needed to get a more exact result.

The sample size of previous study was larger and more selective on the repositories they analyzed which would have given previous study a more accurate results than the one produce in this study.

5.2 Internal validity

There are some considerations be aware of when analyzing the results. As can be seen from the results, Puppet has way more security smells than Ansible. This is because of the larger amount of Puppet manifests per repository that were analyzed. Remember, 200 of each type were gathered, but the number of scripts inside may vary. This does **NOT** necessarily mean that Puppet is more insecure, but that more manifests are used per repository to accomplish some tasks. While less Playbooks for Ansible is needed. That is a good point for further research, is it possible to accomplish the same task with less code using Ansible instead of Puppet?

Regarding the results and the numbers, Rahman et al (2019b, 2020c) used more refined and selective files for their analysis. This would yield a more precise and accurate result. Previous studies also used

other open-source platform for gathering data while this study only used GitHub. Therefore, the author only compares the results from the present study with the results Rahman et al (2019b, 2020c) got from GitHub.

5.3 External validity

In comparing the results from this study with results from Rahman et al (2019b, 2020c), the author will observe if the trend points to having a lower defects per line of code or if they in any way perform differently. When the result from this study is in, the author takes these and calculate the value from Rahman et al (2019b, 2020c) and present the differences.

Defects in code is a topic that has been debated and research a lot, it is now in recent times that this has been adopted as a research area in IaC as well. Bugs and security smells is different from one another. Code smells and security smells are not technically wrong as compared to code bugs and a program can function correctly with many security smells (Suryanarayana et al, 2014, p. 1). However, as stated before they are indicative to weaknesses that causes vulnerabilities that can later be exploited.

As McConnell (2004, p. 521) writes about industry averages and how many errors coders can expect per 1000 lines of code he concludes that an average of 1 to 25 errors per 1000 lines of code can be excepted.

5.4 Puppet period 1 vs period 2

Here the author begins with showing the different categories of security smells from the results compared to the one from (Rahman et al, 2019b). The author compares the security smells that were derived from GitHub as it is the same data source that are used. As can be seen below from the graphs, the hard-coded secrets divided up to their own graph because of the numbers of matches in that category. It is done to better see the values from the other categories in the first graph. Time period 1 for Puppet being the date of 2017-07-30 (Rahman et al, 2019b, p. 3). The time for period 2 that is this study, being 2019-01-01 to 2021-03-29 as mentioned in the method section.

It is clear when visualized that the number of security smells derived from our data gathering is noticeable less then Rahman et al (2019b) results.

Figure 8. Bar chart of security smells comparing occurrences for Puppet.

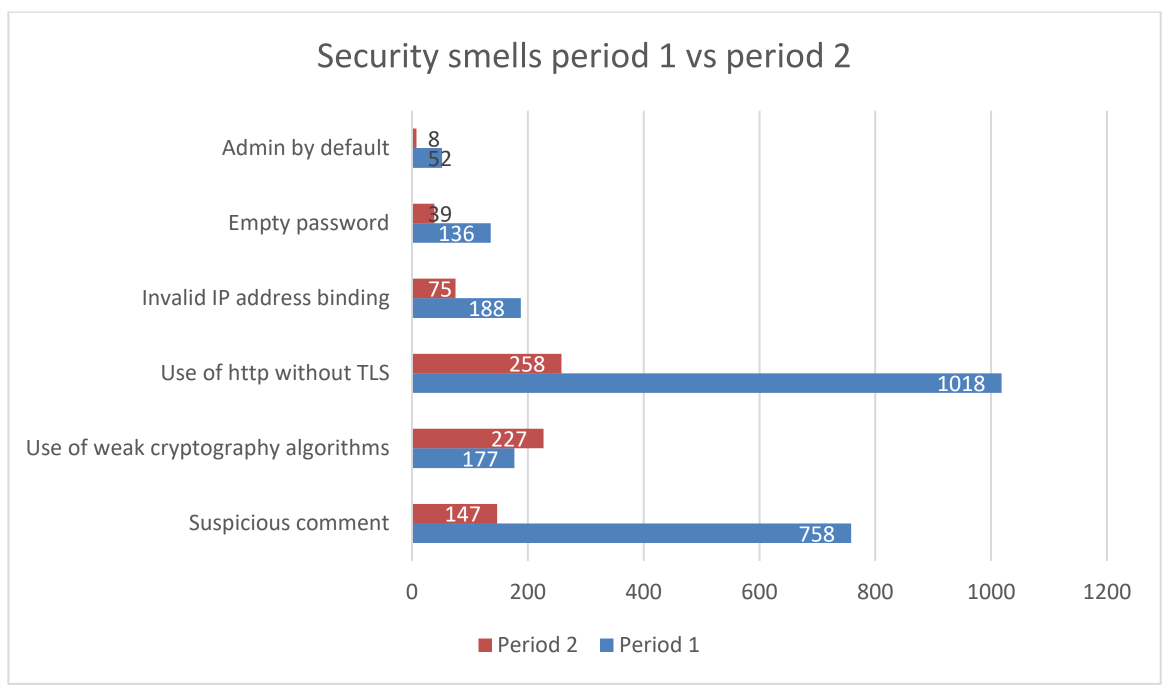


Figure 9. Bar chart of security smell category hard-coded secret comparing occurrences for Puppet.

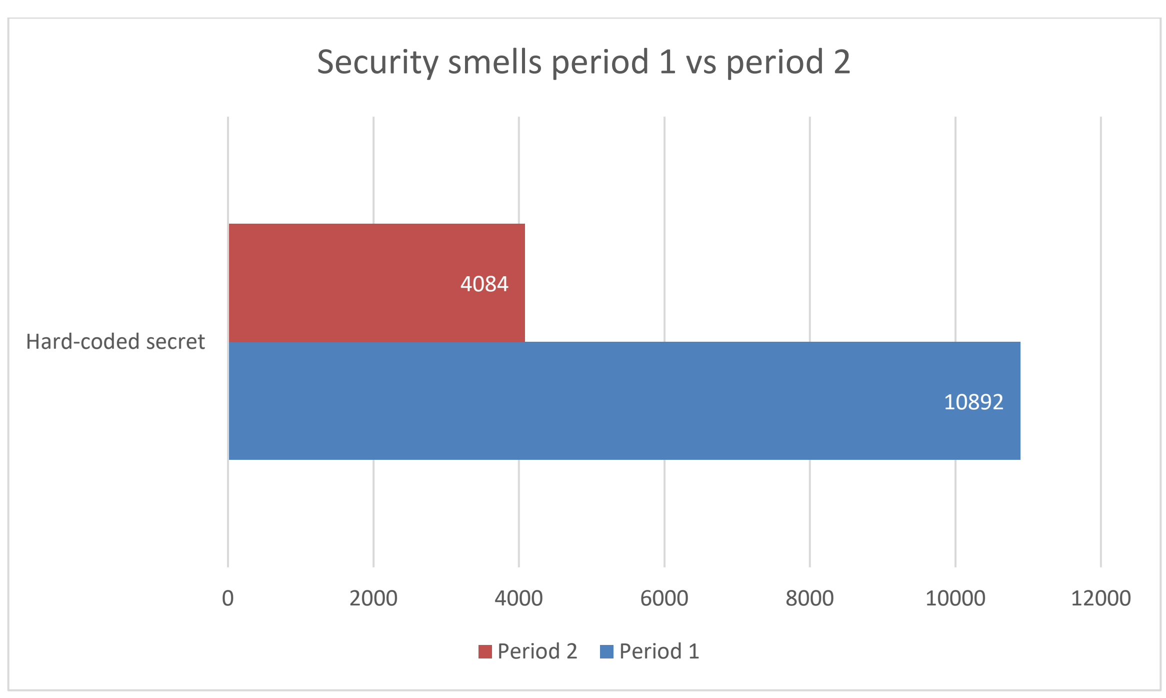
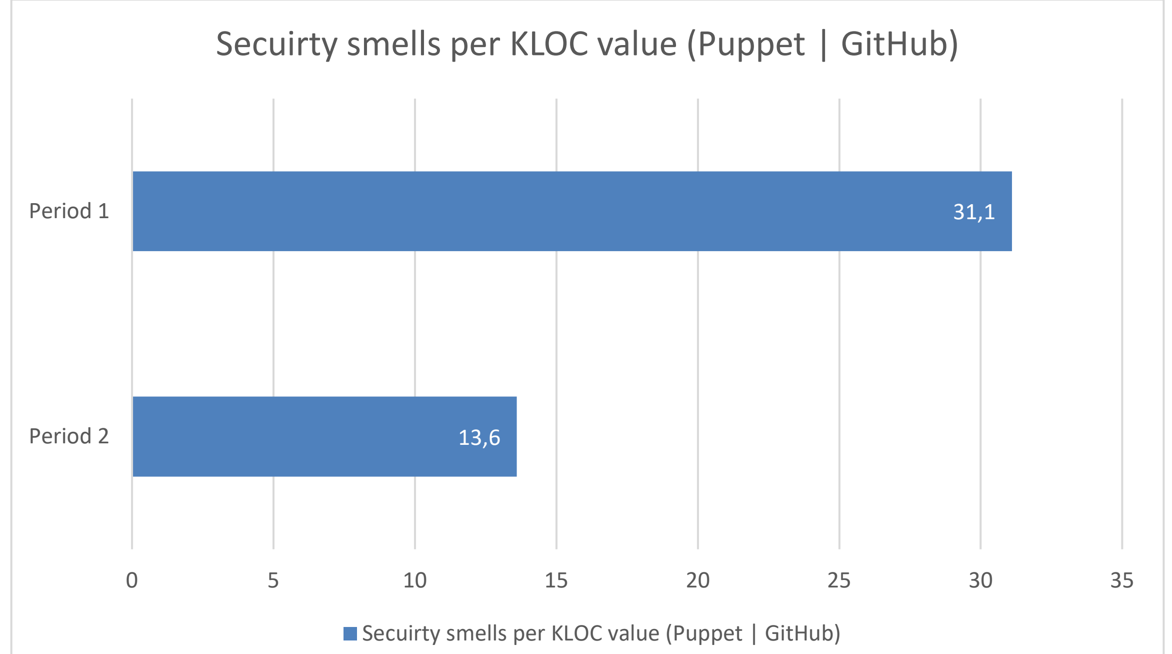


Figure 10 shows the security smells per KLOC also derived from GitHub as data source. As the figures above also show the same trend that of it being less matches for period 2.

Figure 10. Bar chart of security smells per thousand lines of code for Puppet on previous study vs this study.



5.5 Ansible period 1 vs period 2

Here the author presents the same the categories in the same way as above but for Ansible. The author split the hard-coded secrets to their own chart as before because the numbers are larger for that category than for any other category and would not visualize as well. As can be seen from the charts there that the result from the present study produced lower number of security smells than Rahman et al (2020c) study. The difference is even bigger than the difference against Puppet. This can be because of the different time period or the amount of manifest analyzed. Here the time period 1 for Ansible is the date 2014-02 to 2018-11 (Rahman et al, 2020c, p. 3). The time for period 2 that is this study, being 2019-01-01 to 2021-03-29 as mentioned in the method section.

Figure 11. Bar chart of security smells comparing occurrences for Ansible.

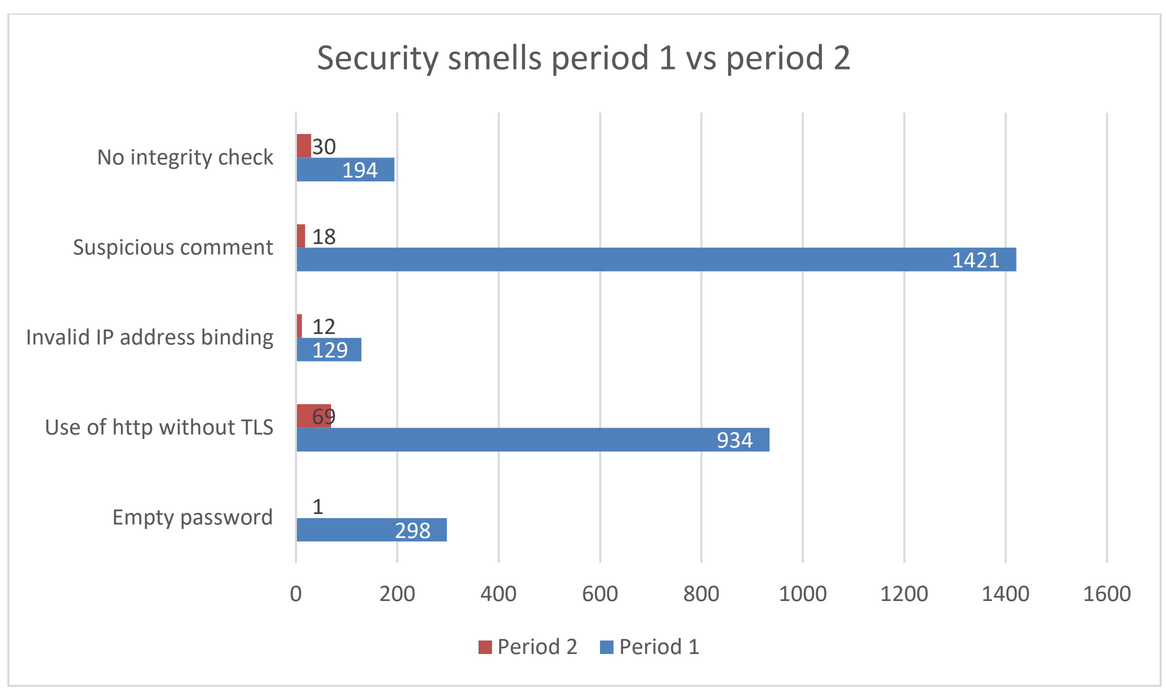


Figure 12. Bar chart of security smell category hard-coded secret comparing occurrences for Ansible.

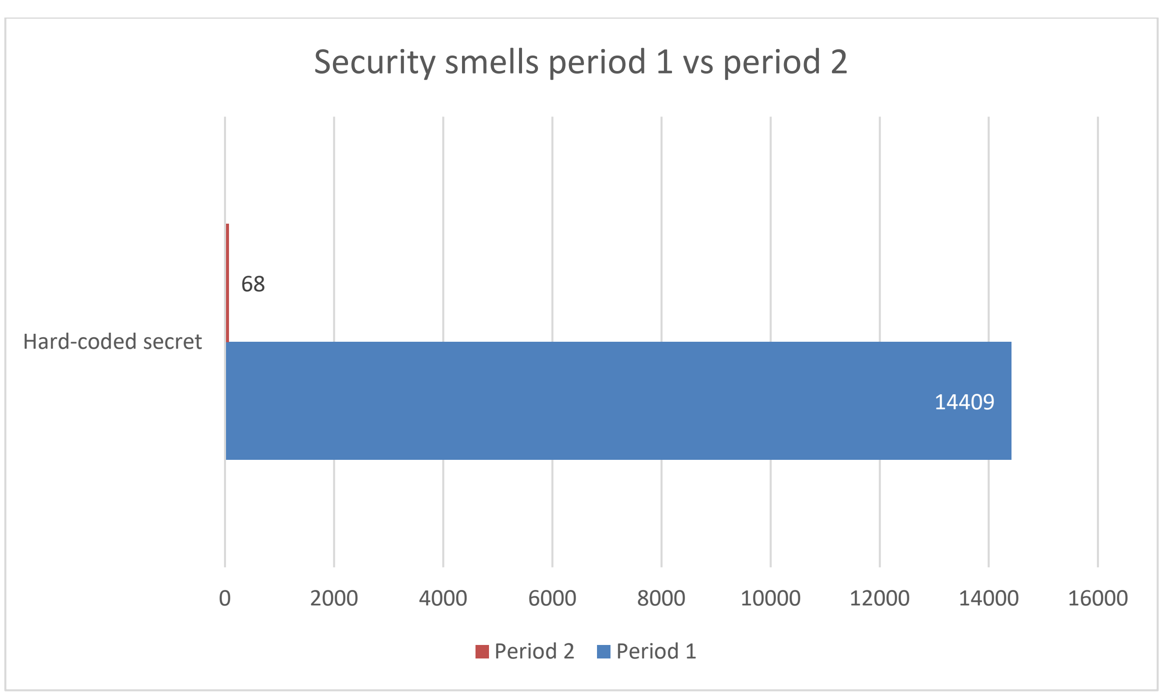
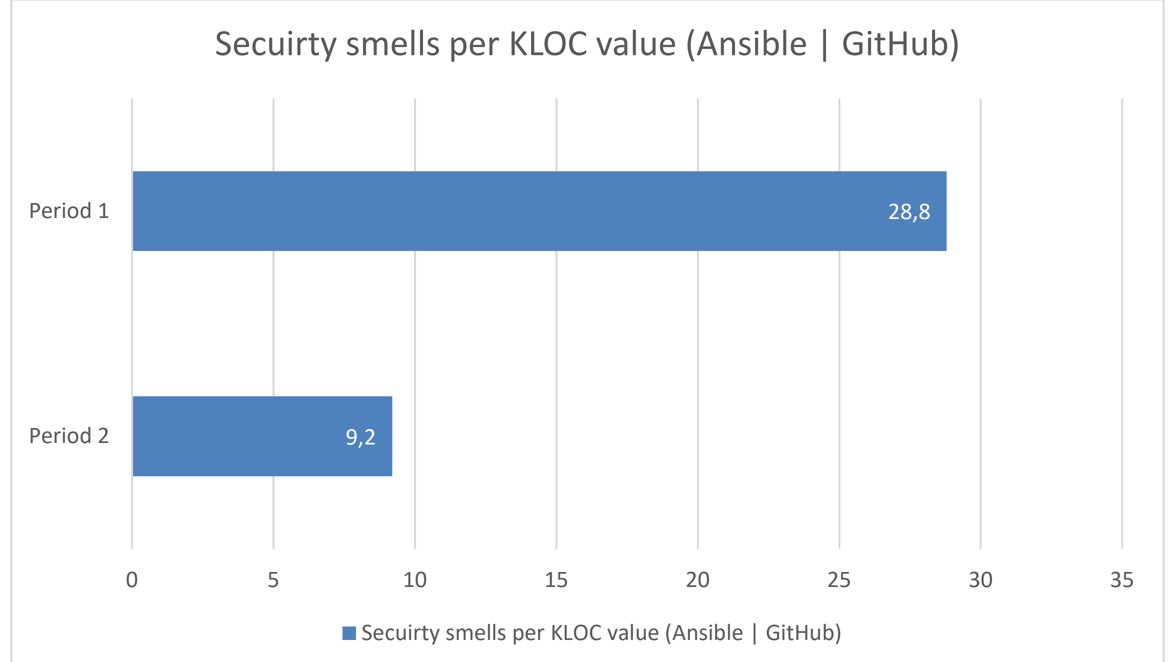


Figure 13 shows the security smells per KLOC also derived from GitHub as data source. As the graphs above it show the same trend that it being less matches. Same as with Puppet.

Figure 13. Bar chart of security smells per thousand lines of code for Ansible on previous study vs this study.



5.6 Trends

This shows that the overall security smells within Puppet and Ansible scripts are decreasing, this can have several factors behind it. This is a field with minimal research done in it according to Rahman et al (2019a) with increased knowledge about the risks of creating security smells it will also decrease the occurrences. The maturity level of the tools used will also help reduce the numbers of security smells because it will bring functionalities to these tools that help mitigate them. Also, the rise of static code analysis tools will help further in this matter. The help from big tech companies and their willingness to help open-source project also helps this trend (Tung, 2020). With GitHub also implementing security features as Cimpanu (2019) writes about it will most certainly be a lot of development in this area to help decrease security smells in the fututre.

5.7 Possible solutions

When looking for solutions to mitigate security smells early and before committing anything to public open-source repositories there are a number of static code analysis tools available to use. The language they support varies and as of writing YAML that Ansible uses was not supported. Ruby however, was and

therefore this process would help Puppet scripts from introducing security smells into their repositories. The one that support the most languages are called SonarQube and is and open-source tool. A tool for finding and reporting bugs, code smells and other anti-patterns during the development process and before committing code to repositories.

This would help tremendously with identifying security smells and is a good supplement to the mitigation techniques that Rahman et al (2019b, 2020c) discuss. Howard et al (2009) also discuss different mitigation strategies to cryptographic and authorization issues.

If for example in a Puppet script, the developers declare a user as admin and declares a password right in the script, the SLIC tool will flag this as a security smell correctly. However, if the same is done but via a safe configuration file the same rule will be flagged, without being a direct security smell. These are the false positives that can occur and would need extensive manual review to find all of them. That unfortunately is not supported by the timeframe of this study. The string pattern still matches which holds true to the accuracy of the SLIC tool, but the technique used to declare that user is considered correct. This might be a debate of what is considered correct and what is not, but this is seen as recommended by both Howard et al (2009) and Rahman et al (2019b, 2020c).

Below is one example from declaring a user the recommended way in Puppet, this would the preferable way as mentioned by earlier descriptions. This is from repo archive /var/puppetRepos/7/modules/users/manifests/hashuser.pp:

```
users::user { $name:
      ensure     => $uinfo['ensure'],
      uid        => $uinfo['uid'],
      groups     => $uinfo['groups'],
      comment    => $uinfo['realname'],
      shell      => $uinfo['shell'],
      privileges => $uinfo['privileges'],
      ssh_keys   => $key_set,
  }
```

Below is one example from declaring users considered as the right way. Using Ansible vault functionality unlockable with a password assigned when encrypting the contents. This example did not produce any security smells. This is from Ansible repo archive /var/ansibleRepos/92/playbooks/vars/40-users.yaml:

```
$ANSIBLE_VAULT;1.1;AES256
65396530316537653834373639323333333037643565353135373365343561326163303337643938
34633536663632313430303263363066663265633635663 40a3864343661396639323535363361 31
```

When checking repository 92 from our analyzed data, the one used in Ansible example above. This repository seems to be used in some kind of educational purpose and this is one of the limitations also. That not all these repositories might not reflect production use of code but can be of use to show how these examples should be done.

6 Conclusion

The present study contributed mainly to identify the new security smells in a two-year period and compare these with the results from the previous study from Rahman et al (2019b, 2020c). Observing the trend and changes will bring new knowledge to the independent developers on the current status of the security smells identification and key issues to continue to work with. Researchers can take the knowledge development to a new scale with further research. There are of course a number of limitations to the study. The use of the SLIC tool used to analyze the repositories can be listed as one of these. Another limitation being that the rulesets that are the tools checks against are static, they are hard-coded in and must be changed in the source code. These rulesets are made from categories identified from research from (Rahman et al, 2019b). These rulesets were largely agreed upon from the community. The problem is that they are check against a certain string pattern that is hard-coded and that is not always how developers use naming conventions and can differ from repositories. These string patterns are shown in table 3 and table 5. Moreover, the limitation of only looking at Ansible and Puppet does not give the full perspective on how the situation are on other tools.

The answer to research question one would be that the results show that practitioners that produced these repositories makes less security smells in their scripts than previous studies. Generally, all of the security smells have decreased in this study but are still there, there are no category that does not show up. For Ansible the empty password security smells is almost eradicated with only one match on 200 repositories. Puppet have also all the same categories and matches in all of them, none of them are completely gone. Surprisingly one category had more security smells in them than previous study period, period 1. In Puppet the security smell of using weak cryptographic algorithms were higher now in period 2. This points to a need of awareness and knowledge in cryptographic use and the dangers of using weak algorithms to protect important data.

Why many but not all security smells are decreasing can have many factors to it. As mentioned in the limitations of the study it is possible it has to do with the sample size of this study and relying on GitHub search sorting. This might not represent the production environments exactly. One of the biggest factors

however could be the big tech companies helping with open-source development. According to Tung (2020) in 2019 there was a record in disclosed vulnerabilities in open-source projects. As Tung (2020) writes the disclosure has gone up ten times since the last 10 years. This is in part thanks to the core infrastructure initiative backed by many big tech companies (Tung, 2020).

GitHub has also implemented security functionality to find errors in code (Cimpanu, 2019). This automatic check for vulnerabilities within open-source projects are a big part to why such defects are decreasing. Not necessarily because they are not implemented but because they are detected faster and before they are put to production use. Then the GitHub tool automatically analysis any new information regarding vulnerabilities and can in alert developers when new information is available. These functions do in the first-place target technically correct bugs and not code smells but can help in pointing out semantic errors in code and that is what code smells and security smells are.

Therefore, it is not possible to determine if the decreasing number of security smells is because of developer performance that have gotten better. Neither can it be determined if it is because of external help from tools that GitHub and other big tech companies have implemented and invested in. More research of the exact factors for the decreasing number of security smells in IaC scripts would be of interest as a future research topic and could study developer performance in a more focused way.

To mitigate security smells further, the help from big tech described above will further improve this work. But individual developers need to take the advantage of tools that are available. The tool used for this study and Rahman et al (2019b, 2020c) is a static code analysis tool. These sorts of tools can help developers further mitigate and discover security smells. There are some commercially available tools such as SonarQube that does just this.

Using this in combination with mitigation strategies mentioned in Rahman et al (2019b, 2020c) and relevant strategies from Howard et al (2009) with code reviews and awareness training is the best ground to mitigate security smells as much as possible. As the debated Linus law says, “Given enough eyeballs, all bugs are shallow.” from Raymond (2001, p. 30). That means that the more people that are involved with fixing a vulnerability the higher the chance are that the error will be seen and fixed. The validity to this claim is however disputed.

For future research there are several areas that are interesting to pursue. As stated in Rahman et al (2019a) These areas are still highly relevant and in need of more research:

- Anti-Patterns
- Defect analysis
- Security
- Knowledge and training
- Industry best practices

One more category that after this study the author would add to this list is the research of what factors that affects developer performance. Can knowledge and training be enough for mitigating defects in code or are external help from automatic tools needed to push this progress forward and increase code quality. What factors affects code quality the most?

The research about how the SLIC tool can be improved to allow dynamic discovery of security smells. Since it now has a static set of hard-coded rules it has limits on false positives and if hard-coded secrets actually are fetched from configuration files instead of being hard-coded. Can this be done by implementing machine learning to dynamically update these string patterns. Maybe other detections methods to help more precisely detect and alert of existing and new security smells.

Research on how open-source intelligence have affected security research in IaC and responsible disclosure of such findings. All these topics are closely related and further research in those fields will help security overall.

Bibliography

Ansible. (2021, May 19). *Ansible Documentation*. Retrieved May 19, 2021, from ansible.com: https://docs.ansible.com/

Chess, B., & West, J. (2007). *Secure Programming with Static Analysis.* Crawfordsville, Indiana, United States of America: Addison-Wesley Professional.

Cimpanu, C. (2019, November 14). *GitHub launches 'Security Lab' to help secure open source ecosystem*. Retrieved May 26, 2021, from www.zdnet.com: https://www.zdnet.com/article/github-launches-security-lab-to-help-secure-open-source-ecosystem/

CLASP. (2006a, July 19). *CWE-327: Use of a Broken or Risky Cryptographic Algorithm*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/327.html

CLASP. (2006b, July 19). *CWE-353: Missing Support for Integrity Check*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/353.html

Datanyze Team. (n.d.). *CONFIGURATION MANAGEMENT MARKET SHARE TABLE*. Retrieved Febuary 2021, from datanyze.com: https://www.datanyze.com/market-share/configuration-management--313

Fryman, J. (2014, January 18). *DNS Outage Post Mortem.* Retrieved Febuary 25, 2021, from Github.blog: https://github.blog/2014-01-18-dns-outage-post-mortem/

Gattani, S. (2020, April 13). *Infrastructure as Code — Orchestration, Provisioning & Configuration Management (Ansible & Terraform)*. Retrieved Febuary 2021, from medium.com: https://medium.com/@golibraryco/infrastructure-as-code-orchestration-provisioning-configuration-management-ansible-7cd0615e49c5

Guerriero, M., Garriga, M., Tamburri, D. A., & Palomba, F. (2019). Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 580-589). Cleveland: IEEE.

Howard, M., LeBlanc, D., & Viega, J. (2009). *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* McGraw-Hill Education.

IBM Cloud Education. (2019, December 2). *Infrastructure as Code*. Retrieved May 15, 2021, from https://www.ibm.com: https://www.ibm.com/cloud/learn/infrastructure-as-code

Kingdoms, 7. P. (2006a, July 19). *CWE-250: Execution with Unnecessary Privileges*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/250.html

Kingdoms, 7. P. (2006b, July 19). *CWE-258: Empty Password in Configuration File*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/258.html

Kingdoms, 7. P. (2006c, July 19). *CWE-259: Use of Hard-coded Password*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/259.html

Lalithraj, K. (2020, September 18). *Static vs Dynamic Code Analysis: How to Choose Between Them*. Retrieved May 26, 2021, from www.overops.com: https://www.overops.com/blog/static-vs-dynamic-code-analysis-how-to-choose-between-them/

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (Vol. 2). Redmond, Washington: Microsoft Press.

MITRE. (2006, July 19). *CWE-546: Suspicious Comment*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/546.html

MITRE. (2010, January 15). *CWE-798: Use of Hard-coded Credentials*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/798.html

Morris, K. (2020). *Infrastrucutre as Code: Dynamic Systems for the Cloud Age* (Vol. 2). Sebastopol: O'Reilly.

Nuñez, C. (2017, Febuary 7). *Why Configuration Management and Provisioning are Different*. Retrieved Febuary 25, 2021, from thoughtworks.com: https://www.thoughtworks.com/insights/blog/why-configuration-management-and-provisioning-are-different

OpenStack. (n.d.). *Security/Guidelines/crypto algorithms*. Retrieved May 15, 2021, from https://wiki.openstack.org: https://wiki.openstack.org/wiki/Security/Guidelines/crypto_algorithms

PLOVER. (2006a, July 19). *CWE-284: Improper Access Control*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/284.html

PLOVER. (2006b, July 19). *CWE-319: Cleartext Transmission of Sensitive Information*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/319.html

PLOVER. (2006c, July 19). *CWE-326: Inadequate Encryption Strength*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/326.html

Portnoy, M. (2016). *Virtualization Essentials* (Vol. 2). Indianapolis: John Wiley & Sons, Inc.

Puppet. (n.d.). *Welcome to Puppet 7.4.1*. Retrieved Febuary 20, 2021, from puppet.com: https://puppet.com/docs/puppet/7.4/puppet_index.html

Rahman, A. (2020a, September 8). *CWE-1327: Binding to an Unrestricted IP Address*. Retrieved April 7, 2021, from https://cwe.mitre.org: https://cwe.mitre.org/data/definitions/1327.html

Rahman, A. (n.d.). *akondrahman/ruby_for_sp*. Retrieved April 25, 2021, from https://hub.docker.com: https://hub.docker.com/r/akondrahman/ruby_for_sp

Rahman, A. (n.d.). *akondrahman/slic_ansible*. Retrieved April 25, 2021, from https://hub.docker.com: https://hub.docker.com/r/akondrahman/slic_ansible

Rahman, A., & Ahamed, F. B. (2020b). Characterizing co-located insecure coding patterns in infrastructure as code scripts. *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)* (p. 32). Melbourne: IEEE.

Rahman, A., Mahdavi-Hezaveh, R., & Williams, L. (2019a). A systematic mapping study of infrastructure as code research. *Information and Software Technology Journal*, 13.

Rahman, A., Parnin, C., & Williams, L. (2019b). The Seven Sins: Security Smells in Infrastructure as Code Scripts. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (p. 12). Canada: IEEE.

Rahman, A., Rahman, R., Parnin, C., & Williams, L. (2020c). Security Smells in Ansible and Chef Scripts: A Replication Study. *Transactions on Software Engineering and Methodology, 30*(1), 30.

Raymond, E. S. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* (Vol. 2). Sebastopol, California, United States of America: O'Reilly Media.

Ross, R. S. (2020, September 23). *Security and Privacy Controls for Information Systems and Organizations.* Retrieved April 7, 2021, from https://www.nist.gov: https://www.nist.gov/publications/security-and-privacy-controls-information-systems-and-organizations

Schults, C. (2019, September 5). *What Is Infrastructure as Code? How It Works, Best Practices, Tutorials*. Retrieved Febuary 20, 2021, from Stackify.com: https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/

Shvetcova, V., Borisenko, O., & Polischuk, M. (2019). Domain-Specific Language for Infrastructure as Code. *2019 Ivannikov Memorial Workshop (IVMEM)* (pp. 39-45). Velikiy Novgorod: IEEE.

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for Software Design Smells: Managing Technical Debt.* Waltham, Massachusetts, USA: Morgan Kaufmann.

Torberntsson, K., & Rydin, Y. (2014). *A Study of Configuration Management Systems: Solutions for Deployment and Configuration of Software in a Cloud Environment.* Uppsala: DiVA.

Tracy, S. J. (2019). *Qualitative Research Methods: Collecting Evidence, Crafting Analysis, Communicating Impact* (Vol. 2). Hoboken: Wiley-Blackwell.

Tung, L. (2020, March 13). *Open-source security: This is why bugs in open-source software have hit a record high*. Retrieved May 26, 2021, from www.zdnet.com: https://www.zdnet.com/article/open-source-security-this-is-why-bugs-in-open-source-software-have-hit-a-record-high/

UpGuard Team. (2020, August 5). *Ansible vs Puppet*. Retrieved Febuary 20, 2021, from upguard.com: https://www.upguard.com/blog/ansible-puppet

Appendices

Appendix 1: Scraper code snippet

```
def scrape(search, clonePath):
    print("Scraping...")
    for x in range (20):
        if(search == "ansible"):
            response = requests.get(

'https://github.com/search?l=&p='+str(x+1)+'&q=ansible+created%3
A%3E%3D2019-01-01+extension%3A.yml&type=Repositories')
        elif(search == "puppet"):
            response = requests.get(

'https://github.com/search?l=&p='+str(x+1)+'&q=puppet+created%3A
%3E%3D2019-01-
01+extension%3A.pp+language%3ARuby&type=Repositories')
        response_code = response.status_code
        if response_code != 200:
            print(response_code)
            print("Error code detected...\nExiting")
            sys.exit()
        html_content = response.content
        dom = BeautifulSoup(html_content, 'html.parser')
        reposSearch = dom.select("a.v-align-middle")
        time.sleep(20)
        end = time.time()
        print("Scraping... - Time elapsed: " + str((round(end-
start,2))) + " seconds")
        for each_repository in reposSearch:
            repositories.append(each_repository.get('href'))
    end = time.time()
    print("Scraping finished - Time elapsed: " + str((round(end-
start,2))) + " seconds")
```

```
print("Beginning cloning...")
getList()
clone(clonePath)
```

Appendix 2: List of analyzed Puppet repositories (Precede URL with github.com)

/ACDoyle/puppet02

/adobe/puppet-dispatcher

/AheadAviation/reporting_servicenow

/ajoybharath/puppet-demo

/albankerloch/puppet

/AleGuilherme/puppet

/alexoliveira88/puppet

/andeman/puppet-foreman_inventory

/andersonfernandes/puppet_pdf

/anonymus2019/puppet

/Ashish-GK/PuppetEnterprise

/avito-tech/puppet-controlrepo-template

/avito-tech/puppet-module-template

/balasharan/repo

/bamishr/Puppet-Docker-Setup

/BCarette/puppetdb2

/beare84/puppet

/bharatmicrosystems/puppet

/binford2k/puppet-function-updater

/broadinstitute/puppet-qualys_agent

/bulletsingh/puppet

/bulletsingh/puppet-ssh

/camptocamp/puppet-es_lite

/camptocamp/puppet-podman

/camptocamp/puppet-restic

/camptocamp/puppet-walg

/camptocamp/puppet-windows_securityoptions

/clover/puppet-mdb

/codemato/puppet

/dan-tabarca/puppet

/danf425/InSpec_Local

/datadiskpfv/puppet1

/dbsectrainer/Puppet

/dderdeynhalfaker/puppet3

/deric/puppet-sysctl_conf

/deshshank/puppet

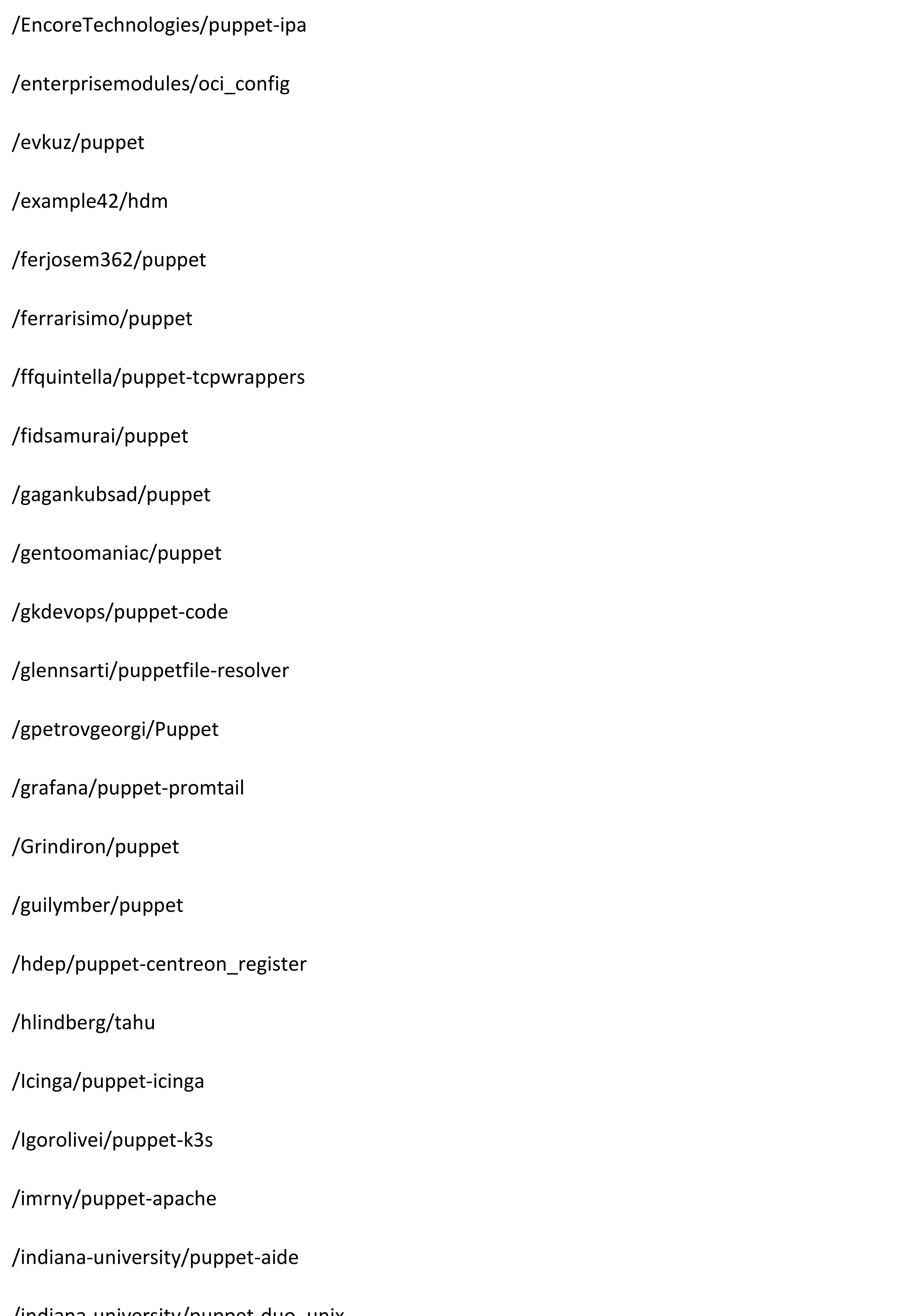
/DevOpsOneself/role

/douglasqsantos/Puppet

/dylanratcliffe/bolt_vagrant

/dylanratcliffe/puppet_ca_gen

/EncoreTechnologies/puppet-inventory_utils



/EncoreTechnologies/puppet-ipa

/enterprisemodules/oci_config

/evkuz/puppet

/example42/hdm

/ferjosem362/puppet

/ferrarisimo/puppet

/ffquintella/puppet-tcpwrappers

/fidsamurai/puppet

/gagankubsad/puppet

/gentoomaniac/puppet

/gkdevops/puppet-code

/glennsarti/puppetfile-resolver

/gpetrovgeorgi/Puppet

/grafana/puppet-promtail

/Grindiron/puppet

/guilymber/puppet

/hdep/puppet-centreon_register

/hlindberg/tahu

/Icinga/puppet-icinga

/Igorolivei/puppet-k3s

/imrny/puppet-apache

/indiana-university/puppet-aide

/indiana-university/puppet-duo_unix

/innogames/puppet-adminapi

/irasnyd/puppet-lint-action

/jaxxstorm/puppet-coredns

/jessereynolds/puppet-os_compliance

/jinostrozam/puppet

/jpogran/awesomesauce

/kcl-nmssys/puppet-fahclient

/KhanSahab/puppet

/kobybr/puppet-apmserver

/lavalamp231/Puppet

/lingeek/tomcat-instances

/lucraftm/puppet

/madprinciple/puppet

/magicmemories/puppet-graylog_api

/magnuslarsen/puppet-influxdb

/mahamot/puppet

/maheshkharwadkar/Puppet

/manthasri/puppet

/marcelocmotta/puppet

/mariusgh/puppet5

/MartyEwings/ca_expiry_check

/matthewrstone/gpno

/mattiashellstrom/contro_repo

/MaxFedotov/puppet-clickhouse

/mckafeh/puppet

/memcmahon/sock_puppet

/MetaCenterCloudPuppet/cesnet-kerberos

/mhnmoa/puppet

/microserve-io/puppet-docksal

/mihailzarankov/puppet

/MirahezeBots/puppet

/muneeb0011/puppeteer_pdf

/ncsa/puppet-profile_puppet_master

/ninan-trials/puppetTrials

/olegmakovey/LinuxA-puppet_course-ssh_module-

/OpenVPN/puppet-openvpnas

/othalla/puppet-coredns

/ovh/lxd-puppet-module

/ovh/puppet-thebastion

/PacktPublishing/Puppet-5---The-Complete-Beginner-s-Guide

/pavanpotluris/PuppetMaster

/pbihq/puppet-module-ignore_update

/pixelpark/puppet-networkmanager

/prshuu/puppet

/psanthoshkumar/PuppetController

/puppetlabs/dellemc-powerstore

/puppetlabs/iac

/puppetlabs/litmus

/puppetlabs/pdk-docker

/puppetlabs/PIE_tools

/puppetlabs/puppet_ciamohe

/puppetlabs/puppet-modulebuilder

/puppetlabs/puppet-release_manager

/puppetlabs/puppetlabs-aws_inventory

/puppetlabs/puppetlabs-cd4pe_deployments

/puppetlabs/puppetlabs-cd4pe_jobs

/puppetlabs/puppetlabs-common_events

/puppetlabs/puppetlabs-dropsonde

/puppetlabs/puppetlabs-gcloud_inventory

/puppetlabs/puppetlabs-gcloud_inventory

/puppetlabs/puppetlabs-http_request

/puppetlabs/puppetlabs-hue

/puppetlabs/puppetlabs-lidar

/puppetlabs/puppetlabs-pkcs7

/puppetlabs/puppetlabs-ruby_plugin_helper

/puppetlabs/puppetlabs-servicenow_change_requests

/puppetlabs/puppetlabs-servicenow_cmdb_integration

/puppetlabs/puppetlabs-servicenow_reporting_integration

/puppetlabs/puppetlabs-terraform

/puppetlabs/puppetlabs-vault

/puppetlabs/puppetlabs-yaml

/puppetlabs/puppetwash

/puppetlabs/RSAN

/pyama86/pero

/r0039/puppet

/RahulGandhi11/puppet-java

/RahulGandhi11/puppet-tomcat

/Rahulgithub11/puppet

/Rajitha7g/Puppet

/ratbox/puppet-puppet_conf

/reidmv/reidmv-change_risk

/reidmv/reidmv-puppet_data_service

/rengan2017/puppet

/rexroyl/puppet

/ruksat4444/Puppet

/ruksat4444/Puppet_Puppet

/rx8191/puppet

/saayeed/puppet

/sagary2j/puppet

/salluriimi/puppet

/SeanHood/puppet-honeytail

/sebastianrakel/puppet-inwx

/sensu/puppet-module-sensuclassic

/serhio83/puppet_dev

/sfarooqui97/puppetPractice

/shamushamu/puppet

/Sharpie/puppet-build-experiment

/simp/pupmod-simp-crypto_policy

/simp/pupmod-simp-ds389

/Smarteon/puppet-rclone

/SoftwareHeritage/puppet-puppet-cassandra

/SoftwareHeritage/puppet-puppet-letsencrypt

/SoftwareHeritage/puppet-puppet-rabbitmq

/SoftwareHeritage/puppet-puppet-redis

/SonicGarden/puppeteer_helper

/Sp4ceCowb0y/Puppet

/ssm/ssm-nifi

/ssm/ssm-nifi_registry

/ssm/ssm-nifi_toolkit

/SteamedFish/puppet

/tailored-automation/puppet-module-patroni

/tamerlanchik/puppet2

/Telefonica/puppet-github-actions-runner

/theforeman/foreman_puppet

/theforeman/puppet-motd

/theforeman/puppet-pulpcore

/theforeman/puppet-puppetserver_foreman

/tlcowling/puppet-wireguard

/trevor-vaughan/simp-webrick

/treydock/puppet-module-oxidized

/tubluu/Puppet

/voxpupuli/puppet_metadata

/voxpupuli/puppet-dbbackup

/voxpupuli/puppet-earlyoom

/voxpupuli/puppet-erlang

/voxpupuli/puppet-hashi_stack

/voxpupuli/puppet-ipset

/voxpupuli/puppet-mlocate

/voxpupuli/puppet-nftables

/voxpupuli/puppet-xmlfile

/voxpupuli/vox-pupuli-tasks

/voxpupuli/voxpupuli-acceptance

/vrtdev/puppet-aptly_profile

/vrtdev/puppet-keypair

/YusukeIwaki/puppeteer-ruby

/YusukeIwaki/puppeteer-ruby-example

Appendix 3: List of analyzed Ansible repositories (Precede URL with github.com)

/aapit/ansible-k3s-rpi

/abdennour/ansible-course

/aleti-pavan/packer-ansible-terraform-demo

/alivx/CIS-Ubuntu-20.04-Ansible

/AndresBott/ansible-autodoc

/ansible-collections/amazon.aws

/ansible-collections/ansible.netcommon

/ansible-collections/ansible.posix

/ansible-collections/ansible.windows

/ansible-collections/arista.eos

/ansible-collections/cisco.asa

/ansible-collections/cisco.ios

/ansible-collections/cisco.iosxr

/ansible-collections/cisco.nxos

/ansible-collections/community.aws

/ansible-collections/community.crypto

/ansible-collections/community.digitalocean

/ansible-collections/community.general

/ansible-collections/community.grafana

/ansible-collections/community.kubernetes

/ansible-collections/community.mongodb

/ansible-collections/community.mysql

/ansible-collections/community.network

/ansible-collections/community.okd

/ansible-collections/community.vmware

/ansible-collections/community.windows

/ansible-collections/community.zabbix

/ansible-collections/google.cloud

/ansible-collections/ibm_zos_core

/ansible-collections/ibm.qradar

/ansible-collections/junipernetworks.junos

/ansible-collections/vyos.vyos

/ansible-community/ansible-build-data

/ansible-community/toolset

/ansible-lockdown/RHEL8-CIS

/ansible-network/network-runner

/ansible/ansible-builder

/ansible/ansible-hub-ui

/ansible/ansible-lint-action

/ansible/ansible-navigator

/ansible/ansible-zuul-jobs

/ansible/awx-operator

/ansible/content_collector

/ansible/galaxy-dev

/ansible/network

/ansible/project-config

/apenella/go-ansible

/arillso/ansible.restic

/aristanetworks/ansible-avd

/aristanetworks/ansible-cvp

/aruba/aruba-ansible-modules

/asiandevs/OracleDBAwithAnsible

/Azure/ansible-aks

/badtuxx/descomplicando-ansible-2020

/badtuxx/descomplicando-ansible-final

/batfish/ansible

/Bes0n/EX407-Ansible-Automation

/bevhost/ansible-module-pfsense

/bluebanquise/bluebanquise

/bradwilson/ansible-dev-pc

/bregman-arie/devops-exercises

/bregman-arie/devops-resources

/byt3bl33d3r/AnsiblePlaybooks

/caddy-ansible/caddy-ansible

/cisagov/ansible-role-cobalt-strike

/CiscoDevNet/ansible-aci

/CiscoDevNet/ansible-meraki

/CiscoDevNet/ansible-mso

/CiscoDevNet/ansible-pyats

/CiscoDevNet/ansible-viptela

/CivClassic/AnsibleSetup

/containers/ansible-podman

/containers/ansible-podman-collections

/crivetimihai/ansible_virtualization

/csmart/ansible-role-virt-infra

/csmart/virt-infra-ansible

/ctienshi/kubernetes-ansible

/cyberark/ansible-security-automation-collection

/cytopia/docker-ansible

/DadosAbertosDeFeira/iac

/dawidd6/action-ansible-playbook

/DeekshithSN/Ansible

/dell/ansible-powermax

/densuke/ansible3

/densuke/ansible4

/devopstrainingbanglore/ansibleRoles

/dinofizz/picluster-ansible

/do-community/ansible-laravel-demo

/do-community/ansible-playbooks

/dokku/ansible-dokku

/elastic/ansible-elastic-cloud-enterprise

/EmailThis/ansible-rails

/ernesen/infra-ansible

/ernesen/Terraform-Ansible

/fortinet-ansible-dev/ansible-galaxy-fortios-collection

/g0t4/course-ansible-getting-started

/galaxyproject/ansible-slurm

/geerlingguy/ansible-role-docker_arm

/geerlingguy/ansible-role-fluentd

/geerlingguy/docker-centos8-ansible

/geerlingguy/docker-debian10-ansible

/geerlingguy/docker-ubuntu2004-ansible

/gkdevops/ansible-code

/gordonmurray/packer_ansible_inspec_terraform_aws

/guidepointsecurity/RedCommander

/iaasweek/ansible

/iancleary/ansible-desktop

/iarsov/ansible-orapatch

/IBM-Blockchain/ansible-collection

/IBM-Cloud/ansible-collection-ibm

/IBM/ansible-for-i

/IBM/ansible-power-aix

/IBM/z_ansible_collections_samples

/ikke-t/awx_pod

/in28minutes/devops-master-class

/integr8ly/ansible-tower-configuration

/internshipcloud2020/Ansible-Scripts

/IronicBadger/ansible-role-proxmox-nag-removal

/itwars/k3s-ansible

/itwonderlab/ansible-vbox-vagrant-kubernetes

/javahometech/ansible-2019-April-weekend

/javahometech/myapp-ansible

/jmutai/tomcat-ansible

/k3s-io/k3s-ansible

/kriansa/ansible-bundler

/learnitguide/kubernetes-and-ansible

/LearnLinuxTV/personal_ansible_desktop_configs

/leops-china/ansible-wiki

/lerndevops/ansible

/linuxacademy/content-deploying-to-aws-ansible-terraform

/liyongjian5179/k8s-ansible

/lotus-dgas/AnsibleUI

/lvthillo/vagrant-ansible-kubernetes

/M507/Kali-TX

/mailcow/mailcow-ansiblerole

/manasapala/Ansible

/mavrick202/ansibletemplatetesting

/medcl/ansible

/MeilleursAgents/terraform-provider-ansiblevault

/mgbarbero/ansible

/microsoft/AnsibleLabs

/MiteshSharma/PrometheusWithGrafana

/MnrGreg/ansible-kubernetes-kubeadm-ha

/MonolithProjects/ansible-github_actions_runner

/mtlynch/ansible-role-tinypilot

/mysidlabs/ansible-network-labs

/n0emis/ansible-role-bigbluebutton

/ndom91/AnsibleFest2020-Slides

/NetApp/ansible

/netbox-community/ansible_modules

/networknuts/Ansible

/nginxinc/ansible-role-nginx-config

/nicholasamorim/ansible-role-harbor

/nkinder/ansible-keycloak

/NVIDIA/ansible-role-nvidia-driver

/oracle/oci-ansible-collection

/oVirt/ovirt-ansible-collection

/PacktPublishing/Mastering-Ansible-Third-Edition

/PaloAltoNetworks/pan-os-ansible

/particledecay/ansible-jsonpatch

/phips/ansible-hw-bootstrap

/plb-formation/ansible-examples

/PyratLabs/ansible-role-k3s

/r3ap3rpy/ansibler

/raspbernetes/k8s-cluster-installation

/rdbreak/rhcsa8env

/ReconInfoSec/ansible-graylog-modules

/redhat-cop/ansible-middleware-playbooks

/redhat-cop/tower_configuration

/redhat-gpte-devopsautomation/ansible_advanced_osp

/redhat-gpte-devopsautomation/nextgen_ansible_advanced_homework

/redhat-operator-ecosystem/operator-test-playbooks

/RedHatSatellite/satellite-ansible-compilation

/rupanisantosh/ansible-code

/sandervanvugt/ansible-advanced

/sandervanvugt/ansiblefundamentals

/sensu/sensu-go-ansible

/ServiceNowITOM/servicenow-ansible

/shfshanyue/op-note

/softwarefactory-project/dhall-ansible

/splunk/ansible-role-for-splunk

/spurin/diveintoansible

/spurin/diveintoansible-lab

/stadtulm/a13-ansible

/sushilsuresh/ocp4-ansible-roles

/systemli/ansible-role-jitsi-meet

/T-Systems-MMS/ansible-collection-icinga-director

/tdevopsschool/ansible-course

/techno-tim/ansible-homelab

/thomvaill/tads-boilerplate

/Tobewont/ansible-playbook

/tonykay/ansible_flask_app_loader_all_in_one

/tribe29/ansible-checkmk

/twin-bridges/ansible_course

/UNoorul/ansible

/ValaxyTech/ansible

/vitabaks/postgresql_cluster

/xuxiaodong/v2ray-for-ansible

/yankils/ansible

/yankils/ansible_for_beginners