BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SYSTEM FOR AUTOMATIC FILTERING OF TESTS
SYSTÉM PRO AUTOMATICKÉ FILTROVÁNÍ TESTŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. MILAN LYSONĚK

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. VIKTOR MALÍK

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

23098

Student:	**Lysoněk Milan, Bc.**
Programme:	Information Technology Field of study: Computer Networks and Communication
Title:	**System for Automatic Filtering of Tests**
Category:	Software analysis and testing

Assignment:

1. Get acquainted with the Git versioning system and with the way it represents differences between versions of files.
2. Learn about the ComplianceAsCode project. Explore the structure of source code of different types of files present in the project and the dependencies among source files. Concentrate on the types of files that change the most often in the project's history.
3. Design a method that will be able to filter tests of a software project that need to be run if a change in the project code occurs. The filtering should be done based on analysis of the source code of changed files and on computing the set of project files and tests that depend on the changes.
4. Implement the proposed solution in a tool, which can filter the set of tests of the ComplianceAsCode project, whose result might have changed after some change was done. Support analysis of at least 4 different types of source code files.
5. Test the created tool on the history of the ComplianceAsCode project and discuss results of these experiments.
6. Write the final text of the Master's thesis in English.

Recommended literature:

- Waltermire, David & Quinn, Stephen & Booth, Harold & Scarfone, Karen. (2018). NIST SP 800-126 Revision 3, The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.3. 10.6028/NIST.SP.800-126r3.
- ComplianceAsCode Developer Guide, https://github.com/ComplianceAsCode/content/blob/master/docs/manual/developer_guide.adoc

Requirements for the semestral defence:

- First 2 items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malík Viktor, Ing.**
Consultant:	Týč Matej, Ing., Ph.D., RedHatCZ
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	May 20, 2020
Approval date:	October 31, 2019

Abstract

The goal of this thesis is to create a system that automatically determines a set of tests that must be run when a change is done in the ComplianceAsCode project. The proposed method selects a set of tests based on static analysis of the changed sources, taking into account the internal structure of ComplianceAsCode. The created system is divided into four parts — obtaining changes from the versioning system, static analysis of different types of files, computing the set of files affected by the change, and computing the set of tests that must be run to test the given change. We implemented analysis of several types of files and our system is designed to be easily extended by other analyses for other file types. The created implementation is deployed on the server where it automatically analyzes new contributions to the ComplianceAsCode project. The automatic running informs contributors and developers about changes that it found and recommends which tests should be run for the change. This saves the time spent on verifying the correctness of contributions as well as the time spent on running tests.

Abstrakt

Cílem této práce je vytvořit systém, který je schopný automaticky určit množinu testů, které mají být spuštěny, když dojde v ComplianceAsCode projektu ke změně. Navržená metoda vybírá množinu testů na základě statické analýzy změněných zdrojových souborů, přičemž bere v úvahu vnitřní strukturu ComplianceAsCode. Vytvořený systém je rozdělen do čtyř částí — získání změn s využitím verzovacího systému, statická analýza různých typů souborů, zjištění souborů, které jsou ovlivněny těmi změnami, a výpočet množiny testů, které musí být spuštěny pro danou změnu. Naimplementovali jsme analýzu několika různých typů souborů a náš systém je navržen tak, aby byl jednoduše rozšiřitelný o analýzy dalších typů souborů. Vytvořená implementace je nasazena na serveru, kde automaticky analyzuje nové příspěvky do ComplianceAsCode projektu. Automatické spouštění informuje přispěvatelé a vývojáře o nalezených změnách a doporučuje, které testy by pro danou změnu měly být spuštěny. Tím je ušetřen čas strávený při kontrole správnosti příspěvků a čas strávený spouštěním testů.

Keywords

Filtering of tests, ComplianceAsCode, SCAP, OpenSCAP, Git, Github, changes in files, file analysis, YAML, Ansible, Bash, Python, OVAL, XML, Jinja, testing, Jenkins

Klíčová slova

Filtrování testů, ComplianceAsCode, SCAP, OpenSCAP, Git, Github, změny v souborech, analýza souborů, YAML, Ansible, Bash, Python, OVAL, XML, Jinja, testování, Jenkins

Reference

LYSONĚK, Milan. *System for Automatic Filtering of Tests.* Brno, 2020. Master’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

Vývoj softwaru je obor, který se rychle vyvíjí. Každý den se se v tomto oboru objeví něco nového — nová technologie, nový programovací jazyk, nová knihovna a podobně. Zákazníci chtějí tyto novinky v produktech, které používají. Chtějí nové funkce, aby byl software přenositelný na jiné zařízení, aby fungoval na zastaralém hardwaru a taky od softwaru očekávají bezchybnost. Vývojáři proto musí trávit hodně času učením se nových věcí, ale zároveň nesmí zapomínat na kvalitu. Kvalitu softwaru si vývojáři často můžou ověřit s pomocí automatických testů, avšak tyto testy si musí napsat ručně. Proto musí vývojáři trávit čas hledáním slabých míst v jejich softwaru, musí vědět jak se vypořádat s chybnými vstupy, které neobvyklé případy mohou nastat apod. Strávit čas psaním testů se však vyplatí, protože to šetří čas jak zákazníkům, tak paradoxně i vývojářům samotným. Díky testům jsou problémy zachyceny a opraveny mnohem dříve než se dostanou k zákazníkovi. Opravit chybu, která se dostala do výsledného produktu až k zákazníkovi, je dražší než její oprava ve fázi vývoje. U testování softwaru ale nesmíme zapomínat na jednu podstatnou část, která může také trvat dlouhou dobu — spouštění testů. Zatímco u malých aplikací průběh testů trvá zanedbatelnou dobu v řádu několika sekund nebo minut, tak u velkých a komplikovaných systémů může testování trvat hodiny, klidně i několik dnů.

ComplianceAsCode projekt patří do kategorie komplikovaného softwaru. Pokud bychom pro něj chtěli spustit všechny testy, tak to zabere několik hodin, nehledě na to, že se mohlo jednat o změnu jednoho řádku ve zdrojových souborech. Tím je vývoj značně zpomalen, protože vývoj ComplianceAsCode je založen na postupu, kdy přispěvatelé posílají do projektu změny a tyto změny musí být schváleny vývojáři, kteří mají oprávnění pro přidávání změn do daného projektu. U ComplianceAsCode projektu je však problém testování jednotlivých změn, protože automatické spouštění všech testů by trvalo dlouhou dobu a vývoj by tím byl značné zpomalen. Proto musí vývojáři před schválením příspěvků ručně otestovat jejich správnost a bezchybnost. Automaticky je spuštěn pouze test, který otestuje základní funkcionalitu, protože další testy trvají několik hodin. Tento postup není praktický, protože vyžaduje od vývojáře, který příspěvek schvaluje, aby změny analyzoval a určil, které testy musí být spuštěny. Ruční testování zabírá čas a navíc způsobuje, že vývojář může něco přehlédnout a část zapomene otestovat. To pak vede k chybám, jejichž oprava v pozdějších fázích trvá mnohonásobně delší dobu.

Pro vyřešení tohoto problému, tato práce cílí na automatické filtrování testů. Testy jsou filtrovány na základě změn v ComplianceAsCode projektu. Avšak tento úkol není jednoduchý, protože ComplianceAsCode je komplikovaný projekt. V ComplianceAsCode je několik různých typů zdrojových souborů a mnohdy jsou tyto soubory ovlivněny i jinými zdrojovými soubory. To způsobuje komplexní provázání závislostí, díky kterému změna jednoho řádku, který na první pohled nevypadá nijak spojen s jinými soubory, může způsobit, že jiné části projektu přestanou správně fungovat.

Hlavním cílem této práce je vytvořit systém pro automatické filtrování testů, který je schopný se vypořádat s projektem, který má komplexní vnitřní závislosti a obecně nepředvídatelné spojení mezi testy a změnami v souborech. Navržený přístup je založen na jednoduché statické analýze změněných souborů v kombinaci se znalostmi vnitřní struktury ComplianceAsCode. Cílem naší analýzy je vysoká efektivita, byť za cenu nižší přesnosti — navržená analýza může vybrat test, který není nutné spouštět pro danou změnu. Avšak spuštění pár testů navíc, které spuštěny být nemusí, je zanedbatelné v porovnání s velikostí testovací sady projektu. Vytvořený systém se zaměřuje spíše na automatickou analýzu různých typů souborů než na podrobnou analýzu specifického typu souborů.

System for Automatic Filtering of Tests

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of Ing. Viktor Malík. The supplementary information was provided by Ing. Matěj Týč, Ph.D and the Security Compliance team from Red Hat. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Milan Lysoněk
June 2, 2020

Acknowledgements

I would like to thank Viktor Malík and Matěj Týč for their support, guidance, feedback, patience, and advice required for the creation of the thesis.

Contents

Chapter 1

Introduction

The software industry moves fast forward. Something new appears every day in the industry — a new technology, a new programming language, a new library, etc. Customers want these new things in the software that they use. They want new features together with portability to other devices, optimized for outdated hardware, and flawlessness. Developers spend a lot of their time learning new things, but they also must pay attention to quality. Quality is often ensured by automated testing, however, tests are usually created manually. Developers need to spend time thinking which parts are weak, how wrong inputs should be handled, which corner cases can appear, and so on. Time spent on writing good tests is worthy, but there is another time consuming part — test runs. While developing a small software, a test run is an insignificant part of development because it takes few seconds or minutes to finish. On the contrary, big and complicated software is different — a test run may take several hours, even days.

The ComplianceAsCode project, developed primarily at Red Hat, belongs to the inherently complicated software category. Running all test scenarios takes hours, even when one line of the source code is changed. This slows down the development of the project since ComplianceAsCode uses a development workflow based on Pull requests. Every time a Pull request to the upstream repository (located on GitHub) is created or changed, the reviewer must run tests manually based on the knowledge which parts are affected by the change. The only test that is run automatically is a so-called the functional test because running all tests takes several hours, which would slow down the development. This is not a very practical approach since the reviewer needs to manually analyze the change and determine the set of tests to run. This process takes time and, in addition, it may be imperfect since the reviewer may overlook something and not run all necessary tests.

In order to resolve this problem, this work aims at the automatic filtering of the test suite based on the implemented changes in the ComplianceAsCode project. This is quite a complicated task, due to the complexity of the project. The source code contains a large number of files of various types that are often connected to others. The sources form a complex chain of dependencies that can cause a single line change to break parts that at first sight do not seem to be connected to the changed line at all.

The main goal of this thesis is to create a test filtering system with the capability to deal with the heterogeneous project that has complex inner dependencies, and generally unpredictable connection between tests and file changes. The approach that we propose is based on a simple static analysis of changed files, taking into account complicated dependencies in the ComplianceAsCode project. The main target of our analysis is high efficiency, possibly even for the sake of its precision — the proposed analyses may conservatively include a test that does not really need to be run. However, running a single or few extra tests that do not need to be run is negligible from the point of view of the test suite size. The system focuses on automatic analysis of a wider range of different file types rather than a deep analysis of a specific file type.

The ComplianceAsCode project is described in Chapter 2. We focus on its structure, testing, and dependencies among source files. Also, Chapter 2 introduces the Security Content Automation Protocol standard and the OpenSCAP tools that work with the standard and contain, among others, the ComplianceAsCode project. As we already outlined, the ComplianceAsCode project uses Git as a versioning system that we use to acquire basic information about changes. Moreover, the development workflow, into which our system is incorporated, is based on Pull requests, which are connected to Git, too. To this end, we describe important concepts of Git in Chapter 3.

The system for automatic filtering of tests is split into four parts — getting changes, analysis of source codes, computing affected parts, and computing the set of tests. The design of the system and of all the parts is described in Chapter 4, where we also focus on supported file types, on corresponding methods to analyze them, and on methods to deal with dependencies. Chapter 5 describes the implementation of the designed system.

Chapter 6 is dedicated to testing and deployment of the implemented system. The chapter describes the usage of Bats: Bash Automated Testing System for implementation of tests for the created system, an automatic code analyzer that was integrated into the implemented system repository, and deployment of the implementation to ComplianceAsCode Jenkins where it automatically analyzes Pull requests from the ComplianceAsCode repository. The last chapter, Chapter 7, is about benchmarking the implementation using actual historical data of the ComplianceAsCode.

Chapter 2

ComplianceAsCode Project

Since computers started to be used widely, there was a need, especially within the processing of classified information, to process them securely. As a response to the need, The United States Department of Defense (DoD) published the Trusted Computer System Evaluation Criteria (TCSEC). The TCSEC is often referred to as the Orange Book. The Orange Book was the first book from the Rainbow series and it is abandoned now [5].

The current valid standard, providing a common set of requirements for the security of information, is the Common Criteria for Information Technology Security Evaluation[1] (CC). However, it does not provide a guidance on how to configure a system. It provides a guidance and a specification for evaluating information security of a product. The guidance on how to configure the system is provided in other policies that are specific for organizations and data they work with. A few examples of policies are:

- Protection Profile for General Purpose Operating Systems (OSPP) — the policy for a general purpose operating system that can work in a networked environment.
- Payment Card Industry Data Security Standard (PCI DSS) — the policy for securing a system working with payment card information.
- Security Technical Implementation Guide (STIG) — the policy used within the U.S. Department of Defense's IT infrastructure.

Policies contain many requirements that need to be met to be compliant. These requirements define system configurations such as password length, password character variability, file permissions, logging, disabled services, disk encryption, using cipher suites, disabled ssh root login, and so on. For an experienced system administrator, it may take few hours of work to set everything properly. Moreover, the fact that a system is compliant does not mean that it will be compliant the next time. The system evolves — new packages are installed, a new person starts working with the system, a new vulnerability is discovered, and other — it must be checked and properly configured periodically. Moreover, today's IT infrastructures consist of hundreds, even thousands, of computers, of virtual machines, or of containers. It is almost impossible to do that manually. As a response to it, Security Content Automation Protocol has been created [15]. Security Content Automation Protocol is described in detail in Section 2.1.

Security Content Automation Protocol (SCAP) is a standard that defines a format and a terminology for security configurations. SCAP scanners are based on the standard to

[1]https://www.commoncriteriaportal.org/

scan machines to find out if the machines are security compliant. One of these scanners is OpenSCAP, described in Section 2.2.

For SCAP scanners, the most important part is the SCAP content. The content specifies checks that can determine whether a system is configured properly. Each check, called a *rule*, has its description stating how it should be configured. The content needs to be implemented so it can be used as an input for the scanner. One of the content implementations is the ComplianceAsCode project which is the core of this thesis (Section 2.2). The description of the project's structure is in Section 2.3. The structure of the project is complex and consists of several different file types. Selection of the file types is described in Section 2.4. Description of the project's testing is in Section 2.5.

2.1 Security Content Automation Protocol

Security Content Automation Protocol (SCAP) [13] is a set of specifications. It standardizes a format and a terminology for security configuration information. The protocol supports automated configuration, vulnerability checking, patch checking, technical control compliance activities, and security measurement.

SCAP 1.0 was published by the National Institute of Standards and Technology in July 2010 and current version is 1.3 which was released in February 2018. SCAP 1.3 consists of twelve component specifications. The specifications are divided into five categories:

- **Languages** — provide standard vocabularies and conventions for expressing security policies, technical check mechanisms, and assessment results. The specifications are Extensible Configuration Checklist Description Format (XCCDF), Open Vulnerability and Assessment Language (OVAL), and Open Checklist Interactive Language (OCIL).

- **Reporting formats** — provide the necessary constructs to express collected information in standardized formats. The formats are Asset Reporting Format (ARF) and Asset Identification.

- **Identification schemes** — provide a means to identify key concepts such as software products, vulnerabilities, and configuration items using standardized identifiers formats. The schemes are Common Platform Enumeration (CPE), Software Identification Tags (SWID), Common Configuration Enumeration (CCE), and Common Vulnerabilities and Exposures (CVE).

- **Measurement and scoring systems** — refers to evaluating specification characteristics of a security weakness and generating a score that reflects their relative severity. The scoring system specifications are Common Vulnerability Scoring System (CVSS) and Common Configuration Scoring System (CCSS).

- **Integrity** — helps to preserve the integrity of SCAP content and results. The integrity specification is Trust Model for Security Automation Data (TMSAD).

For us, the important category is *Languages*. A security policy is specified in a form of an XCCDF document, which is written in the XCCDF language. An XCCDF document contains, organizes, and aggregates rules into one unit. The OVAL language is used to define rules, which check system configuration. And OCIL is used to define rules, which need people to collect information, for example, transcribe a number from a label.

XCCDF from its name — Extensible Configuration Checklist Description Format — is a checklist with additional data on the top. The most important additional data are information about profiles. A profile is a set of rules from an XCCDF document. The set of rules from a profile can overlap with a set from another profile.

OVAL defines an expected state of a system and reports its actual state. According to OVAL website, OVAL standardizes three main steps [12]:

1. Representing configuration information of systems,
2. Analyzing the system for presence of the specified machine state (vulnerability, configuration, patch state, etc.),
3. Reporting the results of this assessment.

OVAL definitions are usually in separate files, so it is easy to edit them and a user does not need to edit XCCDF documents.

An XCCDF document can be used as an input into SCAP scanners, but often multiple XCCDF documents are wrapped into one file called a *datastream*. The datastream can be also used as an input for the scanner. Using the datastream is more convenient for a publisher and as well for a consumer because it is just one file which can be used for different purposes.

SCAP scanners scan a system state according to a selected profile and present results to a user. There exist few such scanners. From the proprietary side, there are, for example, Qualys[2] and Nessus[3]. From the open-source side, there are OpenVAS[4] and OpenSCAP[5]. We will describe more in detail only the OpenSCAP scanner.

2.2 OpenSCAP Tools

OpenSCAP is a collection of free and open-source tools. The development started in November 2008 within Red Hat and the collection is open-source from the beginning [14]. Currently, it is widely used by many businesses and government organizations. The OpenSCAP collection provides a wide range of tools related to the security compliance. There are:

- OpenSCAP,
- SCAP Workbench,
- ComplianceAsCode,
- OSCAP Anaconda Add-on,
- and more.

Tools from the collection are used within other Red Hat products, for example, Red Hat Satellite[6] or Cloud Management Services for RHEL[7].

[2]https://www.qualys.com/
[3]https://www.tenable.com/products/nessus
[4]http://openvas.org/
[5]https://www.open-scap.org/
[6]https://www.redhat.com/en/technologies/management/satellite
[7]https://cloud.redhat.com/

OpenSCAP

OpenSCAP is a library and a command-line tool used to parse and evaluate components of the SCAP standard. The library implements the majority of the OpenSCAP functionality. It implements a processing of SCAP documents, a system scanning, an evaluation of the scan, and a reporting. The library provides Application Programming Interface (API) for developers to create applications working with the SCAP standard.

The command-line tool is called `oscap` and it is a wrapper using the OpenSCAP library. The usability of the command-line tool is not straightforward. Just base `oscap` offers several options regard to the SCAP standard — `ds` (datastream utilities), `oval`, `xccdf`, `cvss`, `cpe`, `cve`, `cvrf` (Common Vulnerability Reporting Framework), and `info`. The command-line tool also offers an option to fix (remediate) a non-compliant system setting. As an exemplary input and output from `oscap` which fixes selected rules from PCI DSS profile, see Listing 2.1.

```
$ oscap xccdf eval --profile pci-dss --remediate \
--rule xccdf_org.ssgproject.content_rule_rpm_verify_permissions \
ssg-rhel8-ds.xml

    Title Verify and Correct File Permissions with RPM
    Rule xccdf_org.ssgproject.content_rule_rpm_verify_permissions
    Ident CCE-80858-4
    Result fail

     --- Starting Remediation ---
    Title Verify and Correct File Permissions with RPM
    Rule xccdf_org.ssgproject.content_rule_rpm_verify_permissions
    Ident CCE-80858-4
    Result fixed
```

Listing 2.1: Output from oscap xccdf eval command (shortened)

ComplianceAsCode

The most important part of the scanning is the SCAP content. SCAP scanners interpret the SCAP content and if the content is not good enough, it may give false positives or false negative results. With the poor content, a user may have more complications than benefits. Hence, *SCAP Security Guide* was created to provide the open-source SCAP content.

SCAP Security Guide started as a common platform to create the security content. Over time, the project came through many changes to achieve [3]:

- **Easy contribution for non-programmers**,
- **Ease of use**,
- **Flexibility** — for new approaches to testing or creating a new content,
- **Extensibility for different needs** — offers remediation scripts in Bash, Ansible or Anaconda,

- **Community** — for developers, testers, and solution architects to contribute and create together the security content,
- **Quality** — easy way of creating test scenarios.

Scope of the project grew and the SCAP Security Guide's name did not reflect it. As a result, the project was renamed to *ComplianceAsCode*.

ComplianceAsCode offers the SCAP content for 18 products including Red Hat Enterprise Linux, Fedora, Debian, Firefox, and more [3]. It delivers security checks in the OVAL language and remediations in Bash, Ansible, Anaconda, and Puppet. The remediation is a script, which changes a configuration of a non-compliant setting to a compliant setting. Thanks to it, a user can get a system to a compliant state with very little effort.

As the introduction of Chapter 2 mentions, there are many policies specific for organizations and data they work with. These policies are called profiles, and they define guidance on how to configure a system. ComplianceAsCode delivers a lot of profiles specific to individual products. For example, Red Hat Enterprise 8 product has Payment Card Industry Data Security Standard (PCI DSS), Protection Profile for General Purpose Operating Systems (OSPP), Australian Cyber Security Centre Essential Eight (E8), Security Technical Implementation Guide (STIG), Health Insurance Portability and Accountability Act (HIPAA), and more profiles available in ComplianceAsCode. Profiles are there defined as lists of rules. ComplianceAsCode merges all profiles related to a product together and creates one file — datastream.

SCAP Workbench

SCAP Workbench is an application with Graphical User Interface implemented in C++ using Qt. It uses the OpenSCAP library. Intuitive GUI helps inexperienced users when working with the SCAP content. SCAP Workbench scan is shown in Figure 2.1.

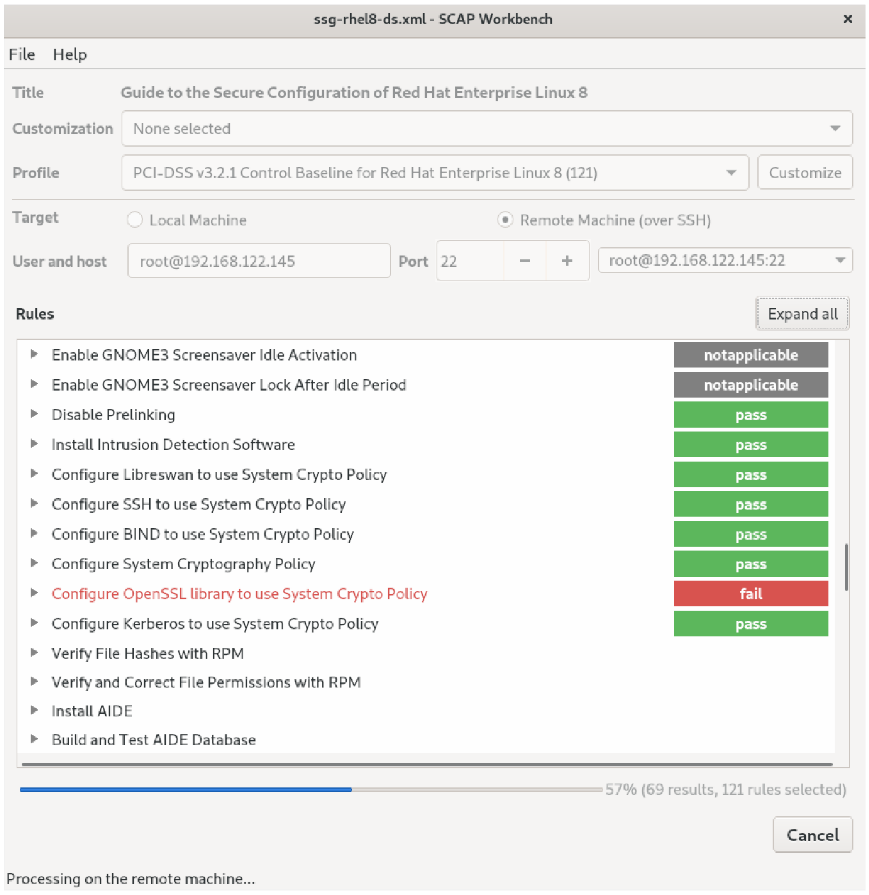
OSCAP Anaconda Add-on

Anaconda is a Red Hat Enterprise Linux installer. OSCAP Anaconda Add-on is a plug-in into the installer. With the plug-in, it is possible to choose a security policy which a user wants to be compliant right from the first boot [14]. The installer configures and installs all policy requirements. Moreover, a user is warned before starting installation if there is any non-compliant setting that the installer cannot change or it is not possible to change it after the installation. An example of such a setting is that `/var/log` must be located on a separate partition.

OSCAP Anaconda Add-on comes with Anaconda remediations. Anaconda remediations are specific for the Anaconda installer and they can easily change the installation run. They can add installed packages, disable services, set options for partitions, and more.

2.3 Project Structure

The ComplianceAsCode project aims to solve security issues that are relevant to various products of various versions while trying to reduce duplication of a content. The structure of the project reflects this complex challenge and even the directory structure seems complicated. The project files can be divided into categories [7]:


ssg-rhel8-ds.xml - SCAP Workbench

File Help

Title	Guide to the Secure Configuration of Red Hat Enterprise Linux 8
Customization	None selected
Profile	PCI-DSS v3.2.1 Control Baseline for Red Hat Enterprise Linux 8 (121)
Target	Local Machine [✓ Remote Machine (over SSH)]
User and host	root@192.168.122.145 Port 22 root@192.168.122.145:22

Rules

Enable GNOME3 Screensaver Idle Activation	notapplicable
Enable GNOME3 Screensaver Lock After Idle Period	notapplicable
Disable Prelinking	pass
Install Intrusion Detection Software	pass
Configure Libreswan to use System Crypto Policy	pass
Configure SSH to use System Crypto Policy	pass
Configure BIND to use System Crypto Policy	pass
Configure System Cryptography Policy	pass
Configure OpenSSL library to use System Crypto Policy	fail
Configure Kerberos to use System Crypto Policy	pass
Verify File Hashes with RPM	
Verify and Correct File Permissions with RPM	
Install AIDE	
Build and Test AIDE Database	

57% (69 results, 121 rules selected)

Processing on the remote machine...

Figure 2.1: Scanning with SCAP Workbench

- **The security content** — rule descriptions, OVAL checks, Bash remediations, Ansible remediations, and more,
- **Tests for the security content** — to assess the quality of the content,
- **Products** — product specific data,
- **Build files** — build configuration files and build scripts,
- **Content templates** — to reduce duplication of the content,
- **Documentations**,
- **SSG Python module** — a project specific module implementing functions for testing, building, processing rules, and more.

Restructuring the project is not simple because the most of the categories are somehow connected — products are built from the security content, some security content is templated, the SSG Python module builds, processes, or tests other parts, and more. The restructuring also includes moving directories around — it makes the project's history convoluted and historical references get lost because of it.

From the items mentioned in the list above, we can group them into three categories — rules, the build system, and tests — which will be more detailed described. Others are part of some category (Products, SSG Python module, or templates) or not important for a detailed description (Documentations).

Rules

Each rule has metadata. The metadata are a title, a description, a rationale, a severity, an OCIL, and identifiers. A rule can have an OVAL check and remediation scripts (Bash, Ansible, and Anaconda), but it does not have to. There are rules which only carry information about a security configuration, but does not have a check nor a remediation script. And there are rules where it does not make sense to have a remediation script. For example, the *Set Boot Loader Password in grub2* rule requires setting a password in `grub2`. Setting it to some default password without a user interaction could make a device unavailable or would not increase the security, because an attacker can know the default password.

In ComplianceAsCode, each rule's identifier is encoded in the containing directory name. Rules are grouped into groups. The groups can be grouped to more generic groups and so on. For example, path to *Set Boot Loader Password in grub2* rule is `linux_os/guide/system/bootloader-grub2/grub2_password/` in ComplianceAsCode. The last folder — `grub2_password` — is the identifier and others are groups where `linux_os` is the most generic.

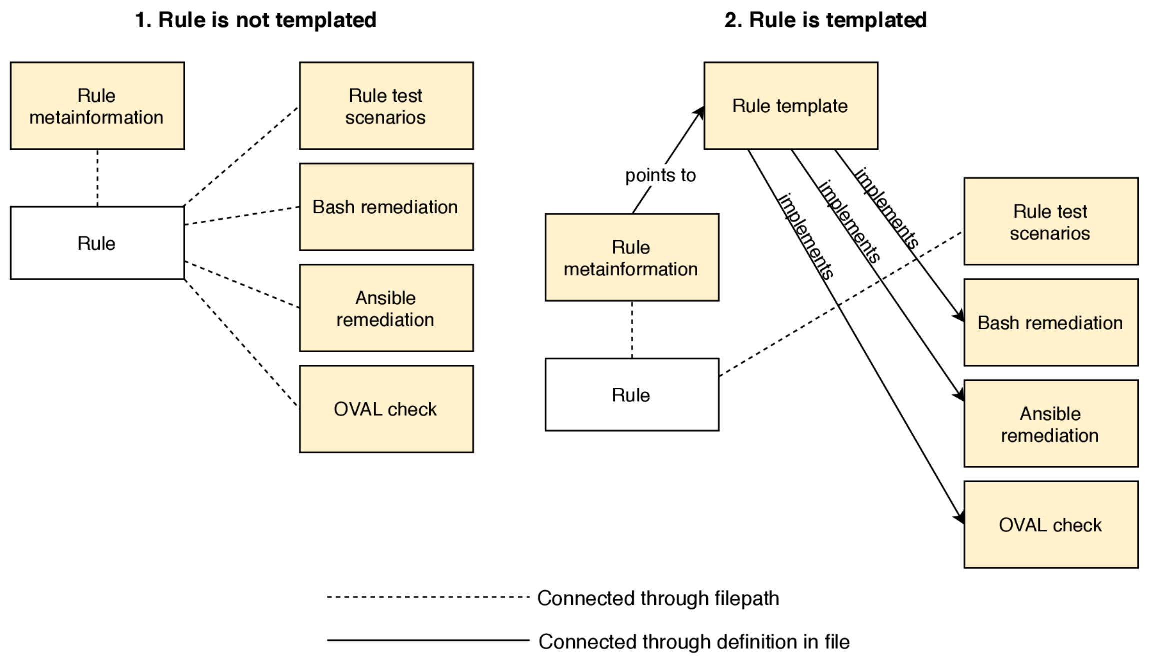


Figure 2.2: Connection of a rule and its files

A lot of rules are similar in terms of what they do. In the project, there are many rules checking a configuration of a file. The principle of checks and fixes of the rules is still the same. It always edits a line in a file. The difference is in a file path, a line, a case sensitivity of keys or values, a separator, and a file type (normal format or INI format). It all can be configured by a template. Thanks to it, redundant code is decreased, and creating new rules is faster. However, templates are under a different folder than rules and the connection between rule's metadata, a rule's check, and remediation scripts differs from not templated rules. The difference is shown in Figure 2.2.

Build System

An output of ComplianceAsCode in the form of a datastream is used as the SCAP scanner input. The datastream contains several parts. The parts include profiles information, OVAL checks, Bash remediations, Ansible remediations, and Anaconda remediation scripts.

Stated components, which assemble the datastream, are divided into individual files in ComplianceAsCode. The reason is an easy development and contributing. A contributor does not need to know the datastream structure and edit an enormous file that mixes multiple programming languages. The assembling part is done by the ComplianceAsCode build system. The build system puts together all individual files and creates the datastream. Figure 2.3 shows the ComplianceAsCode build schema. In the figure is used the old project name — Scap Security Guide (SSG).

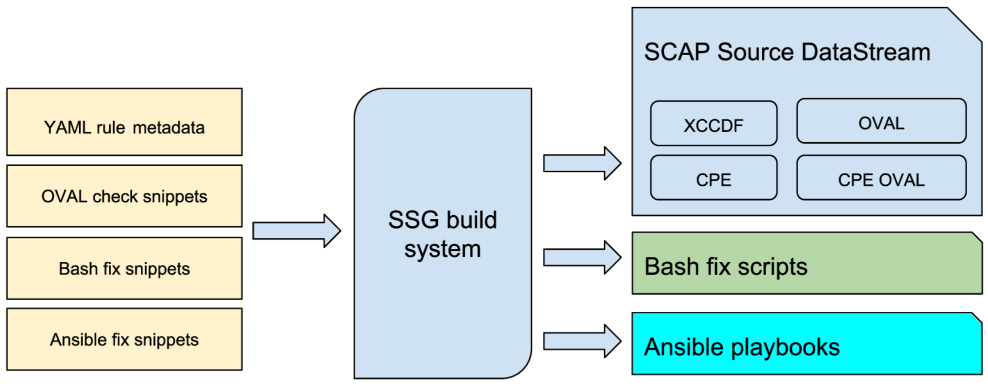


Figure 2.3: ComplianceAsCode build schema [8]

Tests

Tests can be divided into two categories. First are unit tests for the ComplianceAsCode functional part. These tests assure a quality of the build system and the Python `ssg` module. The second category is tests for rules. Tests for a rule are located in the `tests/` folder under the rule's folder. The tests for rules validate OVAL checks and remediation scripts correctness.

2.4 Project's File Types

The system for automatic filtering of tests that we propose in this work is based on the static analysis of changed files. The ComplianceAsCode project contains a number of different types of files, each of them requiring a different analysis approach. The analysis for each type is based on the file's type and the expected structure.

Since ComplianceAsCode contains many file types, support for all of them is a rather difficult task. Therefore, we concentrate on a number of selected types that we describe in this section. The types were chosen based on the following criteria:

- File types are changed the most often in the ComplianceAsCode history — we focus on files whose content changed the most often. Changes that moved files to different folders or that added new files to the project are skipped.

- Recommended by ComplianceAsCode developers — some of the file types were suggested by developers even though the files do not change often. However, analysis of the file types can save developers a lot of time spent on change analysis or on testing.
- File types that can lead to the test selection — the project contains a few file types that do not lead to any test selection. Those file types were skipped, for example, documentation files.

Overall, we chose to support the following file types from ComplianceAsCode:

- **Profile files** — files that define specific security policies. The profile files contain information (title, description, and rule selection) about profiles. The profile files are in the YAML language which is data-serialization language.
- **Ansible remediations** — scripts that change configurations of non-compliant settings to compliant settings. The scripts are written in Ansible automation language that uses the YAML format.
- **Bash remediations** — similar to the Ansible remediations but written in Bash.
- **Python files** — file type that is not used directly in security policies. It is used in the build system, in the test suite, and in utility scripts.
- **OVAL files** — files written in the OVAL language that is used to define a specific system configuration check. The OVAL language is written in XML.
- **Jinja macro files** — file type used for templatization of rule checks, of remediations, or of descriptions. The file type uses Jinja[8] templating language.

2.5 Project Testing

As 2.3 describes, in the ComplianceAsCode project are two categories of tests. The second category — tests for rules — can be difficult to perform without changing a machine state. The tests need to validate that rules are correctly checking and fixing a machine. Thus, tests adapt a machine state to their needs, perform a check, and optionally fix the state. The problem is adapting the machine state. In the worst case, a test scenario can break the machine and the machine is not usable anymore. Hence, tests for rules are executed on virtual machines or containers.

Working with a virtual machine or a container allows easy preservation of the machine state. It allows saving and restoring to the previous state if something breaks. Steps of testing a rule are shown in Figure 2.4. Description of each step:

1. Connect to a virtual machine or a container through a network interface,
2. Save the current state of the machine,
3. Perform a test scenario script,
4. Perform an OVAL check,
5. Check if the result from the OVAL check is expected,

[8]https://palletsprojects.com/p/jinja/

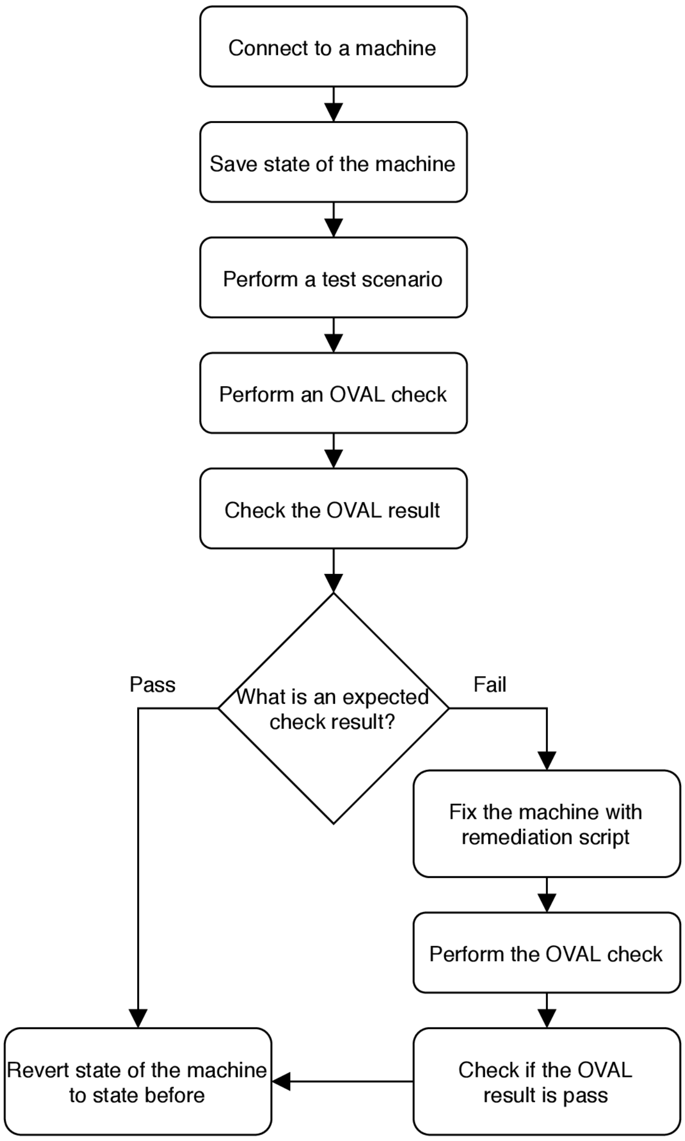

Connect to a machine
Save state of the machine
Perform a test scenario
Perform an OVAL check
Check the OVAL result
What is an expected check result?
Pass
Fail
Fix the machine with remediation script
Perform the OVAL check
Check if the OVAL result is pass
Revert state of the machine to state before

Figure 2.4: Testing a rule in ComplianceAsCode

6. If the expected result is *pass*:
   (a) Revert a state of the machine to the state before performing the test scenario.
7. If the expected result is *fail*:
   (a) Fix the machine with a remediation script,
   (b) Perform the OVAL check again,
   (c) Check if the OVAL result is *pass*,
   (d) Revert a state of the machine to the state before performing the test scenario.

Expected result means a result which should be resulted from an OVAL check after changing a machine state according to a test scenario. For example, *Ensure rsyslog is Installed* rule has test scenarios called `installed.pass.sh` and `notinstalled.fail.sh`.

The *pass* (*fail*) string in filenames indicates the expected result from the OVAL check. The `installed.pass.sh` test scenario installs the *rsyslog* tool. In this case, the OVAL check should return *pass* result. The `notinstalled.fail.sh` scenario uninstalls *rsyslog*, expects *fail* result from the check, performs the remediation script, and performs the OVAL check again with the expected *pass* result.

This way of preserving a machine state is time-consuming, especially in the case of virtual machines. It is possible to test all rules in a profile this way. This functionality is implemented in the ComplianceAsCode test suite and it is called the *combined* mode. However, it should be performed deliberately because the average time of running one test scenario is 2 minutes. The usual profile has over 100 rules. With several test scenarios per a rule, testing one profile takes few hours on a virtual machine using Intel Skylake 6th Generation processor with 2.1 Ghz frequency and 4 GB memory. For running all test scenarios for one rule, the test suite has the *rule* mode.

One of the minimal tests to consider would be a test of a freshly installed, i.e. unmodified, system when running OVAL checks for all rules in a profile and then performing remediations for failed ones. This test is also implemented by the ComplianceAsCode test suite where it is referred to as the *profile* mode. The mode checks if some rules are still failing after a remediation. However, this test is not sufficient to ensure that everything works correctly. Testing a profile as a whole is not as effective as testing each rule with several scenarios. When testing a profile as a whole, some rules influence others and it may lead to misleading results.

Chapter 3

Git Versioning System

Git[1] is an open-source version control system. A version control system records changes to a file or a set of files over time. How version control systems record changes is described in Section 3.1. With the version control system, it is possible to return to a specific version of a file. It allows reverting files to a previous state, to see who last modified a file, when an issue was introduced, compare changes over time, and more. Using a versioning control system means that if something has been broken or lost, it is possible to recover to a sanity state with a very little overhead [10].

The Red Hat Security Compliance team uses Git as the version control system for the ComplianceAsCode project. The project is saved at the GitHub remote repository. The remote repository serves as a common actual state of the project for all developers. Developers can obtain the project from it or save to it. More about remote repositories is described in Section 3.2.

This thesis implements test filtering for the ComplianceAsCode project. Test filtering is done based on changes in the project. Changes are acquired from Git by comparing previous and new versions of files. A file state is found with Git references (Section 3.3) which point to individual states of files. The comparison of states is also performed by Git (Section 3.4). If not stated otherwise, information in this chapter is taken from [10].

3.1 Storing Data in Version Control System

Generally, there are three kinds of version control systems (VCS) — local, centralized, and distributed. The local VCS is located on the local computer together with the files that are controlled, for example, RCS[2]. The centralized VCS has a single server that contains all the versioned files at a central place, for example, Subversion[3]. In the case of distributed VCS, clients fully mirror the version database from a server. Git is one of the distributed version control system.

In centralized version control systems, most operations have a network latency overhead, since it is necessary to synchronize with the central server. On the other hand in Git, most operations need only local files and local resources to operate. The entire history of the project is on the local disk and therefore Git does not need to contact a remote server to obtain, for example, the state of the project from one month before.

[1]https://git-scm.com/
[2]https://www.gnu.org/software/rcs/
[3]https://subversion.apache.org/

The way Git thinks about data is different from other version control systems. Most version control systems store information as a list of file-based changes. On the contrary, Git thinks of its data more like a set of snapshots of a miniature file system. A snapshot, called a *commit*, describes what the project files look like at the given moment.

Each version of a file in Git is represented as a *blob*. Blob (binary large object) holds a file's data, but it does not contain any metadata about the file [4].

A file can have several versions at the same time in Git thanks to *branches*. Branches diverge development from a main line (often called the *master* branch) of development. It is useful for developing without messing the main line. Switching from an actual branch to another is called *checkout*, derived from Git command `git checkout`.

3.2 Remote Repositories

A repository stores files as well as a history of changes made to those files. A repository can be located at a user's computer (a client's repository) or at a server (a remote repository).

With Git, it is technically possible to upload, called *push*, and download, called *pull*, changes from client repositories, but it can be easily confused what others are working on and it is not possible to access the repository if their computer is offline. Therefore, the preferred method for collaborating is to set up an intermediary remote repository, and push to and pull from that. Even though every user has its own copy of the repository, it is agreed that the remote repository contains the newest accepted version of the project.

There exist several options for creating an intermediate repository and they use different protocols for remote synchronization:

- **Local protocol** — the remote repository is in another directory on disk. It is simple, and it uses existing file permissions and network access. However, these methods are difficult to set up and to reach from multiple locations than other network protocols.

- **HTTP protocols** — Git can communicate over HTTP in two different modes — the „Smart“ HTTP protocol or the older called „Dumb“ HTTP. The „Dumb“ HTTP protocol is very simple and generally read-only. The „Smart“ HTTP protocol is very similar to SSH or Git protocols. It can use various HTTP authentication mechanisms and can be also set up to serve anonymously.

- **SSH protocol** — a common transport protocol for Git, because SSH access to servers is already set up in most places and if it is not, then it is easy to do so. The negative aspect is the impossibility to serve anonymous access of the repository over it.

- **Git protocol** — provides a service similar to the SSH protocol and works based on special a daemon that comes packed with Git. The protocol's setup is difficult and it lacks authentication. Generally, it is undesirable for the Git protocol to be the only access to the repository. The Git protocol is used for read-only access.

The single largest host for Git repositories is GitHub[4]. It offers communication via HTTPS, SSH, or Git protocols [6]. GitHub is the central point of collaboration for millions of developers and projects, which use it for hosting, issue tracking, code review, and other things. However, GitHub is not a direct part of the Git open-source project.

[4]https://github.com/

Typically, working with GitHub is designed around a particular collaboration workflow. It is centered on *Pull requests*. Basically, a contributor copies a repository, called *fork*, implements changes, pushes them to the forked repository and creates a Pull request against the original repository in the GitHub's web interface. The flow works whether collaborating with a small team, or a globally distributed company. The detailed process of Pull requests:

1. Fork a project — when contributing to an existing project without push access, then workflow starts with *forking* the project. It creates a copy of the project repository into the user's namespace where the user has push access, regardless of the user's rights in the original repository.

2. Create a branch from the master branch.

3. Make new changes and capture them using commits to the project.

4. Push the branch (new commits) to the GitHub project fork.

5. Open a Pull request from the forked GitHub repository to the project's original repository — the main part of the workflow, which allows a project owner to review the suggested changes before merging them into the project.

6. The project owner merges or closes the Pull request.

GitHub offers dozens of services that can be used to integrate the repository with other commercial or open-source systems. These systems provide, for example, bug tracking, documentation system, issue tracking, or Continuous Integration. Perhaps the most common of these is the Continuous Integration (CI) and testing services. They provide automated testing of the new code. Usually, a CI service is related to Pull requests when the service reacts to new changes added to a Pull request by running tests and reporting the results to the Pull request thread.

3.3 Git References

Each commit has a SHA-1 value. The SHA-1 value is a unique identifier within an entire repository. Each commit points to the previous one thereby creating a tree-like structure of commits. With the SHA-1 value identifying a commit, it is possible to traverse the tree-like structure — go through the history of a project.

A SHA-1 value can be stored under a simple name in a file. The name is an alias to the SHA-1 value. The name can be then used instead of the raw SHA-1 value. In Git, these are called *references* or *refs* and can be found in the Git repository in the `.git/refs` folder. The parent folder, `.git`, groups all necessary information (logs, commits, ...) for the Git version control system. In Git, there are few different reference types:

- **Tag** — an object which is very much like a commit object — it contains a tagger name, a pointer, a message, and a date. The difference is the tag generally points to a commit, while the commit object points to a tree. The tag pointer is like a branch reference, but it always points to the same commit.

- **The HEAD** — a reference to the branch a user is currently on. The `HEAD` is not a reference as others, but it is a symbolic reference. By a symbolic reference, it is meant it does not contain a SHA-1 value but a pointer to another reference. In the case of HEAD, it is a reference to a reference of a branch. As Listing 3.1 shows, HEAD can be found in the .git/ folder.

```
$ cat .git/HEAD
  ref: refs/heads/master
```

Listing 3.1: HEAD pointing to the master branch

When committing a change, it creates a commit object and specifies a parent of the commit object. The parent is the SHA-1 value referenced by HEAD.

- **Remote** — difference between remote references and branches is that remote references are considered read-only. It is possible to checkout to remote reference, but Git will not point HEAD to it and will not update it with the commit command. Git manages these references as bookmarks to the last known state of the branches on remote servers.

3.4 Comparing Git Versions

When comparing versions of two items, the appropriate way of displaying changes is by showing differences between them [4]. For example, the Linux and the Unix diff command compares the file line by line and summarizes the differences, as shown in Listing 3.2.

```
$ cat file1
  Hello
$ cat file2
  Hello World!
$ diff file1 file2
  1c1
  < Hello
  ---
  > Hello World!
```

Listing 3.2: Differences between two files displayed by diff utility

The -u option used with the diff command, produces the output in a unified diff format. The unified diff format is a standardized format used to share modifications. Minus signs indicate lines from the old file and plus signs indicate the new file. The output also contains a date, a time zone, and the differences are divided into chunks. Each chunk starts with @@, the number of the starting line in the new or in the old file, and the chunk length [9]. The output of a unified diff is shown in Listing 3.3.

```
$ diff -u file1 file2
  --- file1 2019-11-26 18:35:50.202960384 +0100
  +++ file2 2019-11-26 18:34:41.964413558 +0100
  @@ -1 +1 @@
  -Hello
  +Hello World!
```

Listing 3.3: Differences between two files displayed in unified diff format

The diff command can be extended to show differences among multiple files and entire directory hierarchies with the -r option. It computes differences of all pairs of files found in two directory hierarchies. The command traverses each hierarchy, pairs files by path names, summarizes the differences between each pair, and produces a set of unified diffs [4].

Git has its own diff facility under the git diff command. The command is similar to diff -r. It traverses two tree objects and generate a representation of the variances. The git diff command has its own powerful features tailored to the particular needs of a Git user. There are three basic sources for trees or tree-like objects to use with git diff [4]:

- Any tree object within the commit graph — a tree object represents directory information. It contains path names, blob identifiers, and metadata for files in the directory.
- The working directory.
- The index — the index captures the project's structure at some moment in time. It records and retains changes until they are committed.

Typically, trees compared are named via commits, tags, or branch names. Also, a file and a directory hierarchy of the working directory can be treated as trees. The git diff input can also be specified as a range. The most common range specification is the double-dot syntax [4]. For example, git diff master..feature shows changes included in the feature branch that are not in the master branch.

The git diff output is similar to the unified diff format with a few additional information. In the beginning, there is a git diff header followed by extended header lines. The extended header lines contain information related to the displayed change. There can be a path from which the file was copied, how the file mode (a file type and file permission bits) changed, how the file was renamed, and so on [2]. Listing 3.4 shows a git diff output when a file got changed.

```
$ git init
Initialized empty Git repository in /home/Example/.git/
$ echo "Hello" > file; git add file; git commit -m "Adds file"
  [master (root-commit) 8368ed6] Adds file
   1 file changed, 1 insertion(+)
   create mode 100644 file
$ echo "Hello World!" > file # Change the file
$ git diff
  diff --git a/file b/file
  index e965047..980a0d5 100644
  --- a/file
  +++ b/file
  @@ -1 +1 @@
  -Hello
  +Hello World!
```

Listing 3.4: Differences between two files displayed by git diff command

Chapter 4

Design of Automatic Tests Filtering for ComplianceAsCode

The base of test filtering is knowledge which tests cover which parts of the project's behavior. With such knowledge, it is possible to determine the set of tests that are sufficient to be run for a certain change. A manual way of choosing tests is in most cases easy for a developer that knows the implementation, because he knows which parts of the implementation are covered by which tests.

On the other hand, for an automatic test filtering, two different approaches are available:

- **General approach** — it is insensitive to any internal dependencies of the project. It is usually based on automatic analysis (static, dynamic, or both) of changes in the project and of its tests. The selection of tests is based then purely on results of such analysis,
- **Project-specific approach** — it operates with the knowledge of the project structure and of dependencies among project files. It extends the first case since some kind of analysis must be still performed, however, project specifics are now used for interpretation of the results.

Each of the cases has advantages and disadvantages. The first case is general and can be used on any project but it is usually not as accurate at tests filtering as the second case. The second, usually more accurate approach, works only for a specific project and a change in the project structure or in its dependencies can cause that the filtering will not work properly and that it must be adjusted to these changes.

In ComplianceAsCode, the general approach, is not sufficient because connections between the implementation and the tests depend on several factors. General static analysis of ComplianceAsCode content can be performed, however, knowledge of internal dependencies must be taken in mind — comments can contain metadata, content of files can be generated from a template, a file content can be only a reference to a macro that is expanded during the project build (thus the file content is not the resulting content), and more — more information on different file types from ComplianceAsCode are described in Section 2.4. With respect to this, the automatic filtering of tests that we propose in this chapter uses the project-specific approach. We focus on automatically analyzing the changed source files, using a combination of general static analysis with project knowledge. The input to the analysis are new changes in the ComplianceAsCode project. The output is a list of detected changes and a list of tests that are recommended to be run for the changes. A high-level description of the proposed method workflow is presented in Section 4.1. The following sections describe individual parts in more detail — Section 4.2 describes how changes are obtained, Section 4.3 describes methods for analysis of the changes, Section 4.4 is about computing parts of the project affected by the changes, and Section 4.5 describes the last part that computes the resulting set of tests.

4.1 High-Level Method for Automatic Test Filtering

The design of the system for automatic filtering of tests is shown in Figure 4.1. The input to the system is a Pull request number that serves as the reference to new changes in the project. From the reference, the actual changes are obtained. These changes are analyzed and other affected parts are computed. The computing of affected parts can output additional files that need to be analyzed. At the end, the analysis results and the information about other affected parts are used to compute a set of tests that are sufficient to be executed to test the given change.

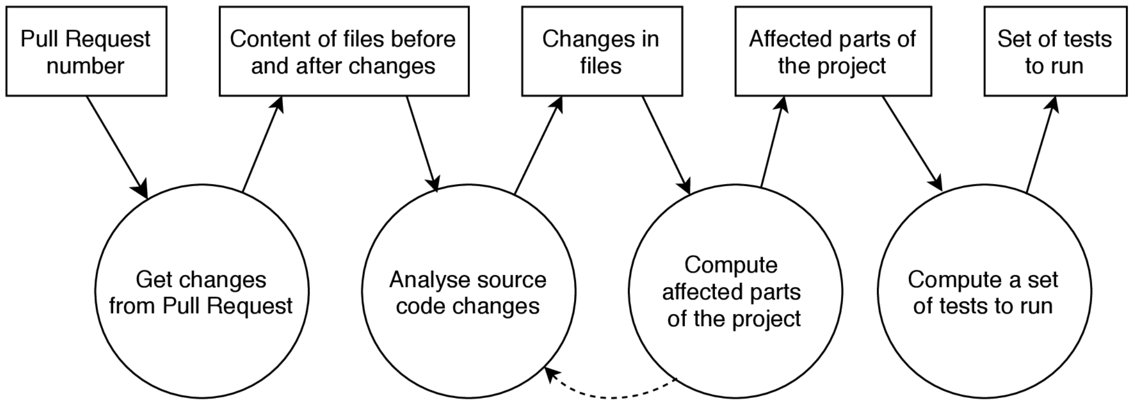

Pull Request number
Content of files before and after changes
Changes in files
Affected parts of the project
Set of tests to run
Get changes from Pull Request
Analyse source code changes
Compute affected parts of the project
Compute a set of tests to run

Figure 4.1: Diagram of the designed system for automatic filtering of tests

4.2 Getting Changes

The static analysis phase is the most computationally expensive part of the implemented system. Each file is analyzed twice — in the state before the changes and after the changes. The majority of the analysis is unnecessary because it finds out changes in few files only and in most of the files, nothing changes. Therefore, it is convenient to pre-process the changes and find out which files changed and hence must be analyzed. To this, we can use information from the version control system. Moreover, since files can be moved, renamed, etc., it is essential to know the mapping between files from the two versions. As an example, Figure 4.2 shows a change in a source tree. Without any information about the mapping, we do not know if `File B` was removed and `File D` added or if `File B` was moved to `File D`. Again, this can be determined from the version control system.

Analyzing each file in ComplianceAsCode could be useful for analyzing built files that are created from its build system. The build system is complicated, it creates intermediate files from sources and then connects them to datastreams (documents used as an input to SCAP scanners that wrap multiple XCCDF documents that contain rules for profiles). The

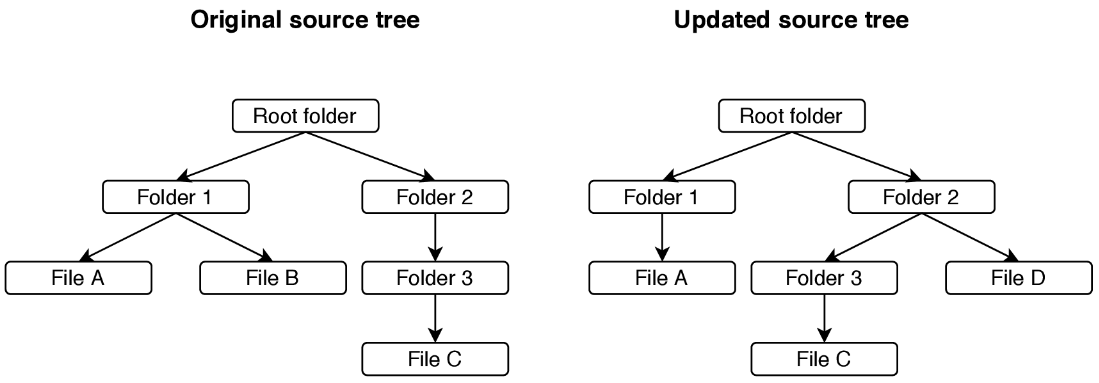


Figure 4.2: File moved within a source tree

analysis would reveal connections among the source and the built files. It could be useful, even for an experienced developer, to see these connections.

The ComplianceAsCode build takes several minutes and even though the time is negligible when compared to duration of test runs, we decided not to analyze each file including the built ones because it is computationally expensive. If we will need to analyze built files for a specific analysis, then we will build ComplianceAsCode only during the analysis. Thus, we save time and we do not perform unnecessary builds. The information about changed files is obtained from the version control system, Git. Another reason for using Git is that the ComplianceAsCode project uses Git for version control and all contributions to the project happen on GitHub.

The input for getting changes is an identifier of a Pull request, i.e a number. The remote repository is the main place for the ComplianceAsCode project where all new features, new rules, bug fixes, and new tests are introduced, reviewed, and approved. The workflow starts with creating a new Pull Request. Hence, a convenient way to refer the new contribution is the Pull request number.

From the Pull request, changes are obtained with Git version control system and saved to a record that represents a file record. Each changed file is saved to an individual file record. The output from this part are these file records.

4.3 Analyze Source Code Changes

The ComplianceAsCode project contains different file types as Section 2.4 describes. For each file type, we propose a separate approach to analysis of changes, with respect to the file's language and expected structure. The proposed analyses are described in the rest of this section.

The input to this phase are file records that contain contents of changed files before and after the changes. The output is a record that contains information about importance of the changes in each file — if the change affects the file thus a test will be selected or the change is not important thus no test will be selected — and additional information for following computation of affected parts and tests selection. Each changed file has the information in its own record and the form of the information differs by the kind of a source file.

Profiles

A profile file is a file in YAML format that defines a specific security policy. In a profile file, several keys can be used — `documentation_complete`, `title`, `description`, `extends`, and `selections`. Most of them serve for documentation purposes, only the `selections` key is important for the analysis. Under the key, there is a list of rules that are selected for the policy. If the list of rules changes, the profile test should be run because an added or a removed rule can cause the profile not to work as a whole.

YAML format uses lists and dictionaries (each key has a value). These can be nested and mixed together — it forms a tree structure. The tree structure root node is the document itself, an unlabeled edge to a node represents an item from a list where a single node corresponds to one value from the list, and a labeled edge to a node represents an item from a dictionary where the edge label corresponds to the key and the node is the key's value. The comparison of the old and the new tree traverses recursively from the root node to leaf nodes in both trees and tries to find most similar paths in them. The order of edges and nodes is not important for profile files because they are order insensitive — it does not matter if `description` precedes `title` and also the order of rules in the rule selection does not affect their order in the built datastream file.

Ansible Remediations

Ansible is an automation language used for deployment, orchestration, and, most importantly for the ComplianceAsCode project, for configuration management.

Ansible remediations use Ansible, however, they can be also templated with Jinja macros and hence the content of the file does not have to be an actual Ansible. If it is templated, then during the build the macro is expanded, a new file for remediation is created within the build files, and the new file is used for remediating. Remediations can contain metadata that are present at the beginning of the files in the form of comments. These metadata are used during a build process and the analysis should record if they have changed.

Ansible remediations are in the YAML format, however, contrary to the profile files, they do not have a defined structure and they can contain arbitrary content. The only known key is `name` that specifies the name of the remediation. Hence, we use a more straightforward way to analyze Ansible remediations. We compare the old and the new content as strings, obtain the differences in the unified format, and analyze the output. In the output, metadata changes can be seen and it does not matter if the file is templated with a Jinja macro or not.

We distinguish the following types of changes of Ansible remediations:

- changed line has a pattern that starts with `{{{` and ends with `}}}` — usage of Jinja macro changed,
- changed line starts with `#` and `platform`, `disruption`, `strategy`, `complexity`, or `reboot` keyword follows — metadata changed,
- changed line is empty or starts with `#` and the next word is not a keyword — an irrelevant change,
- all other cases — the usage of Ansible changed.

The particular changes can be identified using regular expressions (see Chapter 5 for more details).

A regular expression can also find that the remediation templatization changed — the remediation did not use a Jinja macro and now it uses the macro, or vice versa. This type of a change is important and if it happens, no more analysis has to be performed because the file content completely changed.

Bash Remediations

Bash remediations are similar to Ansible remediations — they are used for configuration management and they can have the same metadata. However, as the name implies, they are written in Bash language or templated with Jinja macros.

If a Bash remediation is templated, then it can be analyzed the same way as the Ansible remediations (see description of Ansible remediations analysis). Similarly, if the remediation templatization changes, then it is an important change because the file content was completely changed.

A different case is if the Bash remediation is not templated and contains actual Bash code. Bash code can be compared using lexical analysis. The code is parsed into tokens and the individual tokens from the old Bash code are compared with the tokens from the new code. If a change in a token is found, then the analysis determines that the source code has changed. Changes in comments and indentations do not affect the lexical analysis, hence these types of changes are filtered out. Such filtering is sufficient for our analysis because Bash remediations are short scripts that fix specific configuration mostly within few lines and in a vast majority of cases, a change in the code causes an actual change of the script semantics.

Python Files

Python code is used in the build system, in the test system, and in utility scripts. When a change is found in any Python file, it is sufficient to run a single specific set of basic tests from the project. A more important decision, in this case, is if these tests should or should not be run.

Python source code is parsed to an Abstract Syntax Tree (AST). It is an abstract structure in the form of a tree that represents the code. Afterwards, we compare ASTs for tree isomorphism. This allows us to filter out, e.g. changes in comments.

OVAL Files

OVAL language is written in XML, therefore, a general analysis for XML files can be used. Each OVAL file contains an OVAL check that represents a rule check.

Before we start analyzing an OVAL file, we must add the XML header and the XML footer to the file. It creates the valid XML file that can be parsed. In ComplianceAsCode, the header and the footer are added during the build.

XML is a markup language. It has start and end tags, these tags can have attributes, and they form elements. Elements can be nested together and they form a tree-like structure. For analysis of the differences, we use an algorithm for detection of changes in hierarchically structured information [1].

Afterwards, we distinguish a number of types of changes:

- New node inserted or node removed — if the node was added to or removed from a `metadata` node, then the change is irrelevant for the OVAL check. If it was added to or removed from other node, then it is considered an important change.

- Node moved — if the node was moved within the same node that specifies one part of the check — `metadata`, `criteria`, `textfilecontent54_test`, `textfilecontent-54_object`, or `textfilecontent54_state` node — then the change does not affect it. However, if the node was moved from one part of the check to another, then the check could not work at all, thus this must lead to testing every time.
- New attribute added — a new attribute does not affect the OVAL check.
- Attribute removed, renamed, or attribute's value changed — if the changed attribute is `comment` or `version` then the check is not affected by the change but other attributes could affect it thus they are considered as important changes.
- Text within a node changed — if the changed text is a `title`, `description`, or `platform` node then the check is not affected, otherwise it could be.
- Text added after a node — adding a text after a node, not to a node, is not a typical type of change in an OVAL check and must be verified every time.
- Inserted comment — this type of change can be ignored, because comments in OVAL checks do not contain important information.

Jinja Macros

Jinja is a templating language. In the ComplianceAsCode project, it is used for templatization of rule's checks, of remediations, or of descriptions, which differ only in a part that can be parameterized. For example, package installation rules and their remediations are usually templated — the package name is parameterized and the other parts are the same. Therefore, a macro can contain any string and it does not distinguish if it is a text or a source code. The important knowledge is where the macro is used and whether it makes sense in the context. Jinja macros can be also used within other Jinja macros.

The only certainty for analyzing Jinja macro definitions are the `macro` keyword, the macro name, the parameters at the beginning of the macro, and the `endmacro` keyword in the ending tag. Even the format of these tags can vary in systems that use Jinja macros because Jinja allows to specify a string for marking the beginning and the ending of a macro — Jinja macro documentation[1] refers to a format that uses `{{` and `}}` to mark the start and the end of the tag, respectively, while ComplianceAsCode uses `{{%` and `%}}`.

The important part in analyzing Jinja macros is finding which macro has changed. Each file with Jinja macros contains several macros and we need to filter out not changed ones.

The file with Jinja macros is analyzed as a string. We use a combination of getting changes in the unified format, analysis of the file content, and regular expressions to parse the changed macros (see Chapter 5 for details).

4.4 Compute Affected Parts

The input to this phase is a record that contains information about the importance of found changes and additional information about them, e.g. the name of the changed rule. Based on the information, it computes the set of files that are affected by the changes. Computing affected parts can lead to different scenarios depending on the file type and

[1]https://jinja.palletsprojects.com/en/2.11.x/templates/#macros

the change — a new file will need to be analyzed, a product that uses changed files will be tested, or nothing will be tested because the changed part is not used in any built output. The most important part in computing affected parts is to determine what type of file was changed. The record from the input is filled with information about other affected parts of the ComplianceAsCode project. With the affected parts, we mean files but the format how the information is saved the record depends on the analyzed file — if a profile file is affected then we save the information as the profile name, if a file from built files is affected, then we save the file path, and so on. The output from this phase is the filled record.

Profiles

Profiles can be extended by other profiles. That means if a profile was changed, then all profiles that are extending it are also affected by the change. These profiles can be found in the same folder as the changed profile and they contain the `extends` key with the value equal to the name of the changed profile. During the computation of affected parts, we check the profile files in the same folder to see if there is any profile that extends the changed one. This check is recursively repeated to find profiles that extend the extending one. All affected profiles, including the changed one and all profiles that (directly or indirectly) extend it are saved to the record.

Ansible Remediations

Ansible remediation is a remediation for a specific rule. The rule must be a part of a profile — the rule must be in the `selections` part of some profile. If the rule is not a part of any profile, then it cannot be tested. This is because the rules to test are selected from datastreams, which are created from profiles, thus if no profile contains the rule, then also no datastream will contain it. During the computation of affected parts of an Ansible remediation change, we must search all profile files (all files in all `profiles` folders) in ComplianceAsCode and find at least one profile that includes the changed rule. If no such profile is found, the rule cannot be tested and we do not add it to the record.

Bash Remediations

Similar to Ansible remediations. A Bash remediation is a remediation for a specific rule, therefore if the changed Bash remediation is from a rule that is not present in any profile, then it cannot be tested and hence we do not add it to the record.

Python Files

If a Python file was changed and during the analysis we found that the change must be tested, then we have recorded to the record that the functional test must be selected in next phase. No computation of affected parts is needed because Python files do not affect rules, profiles, or products. They affect the build system and the testing system that are checked by the functional test.

OVAL Files

The OVAL file represents a check for a rule, thus the rule must be a part of some profile as we described in the Ansible remediations part. The difference is that tests from an OVAL check can be used within other OVAL checks. The OVAL file can have one or more tests and each of them has an `id` attribute. The value of the attribute can be used in other OVAL files via the `definition_ref` attribute that references the `id` value. Thus, we parse values of all `id` attributes from the changed OVAL file and search for all OVAL checks that contain a `definition_ref` attribute that has the same value as any `id` from the changed OVAL. All rules that use a check from the changed or from an affected OVAL file must be added to the record provided they are a part of a datastream. Hence, before the affected rules are added to the record, we search if they are included in any profile.

Jinja Macros

When computing affected parts for a changed Jinja macro, we must first find the name of the macro. From the previous phase, we know which macro changed and hence we can parse from it the name with a regular expression. The format of Jinja macros is the following:

```
{{%[-] macro NAME(PARAMETERS) [-]%}}
MACRO BODY
{{%[-] endmacro [-]%}}
```

The name we are looking for is in the first line.

With the name we can search for usage of the macro. Usage of each macro has format:

```
{{{ NAME(PARAMETERS) }}}
```

There are three cases where a changed Jinja macro can be used:

- The changed macro can be used in other Jinja macros — when the macro is used in other macros, then we need to find usages of those macros in other macros. This process is repeated until we know all macros that (directly or indirectly) use the changed macro.
- The changed macro is used in a specific rule file — the macro is used in an OVAL file, a remediation file, or a rule description file. Such a file must be processed two times through the build system so the macro is expanded — the first time with the value of the macro before the changes and then after the changes. Then, we can create a file record the same way as we have created file records during obtaining changes from Git — create a record that has the content of the file before the changes, the content of the file after the changes, the file path, and the flag that indicates the file was modified. The created file record is added to the record from the input as the affected file that must be analyzed. This is done with all files that use the macro. After the Jinja macro analysis, the system processes the new file records the same way as other changed files — uses them as inputs to the analysis phase.
- The changed macro is used in a template file — the template file is a file that generates files during a build. From the template file name we must parse out the template name and the type of files that are generated from the template (OVAL files, Bash remediations, or Ansible remediations). The template name is used for searching the template usage in rules — rules that have generated checks and remediations from templates have the `template` keyword with a name that is the same as the template name in their rule description file. When we know specific rule names and which type of files are generated from the template, then we know what the names of the created files from the template during the build will look like. The format is the rule name

with a suffix according to the file type (OVAL files have `.xml` and so on). Then, we can proceed similar in a way as we do for the second mentioned case — run the ComplianceAsCode build two times (before and after the changes), get contents of created files (we already found out the names of files that are generated from the changed macro), create file records, and add the file records to the record from the input as affected files. The system then analyzes the new file records the same way as others (inputs them to the analysis phase).

4.5 Compute Set of Tests

The input for computing set of tests are records filled with information from file analyses and from computing of affected parts. The records already contain all important information about what must be tested. The information from the records serves for the selection of specific tests.

The tests in ComplianceAsCode are specific that they do not need paths to test scenarios. Inputs to tests differ for individual test types:

- **rule test** — uses the testing suite in the *rule* mode. Inputs are the name of the tested rule and a datastream of a product that contains a profile with the tested rule,
- **profile test** — uses the testing suite in the *profile* mode. Inputs are the name of the tested profile and a datastream of a product that contains the tested profile,
- **product test** — builds the product's datastream. The only input is the name of the tested product,
- **functional test** — no input to the test.

The record from the input contains information in the form of a rule name with a product name, a profile name with a product name, a product name, or the boolean variable that signalizes a changed functionality. Thus, this phase selects tests based on the information provided in the record, for example, if the record contains a rule name with a product name then the product test and the rule test are selected. In this case, the product test is important for the rule test because it builds the datastream that is used for the rule testing (same for the profile test). Detailed description of testing modes in ComplianceAsCode is in Section 2.5.

The output from the computing is a list of tests. These tests are in the form of commands that are almost ready to be used for ComplianceAsCode project. The user must only change the path to script that runs the test suite and the name of virtual machine where tests will be performed.

Chapter 5

Implementation

This chapter describes the implementation of the system for the automatic filtering of tests. The implementation and the description of it correspond to the design described in Section 4.1 with a difference in the analysis part and the computing affected parts part. These two parts overlap because some findings during the analysis are actually part of the computing affected parts and vice versa. The implementation is open-source and available on GitHub[1].

The system is implemented in Python 3. Python 3 was selected because the ComplianceAsCode project uses it for the build system, the testing library, and utility scripts. Another reason is its library variability.

In Section 5.1, the main control of the system is described. The following sections deeply dive into individual parts that are run from the main control. Section 5.2 describes how changes are obtained from the ComplianceAsCode project with Git commands. The next section, Section 5.3, describes the implementation of analyses of file types that are described in the previous chapter, in Section 4.3. In the analysis part, we have focused on extensibility and thus more analyzers can be added easily without changes in already implemented parts. For the extensibility, we have used polymorphism for the analyzers and for their selection. Section 5.4 outlines how specific test commands are obtained from results from an analysis. The last section 5.5 is dedicated to printing results and commands from the run of the system.

5.1 Main Control

The `content_test_filtering.py` file implements the highest level of the filtering algorithm. In the beginning, logging and variables are initialized. Then the main part starts.

At Algorithm 1 is the main control of the implementation. At line 3 starts the main loop that analyzes each changed file, computes its affected parts, saves results, and extends changed files with affected files. In the algorithm, the analysis and finding of affected parts are divided, but in the implementation, those parts are in a single step.

5.2 Getting Changes

Changes are obtained with Git commands. Python has the `GitPython`[2] library for working with Git. Git commands in the library are implemented either in a pure Python or through

[1]https://github.com/mildas/content-test-filtering
[2]https://github.com/gitpython-developers/GitPython

```
initialize Git repository R;
Files = changed files from R;
do
    File = Files.pop();
    result = analyze(File);
    affected_files = result.findAffected();
    tests.append(result.tests);
    logs.append(results.logs);
    Files.extend(affected_files);
while Files not empty;
print(tests.getCommands());
print(logs);
```

Algorithm 1: Main control of automatic test filtering

Git commands. The pure Python implementation is less resource-intensive but some of the commands are not implemented in that way or they miss some of the options implemented. If such commands are needed, the implementation directly uses Git commands. In this section, we present the used Git commands, however, in the implementation we use `GitPython` methods that are equivalent to the presented Git commands.

This phase consists of five steps:

1. Obtaining changes from the remote repository to the local repository,
2. Finding the last common commit of the master branch and the branch with changes,
3. Finding files that are changed in the branch with changes,
4. Obtaining contents of the changed files,
5. Creating file records with the contents of the changed files.

The source code for this phase is in the `ctf/diff.py` file. The basis of the code is the `GitDiffWrapper` class that wraps the `GitPython` library and provides methods important for the filtering system. The class is a singleton because some analyses need to know the repository path, need to get the state of the project before and after changes, or build the project. As a singleton the class can be imported and used without a need to re-initialize the repository or pass it in as an argument. The rest of this section describes the above five steps in detail.

Obtaining Changes from the Remote Repository to the Local Repository

The `GitDiffWrapper` class has a method that prepares the ComplianceAsCode repository. It creates a directory and clones the repository to a local repository. If the local repository exists on a disk and its path is provided to the system with the `--repository <path>` argument, then nothing is cloned. It saves time and it is also useful for filtering tests for a local change that is not available at the remote repository yet.

The `git_diff_files` method from `GitDiffWrapper` manages obtaining changes. First, it finds a remote repository that will be used for pulling changes and updating the local repository. As a default value is used the ComplianceAsCode repository[3] but it is possible to

[3]https://github.com/ComplianceAsCode/content

use an arbitrary fork of the remote ComplianceAsCode repository with the `--remote_repo <repository_url>` argument.

The input to the `git_diff_files` method, thus the input to this phase, is a name of a branch or a Pull request number. The input depends if the application was run with the sub-command `branch` or `pr`. The difference is how the changes are obtained from the remote repository and to which local branch they are saved:

- **branch** — obtains changes from a branch. It works with the exact name of the branch and saves it to a local branch with the same name.
- **pr** — obtains changes from a Pull request. It pulls changes from a remote branch `pull/ID/head` where `ID` is a Pull request number. This reference is specific for repositories saved on GitHub. Changes are saved to a branch with the name `pr-ID`.

Now, we have a new local branch that contains the changes of interest that we call the *changes branch*.

Finding the Last Common Commit

After the *changes branch* is created, it is necessary to find the commit at which it diverged from the master branch. The common commit, against which the changes need to be compared, is shown in Figure 5.1. It is not possible to compare the *changes branch* with the master branch directly because the master branch can contain additional changes that are not present in the *changes branch*.

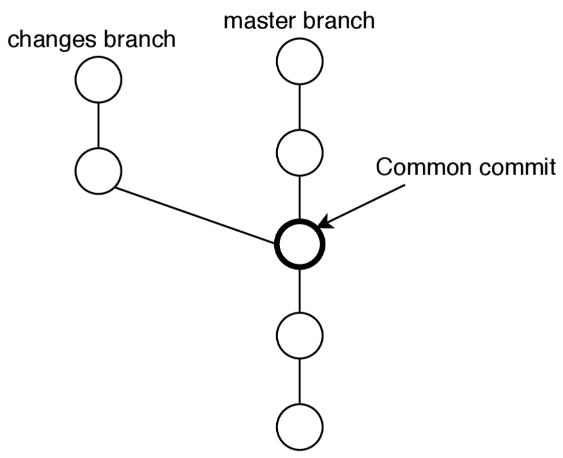


Figure 5.1: Divergence of branch with changes

When finding the common commit, two cases can happen — the *changes branch* can be already merged into the master branch or not. The difference between these cases is shown in Figure 5.2. It can be detected by finding the most recent common commit of these two branches. When the common commit is the same commit as the last commit of the *changes branch*, then the *changes branch* was already merged. When the most recent common commit is not the last commit of the *changes branch*, then the commit is the common commit we were looking for and we can use it for the comparison.

If the *changes branch* is already merged, we need to find the common commit from which was the *changes branch* created. That can be achieved in three steps:

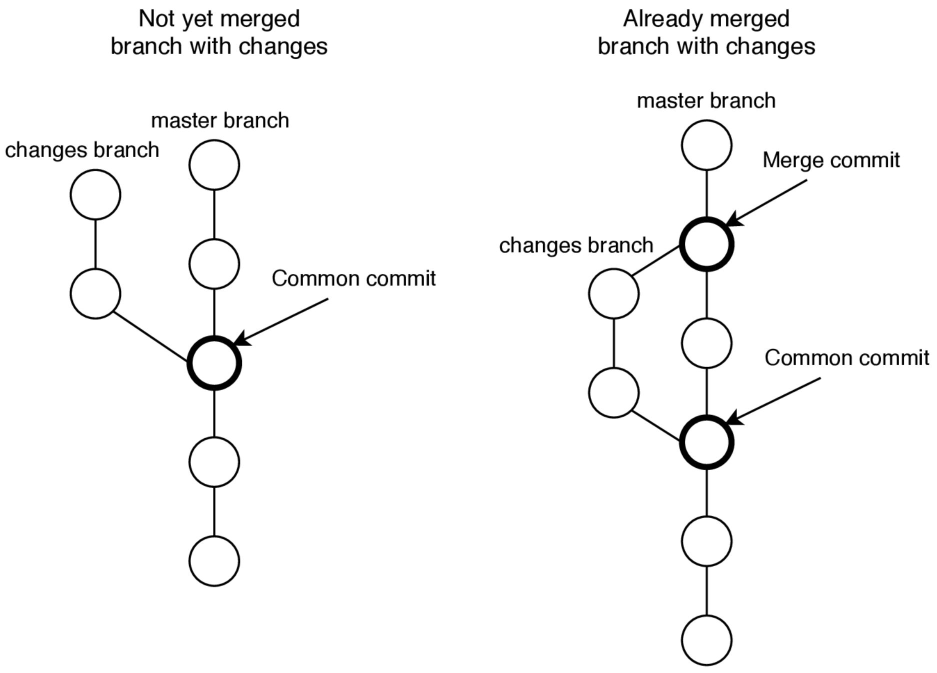


Figure 5.2: Difference between not merged and already merged branch

1. Get commits that are descendants of the *changes branch* and ancestors of the master branch. We can do that by invoking `git log`:

   ```
   git log MASTER_BRANCH ^NEW_BRANCH --ancestry-path \
   --reverse --format=%P
   ```

2. Retrieve the oldest record that has two parents commits associates,

3. Get the common ancestor of the two parents commits, e.g. by invoking `git merge-base`:

   ```
   git merge-base --all commit1 commit2
   ```

The first step gets *changes branch* descendants and the master branch's ancestors with the `--ancestry-path` option. Usually, the returned list of commits is sorted from the newest one to the oldest one. In this case, the oldest commit from the list is needed — that is the merge commit of the master branch with the *changes branch* — thus the list is reversed with the `--reverse` option. With the `--format=%P` option, it is possible to get a hash of the parent commit. In this case, the oldest record's parent commit that we are looking for are actually two commits because it is a merge commit that has two parents — the last commit from the *changes branch* and the last commit from the master branch before the merge commit. The common ancestor of the two commits is the common commit from which was the *changes branch* created — the common commit that we were looking for.

Now, we know the common commit from the two branches and we can use it for finding files that were changed in the *changes branch*.

Finding Files that are Changed in the Branch with Changes

With the hash of the common commit, information about changes can be obtained with `git diff`. When the `--name-status` option is used within the command, it returns a list of changed files in the form of a path or paths (if the file was renamed) with a flag that informs about the change — the `M` flag if the file was modified, `A` flag if the file was added, and so on. The full command, when switched to the *changes branch*, is:

```
git diff --name-status COMMON_COMMIT..HEAD
```

Now, we have the list of all files that were changed in the *changes branch.*

Obtaining Contents of the Changed Files

When we know which files have been changed and how they have been changed, we need to obtain the content of each file before the changes and after the changes. That can be achieved with `git show` commands:

```
git show COMMON_COMMIT:./OLD_FILE_PATH # the content before the changes
git show HEAD:./NEW_FILE_PATH # the content after the changes
```

The commands operate on the *changes branch* so it is not necessary to use a specific commit hash for the last commit from the *changes branch* — we can use the HEAD reference. These commands vary on a type of the change of the file. If the file path did not change, then `OLD_FILE_PATH` and `NEW_FILE_PATH` are the same. If the file was added, then the first command is not relevant because the file did not exist before. If the file was removed, then the second command is not relevant because the file does not exist anymore.

Creating File Records

The information about the changed files we found must be wrapped to records. Then we can output these records from this phase and use them in the next phases.

For each changed file we create a single record. The records represent file records with all information that is needed for the analysis. Each record contains the file path (if the file was renamed, then the new file path is used, otherwise the file path is the same for the file before and after changes), the modification flag, the content of the file before the changes, and the content of the file after the changes. If the file was added, then we put the empty string to the content of the file before changes. If the file was removed, then we put the empty string to the content of the file after changes.

Now, we have everything for the analysis. The output from this phase are the created file records. They are returned to the main control for the next phase.

5.3 Analysis and Computing Affected Parts

The input to this part is a file record structure with changes. It starts at the `ctf/diff_analysis.py` file where each analysis class decides if it can analyze the file — at most one class can be chosen.

The selection of the analysis (if the file will be analyzed as Bash, as Ansible, and so on) is implemented by a plug-in system that allows extension by other analyzers as long as they conform to the interface. The selection of analyzer is done in four steps:

1. Import all analysis classes from modules located in the `ctf/analysis` folder,
2. Each of analysis classes with the `can_analyse` method runs the method on a file path of the changed file (the information is contained in the file record),
3. If the method returns `True`, then the class knows it can perform the analysis,
4. The class is selected and will perform the analysis.

Thus, the newly added analysis class does not need to be added to `ctf/diff_analysis.py`. The class only needs to be defined in the `ctf/analysis` folder and to have implemented the `can_analyse` method.

If no analyzer can perform analysis of a file, then no class returns `True`, no analysis will be performed, and the file is skipped.

All analysis classes inherit from the `AbstractAnalysis` class. Besides few methods useful for analysis (to check if the file was added and to check if the file was deleted), there are static abstract `can_analyse` method and abstract `process_analysis` method. Abstract methods enforce the implementation of the methods in subclasses. The static method, `can_analyse`, allows us to use the method without an instance of the analysis class and hence we do not need instances of analysis classes when selecting the analyzer.

Each analysis records changes, logs, and affected parts to an instance of the `DiffStruct` class, which is the output from the analysis. A detailed description of the `DiffStruct` class is in Section 5.4. The rest of this section presents implementation details of analyses of different file types.

Profile Analysis

A profile file is detected if its file path has the `.profile` suffix. The initialization of the class for profile analysis starts with parsing the name of the profile and of the product using the profile file path. Each profile file has the same file path format. The format relative to a repository path is `PRODUCT_NAME/profiles/PROFILE_NAME.profile`.

The analysis starts with checks if the profile file was added or removed. Three cases may occur:

- Profile file was added but there is no list of rules — the new profile does not have the `selections` section with rules. We only record the logging information to the instance of `DiffStruct` but we do not record the change that would lead to test selection in the next phase.
- Profile file with rules was added — the new profile has rules in the rule list. We record the logging information and the change to `DiffStruct`, which will lead to test selection in the next phase.
- Profile file was removed — we record the logging information but we do not record the change, thus this will not lead to test selection.

When the profile file is modified, the previous and the new content of the file are parsed to dictionaries with the `PyYAML`[4] library. These dictionaries are then compared using the `DeepDiff` class from the `DeepDif`[5] library. The `DeepDiff` class compares the dictionaries

[4]https://pyyaml.org/
[5]https://github.com/seperman/deepdiff

and for each change saves a change type and how it changed. Each change type is handled differently in the profile analysis:

- **dictionary_item_added** or **dictionary_item_removed** — a new key was added to the profile or a key was removed from it. If the key is `title`, `description`, or `documentation_complete`, no change is recorded to the instance of `DiffStruct`. If anything else changes, then the change is recorded to the instance.
- **type_changes** — type of a value changed. This change is not typical thus, in this case, the change is always recorded to `DiffStruct` and it will lead to the test selection.
- **values_changed** — some value was changed. If the value is from the `selections` key, then the change is recorded to `DiffStruct` for following test selection. Otherwise, no change is recorded because the change does not affect the profile functionality.
- **iterable_item_added** or **iterable_item_removed** — an iterable value was added to the profile or removed from it. The only key that contains iterable values is `selections` with rules. We obtain names of rules that were added or removed, add them to the instance of `DiffStruct` as logging information for the user, and we record the change to the instance thus the following phase will select the profile test.

When the change is being recorded to the `DiffStruct` instance, we already know that the profile file changed. Thus, as a part of computing the affected parts, we must find all profiles that extend the changed one. All relevant profiles are found in the same folder as the changed one. We open them and try to find the `extends` key. If a profile has the key and the key's value is the name of the changed profile then we record the affected profile name to the `DiffStruct` instance, thus the test for the profile is also selected in the next phase. The process is repeated until all profiles that directly or indirectly extend the changed profile are collected.

After analyzing all the changes, the analysis ends and the `DiffStruct` instance with information about changes and logging information are returned to the main control.

Ansible Remediation Analysis

An Ansible remediation file is recognized with a regular expression that matches a file path if it has the `.yml` suffix and if it is in the `ansible` folder:

```
.+/ansible/\w+\.yml$
```

The name of the rule that the remediation corresponds to is detected with a similar regular expression because the rule name is the name of the parent folder of the `ansible` folder. The regular expression that captures the rule name with a capturing group is:

```
.+/((?:\w|-)+)/ansible/\w+\.yml$
```

The analysis starts with a detection if the Ansible remediation is newly added or removed. If it is added, we record it to the instance of `DiffStruct` so the test for the rule is selected in the next phase and the analysis ends. If the file is removed, we record the logging information to inform the user but we do not record any change so no test will be selected and the analysis ends.

When the remediation is modified, first of all, the analysis detects if the remediation used a Jinja macro in the previous state and if it uses in the new state. The fact is detected with a regular expressions that remove the Jinja macro usage (everything from {{{ to }}}), empty lines, and comments. If no lines are left, then the file uses a Jinja macro. If the file used a macro in the previous state and the new state it does not use any macro or vice versa, then it is an important change that is recorded to the DiffStruct instance, the analysis ends, and next phase will select the test scenario for it.

If the file uses a Jinja macro both in the new and the old state or it does not use the macro in any state, then we use the old and the new file contents as inputs to the DeepDiff class to analyze the changes. Both inputs to the class are strings and in such case, the output from the analysis is in a unified format. All unnecessary lines (unified format header and empty lines) for the analysis are deleted from it and only changed lines are kept — lines starting with + or -. The changed lines are then analyzed one by one according to the description of Ansible remediation file type in Section 4.3:

- changed line starts with # — it is a comment. If the comment starts with platform, reboot, strategy, complexity, or disruption keyword, then it is a change in metadata. The change is recorded in DiffStruct and in the next phase the test for the product will be selected. Otherwise, no test is selected and no change recorded,
- changed line is empty — no change is recorded to DiffStruct,
- other cases — the remediation was changed. If the remediation uses a Jinja macro, then we record the change and it will always lead to the rule test selection in the next phase. If the remediation does not use any Jinja macro, then we first check if the changed line does not start with the name: string. If it starts with the string, then we do not record anything and analysis ends. Otherwise, we record the change so that the rule test is selected in the next phase.

During recording of the changes to the DiffStruct instance, we first check if the rule is used in any profile. That is achieved by finding all profiles folders in ComplianceAsCode and searching all files from the folders. If any profile has the rule in the rule list, then we choose the profile's product (because datastreams are created for products and datastreams contain all product's profiles) and record the rule and the product to DiffStruct so that the test selection phase knows the rule is used in some profile and thus it can be tested. If the rule is not contained in any profile, then even though we found a change in the rule's remediation we do not record the change so that the test selection phase does not select any test for it because the change cannot be tested.

After analyzing and recording changes to the instance of DiffStruct, the Ansible analysis ends and the instance is returned to the main control.

Bash Remediation Analysis

A Bash remediation file has the .sh suffix and is located in the bash folder. The regular expression that recognizes the remediation is:

```
.*/bash/\w+\.sh$
```

Most of the Bash remediation analysis is similar to the Ansible remediation analysis — first we detect if the file is added or removed, then if the remediation used a Jinja macro in the previous state and in the new state it does not or vice versa. Also if the remediation uses a Jinja macro in the old and in the new state, then the steps are the same as for the Ansible remediation that uses a Jinja macro. Recording of the changes to the DeepDiff instance and computing affected parts (if the rule is contained in any profile) is also the same. Thus, for more information see the Ansible remediation implementation part.

The difference from the Ansible remediation analysis is when the Bash remediation does not use a Jinja macro — the remediation is written in Bash. In such case, we compare outputs of a lexical analysis of two versions of the file. For the lexical analysis of the code the `shlex`[6] library is used. The algorithm for comparison of Bash remediations is shown at Algorithm 2.

```
1  token_before = getNextToken(file_before);
2  token_after = getNextToken(file_after);
3  while $token\_before$ not empty $\wedge$ $token\_after$ not empty do
4      if $token\_before \neq token\_after$ then
5          break;
6      end
7      token_before = getNextToken(file_before);
8      token_after = getNextToken(file_after);
9  end
10 if $token\_before \neq token\_after$ then
11     code has changed;
12 end
```

Algorithm 2: Comparison of Bash source codes

If the analysis finds a change in the remediation, then it is recorded to the `DiffStruct` instance so the test for the rule is selected in the next phase. The instance is returned to the main control.

Python Analysis

A Python file is recognized if it has the `.py` suffix. If a change in a Python file is found during the analysis, then we record the change to the instance of `DiffStruct` always as the functionality change.

The analysis starts with checking if the file was added or removed. In both cases, we record it to the `DiffStruct` instance and the analysis ends.

If the file is modified, then the old and the new Python source codes are parsed to Abstract Syntax Trees (ASTs) with the `ast`[7] library and these two trees are compared. Nodes of a Python AST are described in the Green Tree Snakes documentation[8].

Algorithm 3 shows the comparison of two ASTs. The comparison returns a boolean value whether the trees are the same. The comparison starts with comparing root nodes of both trees and recursively dives deeper to child nodes until it hits leaf nodes. First, the algorithm checks if both nodes are of the same type. If so, and the nodes are instances of the `AST` class, then items of the nodes are further analyzed. For each item, the algorithm checks if the item's key is a line number or an offset of a token. These items can be changed and the AST remains the same thus they are not compared. The context of the node is also ignored because if a context of the node changes, then the type changes and the first condition with the type check fails. If the whole content of the first check passes, then

[6]https://docs.python.org/3/library/shlex.html
[7]https://docs.python.org/3/library/ast.html
[8]https://greentreesnakes.readthedocs.io/en/latest/nodes.html

```
Input: $TreesToCompare$
Output: $TreesEquality$
Function ASTEquality($n1, n2$):
    if $type(n1) \neq type(n2)$ then
        return False;
    end
    if type(n1) == AST class then
        foreach $key, v1 \in n1.items$ do
            if key == „number of line“ ∨ „offset of token“ ∨ „context“ then
                continue;
            end
            v2 = n2[key];
            if not ASTEquality(v1, v2) then
                return False;
            end
        end
        return True;
    end
    if type(n1) == list then
        foreach $(v1, v2) \in zip(n1.items, n2.items)$ do
            if not ASTEquality(v1, v2) then
                return False;
            end
        end
    else
        return n1 == n2;
    end
End Function
```

Algorithm 3: Finding Abstract Syntax Trees equality

`True` is returned. If the nodes are not the instances of the `AST` class but they are lists, then `foreach` loop creates pairs of *n*th item from the first node with *n*th item from the second node and uses them as the input to the function. If none of the previous conditions were fulfilled, then the equality of nodes if returned.

If the algorithm found a change in the AST, then the functionality change is recorded to the `DiffStruct` instance, and the test selection phase selects the functionality test that runs `ctest` from ComplianceAsCode. Hence, we do not need to check which specific Python file changed and which other Python files are affected because we ensure the functionality is correct with the test.

OVAL Analysis

An OVAL file path has the `.xml` suffix and it can be found in the `oval` folder that is in a rule folder. The regular expression that checks if a file is a rule check is:

```
.*/oval/\w+\.xml$
```

OVAL files themselves are not valid XML files because:

- they can be fully written in a Jinja macro, same as Ansible and Bash remediations,
- they can use Jinja macro for conditions — majority of such files is in XML but a few parts are conditioned with a Jinja macro, see example at Listing 5.1 from the `ensure_redhat_gpgkey_installed` rule's check,
- they do not have the XML header and the XML footer.

All the cases are resolved during the ComplianceAsCode build process — macros are rendered, the XML header is prepended, and the XML footer is appended to the file content.

```
{{%- if product == "rhel6" %}}
  <extend_definition comment="SL6 installed"
  definition_ref="installed_OS_is_sl6" />
{{%- endif %}}
{{%- if product == "rhel7" %}}
  <extend_definition comment="SL7 installed"
  definition_ref="installed_OS_is_sl7" />
{{%- endif %}}
```

Listing 5.1: Partially templated OVAL check

First, we check if the file is newly added or removed. If it is added, we record it to `DiffStruct` for the test selection and the analysis ends. If it is removed, the analysis ends. Then, we check if the file is fully written in a Jinja macro, same as we do for Bash and Ansible remediations with regular expressions. Also, if the check did not use a Jinja macro and now it uses a macro or vice versa, then we record the fact to the `DiffStruct` instance and the analysis ends.

If the check is not fully written in a Jinja macro, then the XML format must be analyzed. Since the content of the OVAL file is not a valid XML, we must create a valid XML from it. The analysis does partially the job of the build system:

- completely removes lines with Jinja macros — from Listing 5.1, it creates Listing 5.2, thus the check may not be proper (conditions may be contradictory) but the XML is valid and no part of the check is overlooked,
- prepends the XML header,
- appends the XML footer.

```
<extend_definition comment="SL6 installed"
definition_ref="installed_OS_is_sl6" />
<extend_definition comment="SL7 installed"
definition_ref="installed_OS_is_sl7" />
```

Listing 5.2: OVAL check after removing Jinja macros

When the check is a valid XML, then the old and the new XMLs are compared with the `diff_texts` function from the `xmldiff`[9] library. The function compares XMLs and returns changes in a form of an object for each change. Each change class is handled differently:

[9]https://xmldiff.readthedocs.io/en/stable/index.html

- **InsertNode** or **DeleteNode** — a new node was added to the check or removed from it. If the changed node is from the `metadata` node (a node that contains name, platforms, and description of the check), then we do not record the change to the `DiffStruct` instance. Otherwise, we record the change and a test will be selected.
- **MoveNode** — a node was moved within the check. If the node was moved within a single node, for example, a node was moved within the `metadata` node, then no change is recorded. Otherwise, it is recorded and the next phase will select tests.
- **InsertAttrib** — a new attribute was added to a node. For this change, we do not record any change, because a new attribute does not affect an already working check.
- **DeleteAttrib**, **RenameAttrib**, or **UpdateAttrib** — an attribute was removed, renamed, or changed. If the attribute change is the `comment` or the `version` attribute, no change is recorded. In other cases, we record the change for the test selection.
- **UpdateTextIn** — a text within a node was updated. If the text is from the `title` node, the `description` node, or the `platform` node, then we do not record the change. Otherwise, we record it for test selection.
- **UpdateTextAfter** — a text was added after a node. This change is not typical, because texts in OVAL checks are usually within a node, thus, in this case, it is always recorded and it leads to the test selection,
- **InsertComment** — a comment was added. This change is ignored.

When a change is being recorded to the `DiffStruct` instance, first we check which other rules use a check from the changed check:

1. Get all values from `id` attributes with XPath[10] `.//*[@id]`,
2. Search through all OVAL files from ComplianceAsCode and in each of them, try to find if it contains the `definition_ref` attribute,
3. Check the `definition_ref` value, if it is the same as some of `id` attribute's values from the changed OVAL,
4. Save all rules that reference any part from the changed rule because they are affected by the change.

Then we check if the changed rule and the affected rules are part of any profile, same as we do for remediations to know that we can test them.

Jinja Analysis

File with Jinja macros is recognized if it has the `.jinja` suffix. The analysis starts with checking if the file is newly added or removed. If the file with macros is newly added, no record is added to the `DiffStruct` instance because macros in the file are new and not used anywhere. In case the change also adds files that use the new macros, then these files are analyzed separately. If the file was removed, then we record to `DiffStruct` a change in functionality thus the functionality test will be selected in the next phase. If any file uses

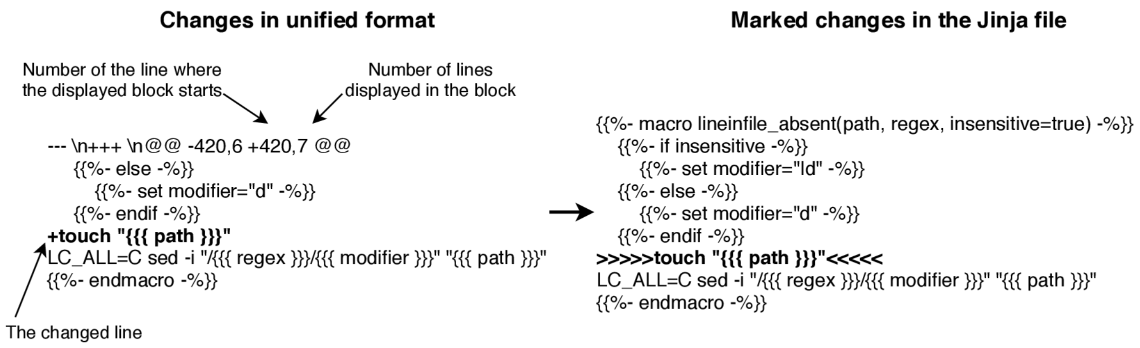
[10]https://www.w3.org/TR/xpath-31/

a macro that was removed, the test reveals the fact because it renders macros, and if any macro usage misses the definition of the macro, the test fails.

When a Jinja macro is changed, we need to:

1. Find the name of the macro,
2. Find which other macros use the changed one,
3. Find usages of all affected macros in rule files (remediations, checks, and descriptions),
4. Find all usages of all affected macros in template files that generate rule files during the ComplianceAsCode build.

When finding the name of the changed macro, we have chosen the approach that based on differences in the unified format marks changed lines in the file with macros, splits macros from the file to individual macros, and then finds which macros have those marked lines. However, more approaches are available, for example, to load the Jinja macro to bytecode and compare bytecodes of each macro before and after changes.


Changes in unified format
Marked changes in the Jinja file
Number of the line where the displayed block starts
Number of lines displayed in the block
The changed line

```
--- \n+++ \n@@ -420,6 +420,7 @@
    {{%- else -%}}
        {{%- set modifier="d" -%}}
    {{%- endif -%}}
+touch "{{{ path }}}"
LC_ALL=C sed -i "/{{{ regex }}}/{{{ modifier }}}" "{{{ path }}}"
{{%- endmacro -%}}
```

```
{{%- macro lineinfile_absent(path, regex, insensitive=true) -%}}
    {{%- if insensitive -%}}
        {{%- set modifier="Id" -%}}
    {{%- else -%}}
        {{%- set modifier="d" -%}}
    {{%- endif -%}}
>>>>>touch "{{{ path }}}"<<<<<
LC_ALL=C sed -i "/{{{ regex }}}/{{{ modifier }}}" "{{{ path }}}"
{{%- endmacro -%}}
```

Figure 5.3: Marking change in Jinja file with unified format input

Our approach uses the `DeepDiff` class from the `DeepDiff` library to compare the file with Jinja macros in the state before changes with the state after changes. It outputs differences in the unified format that we use for finding the macro name:

1. Parse information about changes from the unified format — the number of the line where the unified format block starts, number of lines displayed in the unified format, and specific lines that are changed. Figure 5.3 shows the information in a unified format output.
2. With the information, mark the changed lines both in the content before the changes and in the content after the changes. We have the contents in the file record from the input to the analysis. We need to try to mark the changed lines in both contents because if the change only removed lines then we will not find the lines in the content after changes (the lines do not exist in the content after the changes). A similar case is added lines with the content before changes (the added lines do not exist in the content before the changes). We need to have marked lines in at least one of the contents for the next steps. See an example of marking changes in Figure 5.3. As marker was have chosen the `>>>>>` prefix and the `<<<<<` suffix but it could be an arbitrary marker that is not used in any Jinja macro's definitions.

3. Split the file contents to individual Jinja macros. That is achieved with a regular expression that captures each Jinja macro and saves it to separate string:

   ```
   {{%(?:\s|-|\n|>|<)+?macro(?:\s|\n|<|>)(?:.|\n)+?
   endmacro(?:\s|-|\n|>|<)+?%}}
   ```

   The regular expression is based on the specific knowledge of ComplianceAsCode Jinja macros — they start with `{{% macro` or `{{%- macro` and end with `endmacro %}}` or `endmacro -%}}`.

4. In each Jinja macro we try to find our markers, in our case `>>>>>` and `<<<<<`. We need to find at least one of the markers because, for example, if the first line of the macro changed, then we did not capture the first marker.

5. Macros that contain a marker are the changed ones that we are looking for. From them, we can capture their names with the following regular expression:

   ```
   (?:\s|-)macro(?:\s|\n)+([^(]+)
   ```

   The regular expression uses the capturing group for the name capturing. Now, we know the names of the macros that were changed.

When we know which macros are the changed ones, we save them to the instance of the `JinjaMacroChange` class. The class constructor uses the macro name for searching macro usages in the ComplianceAsCode project. When the constructor finds that the macro is used in a rule file or in a template, it just saves the rule name or the template name to the attribute. However, if the constructor finds that the macro is used in another Jinja macro, then it captures the name of the macro that uses the changed macro in a similar way as we did for changed macros, creates a new instance of `JinjaMacroChange` for it, and saves the reference to the newly created instance. With that, we achieve a next invocation of the `JinjaMacroChange` constructor, thus searching the usages of the macro, and repeating this process until there are no more macros that are affected with the changed macro. Now, we have a hierarchy of changed macros, where we have references to the lowest-level macros and they point to higher-level macros.

All affected macros already found their usage in rule files and in template files during the initialization of their instances of `JinjaMacroChange`. Now, we need to analyze all those usages.

If the usage is in a rule file (remediation, check, or description):

1. Render the Jinja macro in the file. The rendering process is taken from the `ssg` module that is a part of the ComplianceAsCode project. The module is dynamically imported from the cloned ComplianceAsCode repository. We render the file two times — first time with the macro before the change and then with the macro after the change,

2. The rendered file is used to create the file record in the same way as file records are created from the `GitDiffWrapper` class. We have the rendered file content before and after the change and as the flag we use `M` (modified),

3. The created file record is added to the `DiffStruct` instance to affected files and that causes the main control will analyze the file record as other changed files.

If the usage is in a template file:

1. Find which files are generated from the template file,
2. Generate files from the template,
3. Create file records from the generated files and add them to the `DiffStruct` instance.

The first step, finding which files are generated, starts with parsing the template file name. The name is in the `template_TYPE_TEMPLATE-NAME` format, where `TYPE` is a type of files that are generated from the template (`BASH` for the Bash remediation, `OVAL` for OVAL check, and so on) and `TEMPLATE-NAME` is the template identifier for referencing in rule descriptions. The `TEMPLATE-NAME` part is used for finding rules that use the template. With it, we search all rule descriptions (`rule.yml` files). Rule descriptions that contain the identifier, are the rules that will have generated files. The generated files have the `RULE_NAME.SUFFIX` format, where `RULE_NAME` is found from the rule description's file path (the parent folder of the rule description file is the rule name). The `SUFFIX` part is obtained from the template's `TYPE` (the `.sh` suffix for the `BASH` type, etc.). Now, we know the exact names of generated files from the template.

The generated files do not exist in the ComplianceAsCode project until the project is built. Therefore building is needed. The project is built with the `GitDiffWrapper` class from `diff.py` that takes care of getting changes from Git. The class is a singleton and has all information about the repository, the previous state of the project, the new state of the project, and the `build_project` method that builds only the templated content. When the method is called, it deletes build folders from previous builds, creates empty folders, runs `cmake`, and then builds the templated content. Actually, the project is built two times to two folders — first time for the state of the project before the change and other time for the state after the change. Now, ComplianceAsCode has generated files from templates.

In folders with built contents are now files with the `RULE_NAME.SUFFIX` name format. These files contain complete rule contents and they can be used to create file records. One build folder has the content of the file before changes and the second folder has the content after changes. As a file path for the file record is used the path to the folder with the build after changes and the flag is set to `M` as modified. The created file records are added to `DiffStruct` to affected files and the main control will get them and handle them as other changed files.

When all macros are analyzed, the important output from this analysis are affected files in the `DiffStruct` instance. Those affected files analyses lead to test selections.

5.4 Computing a Set of Tests to Run

During the analysis, information which tests will be selected and logging information are saved to the instance of the `DiffStruct` class. Each analysis result is saved to a separate instance of the class. The class is defined in the `ctf/DiffStruct.py` file.

The `DiffStruct` class has an attribute for each affected part — rules, profiles, products, changed functionality, and affected files. The type and the purpose of attributes differ depending on a data required for selecting a test:

- `file_type` — contains information about the type of the changed file, for example, Bash, Python and so on,

- `changed_rules` — a dictionary with product names as keys. Each key has a list of changed rules for the product. For testing a rule, information which product includes the rule must be known because if no product contains the rule, we cannot test it,

- `changed_profiles` — a dictionary with product names as keys. Each key has a list of changed profiles for the product. Similar to `changed_rules`, a product that includes the profile must be known for the testing,

- `changed_products` — set of changed products,

- `functionality_changes` — a boolean variable, that indicates that the functionality has changed and the fact that it must be tested with the functional test,

- `affected_files` — a list of file records that contains the affected files. This attribute is used in the main control for analyzing affected files found during analyses.

These attributes are not filled directly in the analysis. The analysis uses methods from the `DiffStruct` class and the methods fills the attributes. The methods ensure the attributes are properly filled — in most of the cases, during the analysis, information about which product includes the changed rule or the changed profile is unknown. The input to these methods is the name of a changed rule, of a profile, or of a product and if the attribute format requires the product name the method finds the product and then fills the attribute. Other attributes that do not require a product name they serve as a wrapper for manipulating with the attributes.

An optional input to the methods that fill attributes is a string where can be justified why the test was selected. The string is printed to the standard output at the end of the implemented system run.

Individual steps of the computing a set of tests are:

1. `DiffStruct` instances are used as the input to the instance of `ContentTests` class in the main control,

2. The `ContentTests` class goes through all attributes from `DiffStruct` and transfers them to `ContentTests` attributes in a form of instances of classes that represent specific tests. Each of the classes that represent specific tests has the `get_tests` method that knows how to get test commands for the test the class presents,

3. The filled `ContentTests` instance is used as the input to the `get_labels` function that gets all test commands. The function prepares the Jinja environment for loading the file with predefined test commands. The function goes through affected products (products their part must be tested — rule, profile, or the product itself) from the `ContentTests` instance and for each individual product the function renders the file with predefined test commands. It is because the file contains a placeholder that needs to be replaced with the product name so the file content has real commands that are ready to use. From the rendered file content, we find all tests that use the current product. Test commands of those test are get with the `get_tests` method. After selecting all relevant test commands, the function outputs the list of the test commands. The `get_labels` function is shown in Algorithm 4.

The list of test commands is returned to the main control.

```
Input: ContentTests instance
Output: List of test commands
Function get_labels(input):
    tests = list();
    foreach product ∈ input.affected_products do
        all_commands = file_all_commands.render(placeholder.replace(product));
        foreach test ∈ input.tests do
            if test.product == product then
                command = test.get_tests(all_commands);
                tests.extend(command);
            end
        end
    end
    return tests;
```

Algorithm 4: Getting test commands from instance of test class

5.5 Logging and Printing Results

All results are printed at one place after performing all analyses and computing the list of test commands to run. The `DiffLogging` class defined in the `ctf/DiffLogging.py` file handles the printing. The instance of the class is filled similar way as `ContentTests` — `DiffStruct` instances are used as the input to `DiffLogging` and `DiffLogging` goes through all attributes from `DiffStruct` and transfers them to `DiffLogging` attributes.

The `DiffStruct` class has attributes and methods for saving logging information about findings. The attributes are similar to attributes where information for test selection is saved. Thus there are attributes for logging information of rules, of profiles, of products, of macros, and of functionality, so the `DiffLogging` class can use them for creation of the structured log.

At the end of the implemented system run, the `print_all_logs` method from the `DiffLogging` instance filled with logging strings is called with the list of tests as arguments. It prints structured information about findings and then the list of test commands.

Chapter 6

Testing and Deployment

When adding new features or changing an already implemented behavior to a project, it is good to have tests that will ensure that the change did not break the expected functionality. One of test types that tests an application against functional requirements is called the functional testing. It is a testing that is based on providing input to the algorithm implementation, and comparing the actual output to a reference one. Section 6.1, describes how Bats: Bash Automated Testing System is used for the functional testing of our implementation.

Large projects with a long life cycle undergo major changes — new features are added, issues are fixed, unused code is removed, files are moved to different folders, and so on. A long exposure to short-sighted code modifications can impart its readability and quality. For better readability of the code, there are guides, such as PEP-8[1] for Python, that standardize a coding style (naming convention, code lay-out, comments, etc.). When an implementation meets the coding style, the readability is better even for people that are not familiar with the project. However, the coding style does not prevent anti-patterns or security risks. For that, there are automatic code analyzers. They can be integrated, for example, to the GitHub repository, and then they automatically analyze changes in the repository. One of code analyzers, DeepSource[2], is used for our implementation. Section 6.2 describes the analyzer, integration of the analyzer to the GitHub repository, and findings from the analysis in our implementation.

To ease users their work, we have automated running of our implementation to analyze new contributions in the ComplianceAsCode repository. The ComplianceAsCode project is hosted on GitHub, and all contributions have form of GitHub Pull requests. The Security Compliance team uses Jenkins[3] for automatic testing of projects in the whole ecosystem (ComplianceAsCode, OpenSCAP, etc.). Jenkins is an open-source automation server that can automate tasks related to building, testing, or deployment a software. Section 6.3 describes how we have integrated our implementation to Jenkins.

6.1 Testing Implementation with Bats

Bats: Bash Automated Testing System[4] is a testing framework that provides a simple way to test UNIX programs based on the comparison of actual and expected output to a given

[1]https://www.python.org/dev/peps/pep-0008/
[2]https://deepsource.io/
[3]https://www.jenkins.io/
[4]https://github.com/sstephenson/bats

input. Bats comes as an executable script, and it features the following commands to use for test scenarios [11]:

- `run` — invokes its arguments as a command, saves the status code, and returns the status code,
- `load` — command for sourcing Bash source files,
- `skip` — command for skipping tests,
- `setup` and `teardown` — pre-test (`setup`) and post-test (`teardown`) hooks that run before and after each test scenario.

We have implemented few test scenarios using Bats for each file type analyzer. All test scenarios are located in the `tests` folder. At the beginning of each test run, the test run clones the ComplianceAsCode repository that is used for creating changes and used as the input to our implemented system. The user can also export the `repo_dir` variable with the path to the local ComplianceAsCode repository. The variable saves time because then test scenarios do not have to clone the repository. Then, a test run creates the `test_branch` branch from the master branch for testing purposes. After that, the testing environment is prepared and individual test scenarios are run. Each test scenarios has a similar process that varies in the part which file is changed and what results are expected:

1. `setup` is called by Bats — it creates an empty file for saving the output,
2. a file that will be changed is specified,
3. the file is changed — most of the time using the `sed` command. For example, a specific line is changed, a comment is added, or an empty line is removed,
4. the changed file is added and committed to the `test_branch` branch,
5. the implemented system is run:

   ```
   content_test_filtering.py branch --local --repository "$repo_dir" \
   test_branch &> "$tmp_file"
   ```

   the `--local` option forces the implementation to find the changed branch at the local repository and not to pull the changes from the remote repository. The `test_branch` branch is used for the comparison against the master branch (the default comparison) and the standard output with the standard error are redirected to the empty file,
6. check if the return code from the system run is `0`,
7. check if the output from the implementation has the expected string — this is done with regular expressions that search, for example, for a specific test scenario or for an empty output,
8. `teardown` is called by Bats — it resets the state of the `test_branch` branch to the state before changes so the next test scenario is not affected by the changes.

The `setup` and `teardown` functions are defined in the `tests/test_utils.bash` file and the file is loaded to test scenarios with `load test_utils`.

6.2 Automatic Code Analysis of the Implementation

For code analysis of our implementation, we used the DeepSource[5] tool. It is free to use for open-source projects.

When the DeepSource analyzer is integrated to a remote repository, it analyzes commits that are pushed to the remote repository. The user can see the issues and metrics of his repository at the DeepSource page after logging in with the GitHub account.

For our implemented system, it found several issues. Most of them were in the style category — too many blank lines between classes or methods, too long lines, wrong indentation in continuation line, and so on. These issues were easy to fix. Then, other easy fixes were in the anti-pattern category that revealed few unused variables and unused imports.

The security issues category revealed issues in usage of insecure libraries, commands, or options, and the bug risks category revealed, for example, usage of protected members. Not all of those issues were fixed because some of them are about a problem we could not avoid — for example, using the `subprocess` library and its `run` function for running the Shell command is generally considered as insecure. However, in our implementation the usage was intended because we use the ComplianceAsCode build that works with `cmake` tool and we run the `cmake` commands with the `run` function. Another example of an issue that was not fixed is usage of a protected member (Python does not have the concept of protected members but the naming convention is that a variable name with the underscore prefix indicates the variable is a protected member) caused by using the `_get_jinja_environment` function that we used for creating same Jinja environment as ComplianceAsCode does (that could be avoided by copy-pasting the function to our implementation but it could cause problems with the code redundancy).

Overall, we did not fix all issues and suggestions from DeepSource because those are general suggestions and each project is unique. The developer must decide if the fix is worth it or if it causes more problems than benefits. However, a lot of issues and suggestions were fixed and thanks to that the overall code quality has been improved.

6.3 Integration to ComplianceAsCode Jenkins

To automate usage of the implemented system on newly created Pull requests for the ComplianceAsCode repository, we have deployed the implementation to the ComplianceAsCode Jenkins[6] where all contributions are tested.

The Red Hat Security Compliance team has their own machines where the Jenkins software is deployed and where tests are run. The infrastructure consists of the master machine and slave machines. The master machine runs Jenkins software[7], provides a web interface, starts jobs on slaves, and is the only machine that is accessible from outside networks. The slave machines are only accessible from the master machine and they test projects that are developed by the team (ComplianceAsCode, OpenSCAP, SCAP Workbench, etc.).

For the deployment we needed the following:

1. to prepare a slave machine for running our implementation,

2. to run our implementation and to collect the results,

[5]https://deepsource.io/
[6]https://jenkins.complianceascode.io/
[7]https://www.jenkins.io/

3. to post the results as an always-single Pull request comment,
4. to create a Jenkins job that hooks to the ComplianceAsCode project and make sure that the job is triggered whenever applicable (new Pull request or new push).

For the machine preparation, all requirements of our implementation were installed on all Fedora slave machines because Fedora is the newest system available on the slave machines. Currently, we did not prepare other available slave machines because it is not needed to load the balance to multiple machines since the run of the filtering is not time expensive (it takes from several seconds to few minutes). Implementation requirements were also added to the Ansible playbook that is used for the automatic configuration of slave machines. The playbook is available at the OpenSCAP Jenkins repository[8]. In this part, we cloned neither ComplianceAsCode nor our implementation — it will be done by the Jenkins job.

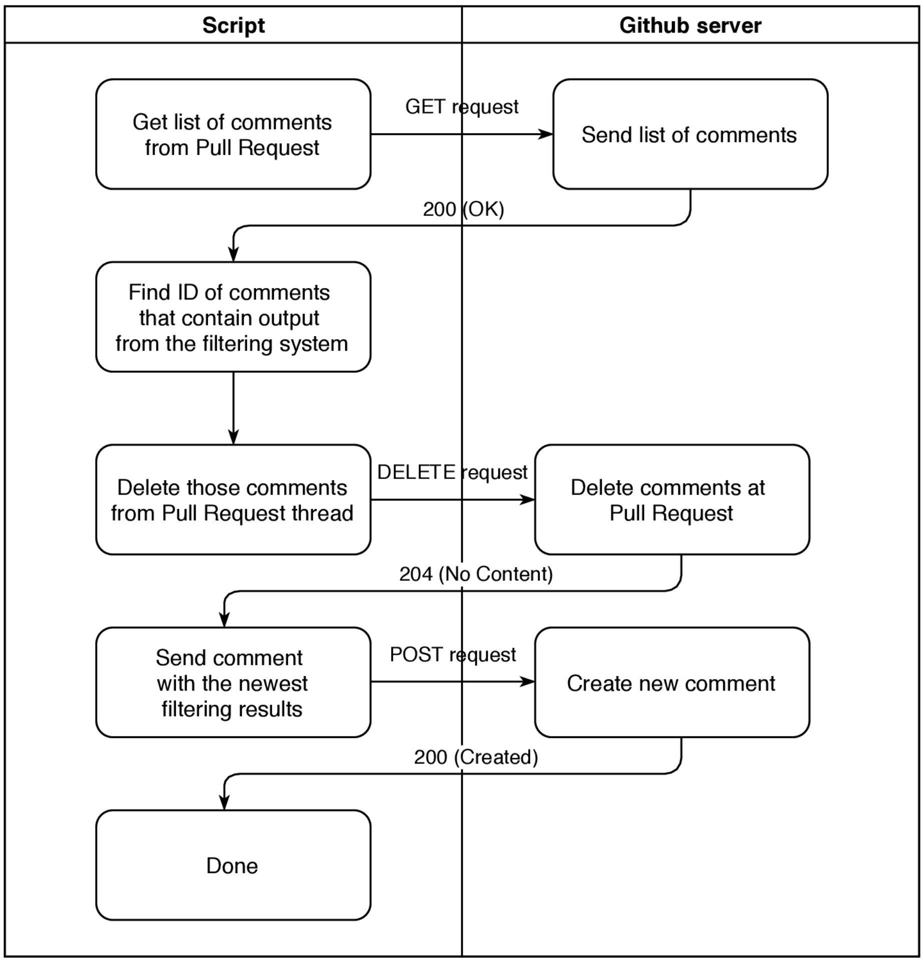

Script
Github server
Get list of comments from Pull Request
GET request
Send list of comments
200 (OK)
Find ID of comments that contain output from the filtering system
Delete those comments from Pull Request thread
DELETE request
Delete comments at Pull Request
204 (No Content)
Send comment with the newest filtering results
POST request
Create new comment
200 (Created)
Done

Figure 6.1: Process of script that creates comment in Pull request thread

The `utility_scripts/comment_pr.py` file contains a script that creates the Pull request comment. Figure 6.1 shows the script process — the script deletes comments that contain an already existing output from our implementation in the Pull request thread and

[8]https://github.com/OpenSCAP/jenkins

sends the latest output. Therefore, each Pull request has only a single comment with the newest output. Inputs to the script are:

- number of the Pull request that will be commented,
- a file with the content for the comment,
- a personal access token that serves as an authentication of the account from which the comment will be created.

In case our implementation cannot analyze anything, we do not post comments. That can happen if there is no change or only unsupported files are changed. We decided not to create unnecessary comments with information that our filtering system does not know how to analyze the changed file.

The last part is creating the Jenkins job that will use the prepared machines and the script that we have created. Figure 6.2 shows the process of the job that we have created. The job uses GitHub Pull request Builder plugin[9]. The job was configured according to the

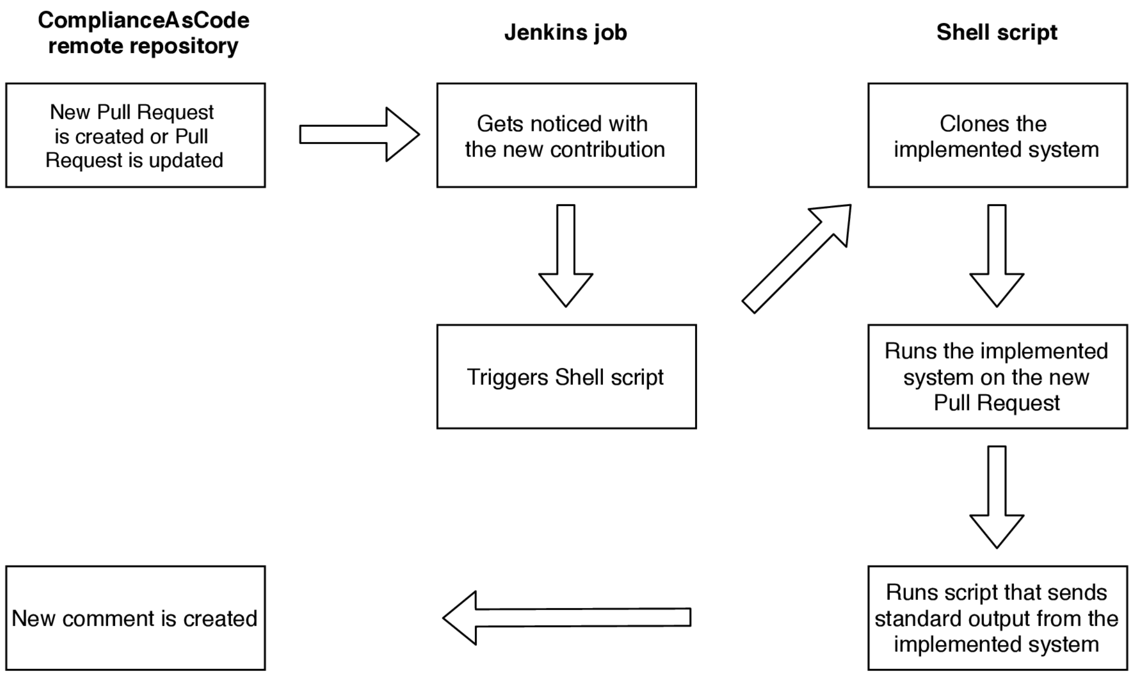

ComplianceAsCode remote repository
Jenkins job
Shell script
New Pull Request is created or Pull Request is updated
Gets noticed with the new contribution
Clones the implemented system
Triggers Shell script
Runs the implemented system on the new Pull Request
New comment is created
Runs script that sends standard output from the implemented system

Figure 6.2: Process of creating new Pull request comment

plugin documentation and as the repository for the job was selected the ComplianceAsCode repository thus the plugin is hooked to it and gets triggered when a new Pull request is created or when a change is pushed to an existing Pull Request in the ComplianceAsCode repository. We have restricted the job to run only on Fedora machines with specifying `fedora` label thus the master machine will start the job only on machines with the label. At the end, we have specified a list of administrators that can configure and re-run the job.

The created job runs on the ComplianceAsCode Jenkins, comments Pull requests that our implementation is able to analyze (it needs to have implemented analyzer for the changed file types), and helps us to improve the created filtering system (find cases that are problematic for the filtering).

[9]https://plugins.jenkins.io/ghprb/

Chapter 7

Testing the Filtering of Tests on the ComplianceAsCode History

We have decided to validate our system against historical data. We propose to select a sample of Pull requests and compare the time needed for execution of the complete test suite with the time needed for the execution of recommended tests.

```
all_time = sum(all test scenarios times);
selection_time = sum(selection of test scenarios times);
pr_list = get 500 Pull requests from the ComplianceAsCode repository;
foreach pr ∈ pr_list do
    filtered_tests = filter_tests(pr);
    filtered_time = calculate_duration(filtered_tests);
    complete_filtered = complete_filtered + filtered_time;
    complete_all = complete_all + all_time;
    complete_selection = complete_selection + selection_time;
end
```

Algorithm 5: Calculation of complete and average time spent on tests

For the testing, we have created a Python script `utility_scripts/experiments.py` that is shown at Algorithm 5. Our implemented system is not able to analyze all historical Pull requests because the file structure of the ComplianceAsCode project changed significantly a couple of times in the project's history. Recently, new types of files whose file analysis we do not support started to be used in ComplianceAsCode. Hence, we have selected 500 Pull requests over a longer time period. The Pull requests are not the latest ones but from a time period before the new file types started to be used in ComplianceAsCode (December 2019 and earlier).

The Pull requests numbers are obtained with GitHub API[1]. The testing algorithm has three different calculations of time spent on tests:

- **filtered tests** — calculates how much time would take running of tests that were selected by our tool. If the filtering could not analyze files in a Pull request because it does not have implemented analyzers for the changed files, the time is taken from the selection of tests described below,

[1]https://developer.github.com/v3/pulls/

- **all tests** — calculates how much time would take running all tests (all rule tests, all profiles tests, and the functional test),
- **selection of tests** — calculates how much time would take running a specific selection of tests (a subset of all tests) that is used the most often in the project.

The second and the third time are constants, while filtered tests can vary for every Pull request. The second case counts every profile test, every rule with tests, and the functionality check — currently, the entire ComplianceAsCode has 132 profiles, 248 rules with tests, and one functionality test. The number of profiles in ComplianceAsCode was found with the command:

```
find -name *.profile | wc -l
```

The number of rules that have at least one test scenario was found with the command:

```
find -name tests | wc -l
```

The third case — selection of tests — calculates the time that would be spent on running tests of 25 profiles and 94 rules. The profiles represent profiles that contributors of ComplianceAsCode are working on the most — currently, most of the profiles are from the Red Hat Enterprise Linux 7 and the Red Hat Enterprise Linux 8 products and other products are represented with zero to two profiles. The rules represent a selection of rules that have at least one test scenario and are a part of the most contributed profiles.

The time for each test type was calculated as the average duration of it on a virtual machine using Intel Skylake 6th Generation processor with 2.1 Ghz frequency and 4 GB memory. Table 7.1 shows times for each test. A rule test is a ComplianceAsCode testing in the *rule* mode where the rule is tested for each test scenario available for it. A profile test is a testing in the *profile* mode where a machine is scanned to assess compliance of the default configuration, then all failed rules from the profile are fixed, and then tested again if it complies to the profile. The functionality test builds all products in ComplianceAsCode and runs the `ctest` tests. The build time in the table refers to the product build. The implemented filtering system can select a build of a specific product if it needs the built datastream for testing, thus it does not have to build all products (as the functional test always does). The detailed description of testing modes is in Section 2.5.

Rule test [min]	**Profile test [min]**	**Functionality test [min]**	**Build [min]**
5.3	11.3	31.4	1.6

Table 7.1: Average duration of individual test types

Evaluation of the testing is in Section 7.1. During the testing, we have encountered few issues that appeared only for certain Pull requests. Section 7.2 describes those issues and their solutions.

7.1 Evaluation

Table 7.2 shows the average duration for each type of test selection. Running all tests from ComplianceAsCode takes too much time. Actually, all tests are never run because of the length and because many profiles and rules are obsolete and not maintained for a long time. The selection of tests is run when a new version of ComplianceAsCode is released but the selection is not run for each Pull request because 13.5 hours is still a lot. Currently, only the functional test is run automatically and developers manually choose other tests to run.

All tests [h]	Selection of tests [h]	Filtered tests [h]
47.1	13.5	5.0

Table 7.2: Average duration of testing each Pull request

From the 500 tested Pull requests, our tool was able successfully process 337 Pull Requests (64.7 % of all Pull requests) with an average time of 5 hours of testing per a Pull request. If the filtering system could not analyze any file from a Pull request because the system does not implement the analyzer for the type of the changed file, it assumed that the selection of tests needs to be executed. With more implemented analyzers, the resulting time could decrease substantially.

Even though the implemented system does not automatically run recommended tests, it serves as an assistant that automatically analyses each Pull request, recommends tests to run, and presents findings of analyzed changes in a human-readable form. Currently, it saves developers their time for test selection. Based on developers experience, the test selection for a Pull request takes approximately 3 minutes and approximately 5 Pull requests are opened each working day. With the implemented analyzers, our implementation analyzes a little over 60 % thus 3 of 5 Pull requests on average each day — it saves almost 10 minutes of developers time spent on test selection. However, the implemented system also calculates the set of files affected by the changes while developers do not usually do it. For example, if a Pull request changes a profile, then a reviewer tests the changed profile. However, he does not analyze dependencies (finding profiles that extend the profile) and, therefore, he does not run tests for the dependencies which can lead to missing bugs for which it is difficult to find what caused them.

The saved time for developers would greatly increase if the implementation could run recommended tests automatically. Running test scenarios manually and waiting for test results takes approximately 15 minutes on average. With the automatic running of recommended tests, developers could start reviewing Pull requests after they have all test results available. In the current state of the implementation (analyzed 3 of 5 Pull requests each day), it would save almost 1 hour of developers time every day.

7.2 Encountered Issues

This section describes issues that appeared during the testing of the implemented system. These issues are rare because they depend on several conditions to appear, for example, a merged Pull request with a commit that is also part of another Pull request (author pulled changes from the master branch to the branch connected to the Pull request).

Next commit after the merged branch is not a merge commit

In Section 5.2, we have described the method to obtain the common commit of the master branch and the branch we are analyzing that is already merged to the master branch. The method is based on obtaining descendants of the analyzed branch and it relies on the fact that the commit after the analyzed branch is the merge commit of the analyzed branch with the master branch. Actually, it appears that there are some specific cases when that is not true. Figure 7.1 shows such a case — we are analyzing the *analyzed branch* that contains a commit with changes. Before the *analyzed branch* was created, another contributor has created the *updated branch.* The *analyzed branch* got merged and the creator of the *updated branch* wants to update his branch and he pulls changes directly to the *updated branch* causing the commit from the *analyzed branch* to appear also in the *updated branch.*

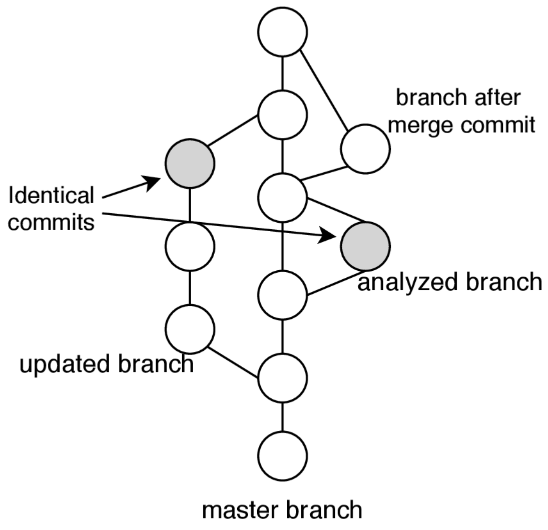


Figure 7.1: Problematic branch that puts a commit before a merge commit

In such a case, we observe that `git log --ancestry-path --reverse` does not show the merge commit that we are looking for as the first commit in the log. The `git log` command first shows commits (from the *branch after merge commit*) that were added after the merge commit of the *analyzed branch* but before the merge commit of the *updated branch.* After those commits, the command shows the merge commit we are looking for. However, none of the commits before the merge commit is a merge commit — the first merge commit (a commit that has two parents) in the log is the commit we want. Hence, a solution for those situations is not to refer only to a single commit from the log, but to *find the oldest merge commit in the log.*

Machine does not have installed ComplianceAsCode dependencies

The testing was performed on a clean machine where we have cloned ComplianceAsCode, cloned our implementation, and installed the required packages for our implementation.

The problem appeared when our implementation needed to run the ComplianceAsCode build. Without ComplianceAsCode dependencies, the build fails, and we do not have built files for the analysis. The solution is simple — install ComplianceAsCode dependencies — because without them, some of our analyzers do not work.

The ComplianceAsCode build system did not support used function

Because the ComplianceAsCode build system is still developed, we encountered the issue when running the ComplianceAsCode build for old Pull requests.

In our implementation, we run the ComplianceAsCode build when we need to build files for the analysis. We build with a command that builds only templated files, which saves time and does not create files that we do not need. However, the command was added to the project in September 2019. Hence, when our implementation tries to build files using the ComplianceAsCode state older than the date, it fails.

The building issue is one of the problems that prevent us from analyzing a longer history of the ComplianceAsCode project. We aim to analyze newly created contributions and not to analyze the project's history. However, because of these types of changes in the ComplianceAsCode project, we must keep in mind that in the future, changes that will require adjustment of our implemented filtering system can appear.

Chapter 8

Conclusion

In the master thesis, we have designed, implemented, integrated, and tested the system for automatic filtering of tests for the ComplianceAsCode project. ComplianceAsCode is a complex and large project whose testing may take several hours, even when running basic test scenarios. To automatically filter tests for the project, we have created a system that uses static analysis of different file types with the knowledge of ComplianceAsCode internals to produce a set of tests that must be run to verify a specific change.

The implemented system uses a version control system for obtaining changes from the ComplianceAsCode repository. The changes are used as the input to the static analysis that evaluates the importance of the changes. The results from the static analysis are used for computation of affected parts of ComplianceAsCode. With the analysis results and the knowledge of affected parts, we compute a set of tests that are recommended to run.

The implementation was deployed to a server and it automatically analyzes new contributions to the ComplianceAsCode repository. If the analysis of a contribution is successful and it found important changes, then the deployment is set to automatically create a Pull request comment containing a message about the found changes and the recommended tests to run. To evaluate benefits of our solution, we have run the implementation on past contributions and estimated that, currently, it saves developers approximately 10 minutes each day spent on test selections.

The filtering system is implemented in Python 3. The implementation is open-source and available on GitHub. In the implementation, we have focused on static analysis of different file types and on the possibility to easily extend the system by new analyzers for other file types. Currently, the system supports analyses for six file types from the ComplianceAsCode project — Profile files, Ansible remediations, Bash remediations, Python files, OVAL files, and files with Jinja macros. For the implemented system, we have created tests with Bats: Bash Automated Testing System, and to improve the quality of source code, we have integrated the DeepSource code analyzer to the GitHub repository of the project.

Even though the implemented system supports six different file types, there are still several file types that are missing. With more implemented analyzers, more contributions would be analyzed and more time could be saved for the developers.

The next steps as a follow-up for the master thesis can be implementing analyzers for other file types, or preparation of infrastructure for integration of the filtering into Continuous Integration of the project (to run the selected tests automatically). The automatic running of tests would save developers the time they spend on running the tests manually.

Bibliography

[1] Chawathe, S. S., Rajaraman, A., Garcia Molina, H. and Widom, J. Change detection in hierarchically structured information. *Acm Sigmod Record*. ACM New York, NY, USA. 1996, vol. 25, no. 2, p. 493–504.

[2] Git community. *Git - diff-format Documentation* [online]. 2019 [cit. 2019-12-02]. Available at: https://git-scm.com/docs/diff-format.

[3] Haičman, M. *SCAP Security Guide intro pitch* [Red Hat internal slides]. September 2018.

[4] Jon Loeliger, M. M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. 2nd ed. O'Reilly Media, 2012. ISBN 978-1449316389.

[5] Lipner, S. B. The Birth and Death of the Orange Book. *IEEE Annals of the History of Computing*. Apr 2015, vol. 37, no. 2, p. 19–31. DOI: 10.1109/MAHC.2015.27. ISSN 1934-1547.

[6] Mikulėnas, M. *A comparison of protocols offered by GitHub* [online]. 2013 [cit. 2019-11-26]. Available at: https://gist.github.com/grawity/4392747.

[7] Red Hat Security Compliance Team. *ComplianceAsCode Developer Guide* [online]. 2019 [cit. 2019-12-08]. Available at: https://github.com/ComplianceAsCode/content/blob/master/docs/manual/developer_guide.adoc.

[8] Red Hat Security Compliance Team. *Security compliance content in SCAP, Bash, Ansible, and other formats* [online]. 2019 [cit. 2019-12-09]. Available at: https://github.com/ComplianceAsCode/content.

[9] Rossum, G. van van. *Unified Diff Format* [online]. 2006 [cit. 2019-12-02]. Available at: https://www.artima.com/weblogs/viewpost.jsp?thread=164293.

[10] Scott Chacon, B. S. *Pro Git*. 2nd ed. Apress, 2014. ISBN 1484200772, 978-1484200773.

[11] Stephenson, S. *Bash Automated Testing System* [online]. 2014 [cit. 2020-12-05]. Available at: https://github.com/sstephenson/bats.

[12] The MITRE Corporation. *OVAL - Open Vulnerability and Assessment Language* [online]. 2016 [cit. 2019-12-17]. Available at: https://oval.mitre.org/.

[13] Waltermire, D., Quinn, S., Booth, H., Scarfone, K. and Prisaca, D. *The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.3*. National Institute of Standards and Technology, 2016.

[14] Černý, J. *Nástroj pro tvorbu definic OVAL v projektu OpenSCAP*. Brno, CZ, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/18235/.

[15] Černý, J. *Automatizované ověřování konfigurace operačního systému MS Windows pomocí projektu OpenSCAP*. Brno, CZ, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/20842/.

Appendix A

Contents of the Included Storage Media

The storage media contains the following files and directories:

```
/ ............................................................ Storage media
├── content-test-filtering/ . Complete source codes of the implementation
│   ├── content_test_filtering.py ......................... Starting script
│   ├── ctf/ .......................................... Implemented modules
│   ├── README.md ......... Description of installation and usage of the tool
│   ├── tests/ ...................................... Source codes of tests
│   └── utility_scripts/ ............................ Folder with scripts
│       ├── comment_pr.py ............ Script that creates comment at GitHub
│       └── experiments.py ................. Script for experimental testing
├── dp.pdf ............................. Electronic version of this document
└── thesis_sources/ ........................ Source codes of this document
```