FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Assessing Software Vulnerabilities using Naturally Occurring Defects

Sofia Oliveira Reis

DISSERTATION

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão - *Faculdade de Engenharia da Universidade do Porto*

July 23, 2017

Assessing Software Vulnerabilities using Naturally Occurring Defects

Sofia Oliveira Reis

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: João Pascoal Faria - *Faculdade de Engenharia da Universidade do Porto*
External Examiner: José Campos - *Universidade do Minho*
Supervisor: Rui Maranhão - *Faculdade de Engenharia da Universidade do Porto*

July 23, 2017

Abstract

Currently, to satisfy the high number of system requirements, complex software is created which makes its development cost-intensive and more susceptible to security vulnerabilities.

In software security testing, empirical studies typically use faulty artificial programs because of the challenges involved in the extraction or reproduction of real security vulnerabilities. Thus, researchers tend to use hand-seeded faults or mutations to overcome these issues which might not be suitable for software testing techniques since the two approaches can create samples that inadvertently differ from the real vulnerabilities. Secbench is a database of security vulnerabilities mined from Github which hosts millions of open-source projects carrying a considerable number of security vulnerabilities.

The majority of software development costs is in identifying and correcting defects. To minimize such costs, software engineers responded creating static analysis tools that allow the detection of defects in the source code before being sent to production or even executed. Despite the promising future of these tools on reducing costs, during the software development phase, there are studies that show that the tools' vulnerabilities detection capability is comparable or even worse than random guessing, i.e., these tools are still far from their higher level of maturity, since the percentage of undetected security vulnerabilities is high and the number of correctly detected defects is lower than the false ones.

This study evaluates the performance and coverage of some static analysis tools when scanning for real security vulnerabilities mined from Github. Each vulnerability represents a test case containing the vulnerable code (Vvul) which can or cannot be exposed; and, the non-vulnerable code (Vfix) - fix or patch - which is not exposed. These test cases were executed by the static analysis tools and yielded a better analysis regarding performance and security vulnerabilities coverage. This methodology allowed the identification of improvements in the static analysis tools that were studied.

Besides contributing to the improvement of these tools, it also contributes to a more confident tools choice by security consultants, programmers and companies.

Resumo

Atualmente, de forma a satisfazer o elevado número de requisitos de um sistema, é produzido software complexo que torna o seu desenvolvimento custoso e mais susceptível a vulnerabilidades de segurança.

Na área de testes aplicados à segurança, estudo empíricos usam tipicamente programas com falhas artificiais devido aos desafios envolvidos na extração e reprodução de vulnerabilidades de segurança reais. Portanto, os investigadores tendem a usar falhas artificialmente plantadas ou mutações para ultrapassar esses problemas que podem não ser o mais adequado às técnicas na área de testes de software dado que as duas abordagens podem criar amostras que inadvertidamente diferem de vulnerabilidades reais. Secbench é uma base de dados de vulnerabilidades de segurança extraídas do Github que inclui milhões de projetos open-source que possuem um número considerável de vulnerabilidades de segurança.

A maioria dos custos no desenvolvimento de software é na identificação e correção de defeitos. De forma a minimizar esses custos, os engenheiros de software responderam com a criação de ferramentas de análise estática que permitem a deteção de defeitos no código fonte antes de ser enviado para produção or até executado. Apesar do futuro promissor destas ferramentas na redução de custos durante a fase de desenvolvimento de software, há estudos que mostram que a sua capacidade na deteção de vulnerabilidades é comparável ou até pior que deteção aleatória, i.e., estas ferramentas estão ainda longe do seu maior nível de maturidade, visto que a percentagem de vulnerabilidades de segurança não detetadas é maior e o número de vulnerabilidades detetadas corretamente é menor que as detetadas falsamente.

Este estudo avalia a performance e cobertura de algumas ferramentas de análise estática quando procurando por vulnerabilidades de segurança reais extraídas do Github. Cada vulnerabilidade representa um caso de teste que contém o código vulnerável (Vvul) que pode ou não ser exposto; e, o código não vulnerável (Vfix) - correção ou patch - que não é exposto. Estes casos de teste foram executados pelas ferramentas de análise estática e levaram a uma melhor análise em termos de perfomance e cobertura de ferramentas de análise estática. Esta metodologia permitiu identificar melhoramentos nas ferramentas de análise estática que foram estudadas.

Além de contribuir para o melhoramento destas ferramentas, também contribui para uma maior confiança na escolha das mesmas por consultores na área de segurança, programadores and companies.

Acknowledgements

Firstly, I would like to thank my mom and dad for always believing in me and for assuring me that no matter what, they would always stay by my side. Mom, thank you for all the times you told me to believe in myself, mainly, when I was almost giving up.

To my aunt Cristina and uncle Rui, I want to leave a huge thanks for all the support given in the last months and for the hard laughs in our crazy dinners at home.

To my little cousin who will not be able to read this since it almost failed on his English test, I want to leave a hug for always making me feel better when I was stressing out with my thesis. Hopefully, you will be able to read this one day!

To the rest of my awesome and big family, thank you for all the support. You are the best!

To my hometown friends, thank you for keeping me motivated and being the best friends someone could have. To my college friends, thank you for all the support through this past journey and for the wonderful moments for the last 5 years.

Lastly, I would like to thank my supervisor, Prof. Rui Maranhão a.k.a "Kim jong-Rui" for not losing his mind with my walls of text stressing out with the project. I am very thankful for your patience. I would also like to thank you for giving me the last push that I needed to go to the Netherlands for 6 months. It was indeed one of the best experiences of my life. I learned and grown up a lot, not only in person but mainly academically. Thank you for the 99999 questions answered, for the constant motivation and for believing in my work even when I did not.

Sofia Reis

"The only system which is truly secure is one which is switched off and unplugged, locked in a titanium lined safe, buried in a concrete bunker, and is surrounded by nerve gas and very highly paid armed guards. Even then, I wouldn't stake my life on it."

Dr. Eugene Spafford

Contents

1 Introduction **1**
- 1.1 Context . . . 2
- 1.2 Motivation . . . 3
- 1.3 Scientific Concepts and Definitions . . . 4
- 1.4 Methodology Overview . . . 6
- 1.5 Research Questions . . . 6
- 1.6 Goals . . . 7
- 1.7 Thesis Structure . . . 8

2 Literature Review **9**
- 2.1 Security Reports . . . 9
  - 2.1.1 IBM X-Force Threat Intelligence 2017 . . . 9
  - 2.1.2 Open Web Application Security Project Top 10 2017 . . . 10
  - 2.1.3 ENISA Threat Landscape Report 2016 . . . 11
- 2.2 Github . . . 12
- 2.3 Defects Prediction . . . 13
- 2.4 Defects Databases . . . 14
- 2.5 Static Analysis . . . 14
  - 2.5.1 Why Static Analysis? . . . 14
  - 2.5.2 How does a SAT work? . . . 15
  - 2.5.3 Semantic Analysis . . . 16
  - 2.5.4 Static Analysis Tools . . . 18

3 Methodology **25**
- 3.1 Identification and Extraction of Vulnerabilities from Github . . . 25
  - 3.1.1 Security Patterns . . . 26
  - 3.1.2 Test Cases Structure . . . 27
  - 3.1.3 Samples Evaluation and Acceptance Criteria . . . 28
  - 3.1.4 Database Manual Evaluation Limitations . . . 29
- 3.2 SATs Evaluation and Study . . . 30
  - 3.2.1 Identifying Good Candidates . . . 30
  - 3.2.2 Identifying Vulnerabilities and SAT Results Evaluation . . . 31
  - 3.2.3 SATs Evaluation Limitations . . . 32
- 3.3 SATs Modernization . . . 33
- 3.4 Conclusions . . . 33

4 Tools and Database Implementation **35**
4.1 Overview . . . 35
4.2 Database Structure . . . 37
4.2.1 Repository . . . 37
4.2.2 Commit . . . 38
4.2.3 Tool Results . . . 38
4.2.4 Experiments . . . 39
4.3 Mining Tool . . . 40
4.3.1 Mining Repositories . . . 40
4.3.2 Test Cases Structure on the Cloud . . . 41
4.4 Data Validation . . . 41
4.5 Data Visualization . . . 42

5 Empirical Evaluation **43**
5.1 SecBench Report . . . 43
5.1.1 How was the database sample chosen? . . . 43
5.1.2 Real Security Vulnerabilities Distributions . . . 45
5.1.3 Identified Vulnerabilities by CVE and CWE . . . 49
5.1.4 Mined and Accepted Vunerabilities within Repositories . . . 49
5.1.5 Regular Expression Efficiency . . . 50
5.1.6 Correlations . . . 52
5.2 Static Analysis Tools Report . . . 53
5.2.1 SAT: Infer . . . 53
5.2.2 SAT: Find-Sec-Bugs . . . 56
5.3 Conclusions . . . 57

6 SAT Problems **59**
6.1 Infer . . . 59
6.2 Infer: How does it work? . . . 59
6.3 Results and Source Code Comparison . . . 60
6.3.1 Case 1 . . . 60
6.3.2 Case 2 . . . 61
6.3.3 Case 3 . . . 62
6.3.4 Case 4 . . . 63
6.4 Conclusions . . . 63

7 Conclusions and Future Work **65**
7.1 Answer Research Questions . . . 66
7.2 Limitations . . . 67
7.3 Contributions . . . 68
7.4 Future Work . . . 68
7.5 Publications . . . 68

References **71**

A Patterns **75**
A.1 Top 10 OWASP 2017 . . . 76
A.1.1 A1 - Injection . . . 77
A.1.2 A2 - Broken Authentication and Session Management . . . 78

A.1.3 A3 - Cross-Site Scripting (XSS) . . . 79
A.1.4 A4 - Broken Access Control . . . 80
A.1.5 A5 - Security Misconfiguration . . . 80
A.1.6 A6 - Sensitive Data Exposure . . . 81
A.1.7 A7 - Insufficient Attack Protection . . . 82
A.1.8 A8 - Cross-Site Request Forgery (CSRF) . . . 82
A.1.9 A9 - Using Components with Known Vulnerabilities . . . 83
A.1.10 A10 - Underprotected APIs . . . 83
A.2 Others . . . 83
A.2.1 Memory Leaks . . . 84
A.2.2 Resource Leaks . . . 84
A.2.3 Context Leaks . . . 85
A.2.4 Path Traversal . . . 85
A.2.5 Denial-of-Service . . . 86
A.2.6 Overflow . . . 86
A.2.7 Miscellaneous . . . 87

B Scientific Paper **89**
B.1 SECBENCH: A Database of Real Security Vulnerabilities . . . 89

List of Figures

1.1 Software development lifecycle (SDLC) . . . 1
1.2 Overview of the different layers of issues and their dependencies that need to be solved in order to produce safer and cheaper software . . . 3
1.3 Methodology overview . . . 6

2.1 Security incidents by attack type by time and impact from 2014 to 2017 [USA17] 10
2.2 Everything that changed on the OWASP Top 10 from 2013 to 2017 [Fou17] . . . 11
2.3 An example of the information provided by Github for a sucessfully caught test case: commit 8414fe1 from linux developed by torvalds . . . 13
2.4 Basic operation of a SAT . . . 16

3.1 Workflow to extract and identify real security vulnerabilities . . . 26
3.2 Difference between V_{fix}, V_{vul} and V_{diff} . . . 28
3.3 SATs study methodology overview . . . 30
3.4 Example of several vulnerabilities identification . . . 32

4.1 Different modules and applications produced to obtain and visualize the results . 36
4.2 Github structure and vulnerabilities detection . . . 40
4.3 Merge of more than one commit at the same time . . . 41
4.4 Difference between V_{fix}, V_{vul} and V_{diff} on the cloud . . . 41
4.5 Example of bar chart uisng `D3.js` . . . 42

5.1 Top 13 of the most mined languages in bytes of code (BOC) . . . 44
5.2 Distribution of real security vulnerabilities per year . . . 46
5.3 Distribution of real security vulnerabilities per language . . . 46
5.4 Distribution of real security vulnerabilities by pattern . . . 48
5.5 Distribution between mined commits and their current state without reallocating vulnerabilities . . . 51
5.6 Distribution between mined commits and their current state after reallocating vulnerabilities . . . 51
5.7 Distribution between mined commits and the time of development (1) and vulnerabilities accepted and the time of development (2) . . . 52
5.8 Distribution between mined commits and the repository size (1) and vulnerabilities accepted and the repository size (2) . . . 52
5.9 Distribution of FP, TP and FN on detecting memory leaks by `Infer` for C (1) and Objective-C (2) . . . 54
5.10 Distribution of FP, TP and FN on detecting resource leaks by `Infer` for C (1) and Java (2) . . . 55

5.11 Distribution of FP, TP and FN on detecting memory leaks (1) and resource leaks (2) by `Infer` 56

6.1 Infer Case 1: 2 FP for memory leaks in Objective-C 61
6.2 Infer Case 1: Trace bugs 61
6.3 Infer Case 2: 2 FP for resource leaks in C 62
6.4 Infer Case 3: 1 FN for resource leaks on C 63
6.5 Infer Case 4: 1 FN for resource leaks in C 63

A.1 Injection pattern 77
A.2 Broken Authentication and Session Management pattern 78
A.3 Cross-Site Scripting pattern 79
A.4 Broken Access Control pattern 80
A.5 Security Misconfiguration pattern 80
A.6 Sensitive Data Exposure pattern 81
A.7 Insufficient Attack Protection pattern 82
A.8 Cross-Site Request Forgery pattern 82
A.9 Using Components with Known Vulnerabilities pattern 83
A.10 Underprotected APIs pattern 83
A.11 Memory Leaks pattern 84
A.12 Resource Leaks pattern 84
A.13 Context Leaks pattern 85
A.14 Path Traversal pattern 85
A.15 Denial-of-Service pattern 86
A.16 Overflow pattern 86
A.17 Miscellaneous pattern 87

List of Tables

2.1 IBM's executive overview about the different types of security events 10
2.2 10 of the most popular programming languages on Github statistics 12
2.3 Defects databases state-of-the-art and requirements fulfilled 14
2.5 SATs advantages and disadvantages 15
2.6 Different types of SATs analyzers 16
2.4 Defects databases comparison and overview 19
2.7 Static Analysis Tools state-of-the-art and comparison (I) 20
2.8 Static Analysis Tools state-of-the-art and comparison (II) 21
2.9 Static Analysis Tools state-of-the-art and comparison (III) 22
2.10 Static Analysis Tools state-of-the-art and comparison (IV) 23
2.11 Static Analysis Tools state-of-the-art and comparison (V) 24

3.1 Top 10 OWASP 2017 27
3.2 Other Security Issues/Attacks 27

4.1 Repository hash fields 37
4.2 Commit hash fields 38
4.3 Vulnerability hash fields 39
4.4 Experiment hash fields 39

5.1 Vulnerabilities identified with CVE 49
5.2 Mined Vulnerabilities Distribution 49
5.3 Accepted Vulnerabilities (AVulns) Distribution 50
5.4 `Infer`: Resulting metrics for memory leaks 54
5.5 `Infer`: Resulting metrics for resource leaks 55
5.6 `Infer`: Resulting metrics for memory leaks and resource leaks 56
5.7 Resulting `Infer` metrics 56

Abbreviations

API	Application Programming Interface
BOC	Bytes of Code
OSS	Open Source Software
OWASP	Open Web Application Security Project
SAT	Static Analysis Tool
SDLC	Software Development Life Cycle

Chapter 1

Introduction

From holding hostage worldwide companies information to keeping several distributed systems down for hours, the last two years were marked by several security attacks which are the result of complex software and its fast production.

Under the rush of companies trying to outdo each other, fast updates and software supporting new features with every new release, testing tends to be one of the most forgotten phases of the software development lifecycle (SDLC). Not only due to the lack of reliable automated tools to test software but also due to the high costs associated with the task. Some companies do not even pay attention to the process of finding security vulnerabilities before shipping software. The ones that do care, usually have two choices:

- They try to tackle the issue hiring testers and the best developers they can; making extensive manual code reviews or even paying money to external people to find bugs (bug bounty programs);
- Or, they ship the product based on a possible "well-thought-out" balance between the damages of a vulnerable version and the fact that the vulnerable code would never be disclosed.

Companies tend to choose the second one since the first one is too time-consuming and monetarily unbearable. Although dangerous, companies would escape attacks with the second approach, but now, and more than ever, it rarely happens, and the proof is the increasing of security vulnerabilities reported by annual security reports [USA17, CIS07, fNS17].

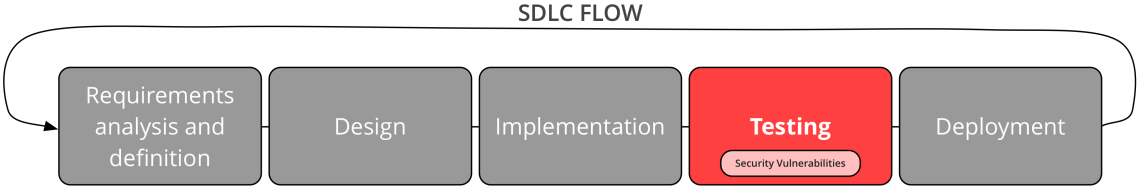

Figure 1.1: Software development lifecycle (SDLC)

Testing (Fig. 1.1) is one of the most important activities in SDLC since it is responsible for ensuring software's quality through the detection of the conditions which may lead to software failures.

Static Analysis Tools (SATs) are one of the most promising ways of fighting the costs associated with the identification and correction of software vulnerabilities. But, despite their potential, the results are still far from being reliable and sometimes even understandable [JSMHB13].

This thesis' main goal is to understand and discuss if the currently used methodology (Chap. 3) allows researchers to study and find points of modernization on SATs using real security vulnerabilities. On the other hand, there are no databases containing a considerable amount of real security vulnerabilities to study these tools. So, another goal will be the analysis of source code hosting websites to see if there is available information to create a benchmark of real security vulnerabilities.

This chapter aims the topics contextualization and motivation, a better explanation of the problems and goals behind this thesis and also a brief introduction to the used terminology.

1.1 Context

More than one century ago, Nevil Maskelyne disrupted John Fleming's public demonstration of a supposedly secure wireless telegraphy technology created by Guglielmo Marconi, sending insulting messages in morse code through the auditorium's projector [Dav15]. This was the first known security attack ever performed, back in the early 1900s.

Since then, technology had an explosive growth in several different fields which not only improved people's lives but turned to be a problem and threat for technology companies regarding holding the consequent and adjacent increase of security vulnerabilities. According to IBM's X-Force Threat Intelligence 2017 Report [USA17], the number of vulnerabilities per year has been significantly increasing over the past 6 years. Not only the number of known vulnerabilities (e.g., SQL injection, cross-site scripting and others) but also the number of the unknown ones, which grants even more importance to this field since developers have been struggling already with the disclosed ones.

To satisfy the high amount of system requirements, developers produce software with millions of lines of code which turns its development cost-intensive and more susceptible to vulnerabilities that can result in severe security risks. The identification and correction of defects take more than 50% of the software development costs [Tas02] which is extremely high given all the phases involved in the SDLC (Fig. 1.1). This is due to the lack of reliable automated techniques and tools to detect these issues and the non-knowledge of their existence by developers [YM13].

Several SATs (Tab. 2.7) can detect security vulnerabilities through a source code scan which may help to reduce the time spent on the vulnerabilities identification and correction. Unfortunately, their detection capability is comparable or even worse than random guessing [GPP15], i.e., the percentage of undetected security vulnerabilities of these tools is high, and the number of correctly detected defects is lower than the false ones.

To study and improve these software testing techniques, empirical studies using real security vulnerabilities are crucial [Bri07] to gain a better understanding of what tools are able to detect [BL04].

Yet, performing empirical studies in the software testing research field is challenging due to the lack of widely accepted and easy-to-use databases of real bugs [JJE14, DER05], as well as the fact that it requires human effort and CPU time [Bri07]. Consequently, researchers tend to use databases of hand-seeded vulnerabilities which differ inadvertently from real vulnerabilities and thus might not work with the testing techniques under evaluation [JJI+14, PCJ+17]. According with [JJI+14], it is only possible to represent real security vulnerabilities using mutations under specific conditions, i.e., mutations may not always be a feasible approach to represent real security vulnerabilities. Thus, it is crucial to contribute with new databases of security vulnerabilities to help to increase the number of available test cases to study static analysis tools.

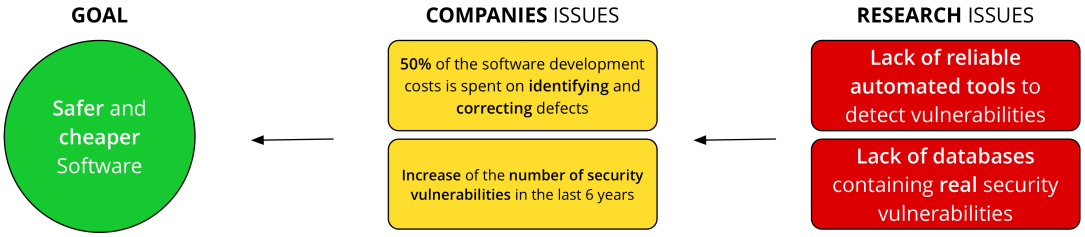

Figure 1.2: Overview of the different layers of issues and their dependencies that need to be solved in order to produce safer and cheaper software

The improvement of the SATs can be a major step not only on the decreasing of time and money spent on vulnerabilities identification and correction; but also on the number of security vulnerabilities on the released software versions. It is necessary to obtain primitive data (e.g., database of real security vulnerabilities) which is difficult to collect since there are not feasible tools or methodologies to collect them easily; and, because until a few years ago there was no information available (e.g., software source code). Due to the now increasing adhesion to the production of open-source software (OSS), we believe source code hosting websites (e.g., Github, Bitbucket and SVN) contain information to collect test cases to study the SATs.

Thus, research issues must be tackled, in order to smooth companies' wastes, always with the main goal of helping developers producing safer and cheaper software (Fig. 1.2).

1.2 Motivation

It is essential to find means to help researchers minimizing the impact of the lack of reliable SATs since they have strong potential on decreasing the costs associated with testing. While some vulnerabilities can be not even disclosed, others can lead to the user's privacy violation.

The past years have been flooded by news of security attacks from the cybersecurity world:

- Exposure of large amounts of sensitive data (e.g., 17M of zomato accounts stolen in 2015 which were put up for sale on the dark web marketplace only now in 2017) [Dun17].
- Phishing attacks (e.g., a simple link faking to be a shared Google Docs document would provide access to user e-mails and contacts) [Kha17b].
- Denial-of-service attacks like the one experienced last year by Twitter, The Guardian, Netflix, CNN and many other companies around the world. A network of computers infected with a special malware (botnet) was programmed to inject traffic into a server until it collapsed under the strain [Woo16].
- Gotofail vulnerability on iOS 7 (Apple, 2014). This vulnerability kept the system vulnerable for a few months due to an extra gotofail after the end of an if which would make the program jump to the end without verifying the authentication signature [Gua14].
- The shellshock vulnerability was discovered in 2014 on most versions of Linux and Unix operating systems. An attacker would exploit the issue remotely forcing an application to send a malicious environment variable to Bash [ABE+15].
- Or, the one that possibly stamped the year, the ransomware attack which is still very fresh and kept hostage many companies, industries and hospitals information [Kha17a].

All of these attacks were able to succeed due to the presence of security vulnerabilities that were not tackled before someone exploit them.

Early in the current year, the European Union Agency for Network and Information Security (ENISA) released their annual Cyber-Threats and Trends report of 2016 [fNS17] where the agency reports that the last year was marked by cyber attacks whose main goals were mainly monetization and politic impact. Trends are basically the same as in 2015, being the top of the table malwares, web-based attacks, web applications attacks, denial-of-service and botnets.

A survey performed on 400 IT executives from 19 industries revealed that 48% of the firms experienced at least one Internet-of-Things (IoT) security breach with a potential cost of $20 million in a company whose annual revenue is above $2 billion [Sec17].

For these reasons, it is necessary to search and improve automated techniques and tools to help companies reducing not only the costs of identifying and correcting defects but also the costs associated with the breaches that can result from the production of unsafe software.

Improving these tools will be a major step in the adoption of automated techniques for the identification and correction of security vulnerabilities phase.

1.3 Scientific Concepts and Definitions

To better understand the terminology used throughout this thesis, the definitions of a few terms are presented in this section.

- A *vulnerability* is a software defect which can be exploited by an attacker whose goal is to violate the software security policy.
- When the word *real* is used to qualify vulnerabilities, it is to emphasize the idea that they were naturally created by developers, i.e., they were not created with the purpose of testing software (non-artificial test cases).

Definition 1 *A test case t is a 3-tuple (v,f,d), where v is the sample of code which contains the vulnerability (works differently from what was expected); f is the sample of code that fixes the vulnerability (works as expected); and, d is the sample of code which represents the difference between v and f.*

Definition 2 *A test suite* $T = \{t_1, ..., t_n\}$ *is a set of test cases whose function is to test if the program follows the specification (list of requirements).*

Definition 3 *The cardinality of X (#X) is the number of elements of type X.*

The results classification and metrics used to evaluate the tools reliability are based on the National Security Agency SATs study [fAS11]. Three different types of results classifications will be considered:

- A *True Positive (TP)* when the tool correctly identifies and reports the alleged vulnerability.
- A *False Positive (FP)* when the tool reports the target vulnerability in another part of the test case - *false warning*.
- A *False Negative (FN)* when the tool does not report the supposed vulnerability.

Three different metrics will be used to calculate if the tool is good or bad: precision, recall and f-score, where $\#TP$ is the number of true positives, $\#FP$ is the number of false positives, and $\#FN$ is the number of false negatives found studying the tool.

- *Precision (pr)* measures how reliable is the report of the SAT based on the ratio between the number of reported vulnerabilities by the tools and the number of vulnerabilities on the code under analysis. If $pr = 0$, then the SAT is not reliable, but if $pr = 1$, then the SAT is 100% reliable.

$$pr = \frac{\#TP}{\#TP + \#FP}$$

- *Recall (r)* or sensibility measures if the tool correctly identifies the vulnerabilities based on the fraction of real vulnerabilities reported (true positives) from the code under analysis. If $r = 0$, then the SAT does not identify correctly the vulnerabilities, but if $r = 1$, then the SAT identified all the vulnerabilities correctly.

$$r = \frac{\#TP}{\#TP + \#FN}$$

- *F-score (fs)* will help on identifying if a SAT is good or not capturing how many vulnerabilities were found (true positives) and how much noise is produced (false positives). It measures the overall tool performance.

$$fs = 2 \times \frac{pr \times r}{pr + r}$$

1.4 Methodology Overview

This thesis proposes a methodology that allows the study of SATs using real security vulnerabilities in four different steps (Fig. 1.3).

The methodology starts with the creation of a mining tool whose input is a Github repository and a previously defined security pattern. This tool iterates all repository commits, in order to find patterns on its messages. From this process will possibly result candidates to test cases, i.e., real security vulnerabilities. The source code of the candidates is evaluated by the researchers to make sure it is actually a vulnerability and to identify it correctly, since the code may not reflect the message commit. Then, the candidates were identified as real security vulnerabilities or not. If yes, the vulnerability may be considered on the SATs study. Each SAT has its own list of supported vulnerabilities and languages. So, a previous study was elaborate to n order to understand which languages and vulnerabilities each SAT supports (Tab. 2.7). If the vulnerability suits the tool, its source code will be scanned, and the results will go through another manual evaluation where humans try to understand if the tool results are expected or not. After this last evaluation, humans are able to determine if the tool is reliable or not. Thus, the researcher may find points of improvements on the SAT comparing the SAT results and the results from the second manual evaluation.

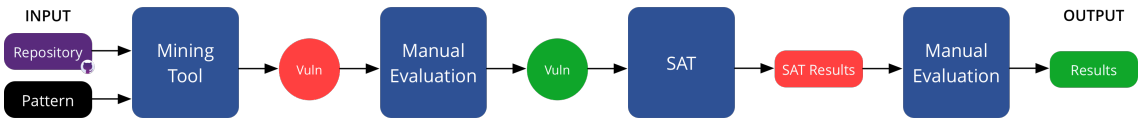

Figure 1.3: Methodology overview

It is a very meticulous methodology which will be implemented, tested and discussed throughout the next chapters.

1.5 Research Questions

As mentioned before, researchers believe SATs are still not reliable due, mainly, to the high rate of false positives and false negatives; and, to the not user-friendly way of how some warnings are presented [JSMHB13]. Researchers believe that to study these tools, empirical studies using real security vulnerabilities must be performed since they are crucial [Bri07] to gain a better understanding of what tools are able to detect [BL04].

The **hypothesis** under this thesis is the one presented below:

> *It is possible to identify points of improvement on a static analysis tool using real security vulnerabilities.*

Based on this, two different groups of research questions will be answered. Firstly, related to the availability of information on OSS repositories:

- **RQ1.1** *Is there enough information available on OSS repositories to create a benchmark of software security vulnerabilities?*
- **RQ1.2** *What are the most prevalent security patterns on OSS repositories?*

Secondly, related to the performance and possible improvements resulting from the study performed on the SATs using real security vulnerabilities collected from OSS repositories:

- **RQ2.1** *Is it viable to study SATs with real security vulnerabilities?*
- **RQ2.2** *Can we understand where SATs can be improved using this methodology?*

This thesis tries to answer the previous research questions for two different fields of research with the aim of satisfying the proposed goals (Sec. 1.6).

1.6 Goals

This thesis has no background tools for mining repositories in this domain or available databases to be used. So, everything needed to perform the study was created from scratch. This thesis' goals are:

- Demonstrate if the chosen methodology works to collect real security vulnerabilities
  - Identify a good and considerable amount of trending security patterns
  - Create a tool which mines the patterns on Github repositories
  - Create a database with real security vulnerabilities
  - Contribute with a new test suite of real security vulnerabilities to the software testing field
- Contribute with a new methodology to study SATs and collect real security vulnerabilities from OSS software hosted by Github
- Understand if this methodology helps to find points of improvement in the SATs
- Answer the research questions

The thesis' main goal is to understand if the methodology provides results that support the hypothesis. Thus, this methodology can be used to study other tools or as a base in the search for better approaches on studying SATs; and, even on collecting real vulnerabilities from source code hosting sites.

1.7 Thesis Structure

This thesis contains another six chapters. In chapter 2, the related work and state-of-the-art are discussed and presented: conclusions obtained from security reports; different techniques available for defects prediction; and, the state-of-the-art of databases and static analysis tools. In chapter 3, the methodology under the study is presented in detail. In chapter 4, the tooling support created to perform the study is presented in detail: different applications created, technologies used, database structure and data validation. In chapter 5, the statistic results from the database and the tools studied are discussed and presented. In chapter 6, a few interesting results are highlighted in order to perform future improvements to the tools studied. In chapter 7, some conclusions, challenges and limitations are presented along with a few points of discussion for future work. Appendix A contains the visualization of each security pattern is presented. Appendix B contains the accepted scientific publications, submitted to *DX'17*[1], *SecSE 2017*[2]; and, invited to *IJSSE*[3].

[1]The 28th International Workshop on Principles of Diagnosis
[2]The International Workshop on Secure Software Engineering in DevOps and Agile Development
[3]The International Journal of Secure Software Engineering

Chapter 2

Literature Review

In this chapter, an overview of all the subjects involved in this study is provided: security reports (Sec. 2.1), Github's interesting information and statistics (Sec. 2.2), defects prediction techniques that can be used in this study (Sec. 2.3), defects databases (Sec. 2.4) and static analysis (Sec. 2.5).

2.1 Security Reports

In this section, interesting conclusions and summaries from several security reports are presented to help understand the trendiest security events and what are the types of vulnerabilities behind them.

Overall, they all report a significant increasing of security vulnerabilities over the last years and more or less in the same patterns depending on the focus field. Although the awareness of security is starting to emerge between developers and having positive results, the attackers are also more clever using attacks with higher impacts and attacking larger sets of companies at each time.

2.1.1 IBM X-Force Threat Intelligence 2017

IBM's X-Force Threat Intelligence 2017 [USA17] is based on information collected on 2016, from $8K$ client devices and industries sensors. They classify security events according to their impact and information available based on the next hierarchy: security event $>$ attack $>$ security incident $>$ breach, being a breach a security event, attack and incident (Tab. 2.1).

According to these results, IBM's questions if the decrease of attacks and security incidents reflects a safer environment and software. It does not, 2016 was one of the most notable years on cybersecurity with malwares leaving all Ukraine without electricity, major phishing attacks and information leaks being the largest ones at Germany (1.9 TB of information about European football players) and Kenya (1TB of Kenyan Ministry of Foreign Affairs including trade secrets and classified information).

Classification	Types of Events	From 2015 to 2016
1	Security Events	↑ 3%
2	Attacks	↓ 12%
3	Security Incidents	↓ 48%
4	Breach	↑↑

↑↑ - 4 billion records leaked (≡ combined total from the 2 previous years)

Table 2.1: IBM's executive overview about the different types of security events

Another interesting point reported by IBM is the large number of undisclosed vulnerabilities, i.e., vulnerabilities that do not belong to any known attack type or class which can be harmful since developers have been struggling already with the disclosed ones.

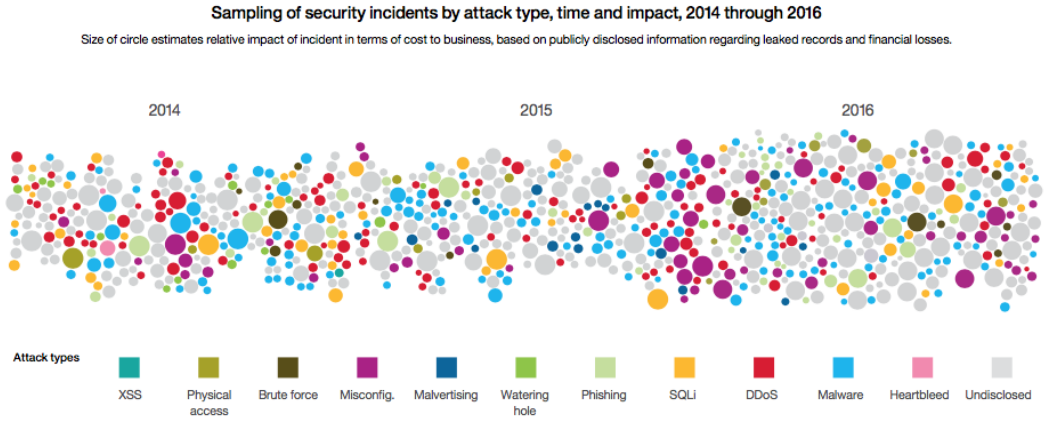

Figure 2.1: Security incidents by attack type by time and impact from 2014 to 2017 [USA17]

Spam with malicious content was always a problem, but since March 2016 the issue suffered a major growth. Based on IBM's database which contains more than $10K$ vulnerabilities, cross-site scripting and SQL injection are the trendiest vulnerabilities and the injection of unexpected items (42%) and data structures manipulations (32%) the most performed attacks to their clients.

Information and communications is the industry that sharply suffered more breaches in 2016. To perform malicious spam, clickjacking and phishing attacks, attackers need to obtain peoples' information. Healthcare is one of the best sources of that kind of sensitive data (e.g., e-mails, phone numbers, name, addresses and more) and has the higher percentage of attackers in 2016.

2.1.2 Open Web Application Security Project Top 10 2017

In this section, it is given an overview of the changes between 2013 and 2017 (Fig. 2.2) in the top 10 of the trendiest security vulnerabilities in web applications created by the Open Web Application Security Project (OWASP).

The fast adoption of new technologies (e.g., cloud), the automation of software development processes (e.g., Agile and DevOps), the explosion of third-party libraries and frameworks, and the advances made by attackers are responsible for the changes on the top 10 OWASP [Fou17].

Insecure direct object reference (A4) and missing function level access control (A7) were merged into the older category broken access control (A4) since the authors felt it is no longer needed to separate the two. This separation had the purpose of giving more attention to the different patterns.

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Figure 2.2: Everything that changed on the OWASP Top 10 from 2013 to 2017 [Fou17]

Due to the lack of capabilities to detect, prevent and respond to manual and automated attacks a new pattern was created Insufficient Attack Protection (A7). Unvalidated redirects and forwards (A10) was dropped due to the lack of data showing that is still a prevalent issue on the current software produced.

Current applications and APIs involved in client applications like mobile applications and JavaScript in the browser are connected to APIs (e.g., SOAP/XML, REST/JSON and many others) which are often unprotected and contain several security vulnerabilities. Underprotected APIs (A10) pattern was added to give awareness to the importance of maintaining their software secure since it is used by many applications.

2.1.3 ENISA Threat Landscape Report 2016

According to [fNS17], the maturity of defenders is increasing, and security is finally gaining attention in the professional education and training market. However, attackers are still one step ahead. They are focused on:

- Larger attacks, i.e., attacks that can reach further using distributed denial-of-service attacks through the infection of IoT devices.
- Extortion attacks, targeting commercial organisations and achieving high levels of ransom and rates of victims.

- Attacks affecting the outcome of democratic processes (e.g., US elections).

The top 5 of threats in 2016 contains malware, web-based attacks, web application attacks, denial-of-service and botnets which reflects this thesis' motivation.

2.2 Github

Github is one of the most used source code hosting websites by developers. It is known as the social network for programmers and developers. Many other source code hosting websites already existed, but Github was the only one able to conquer companies like Google, Microsoft and many others. It contains more than $22M$ of users and $61M$ of OSS repositories, and it shares more than $199M$ issues. So, Github is definitely a huge domain where we can find information.

According to a few statistics collected from the Github blog[1] and GitHut[2], some of the most popular programming languages on Github are JavaScript, Java, Python, Ruby, PHP, CSS, C, C++, C# and Objective-C.

Language	**#Repositories**	**#Closed Issues**	**#Open Issues**
JavaScript	978,726	3,386,650	679,953
Java	739,714	3,168,495	851,421
Python	509,852	2,079,340	399,232
Ruby	497,604	1,877,208	356,441
PHP	457,446	2,017,814	358,568
CSS	351,693	619,493	103,597
C	260,948	1,206,189	450,685
C++	263,881	1,311,363	258,093
C#	241,828	1,010,944	312,306
Objective-C	165,701	394,912	116,723

Table 2.2: 10 of the most popular programming languages on Github statistics

Above (Tab. 2.2) are presented a few statistics collected from Github: number of repositories, number of closed issues and open issues. Issues can also be a good source of information for collecting test cases.

Along with the variety of languages, Github has repositories spanning different sizes produced by millions of different programmers, with good and bad programming skills, which is reflected on the quality of the source code. This can be a threat to finding test cases of good quality since there are not automated ways of checking if a repository is useful or not.

There is no much information on finding security vulnerabilities on Github: if there is any correlation with the repository size, the development time (project maturity) or even the number of pull requests.

[1] https://github.com/blog/2047-language-trends-on-github
[2] http://githut.info/

Figure 2.3: An example of the information provided by Github for a sucessfully caught test case: commit 8414fe1 from linux developed by torvalds

Github hosts millions of repositories containing hundreds even thousands of commits. Each commit has a message associated reflecting the purpose of the commit (Fig. 2.3). Since the message is written by humans, sometimes they may not reflect entirely what is in the source code or vice-versa.

Every time a developer has something new to add to the project, a new commit is made. Thus, Github provides a tree of commits where you can see the difference between the previous commit and the current commit. If a fix of a security vulnerability is caught on the current commit, then the probability of the vulnerability being on the previous commit is high.

2.3 Defects Prediction

To identify bug fixes on software histories, [SZZ05] proposes the syntactic analysis and semantic analysis. Assuming a link (t, b) between a transaction t and a bug b, there are two different levels of confidence associated: syntactic and semantics.

On syntactic analysis, they split all the log file into tokens in order to find links to the bug database. Regular expressions like `bug[# \t]*[0-9]+` are used to represent numbers which are potential links to bugs. Tokens can be *bug numbers*, *plain numbers*, *keywords* matched with `bugs?|defects?|patch` and *words*.

They assume a bug has an initial confidence of 0 and it can be increased until 2. To increase the level of confidence, a few conditions need to be fulfilled: the number needs to be a *bug number* and the log message needs to contain a *keyword* or only *plain* or *bug numbers*.

In terms of semantic analysis, the confidence level is also used but for other conditions: b was set to fix at least once, the bug's report description is contained on the log message of t, the author of t has been assigned to the bug b and, one or more the files affected by t have been attached to b.

In terms of predicting vulnerable software components, specific patterns must be also detected other than only looking for general patterns like buffer overflows. This may not work well with

[SZZ05], thus `Vulture` was created to mine a vulnerability database, the software history and the code base in order to map past vulnerabilities to entities in software projects that can have vulnerabilities[NZHZ07]. Thus, a resulting predictor can predict future vulnerabilities on new components, based on import and function calls.

On [NZHZ07], researchers retrieved all identifiers from advisors which is where vulnerabilities are announced. Then, they searched for references to the database that would take the form of links containing the bug identifier of the defect that caused the vulnerability. It is assumed that a component is characterized by its *imports* and *function calls*.

2.4 Defects Databases

This section mentions the existing related work in the field of databases created to perform empirical studies in the software testing research area. On table 2.3 is presented an overview of a few databases that satisfy the requirements presented below:

The database contains real defects	SIR, Defects4j, Safety-db
The database contains security vulnerabilities	CAS, OSWAP Benchmark, Safety-db
The database was created to study testing tools	CodeChecker

Table 2.3: Defects databases state-of-the-art and requirements fulfilled

Although there are databases targeting security vulnerabilities test cases, only one of them contains real vulnerabilities (Tab. 2.4), the other ones are a mix of real and artificial test cases, or they only contain artificial samples.

2.5 Static Analysis

Static analysis is usually performed using automated tools (SATs) as a part of code review carried out during the SDLC. The SATs were designed to analyze source code and/or binary code and intermediate code versions (e.g., Java bytecodes or Microsoft Intermediate Language) to help to find security flaws. The analysis is named static because it does not involve code execution.

2.5.1 Why Static Analysis?

Simple commands like *grep*, can be used to find vulnerabilities. The function *gets*, from C library, is responsible for a high percentage of buffer overflows because it does not have a buffer size limit, so the code's safety depends on the user to always manage the quantity of inserted characters. This function is highly vulnerable, therefore, is one of the most used functions to recognize the presence of the vulnerability. So, if a developer does *grep gets *.c* on a file where the function is used it may get a vulnerability. This type of commands allows a developer to search for vulnerabilities without executing the code, which is basically what static analysis does.

Using this type of commands has several limitations, like the ability to distinguish between real function calls or equal sequences on comments, variables and strings; and, is non-practicable to developers since they have to run the command for each function and file.

In order to solve those limitations, a few automated tools (RATS[3], Flawfinder[4] and ITS4 [VBKM00]) were created using almost the same mechanism as a compiler on the errors verification but, in this case, for vulnerabilities (source code separated in tokens and matching analysis between them and the dangerous functions). The main component behind of these tools was a database with words or rules for detecting vulnerabilities.

Thus, SATs appeared to automate the coder manual reviews which are one of the most hard-working and costly tasks done by developers during the SDLC [Tas02].

According to [GPP15], SATs performance and coverage are still far from its higher level of maturity. However, they also have its own advantages and promising features (Tab. 2.5): they can make developers lives easier and have a high potential of scalability and improvement. It is a field where researchers must definitely invest. It is not enough to have good tools, researchers must understand their problems and discover how to solve them because *A fool with a tool is still a fool* if it does not know what truly has in hands.

Advantages	**Disadvantages**
Automatize the verification of vulnerabilities in the source code (free of human errors).	False negatives. The tools do not identify all the known vulnerabilities (limited scope).
Allow the detection of vulnerabilities before the software is sent to production.	It is not possible to test all the conditions in useful time (limited analysis).
Focus on the source of the vulnerabilities and not on their effects.	False Positives. Many defects are hard to identify as vulnerabilities or not.
If new vulnerabilities appear, new rules can be added (High Scalability).	It is not easy to run them and understand its results (Not user-friendly).
Detection of vulnerabilities, bugs, code style issues and more.	

Table 2.5: SATs advantages and disadvantages

2.5.2 How does a SAT work?

The analysis is performed based on the vulnerabilities description which, normally, consists on a group of rules (Fig. 2.4). The first stage (*Model Construction*) corresponds to the first functions of a compiler: *scanner*, pre-processing and *parser*. The model is an *abstract syntax tree (AST)* which can be used to represent programming languages. Along the AST, a symbol table is created with the function names, variables, etc. Some tools extend the AST to a control flow graph where each node represents one or more instructions which are executed sequentially. The direction from

[3] https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml
[4] https://security.web.cern.ch/security/recommendations/en/codetools/flawfinder.shtml

each connection between nodes represents the potential control flow. After creating the trees and graphs, the analysis is performed using as input the security rules that need to be verified (Fig. 2.4).

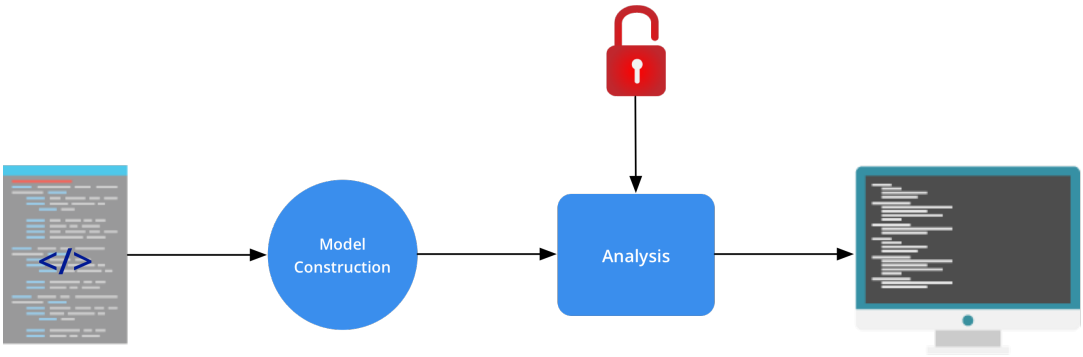

Figure 2.4: Basic operation of a SAT

It is important to note that more advanced techniques exist to perform the analysis in a more efficient way: symbolic execution [YT88], abstract interpretation [CC77] and more.

2.5.3 Semantic Analysis

These are the most prevalent semantic analyzers within security: data types verification, control flow analysis and data flow analysis.

String comparison	Operates directly on source-code in first stage of compilation (*scanner*) like the command *grep*.
Lexical Analyzer	Operates over the tokens generated by the *scanner*, after the first compilation stage. The tools mentioned on sec. 2.5.1 are an example and unlike string comparators they do not mislead a variable of name *getsugar* with the call *gets*.
Semantic Analyzer	Operates over the abstract syntax tree generated by the *parser*, after the third compilation stage. Unlike lexical analyzers, they do not mislead a variable and a function call with the exact same name.

Table 2.6: Different types of SATs analyzers

2.5.3.1 Data Types Verification

In programming, data types limit the way of how data is manipulated (e.g., certain variables are defined as *int* and others as *string* to avoid assignments between them). Thus, the notion of type verification is intrinsically connected with the data types notion. Compilers and languages interpreters that support data types (e.g., C and Java) make this type of verification.

Data types verification based on the original data types of languages like C and Java have a limited utility on the vulnerabilities static analysis because the compilers perform a data type verification which is more restrictive than necessary. One example is the integer vulnerability. Sometimes, the compilers perform an overly restrictive type verification like in Java, where the verification process allows the assignment of a *short* to an *int* but not vice-versa. However, in C the compiler allows integers assignments which can result in a vulnerability.

Two of the integer vulnerabilities that can be detected using this approach are the sinal errors - a signed *int* is assigned to an *int* variable without sign - and the truncation bugs - an *int* represented by a certain number of bits is assigned to a variable with fewer bits. In both cases, the vulnerability is a result of wrong assigning between incompatible data types which can be detected in the process verification simply determining if the incompatibility exists or not.

2.5.3.2 Control Flow Analysis

This type of analysis traverses all possible and different execution paths through the code instructions (conditional, cycles, jumps and function calls) which will result in a control flow graph. The control flow analysis consists of traversing the graph and verifying if certain rules are satisfied. This approach can be used to detect several vulnerabilities and bugs (e.g., memory leaks or resource leaks).

One of the main limitations of this approach is the lack of access to external libraries (e.g., C library). Thus, developers tend to create models to represent their features. For example, to analyze a function that uses `malloc` or `fopen` it is necessary to create models to represent their external behaviour and impact.

There are three different levels of control flow analysis: local analysis (isolated analysis of a function or component); module analysis (module, class or file analysis) and interprocedural analysis (all program is analyzed).The result of local analysis is a function module. Module and interprocedural analysis recursively analyze each function of the module and program until all the modules are created. This can be time-consuming for large projects.

For each chosen path, the process starts with the analysis of the memory state, then the path is simulated throwing or not alarms when vulnerabilities are identified, and in the end the all path is analyzed generating alarms if necessary. After each path analyzed, a model is created to the function.

2.5.3.3 Data Flow Analysis

Data flow analysis is the process of deriving information about the run-time behaviour of a program [KSK09]. This analysis can be done based on a control flow graph using the control flow analysis. There are several ways of data flow analysis in the security domain, but the most common is the *taint analysis*. This type of analysis is one of the most important to discover coding vulnerabilities since it verifies if compromised data - data under the attacker's domain - is used by

dangerous functions. One example is the `strcpy` function which can be vulnerable if the source *string* is compromised.

The main idea is to follow the data flow, to understand if input data reaches dangerous code instructions. This technique classifies if the returning value of a function is potentially compromised (*tainted*) or if an argument from a library function cannot be compromised (*untainted*) based on the parameters.

Another question is the propagation associated with the assign between compromised variables. This is obvious, if a is compromised and $a = b$ then b will be also compromised. However, if d is compromised and $c = func(d)$, then, in this case, it is not possible to immediately say if c is compromised or not. Thus, it is necessary to annotate the function $func$ definition which is a new model to be used like the ones used in the control flow analysis.

Data flow analysis discovers several types of vulnerabilities (e.g., buffer overflows, cross-site scripting and injection).

2.5.4 Static Analysis Tools

Our research counts with more than 25 SATs (Tab. 2.7 - 2.11). It is possible to find free and open-source tools for a wide range of languages (e.g., Java, C/C++, Objective-C, Python, etc) and companies (e.g., Facebook, Google, NASA and more) using different types of analysis (e.g., control flow graph analysis, taint analysis, etc).

Due to the lack of documentation on many tools, it is not possible to obtain specific information like the patterns they identify (at least without exploring the tool source code in deep). Less than 50% have support for Continuous Integration (CI) or can integrate an integrated development environment (IDE).

Overall, the tools are able to identify several different patterns. However, there is a considerable percentage focusing only on the OWASP top 10.

Other systems of classification used by the tools to report the patterns that they identify are the standard dictionaries of weaknesses (CWE[5]) and vulnerabilities (CVE[6]).

To be a good candidate to the study, the tool has to be OSS and needs to accept software from other users. OSS-Fuzz only allows you to submit your projects to review and Codacy only allows you to scan projects for which you contributed. More information will be presented on how to identify a good candidate on section 3.2.

[5] https://cwe.mitre.org/
[6] https://cve.mitre.org/

Table 2.4: Defects databases comparison and overview

Benchmark	Patterns	Languages	Artifical	Real	Vulnerabilities	Observations
Software-artifact Infrastructure Repository (SIR) [DER05]	-	Java, C/C++, C#	✓	✓	N	Most of the test cases are hand-seeded or generated using mutations. It is a repository meant to support experimentation in the software testing domain.
Juliet Test Suites from Center for Assured Software (CAS)[a]	112 (Java), 118 (C/C++)	Java (25,477), C/C++ (61,387)	✓		N	Available through National Institute for Standards and Technology. Each test case has a non-flawed test which will not be caught by the tools and a flawed test which should be detected by the tools.
CodeChecker[b]	-	C/C++	-	-	N	Database of defects which was created by Ericsson with the goal of studying and improving a static analysis tool to possibly test their own code in the future.
OWASP Benchmark[c]	11 (Web)	Web (2,740)	✓		Y	Free and open test suite which was created to study the performance of automated vulnerability detection tools.
Defects4j [JJE14]	-	Java (395)		✓	N	Not only a database but also an extensible framework for Java programs which provides real bugs to enable studies in the software testing research area. The researchers allow the developers to build their framework on top of the program's version control system to add more bugs to their database.
Safety-db [d]	-	Python		✓	Y	Database of vulnerabilities collected from python dependencies. The developers can use continuous integration to check for vulnerabilities in the dependencies of their projects. Data is be analyzed by dependencies and their vulnerabilities or by Common Vulnerabilities and Exposures (CVE) descriptions and URLS.

- means that no information is available to support the fields

[a] https://samate.nist.gov/SRD/testsuite.php
[b] https://github.com/Ericsson/codechecker
[c] https://www.owasp.org/index.php/Benchmark#tab=Main
[d] https://github.com/pyupio/safety-db

Table 2.7: Static Analysis Tools state-of-the-art and comparison (I)

SAT	Languages	Patterns	OSS	CI	IDE	Observations
Infer[a]	Java (Android), C/C++, Objective-C (iOS)	Memory Leaks, Resource Leaks and Null Dereference	✓	✓		Created by Facebook almost two years ago (June 2015). The tool intercepts bugs before mobile applications are shipped to people's phones. There are several companies already using the tool (Facebook, Instagram, Kiuwan, Spotify, Uber and WhatsApp).
Find-Sec-Bugs[b]	Java	OWASP TOP 10 and CWE coverage (113)	✓	✓	✓	Plug-in for the identification of security vulnerabilities in Java web applications. Available for Eclipse and IntelliJ IDEA. It supports CI for Jenkins and SonarQube. Total of 689 unique API signatures.
Symbolic Path Finder[c]	Java	Input Validation and Concurrency	✓		✓	Created at NASA Ames Research Center. It combines symbolic execution with model checking and constraint solving for error detection in Java programs with unspecified inputs. [PR10]
CodePro Analytix[d]	Java and XML	225 security audit rules		✓	✓	It is possibly the oldest SAT founded in this research. It was created in 2005 by Google. It also supports JSP, JSF, Struts and Hibernate. The tool was donated to Eclipse.
OSS-Fuzz[e]	C and C++	Focus on overflow	✓	✓		Uses fuzz testing to uncover defects. Since they deployed the tool, which was less than three months ago (December 2016), they already have found hundreds of security vulnerabilities.
Bandit[f]	Python	-	✓			Tool that finds security vulnerabilities in Python code using Abstract Syntax Trees (AST). For each file, builds an AST and runs appropriate plug-ins against AST nodes. Based on their to-do list, it still has a lot of open doors for improvement.
Safety[g]	Python	OWASP TOP 10	✓	✓		It started on october 2016 but it already gives reports of found security vulnerabilities in the dependencies of a Python project.

- means that no information is available to support the fields

[a] http://fbinfer.com/
[b] http://find-sec-bugs.github.io/
[c] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[d] https://developers.google.com/
[e] https://github.com/google/oss-fuzz
[f] https://wiki.openstack.org/wiki/Security/Projects/Bandit
[g] https://github.com/pyupio/safety

Table 2.8: Static Analysis Tools state-of-the-art and comparison (II)

SAT	Languages	Patterns	OSS	CI	IDE	Observations
Brakeman[a]	Ruby on Rails	> 25 security patterns	✓			It scans ruby on rails application in order to find web security vulnerabilities. This tool has already a limitation reported by the core developers: the high number of false positives. Although already identified as a tool with a few limitations, it will be evaluated together with the others.
Dawnscanner[b]	Ruby on Rails	235 security checks	✓			It scans the source code of ruby projects (gems) for security vulnerabilities. It is able to scan some Model View Controller frameworks like Ruby on Rails, Sinatra and Padrino. The last version has 235 security checks loaded in their knowledge base. Most of them are from Common Vulnerabilities and Exposures (CVE). The core developers are preparing checks from OWASP Ruby on Rails cheat sheet.
RevealDroid[c]	Android(Java)	Android security issues	✓			Detects security vulnerabilities in Android Applications (Java) using a machine-learning approach based on reflection features and some ski-kit learn-based functionalities. It is being developed since December of 2016 and there is already an available version.
QARK[d]	Android(Java)	Android security issues	✓			It was created by LinkedIn and is a Quick Android Review Kit. This tool identifies in source code and packages APKs several security issues related to Android applications vulnerabilities. It is capable of exploiting many of the vulnerabilities it finds.
Checkstyle[e]	Java	Code style issues	✓			Checks code style of Java programs which can lead to serious security risks, e.g., gotofail vulnerability.
Androwarn[f]	Android(Java)	DOS	✓			It was developed academically and is a static analyzer for malicious Android applications. They used Smali[g] to perform detection. Some of their targets are denial of service, PIM data modification, geolocation information leakage, remote connection establishment, etc.

- means that no information is available to support the fields

[a] http://brakemanscanner.org/
[b] https://github.com/thesp0nge/dawnscanner
[c] https://bitbucket.org/joshuaga/revealdroid
[d] https://github.com/linkedin/qark
[e] https://github.com/codacy/codacy-checkstyle
[f] https://github.com/maaaaz/androwarn/
[g] https://github.com/JesusFreke/smali

Table 2.9: Static Analysis Tools state-of-the-art and comparison (III)

SAT	Languages	Patterns	OSS	CI	IDE	Observations
Amandroid[a]	Android(Java)	Sensitive Data	✓	✓		SAT for android applications. It does context-sensitive data flow analysis in an inter-component way to find security vulnerabilities in the software. These vulnerabilities are the result of interactions among multiple components from either the same or different applications.
CFGScanDroid[b]	Android(Java)	Virus	✓			It compares the control flow graph (CFG) signatures with the control flow graphs of Android methods. It was created with the goal of identifying malicious applications.
Opa[c]	Opa	XSS and Injection	✓			It includes its own static analyzer. It is a language intended for web application development. Opa's compiler checks the validity of high-level types for web data and prevents by default many vulnerabilities such as XSS attacks and database code injections.
Frama-C[d]	C	-	✓			Several analysis techniques in a single platform, consisting of a kernel providing a core set of features plus a set of analyzers - the plug-ins. Plug-ins can build upon results computed by other plug-ins in the platform.
Sparse[e]	C	Linux Kernel	✓			Semantic parser that provides a compiler capable of parsing ANSI C and GCC extensions; and, a compiler capable of perform static analysis
Splint[f]	C	-	✓			Open-source version of Lint, for C. It's a tool that checks C programs for security vulnerabilities and coding mistakes.
FlawFinder[g]	C	Buffer Overflow	✓			Examines C source code and reports possible security weaknesses sorted by risk level.

- means that no information is available to support the fields

[a] https://github.com/sireum/amandroid
[b] https://github.com/douggard/CFGScanDroid
[c] http://opalang.org/
[d] https://github.com/Frama-C/Frama-C-snapshot
[e] https://sparse.wiki.kernel.org/index.php/Main_Page
[f] https://github.com/ravenexp/splint
[g] https://sourceforge.net/projects/flawfinder/

Table 2.10: Static Analysis Tools state-of-the-art and comparison (IV)

SAT	Languages	Patterns	OSS	CI	IDE	Observations
Cppcheck[a]	C/C++	Memory Leaks and Null Pointer Dereferences	✓			It does not detect syntax errors in the code but types of bugs that the compilers normally do no detect. It detects memory leaks, null pointer dereferences, unsafe functions suspicious code, etc.
OWASP Dependency-Check[b]	.Net, Java, Ruby, Node.js, Python and C/C++	Dependencies	✓			It detects vulnerabilities within project dependencies. It does this by determining if there is a Common Platform Enumeration (CPE) identifier for a given dependency. If found, it will generate a report linking to the associated CVE entries.
Coverity[c]	Android, C, C++, Objective-C, C#, Java, JavaScript, PHP, Python, Node.js, Ruby, Fortran and Swift	OWASP TOP 10	✓	✓	✓	Created by Synopsys. It finds critical defects and security weaknesses in code. It is capable of finding security vulnerabilities like injection, broken authentication and session management, Cross-Site Scripting (XSS), Insecure Direct Object, etc.
Kiuwan[d]	Java, JavaScript, PHP, SAP, COBOL, C++, Objective-C, Android, iOS, Python, PL/SQL	OWASP and CWE		✓	✓	Provides an action plan with the defects and security flaws that need to be fixed. The code analysis checks code logic.
VCG[e]	C/C++, Java, C#, VB and PL/SQL	Command injection, int overflow, XSS, Android issues	✓			Automated code security review tool. It looks for security vulnerabilities based on comments that indicate broken or unfinished code pieces. It also analyzes code and dangerous functions.

- means that no information is available to support the fields

[a] http://cppcheck.sourceforge.net/
[b] https://github.com/jeremylong/DependencyCheck
[c] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
[d] https://www.kiuwan.com/code-analysis/
[e] https://github.com/nccgroup/VCG

Table 2.11: Static Analysis Tools state-of-the-art and comparison (V)

SAT	Languages	Patterns	OSS	CI	IDE	Observations
RIPS[a]	PHP	40 different types		✓		Tool written in PHP to find security vulnerabilities in PHP applications. By parsing all source code files, the tool transforms the source code into a program model and detects potentially vulnerable functions that can be contaminated by a malicious user during the program flow. You need to pay to have access to the use the standards security checks.
WAP[b]	PHP	8 different types	✓			Tool semantically analyses the source code. More precisely, it does taint analysis (data-flow analysis) to detect the input validation vulnerabilities. After the detection, the tool uses data mining to confirm if the vulnerabilities are real or false positives. In the end, the real vulnerabilities are corrected with the insertion of the fixes (small pieces of code) in the source code.
Codacy[c]	JavaScript, Scala, Java, PHP, Python, CoffeeScript, CSS, Ruby, Swift and C/C++	9 different types	✓	✓		This tool allows you to check your Github and Bitbucket projects in order to find issues. It gives you notifications during the development when a bad commit appears. Checks for security issues for authentications, denial-of-service, HTTP, XSS, SQL injection, validation input and many more.
Garcon[d]	PHP	SQL injec, cmd injec, XSS	✓			It uses Taint Analysis.

- means that no information is available to support the fields

[a] https://sourceforge.net/projects/rips-scanner/files/
[b] http://awap.sourceforge.net/
[c] https://www.codacy.com/
[d] https://github.com/vesuppi/Garcon

Chapter 3

Methodology

This chapter presents in detail the methodology used to prove the thesis hypothesis: *It is possible to identify points of improvement on a static analysis tool using real security vulnerabilities*. Starting with the collection of the patterns; then the mining process and the steps associated with the resulting samples evaluation; the identification process behind the chosen test cases used on the SAT study; and, finally the evaluation of the SAT results to find points of improvement on the tool under study.

3.1 Identification and Extraction of Vulnerabilities from Github

This section describes the methodology used to obtain real security vulnerabilities: from the mining process to the samples evaluation and approval.

The main goal with this approach is the identification and extraction of real security vulnerabilities fixed naturally by developers on their daily basis work. The research for new methodologies to retrieve primitive data in this field is really important due to the lack of databases with a considerable amount of test cases and lack of variety for different defects and languages to support SATs studies. Since there is no database that can support this study, we will have to create ours.

The first step was the identification of a considerable amount of trending security patterns (sec. 3.1.1). Initially, the main focus was the top 10 OWASP 2017 and other trending security vulnerabilities like memory leaks and buffer overflows which are not much prevalent between web applications. After that, more patterns were added and there is still place for many others. For each pattern, there is a collection of words and acronyms which characterizes the security vulnerability. These words were mined on the commits messages, to find possible candidates to test cases. Every time the tool catches a pattern, it saved the sample on the cloud and the information attached (e.g., identifier, type of security vulnerability, etc) on the database. As we can see on figure 3.1, after saving the data there is an evaluation process (3.1.3) to validate whether the caught sample represents the fix of a potential security vulnerability or not. When approved,

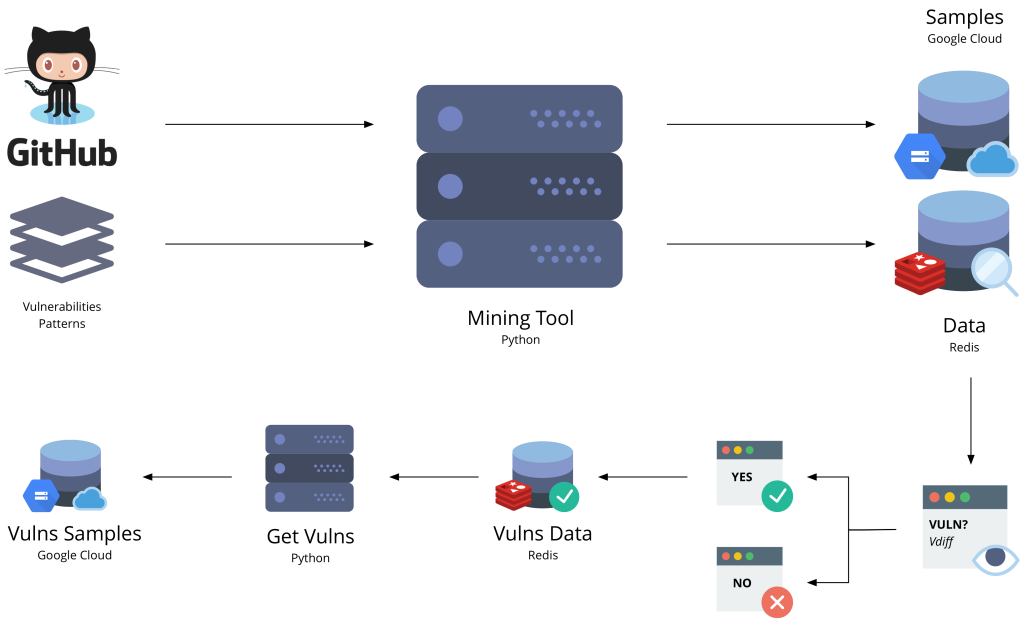

Figure 3.1: Workflow to extract and identify real security vulnerabilities

the sample's information is updated on the database and, consequently, the test case (3.1.2) is added to the final database.

The resulting test cases were organized based on the security vulnerability patterns and the source code language.

3.1.1 Security Patterns

The goal was mining for indications of a vulnerability fix or patch committed by a developer on Github repositories.

Assuming that the trendiest security vulnerabilities will be found more easily on Github we tried to understand if the current security reports actually reflect in the domain. So, the Github search engine was used to see if overall it would be possible to collect information for the chosen patterns. The number of commits for each pattern was checked on the engine which instantly led to a good perception of what patterns would be more difficult to extract.

Then, for each pattern, a regular expression was created joining specific words from its own domain and words highlighting the fix of a vulnerability. To represent the fixing of a vulnerability, words like *fix, patch, found, prevent* and *protect* were used. In certain cases, like the pattern *iap* (Tab. 3.1), it was necessary to change the approach due to the nature of the vulnerability. This pattern represents the lack of automated mechanisms for detecting and protecting applications. So, instead of the normal set, another words were used: *detect, block, answer* and *respond*. It was necessary to adapt the words to each type of vulnerability. To really specify the patterns and

distinguish between them more specific words were added. For example, to characterize cross-site scripting vulnerability tokens like *cross-site scripting*, *xss*, *script attack* and many others were used. Each regular expression can be visualized on the thesis' website[1].

The final set of security patterns used on this study is presented on table 3.1 and table 3.2. The first group represents the top 10 OWASP 2017 and the second one is mainly non-web security vulnerabilities, i.e., which you can see more on applications using programming languages like C, C++ and Java.

ID	**Pattern**
injec	Injection
auth	Broken Authentication and Session Management
xss	Cross-Site Scripting
bac	Broken Access Control
smis	Security Misconfiguration
sde	Sensitive Data Exposure
iap	Insufficient Attack Protection
csrf	Cross-Site Request Forgery
ucwkv	Using Components with Known Vulnerabilities
upapi	Underprotected APIs

Table 3.1: Top 10 OWASP 2017

ID	**Pattern**
ml	Memory Leaks
over	Overflow
rl	Resource Leaks
dos	Denial-of-Service
pathtrav	Path Traversal
misc	Miscellaneous

Table 3.2: Other Security Issues/Attacks

3.1.2 Test Cases Structure

A mining tool was implemented aiming the search of security patterns and resulting information to produce and evaluate candidates to test cases. More detailed information about the mining tool will be provided on chapter 4.

Every time a pattern was found in a commit, a candidate to a test case was added to the main database. Due to the way of how Github is structured, it is easy to track the differences between each pair of commits. This methodology considers the commit containing the pattern as the fix of the vulnerability and the previous one as the actual vulnerability, i.e., the sample containing the

[1] https://tqrg.github.io/secbench/patterns.html

source code to be scanned by a SAT. Each test case has 3 folders: V_{fix} with the non-vulnerable source code from the commit where the pattern was identified (child), V_{vul} with the vulnerable source code from the previous commit (parent) which it is believed to be the real vulnerability; and, V_{diff} with two folders, added and deleted, containing the added lines to fix the vulnerability and the deleted lines representing the security vulnerability (Fig. 3.2).

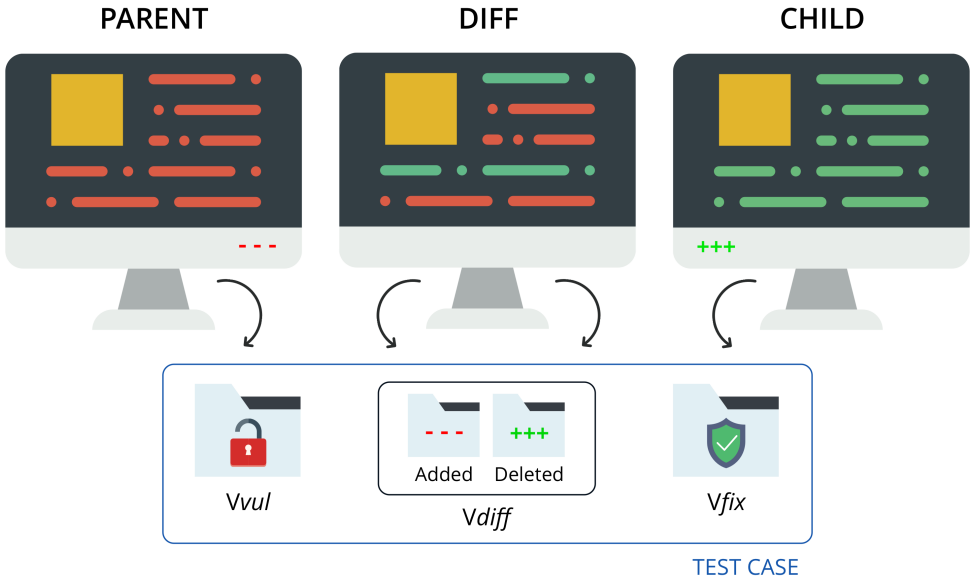

Figure 3.2: Difference between V_{fix}, V_{vul} and V_{diff}

3.1.3 Samples Evaluation and Acceptance Criteria

After obtaining the sample and its information, a manual evaluation was performed on the diff between V_{fix} and V_{vul}. For every single case, firstly it was necessary to evaluate if the message really reflected indications of a vulnerability fix because some of the combinations represented by the regular expressions can lead to false positives, i.e., messages that do not represent the actual vulnerability fix. The fact that the same word can mention different things led to more garbage, i.e., commits referring other things but using the same words. If the conclusion from the first stage (message evaluation) is *No*, then the test case was marked in the database as a non-viable test case. But if the analysis succeeds, the code evaluation is performed through the diff source code analysis. Hopefully, the researcher is capable of isolating manually the functions or problems in the code responsible for the fix and the vulnerability. During the study, several cases were inconclusive, mainly due to the difficulties in understanding the code structure or when the source code did not reflect the message. Normally, these last cases were marked has non-viable, except when there was something that could be the fix but the researcher did not get it. In that cases, they were put on hold as a *doubt* which means that the case needs more research.

To validate the source code much research was made on books, security cheatsheets online, vulnerabilities dictionary websites and many others sources of knowledge. Normally, the process

would be giving a first look on the code trying to highlight a few functions or problems that could represent the vulnerability and then make a search on the internet based on the language, frameworks and information obtained by the diff.

Besides the validation, the set of requirements presented below needs to be fulfilled, in order to approve a test case as viable to the final test suite:

- **The vulnerability belongs to the class where it is being evaluated**

  If it does not belong to the class in evaluation, the vulnerability is put on hold for later study except if the class under evaluation is the miscellaneous class which was made to mine vulnerabilities that might not belong to the other patterns; or, to catch vulnerabilities that may skip in other patterns due to the limitations of using regular expressions in this case.

- **The vulnerability is isolated**

  We accepted vulnerabilities which additionally include the implementation of other features, refactoring or even fixing of several security vulnerabilities. But the majority of security vulnerabilities founded are identified by the files names and lines where they are positioned. We assume all V_{fix} is necessary to fix the security vulnerability.

- **The vulnerability needs to really exist**

  Each sample was evaluated to see if it is a real vulnerability or not. During the analysis of several samples, a few commits were not related to security vulnerabilities and fixes of vulnerabilities, i.e., no real fixes were identified.

3.1.4 Database Manual Evaluation Limitations

Sometimes, developers have hard-times when trying to understand the code from other developers. This is easily explained by the high variety of APIs, frameworks, languages and libraries available to produce software. So, it is safe to say that this methodology will benefit from people with experience in using different components. The requirements were all evaluated manually, hence a threat to the validity as it can lead to human errors (e.g., bad evaluations of the security vulnerabilities). However, we attempted to be really meticulous during the evaluation and when we were not sure about the security vulnerability nature we evaluated as a *doubt* and as a *replica* when we detected a replication of another commit (e.g., merges or the same commit in another pattern). Sometimes it was hard to reassign the commits due to the similarity between some patterns (e.g., *ucwkv* and *upapi*) on OWASP top 10. Another challenge was the trash (i.e., commits that did not represent vulnerabilities) that came with the mining process due to the use of regular expressions. Garbage (non-viable test cases) represent more than 50% of the mining process outcome. We only used regular expressions because of time restrictions.

3.2 SATs Evaluation and Study

The second stage of the methodology corresponds to the SATs study which is based on the CAS Static Analysis Tool Study [fAS11] but instead of artificial test cases, real ones were used, i.e., test cases representing naturally occurring security vulnerabilities. The similarity between this methodology and the one presented by the National Security Agency is the group of metrics used to evaluate the tool: precision, recall and f-score (Sec. 1.3).

In the beginning, it was necessary to do a state-of-the-art study (Sec. 2.5.4) to understand the characteristics of these tools. Later on, some points were taken in consideration to evaluate tools as good candidates to this study (Sec 3.2.1).

After obtaining a good candidate, the vulnerabilities support and models were studied to identify good test cases to study the tool, i.e., good representatives of what vulnerabilities the tool is able to detect (not only at pattern level but also specificity). It does not matter if it is a memory leak if the allocation function is not part of the models.

Leaning on this, each test case suffered a deep manual evaluation to identify possible vulnerabilities on the diff between the two commits. Some test cases were not only representative of one vulnerability. Actually, the percentage of test cases containing more than one vulnerability is lower than the percentage of test cases representing several vulnerabilities. So, it was necessary to identify and isolate the possible and suitable vulnerabilities on the vulnerable source code.

Then, depending on the tool the source code is built and sent to the static analyzer (tool). The tool reports the detected vulnerabilities and another manual evaluation is performed by a researcher to understand if the tool did or did not report the vulnerabilities or if it reported *false alarms*, i.e., non-existent vulnerabilities. Each result is evaluated as *TP*, *FN* and *FP* (Sec. 1.3).

Based on these results, the metrics mentioned before can be calculated, and the tool's quality can be determined.

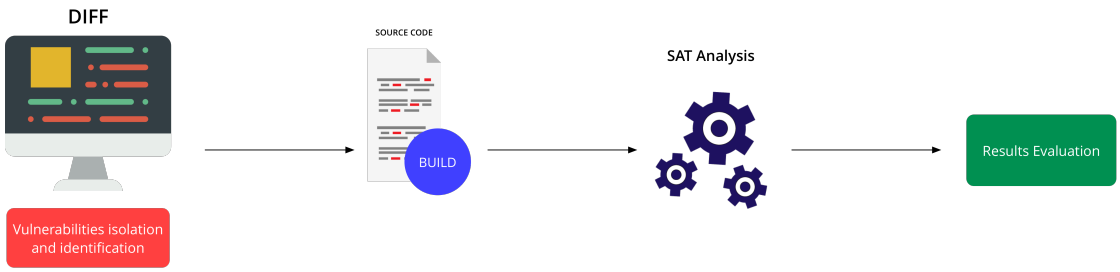

Figure 3.3: SATs study methodology overview

3.2.1 Identifying Good Candidates

To identify static analysis tools that would suit well this methodology, a set of requirements needs to be fulfilled:

- **The SAT must be open-source**

  Since this methodology tries to study SATs, it is indispensable to have access to the code to better understand the models and flows each tool identifies. Due to the primitiveness of these tools, the core developers do not waste too much time on producing documentation, or if it exists it is more an overview than a complex description of what researchers need to know. So it is hard to have a deeper knowledge of how the tool operates and what it could be expected when running a code sample.

- **The SAT must detect vulnerabilities that can be found on Github repositories**

  SATs are being studied using test cases mined from Github. Thus, it is necessary to ensure a minimum set of test cases to be used on the tool study. Due to the vulnerabilities specificity caught by the tool, test cases may not be available.

- **The SAT must be used by a considerable amount of people**

  If the final goal of this methodology is to decrease the costs involved in the identification of vulnerabilities and improve developers lives, then it is mainly important to study and improve tools that are already being used by a good number of developers.

Other than analyzing if the SAT s fulfil the list of requirements, they were also organized and chosen by their popularity, companies support (e.g., Facebook, Google, Spotify and many others), contributors community and the most important the relevance of the vulnerabilities caught. It is important to start with vulnerabilities that have a high impact on the security field since the impact will be higher.

3.2.2 Identifying Vulnerabilities and SAT Results Evaluation

As mentioned before, the group of suitable test cases goes through a deep evaluation in order to isolate possible vulnerabilities on the source code. So, for each type of vulnerability under evaluation, the possible vulnerabilities are highlighted. As you can see on figure 3.4, the socket identifier is no longer available after line 182 and line 190, so it is necessary to close the socket before the function returns, in order to tackle the two resource leaks identified. This is the process used for all the samples before being analyzed with the SATs .

The example presented below is easy to identify. It was not always like this, sometimes it was really difficult to understand where the issues were due to the source code complexity. The source code quality really matters.

Figure 3.4: Example of several vulnerabilities identification

After isolating the vulnerabilities, the sample is scanned with a SAT and the results are also manually evaluated. Based on the comparison between the expected vulnerabilities and the actual results we evaluate each case as *FP* if a vulnerability is caught but it does not exist; *FN* if the tool should have caught the vulnerability but it didn't; and, *TP* if the expected vulnerability was identified. These results were used on the metrics calculation which determines if the tool is reliable (*precision*), sensitive (*recall*) and, mainly, good or bad (*f-score*).

3.2.3 SATs Evaluation Limitations

When identifying security vulnerabilities on the test cases, some complex examples were founded which were not evaluated by a SAT even when it would suit the model's tool. The code structure and external components sometimes did not help on understanding if the tool would catch the vulnerabilities or not, so not all test cases from the test suite were used to test the tool. Some tools build the code before the analysis, so the test cases need to build successfully. The problem is that a few samples were not built successfully due to original software problems. This was an issue because sometimes it was difficult to understand if the problem was from the sample or from the computer used to perform the analysis. This was solved through research on the internet; GitHub repositories and issues. This led to the decrease of available and suitable test cases to perform the study.

The results of the SATs sometimes were not well presented, so it was simply difficult to know how to classify the result. In that times, we dropped the test case from the SAT study. One example, was the SAT pointing the wrong function which nothing had to do with the real vulnerability. This case was not a *false alarm* and other cases[2] like this were reported on the tool GitHub repository issues. The bad interfaces make the study and their application harder.

[2] https://github.com/facebook/infer/issues/648

The specificity of the vulnerabilities scanned by a SAT was also a limitation. For example, it was very hard to find samples to study the `Find-Sec-Bugs` tool since it caught very specific functions and CWE classes. In the sample presented in this study, 0 CWE identifiers were identified. Not successfully or even unsuccessfully. To find vulnerabilities to study this tool, it will be necessary to have other means of identifying such specificity. Maybe checking the import files would be a great path, since developers on Github, normally do not detail so much the commits' messages.

3.3 SATs Modernization

The last stage of the methodology was to evaluate if it is possible to obtain points of improvement within the results of the studied test cases.

Analyzing the SAT results and the information the tool provides online (e.g., source code, documentation, etc), we tried to understand why the tool did not work and how to solve it. It is really important to understand well how the tool really works and how the identification process is made (e.g., `Infer` uses Hoare Logic and Interprocedural analysis). Hopefully, with the problems caught using this methodology, improvements and contributions can be done on the studied tool were we had positive outcomes.

3.4 Conclusions

As observed in the previous sections, this is a very meticulous and long methodology with the potential to succeed but at the same time with several limitations associated. Being the major and probably the most dangerous the high dependency on humans. Github can also be a problem, due to the way of how it is structured. Researchers are not able to identify the best repositories to mine and with the higher quality. Due to its dimension, it is pretty hard to get that kind of information.

It is important to have a benchmark with real security vulnerabilities, but it is also necessary to improve the way of how we identify vulnerabilities since it is too much time-consuming and dependent on humans which inserts high rates of errors.

Chapter 4

Tools and Database Implementation

To support this research, a few components were implemented to save, manage and collect information. First, a database was designed to host the information provided by GitHub. Then, a mining tool was created to extract data. To validate data and obtain non-inflated data, several scripts were created. And, finally, a website was created to present the statistical results.

In the next sections, all of these components will be presented in detail: database structure, mining tool, data validation and data visualization.

4.1 Overview

To perform the study, two different applications were created to communicate with the database:

- `secbench-mining-tool`[1]: Mining tool responsible for extracting information from Github based on a chosen pattern. This tool is responsible for the data collection and its organization.

- `secbench`[2]: Website containing all the statistical results and scripts responsible for validating data. (*Note: At the moment, only the patterns information is available. The website is running locally, but it will be deployed soon.*)

Both communicate with the database to insert and retrieve data which will be used to obtain useful information (Fig. 4.1). For mining Github, there are two different scripts. One to mine repositories information and other to mine patterns on repository commits. `secbench` is responsible for retrieving and provide information in a visual. Validation was a major concern to make sure that the retrieved data was the most reliable as possible.

[1] https://github.com/TQRG/secbench-mining-tool
[2] https://github.com/TQRG/secbench

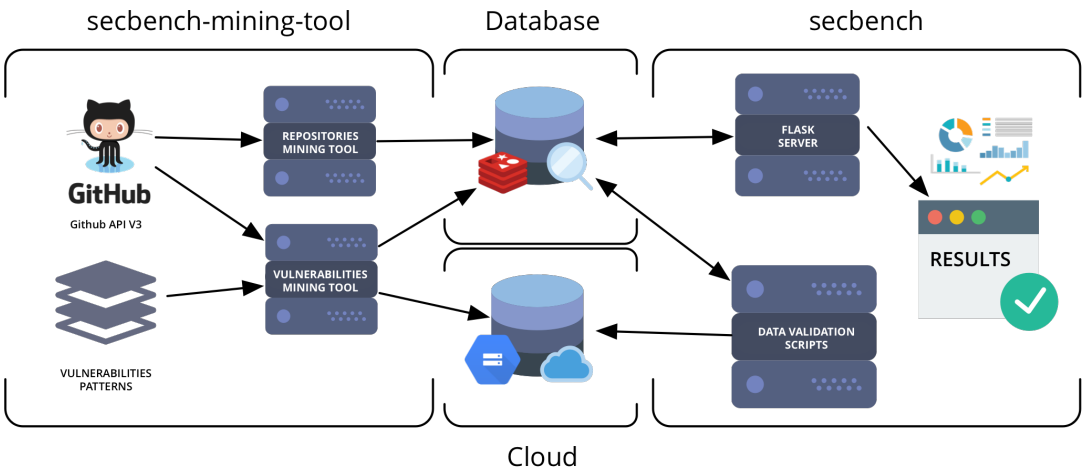

Figure 4.1: Different modules and applications produced to obtain and visualize the results

The majority of technologies used in this thesis are OSS, and the final product will also be OSS. The final versions of the software will be all available on Github. The resulting benchmark is under an MIT License. A wide range of technologies was used to create the components:

- Database
  - `Redis`[3]: Advanced key-store value datastore written in C (very fast). It supports several different data types and handles clients concurrency well. It has one of the most efficient cache management systems within NoSQL databases.
- Mining Tool and Data Validation
  - `Python`[4]: Programming language with a large standard library containing several packages for data mining and statistics.
  - `Github API`[5]: The connection between Github and the application was made through two different packages `GitPython`[6] to download the code samples and `PyGitHub`[7] to obtain the data provided by the API.
  - `Google Cloud Storage API`[8]: The samples of code mined from Github were stored on Google Cloud.
- Data Visualization
  - `D3.js`[9]: `JavaScript` library used to manipulate and visualize information.

[3] https://github.com/antirez/redis
[4] https://www.python.org/
[5] https://developer.github.com/v3/
[6] https://github.com/gitpython-developers/GitPython
[7] https://github.com/PyGithub/PyGithub
[8] https://cloud.google.com/storage/docs/reference/libraries
[9] https://github.com/d3/d3

- `Flask`[10]: Flask is a microframework for `Python` and was used to create the web server behind the website.
- `SciPy`[11]: Group of several `Python` packages used for statistics.

4.2 Database Structure

Currently, the database contains almost 25*K* keys with different types of information. Here, the structure of the database is presented in detail. `Redis` allows you to design and store your data using different data types: *strings*, *hashes*, *lists*, *sets* and *sorted sets*.

4.2.1 Repository

Each repository is represented by a *hash* (map between string fields and string values) with a key identifier equal to:

repo:<repository_owner>:<repository_name>:<repository_status>

where <*repository_owner*> is the name of the owner of the project, <*repository_name*> is the name of the repository and <*repository_status*> is the status of the project which may be **n** (5*K* keys), if it has no problems; **i** (95 keys), if it is empty or blocked; and, **s** (27 keys), if it has less than two commits.

For each repository in general, there are three different fields: *owner* = <*repository_owner*>, *name* = <*repository_name*> and *status* = <*repository_status*>. The repositories with status equal to **n** have more fields than the ones mentioned before (Tab. 4.1).

Field	Value
started	Datetime of the first commit
last_update	Datetime of the last commit
forked?	Is the repository a fork? **Y**es or **N**o
forked_from	Owner and name of the original repository
dev_time	Difference between the first commit and last one (seconds)
branch	Branch name
commits	Number of commits
languages	JSON with all languages and respective BOC

Table 4.1: Repository hash fields

In order to track the patterns mined for each repository, a *list* was created:

class:<repository_owner>:<repository_name>

[10] http://flask.pocoo.org/
[11] https://www.scipy.org/

where a class was added every time it was mined.

For each language (*set*) present in the database, it is possible to see all the repositories which have it in their languages. The key representing a language is:

lang:<language>

where *<language>* is a programming language like Java, JavaScript, C, etc. The keys added to this *set* are the keys identifying the repositories.

4.2.2 Commit

A commit is represented by a *hash* with a key identifier equal to:

commit:<repository_owner>:<repository_name>:<commit_sha>:<pattern>

where *<repository_owner>* is the name of the owner of the project, *<repository_name>* is the name of the repository and *<commit_sha>* is the commit sha key and *<pattern>* is the type of security vulnerability (e.g., *xss*, *csrf*, etc).

There are more than 7.5*k* commits on the database where almost 6*k* are already studied. The fields each commit contains are several (Tab. 4.2).

Field	Value
lang	Language of the file where is the vulnerability
year	Year of the commit
class	Type of vulnerability
message	Commit message
match	Commit match between the message and the regular expression
vuln?	Has the commit a vulnerability? **Y**es, **N**o, **D**oubt or **R**eplication
sha-p	Commit parent
commit_url	URL of the vulnerability
id_f	ID on Google Cloud folder
code	If it is an identified vulnerability by CVE or CWE
observations	Observations of anything about the commit

Table 4.2: Commit hash fields

4.2.3 Tool Results

For each tool is created a *list* with its own name as a key. For example, the key for finding `infer` results keys is **infer**. If you do,

lrange 0 -1 infer

it will retrieve the keys of all vulnerabilities studied for infer. Each vulnerability is represented with a *hash* with the next structure:

vulns:<repository_owner>:<repository_name>:<commit_sha>:<pattern>:<id>

where <*repository_owner*> is the name of the owner of the project, <*repository_name*> is the name of the repository and <*commit_sha*> is the commit sha key, <*pattern*> is the type of security vulnerability (e.g., *xss*, *csrf*, etc) and <*id*> is the number of the vulnerability in the commit.

The fields that were saved for each vulnerability is presented below (Tab. 4.3):

Field	Value
line	Line where the vulnerability is
file	Path of the file which has the vulnerability
lang	Vulnerability language
observations	Normally, the explanation of why it is a vulnerability
id_f	Identifier on Google Cloud
res	Tool report **TP**, **FP** or **FN**

Table 4.3: Vulnerability hash fields

4.2.4 Experiments

Since different experiments were made, a *hash* was created in order to track the database inserts. The key for an experiment is:

exp:<pattern>:<repository_owner>:<repository_name>:<datetime>

where <*pattern*> is the type of security vulnerability (e.g., *xss*, *csrf*, etc), <*repository_owner*> is the name of the owner of the project, <*repository_name*> is the name of the repository and <*datetime*> is the date and time when the experiment ended.

The information tracked for each experiment (Tab. 4.4):

Field	Value
c_time	Datetime when the experiment ended
class	Class for what it was mined
repo_owner	Repository owner name
repo_name	Repository name
time_spent	Time spent on mining the class
n_vuls	Number of vulnerabilities caught

Table 4.4: Experiment hash fields

4.3 Mining Tool

The mining tool allowed to retrieve the necessary information from Github, to create our own database. This tool communicates with several external components: the `Github API` through two python packages (`GitPython` and `PyGithub`); the database in `Redis` where is the information retrieved and the `Google Cloud Storage API` to save the test cases.

This tool has two main scripts:

- *get_github_repositories.py*: Script responsible for collecting Github repositories and adding their information to the database. The mining can be done iteratively starting on a Github page and mining *x* repositories from there. You can search for repositories with the string "Language: <language>" and obtain *x* repositories in the language you want. Or, you can add a specific repository to the database using its owner and name.

- *repos_miner.py*: Script responsible for collecting vulnerabilities based on a pattern. You can mine repositories for all patterns or select a specific pattern. This script collects information only for the repositories on the database.

4.3.1 Mining Repositories

The mining tool iterates all the commits of each repository and matches the message of each commit with a regular expression representing a security pattern. If the match succeeds, it means that a vulnerability fix was caught. If the vulnerability is not already in the database, it is added to the database, and the sample is sent to a bucket on the cloud. As seen in figure 4.2, if a vulnerability fix is caught on a commit, then the real vulnerability will be on the parent. This is what we believe.

Each pattern mined for a repository is considered as a new experiment.

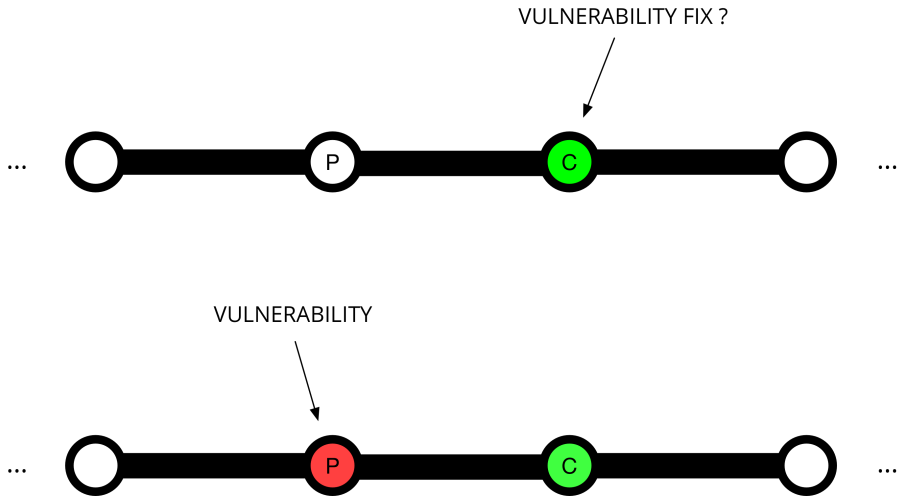

Figure 4.2: Github structure and vulnerabilities detection

There is one limitation associated with the tool due to how Github is structured. When a vulnerability is caught on a merge, the tool is not able to detect which of the parents is the one who has the vulnerability (Fig. 4.3). Research was made and no infallible solution was found. Thus, through the manual evaluation, the parent is checked in order to address the issue and correct *sha-p* value.

```
4 parents cc7e35b + f792943 + 0b0cda4 + 7b821a6   commit 082f6968bb204d1a3d8b2da3c53d6b7a59bbd985
```

Figure 4.3: Merge of more than one commit at the same time

Merges cannot be dropped from the research since in some repositories reflect the correction of bad commits descriptions, i.e., when one of the core developers reviews the code and submits it, it adds specific messages for each fix. This conclusion was taken from the manual analysis of several merges and its parents.

4.3.2 Test Cases Structure on the Cloud

The test cases in the cloud are separated by patterns. Inside of each pattern (e.g., *auth*), there is a folder for each language (e.g., *C*) where the pattern was caught. Then, the user can check the repositories (e.g., *google_google-authenticator*) inside that provided viable test cases following the structure presented in the previous chapter (Fig. 3.2).

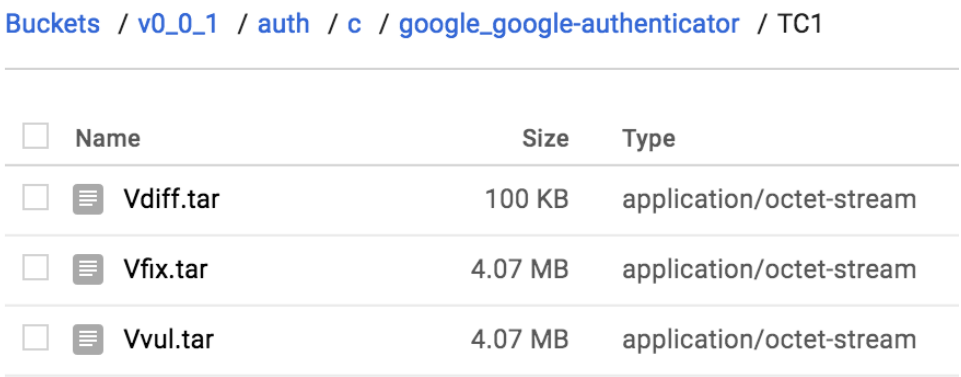

Figure 4.4: Difference between V_{fix}, V_{vul} and V_{diff} on the cloud

4.4 Data Validation

Every single commit studied on our database went through a manual evaluation which was one of the main stages of data validation where all the information was confirmed and complete. To maintain the database and cloud data clean and useful, a few validation methods were used:

- Consistency checks on fields where the values were previously defined (e.g, *vuln?*, *code*, etc).

- Due to the limitation presented in fig. 4.3, the test cases were manually evaluated to see which of the parents could be the one containing the vulnerability.
- File existence check: some test cases were missing files due to the non-correct handling of exceptions in the early stages.
- A few times we corrected the missing packages on the cloud.
- Every two weeks, we ran a script to clean the non-viable samples from the cloud.
- Cardinality checks: Scripts to check if the information is in the right number or if there is any garbage ruining the data (e.g., if the total number of accepted vulnerabilities is equal to the sum of vulnerabilities for each pattern on the database)

4.5 Data Visualization

To visualize data, a website was created with a web server in `Flask`. Several requests were created to obtain data from the database. `D3.js` was used to create different types of graphics (Fig. 4.5) with this information: bar charts, bubble charts, stacked bar charts and scatterplots with regression lines.

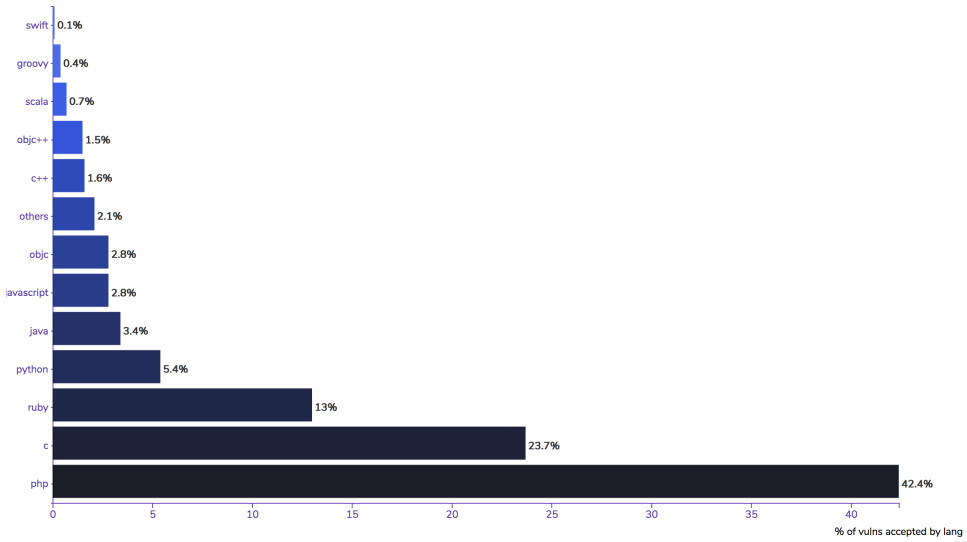

Figure 4.5: Example of bar chart uisng `D3.js`

Chapter 5

Empirical Evaluation

In this section, the resulting statistics of putting the methodology in practice are reported and discussed. Here, it is possible to have an overview of the database characteristics, a few interesting relationships between variables and the statistical reports of the tools studied.

Under this empirical evaluation, there is a sample from the final database which contains the mined and evaluated content for the higher viable number of patterns, i.e., patterns which were successfully able to retrieve test cases. There is a group of repositories that is not yet mined for all the patterns, so they are not being considered but will possibly integrate the final benchmark in the future.

The first version of the benchmark is already available on `https://github.com/TQRG/secbench` containing all the test cases collected from 238 projects. The sample used for the next evaluation has a few more test cases which were the result of mining 10 more repositories.

5.1 SecBench Report

This section provides an overview of the sample characteristics used to the empirical evaluation and all the interesting information and conclusions collected from the study. This sample belongs to the main database whose name is *SecBench* (the junction of **Sec**urity and **Bench**mark) and which already counts with more than 700 accepted vulnerabilities and almost $6K$ mined commits manually evaluated.

The first entry in this database was on March 25 and the last one on June 19 with a mean of mining hours per day equal to 6 hours and 21 minutes.

5.1.1 How was the database sample chosen?

To provide reliable and helpfull information about the data collected through the last months, a sample was chosen based on one requirement:

- **All the chosen repositories must be mined for all patterns which retrieved viable information**, i.e., only repositories mined for all the patterns with more than one vulnerability accepted were taking into account.

For example, the pattern representing context leaks, which is an Android vulnerability, was not taking in consideration for this study since after 248 mined repositories the resulting accepted vulnerabilities were 0.

To obtain a sample which could be a good representative of the population under study, it was ensured the top 5 of most popular programming languages on Github and different sizes of repositories were covered. Due to the large amount of Github repositories ($61M$) and permanent modification, it is very complicate to have an overall of the exact characteristics that the sample under study should have in order to, approximately, represent Github.

The chosen sample supports more than 94 programming languages where the top 5 (JavaScript, Java, Python, Ruby and PHP) was definitely covered, as you can see on figure 5.1 where only the languages with more than $10M$ of bytes of code (BOC) are presented.

The Github API does not allow to retrieve the number of commits in each language for each repository. Instead, it retrieves the number of BOC written in each repository language.

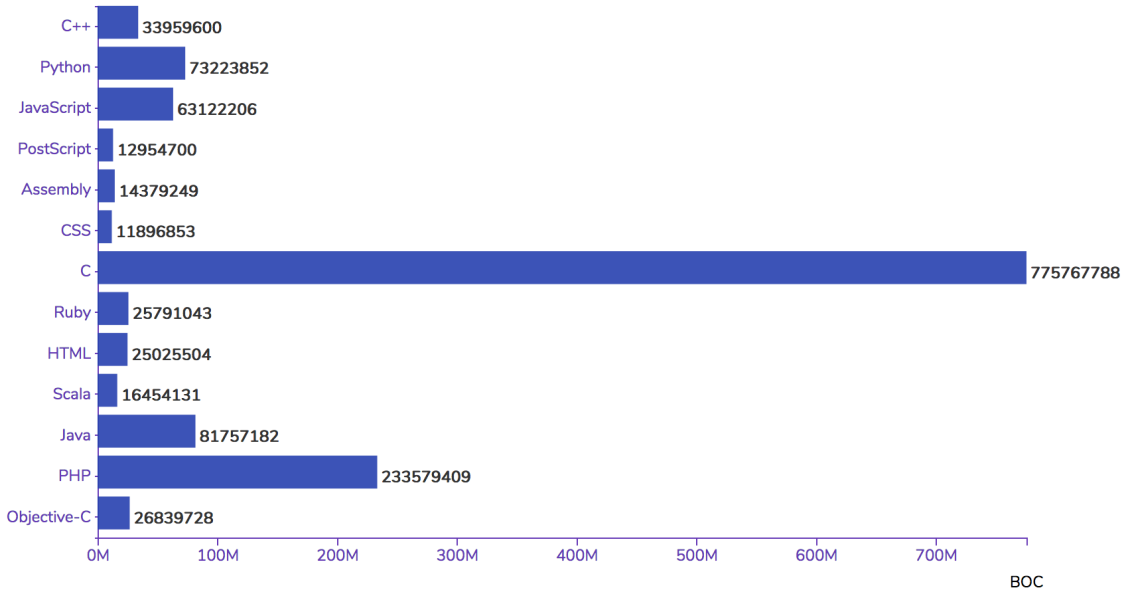

Figure 5.1: Top 13 of the most mined languages in bytes of code (BOC)

There is a high number of BOC for C language due to the extensive search of memory and resource leaks test cases to study a SAT.

The repositories contained in the sample have different sizes from 2 commits to almost $700K$ commits. It would be expected that the result of mining larger repositories would easily lead to more primitive data. But since the goal is to have a good representation of the whole Github, it is necessary to also contain the smaller repositories, to reach balanced conclusions and predictions.

5.1.2 Real Security Vulnerabilities Distributions

As mentioned before, this is already the second version of the benchmark which contains 682 real security vulnerabilities, the result of mining 248 Github repositories - the equivalent to almost $2M$ of commits, more specifically 1978482 commits - covering 16 different and previously defined vulnerability patterns (Tab. 3.1, 3.2).

From the sample under evaluation, a few interesting distributions between the accepted number of vulnerabilities and other variables were obtained. Some of them reflect real information which was already reported by other studies and others contribute with new insights related to OSS. Since a small sample is being evaluated, it is necessary to understand that these values may not represent exactly the domain (Github). The distributions presented below are per year (Sec. 5.1.2.1), language (Sec. 5.1.2.2) and pattern (Sec. 5.1.2.3).

5.1.2.1 Year

SecBench includes security vulnerabilities from 1999 until now (2017), being the group of years between 2012 and 2016 the one with most accepted vulnerabilities (Fig. 5.2), especially 2014 with a percentage of 14.37%. This supports the IBM's X-Force Threat Intelligence 2017 Report [USA17] where it is concluded that in the last 5 years the number of vulnerabilities per year had an overall significant increase compared with the other years. It is also important to highlight that 2017 has already a considerable percentage (4.25%) of accepted vulnerabilities even though we have yet 6 months until the end of the year.

Based on these results, the percentage of real security vulnerabilities has been increasing over almost the past 20 years, with a peak on 2007 which according to the CISCO's Security Anual Report from 2007 [CIS07], it was a year marked by vulnerabilities with higher severity (i.e., the potential impact of a successful exploitation) compared with the previous ones.

The decreasing of security vulnerabilities in the last 2 years does not reflect the news and security reports. However, these reports englobe all kinds of software, and the study is only performed on OSS. The decreasing can reflect the concerns of the core developers within making the code public since the number of attacks is increasing and one of the potential causes is the code availability.

Except for 2000, it was possible to collect test cases for all years between 1999 and 2017 being the last years the most popular (2017, exclusively for now).

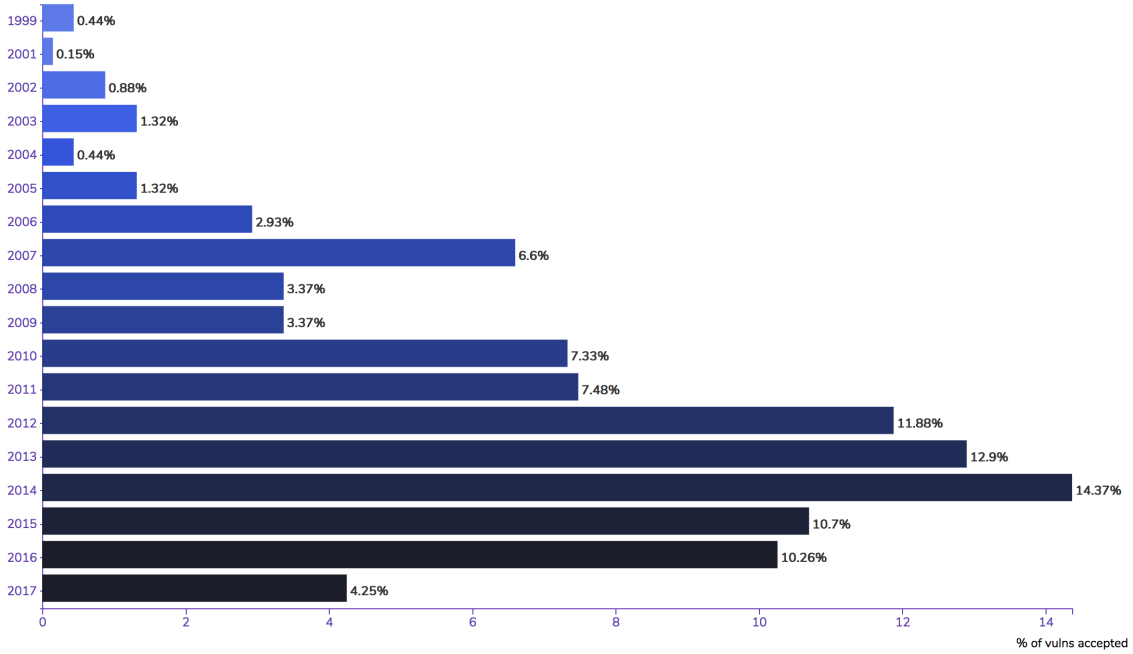

Figure 5.2: Distribution of real security vulnerabilities per year

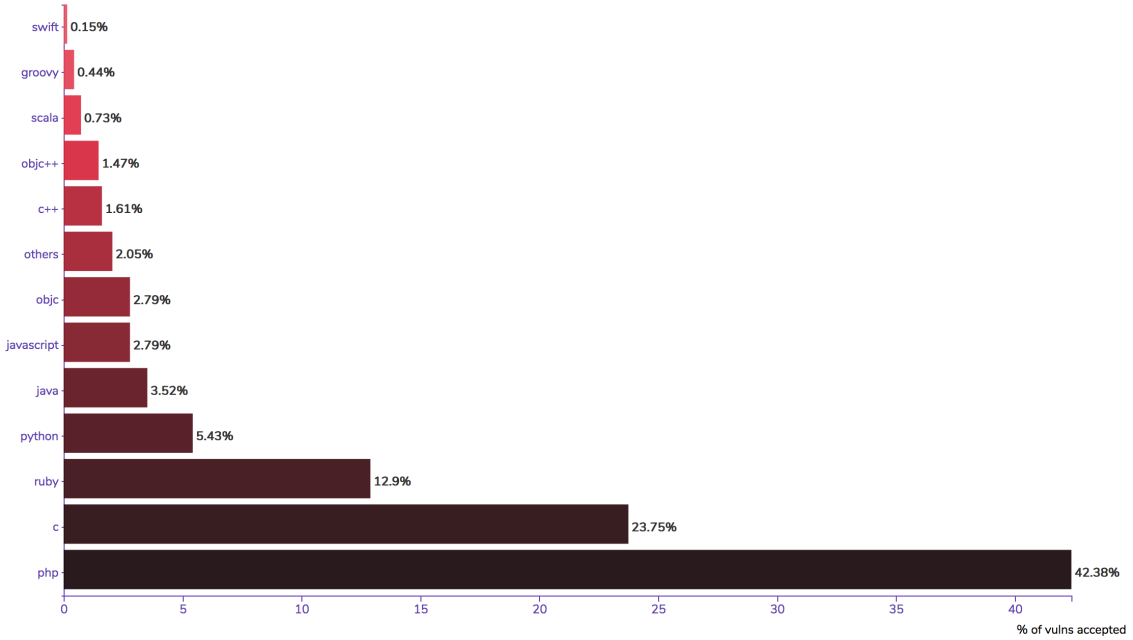

Figure 5.3: Distribution of real security vulnerabilities per language

5.1.2.2 Language

The sample covers more than 12 different languages being PHP (42.38%), C (23.75%), and Ruby (12.9%) the languages with higher number of test cases (Fig. 5.3) which reflects the values

of BOC presented on figure 5.1 for C and PHP. Two of the languages with higher values of BOC are also languages with higher percentages of accepted vulnerabilities. However, C is by far the most mined language, and it is the second language with a higher percentage of accepted vulnerabilities. These values can be the result of several things: the type of mined patterns, since more than 50% of the patterns target web applications; the time when the repositories were migrated or created on Github, i.e., software migrated after years of development with lots of C files but small number of commits; low number of active repositories on Github compared with languages like JavaScript or Ruby; or, even the fact that vulnerabilities from low-level languages are more difficult to identify since their focus is mainly memory management which is not a major concern on higher level languages.

The top 5 of the most popular programming languages are all included on the top 6 of the programming languages with the higher percentage of accepted vulnerabilities which proves that the assumption about retrieving more easily primitive data based on the top of most popular programming languages is true.

This supports the higher percentage of security vulnerabilities caught for *xss* (20.67%), *injec* (14.81%) and *ml* (12.46%) (Fig. 5.4) since C is a language where memory leaks are predominant and Ruby and PHP are scripting languages where Injection and Cross-Site Scripting are popular vulnerabilities.

Although the database contains 94 different languages, it only was able to collect viable information for 12 different languages.

5.1.2.3 Pattern

After mining and evaluating the samples, the results for 16 different patterns were obtained being the two main groups the ones presented on figure 5.4, Top 10 OWASP and others. *xss* (20.67%), *injec* (14.81%) and *ml* (12.46%) are the trendiest patterns on OSS which is curious since *injec* takes the first place on Top 10 OWASP 2017 [Fou17] and *xss* the second. *ml* does not integrate the top because it is not a vulnerability normally found on web applications.

Injection and Cross-Site Scripting are easy vulnerabilities to catch since the exploits are similar and exist, mainly, due to the lack of data sanitization which often is forgotten by the developers. The only difference between the two is the side from where the exploit is done (server or client).

Memory leaks exist because developers do not manage memory allocations and deallocations correctly. These kind of issues are one of the main reasons of *dos* attacks and regularly appeared on the manual evaluations even in the *misc* class. Although these three patterns are easy to fix, the protection against them is also typically forgotten.

Another prevalent pattern and which is not being considered is *misc* because it englobes all the other vulnerabilities and attacks found that do not belong to any of the chosen patterns or whose place was not yet well-defined. One example of vulnerabilities that you can find on *misc* (14.37%) are vulnerabilities that can lead to timing attacks where an attacker can retrieve information about the system through the analysis of the time taken to execute cryptographic algorithms. Normally,

the fix is the exchange of the function by another one with constant-time complexity. There is already material that can result in new patterns through the *misc* class analysis.

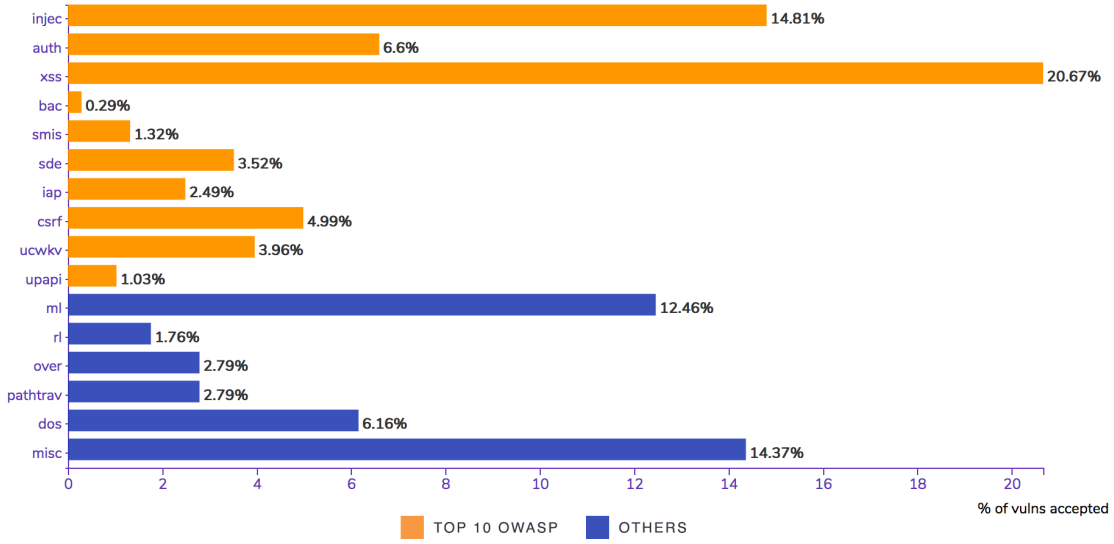

Figure 5.4: Distribution of real security vulnerabilities by pattern

Although *auth* (6.6%) is taking the second place on Top 10 OWASP 2017, it was not easy to find samples that resemble this pattern maybe because of the fact that highlighting these issues on Github can reveal other ones in their session management mechanisms and, consequently, leading to session hijacking attacks. The *csrf* (4.99%) and *dos* (6.16%) patterns are frequently seen among Github repositories: adding protection through unpredictable tokens and fixing several issues which lead to denial-of-service attacks.

The most critical patterns to extract are definitely *bac* (0.29%), which detects unproved access to sensitive data without enforcement; *upapi* (1.03%), which detects the addition of mechanisms to handle and answer to automated attacks; and, *smis* (1.32%) involving default information on unprotected files or unused pages that can give unauthorized access to attackers.

rl (1.76%) is another pattern whose extraction was hard. Although, memory leaks are resource leaks, here only the vulnerabilities related with the need of closing files, sockets, connections, etc, were considered.

The other patterns (e.g., *sde, iap, ucwkv, over and pathtrav*) were pretty common during our evaluation process and also on our Github searches. The *over* pattern contains vulnerabilities for several types of overflow: heap, stack, integer and buffer.

Another interesting point here is the considerable percentage of *iap* (2.49%), which normally is the addition of methods to detect attacks. This is the first time that *iap* makes part of the top 10 OWASP 2017 and still, we were able to detect more vulnerabilities for that pattern, than for *bac* which was already present in 2003 and 2004.

From 248 projects, the methodology was able to collect 682 vulnerabilities for 16 different patterns.

5.1.3 Identified Vulnerabilities by CVE and CWE

Several identified and reported vulnerabilities can also be found in this database. In fact, 15.4%(105) of the current database are vulnerabilities identified by CVE on 12 different years for 98 different CVEs. The database counts with already four identified vulnerabilities for 2017. Two of them are the CVE-2017-7620 where the omission of a backslash allows permalink injection through CSRF attacks; or open redirects via $login\_page.php?return = URI$; and, the CVE-2017-3733 where an encryption vulnerability would lead OpenSSL to crash affecting clients and servers.

Again, there is a significant increase of vulnerabilities in the last years which can be seen on table 5.1. The range between 2016 and 2013 reflects more than 50% of the CVEs identified, i.e., these 4 years ($1/3$ of time) contain 67.62% of the identified vulnerabilities.

Year	2017	2016	2015	2014	2013	2012	2011	2010	2009	2008	2007	2006
#CVE	4	20	13	22	16	9	9	5	2	3	1	1

Table 5.1: Vulnerabilities identified with CVE

In almost $6K$ commits manually evaluated, the identifier for weaknesses (CWE) or reference to that never appeared which reflects the search made through Github that retrieved only $12K$ of commits with messages containing CWE and $2M$ for CVE.

5.1.4 Mined and Accepted Vunerabilities within Repositories

As a result of the mining process for the 16 different patterns (Tab. 5.2), 62.5% of the repositories contain vulnerabilities (VRepositories) and 37.5% do not include vulnerabilities.

#Vulns	#Repositories	Repositories (%)
> 0	155	62.5%
$= 0$	93	37.5%
Total	248	100%

Table 5.2: Mined Vulnerabilities Distribution

After mining the repositories, the manual evaluation was performed where each candidate had to fulfil a group of requirements (Chap. 3). Here, as we can see in table 5.3, the percentage of success, i.e., repositories containing the vulnerability, decreases to 54.19%. The approximated difference of 8% is due to the cleaning process made through the evaluation process where a human tries to understand if the actual code fixes and represents a security vulnerability or not.

Although the decreasing from one process to another, we can still obtain a considerable percentage (more than one half) of VRepositories containing real vulnerabilities.

#AVulns	#VRepositories	VRepositories (%)
> 0	84	54.19%
$= 0$	71	45.81%
Total	155	100%

Table 5.3: Accepted Vulnerabilities (AVulns) Distribution

In the end, we were able to extract vulnerabilities with an existence ratio of ≈ 2.75 $(682/248)$. The current number of repositories on Github is $61M$, so based on the previous ratio we can obtain a database of $\approx$ 168 billion ($167750M$) of real security vulnerabilities which is $\approx$ 246 thousand (245967) times higher than the current database.

5.1.5 Regular Expression Efficiency

Each resulting case from the mining process was evaluated and classified with one of 4 status: **Y**es, when the case fulfilled all the requirements; **N**o, when it has no indications of a vulnerability (e.g., documentation, refactoring and many other causes); **D**oubt, when it seemed to be a vulnerability, but the human evaluating was not sure about it; and, **R**eplication, when the commit was a replication of other commits (e.g., GitHub merges of different commits).

To recognize patterns on commits messages, regular expressions were used. As expected (Fig. 5.5), more than 50% of the results obtained are garbage (N or R) for all patterns. This reflects the difficulty found on getting viable test cases within all the mining resulting commits. This is a highly time-consuming task and not accurate at all. But due to the time restrictions associated to the thesis development, first, the main goal was to prove the hypothesis since it is a long process. To improve these results, it will be necessary to use other techniques like natural language processing to add understanding of semantics to the tool and hopefully decrease the time and effort associated with the task.

The evaluation performed on figure 5.5 and figure 5.6 does not contain one of the projects involved on the sample under study since it is the result of bad mining and would compromise the analysis.

These are the results for all the variations of regular expressions, since their early stages. However, they were adapted according to the level of garbage collected through the mining process. For example, for the *misc* pattern the program would check `fix.* sec.* warning` but in a specific project this got a lot of commits whose messages contained the string *fix second mismatching warning* which had nothing to do with a vulnerability but it appeared a lot through the manual evaluation. Thus, the regular expression was changed to `fix.* secur.* warning` and the issue was solved ignoring those type of commits. There was a problem in balancing the specificity of some words used on the patterns and the garbage or non-caught vulnerabilities. Sometimes we would specify too much a pattern, and then no commits would be caught when the vulnerabilities were there. So, we tried to have a balance.

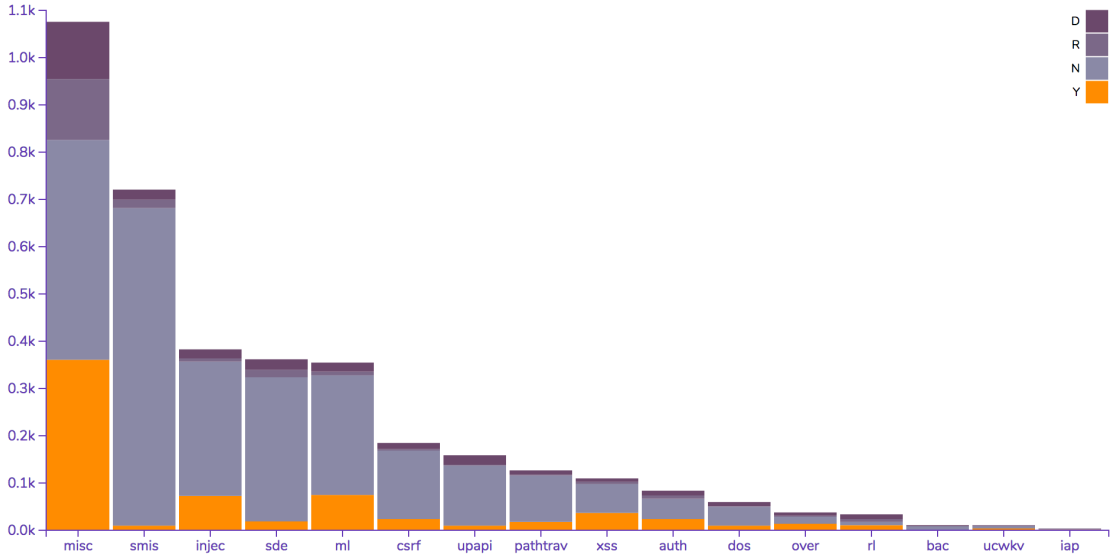

Figure 5.5: Distribution between mined commits and their current state without reallocating vulnerabilities

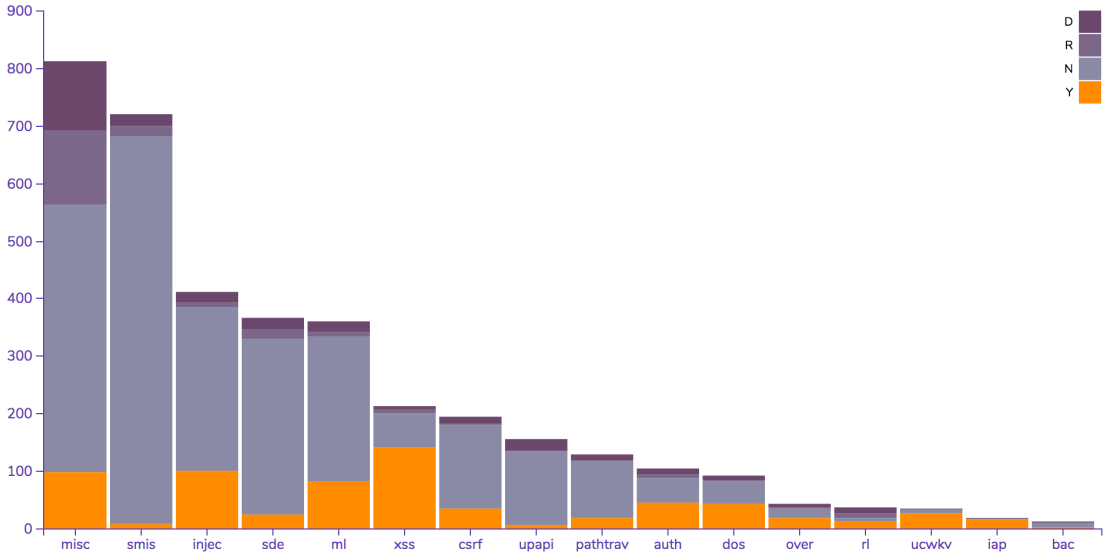

Figure 5.6: Distribution between mined commits and their current state after reallocating vulnerabilities

The difference between figure 5.5 and figure 5.6 is that in the first one the values do not consider when the pattern vulnerability was changed, i.e., if a vulnerability was caught in the wrong place and then changed to the right one. Through the vulnerabilities manual evaluation, some of them were reallocated to other patterns, mainly, vulnerabilities caught on *misc* pattern. On figure 5.6, we can see higher values of Y for the majority of patterns.

Many vulnerabilities caught on *misc* belonged to other patterns because of the identified vulnerabilities (CVE) and the use of words like *attack* and *vulnerable*.

5.1.6 Correlations

Based on the current sample, we tried to understand if there is any correlation between the number of mined commits detecting a vulnerability and the time of development; and, between the number of vulnerabilities accepted and the time of development.

The correlation coefficient value increases after the manual evaluation, i.e., after filtering commits and accepting vulnerabilities. There is a medium positive correlation between both dependent variables. It seems that the correlation is stronger for projects with a time of development with a size lower than 12 years. After 12 years, there is a less number of repositories without commits and vulnerabilities accepted. But at the same time, the correlation it is lower since there are repositories with less time of development with a higher number of test cases. The level of dispersion is higher after 12 years (Fig. 5.7).

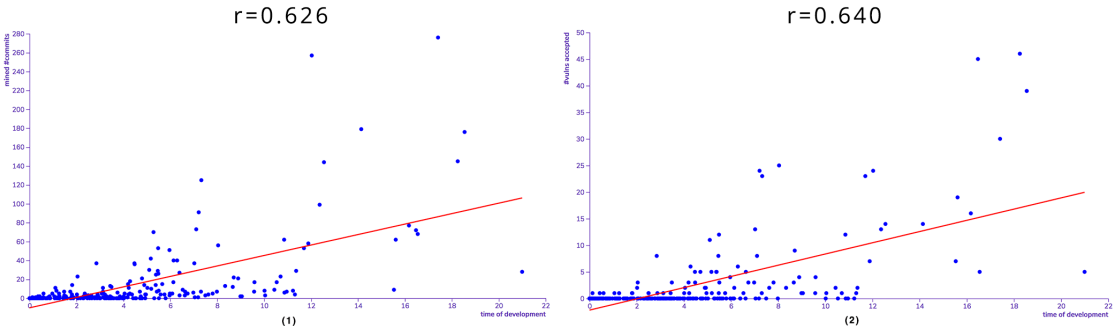

Figure 5.7: Distribution between mined commits and the time of development (1) and vulnerabilities accepted and the time of development (2)

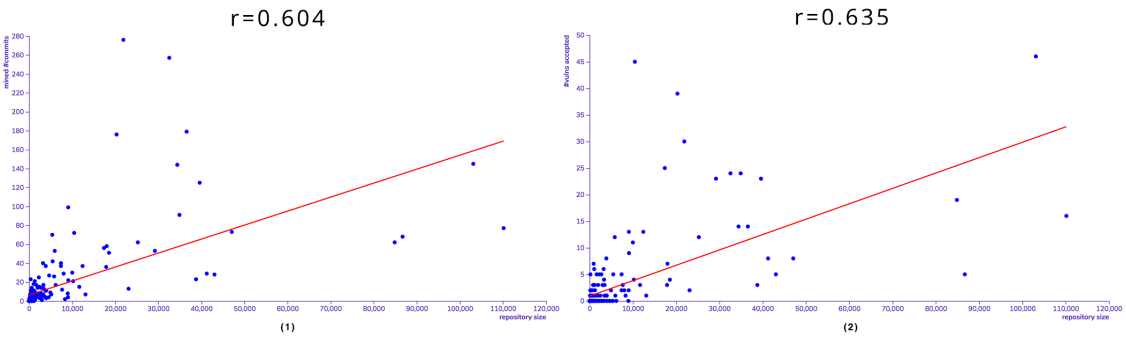

Figure 5.8: Distribution between mined commits and the repository size (1) and vulnerabilities accepted and the repository size (2)

Comparing the repository size and the number of mined commits and the repository size with the number of vulnerabilities accepted, it does not exist correlation for larger repositories. However, it is a positive correlation which says that the dependent variables increase along with the repository size (number of commits) (Fig. 5.8).

Except for lower values of repository size and time of development, the correlations are too small. The medium correlations are due to the results obtained for lower values of independent variables. But it seems that in general, the number of mined commits and vulnerabilities accepted increases according to the independent variables (repository size and time of development).

The sample under study is too small and will benefit from adding more repositories to the sample since this may not reflect all domain.

5.2 Static Analysis Tools Report

In order to understand if it is viable to study SATs using this methodology, two tools were studied in the best way possible using this sample: `Infer`[1] and `Find-Sec-Bugs`[2].

5.2.1 SAT: Infer

This tool was chosen due to its popularity on Github, the company behind (Facebook) and because it tries to catch one of the vulnerabilities that are one of the main causes of Denial-of-Service attacks.

`Infer` catches memory leaks, resource leaks and null dereference vulnerabilities. Unfortunately, this methodology was not able to find a considerable amount of null dereference vulnerabilities. So, they won't be studied.

This tool also searches for memory leaks in C++ but it is still very primitive. On Github Issues, the core developers advised several times developers to not use the C++ analysis yet. They are still working on it.

5.2.1.1 Memory Leaks

27 vulnerabilities were extracted from 10 test cases (4 for Objective-C and 6 for C). `Infer` only identifies memory leaks in C if variables are initilialized with *malloc*, *realloc* and *calloc*. Objective-C memory leaks are only reported if objects were created using Core Foundation or Core Graphics and not released.

[1] http://fbinfer.com/
[2] http://find-sec-bugs.github.io/

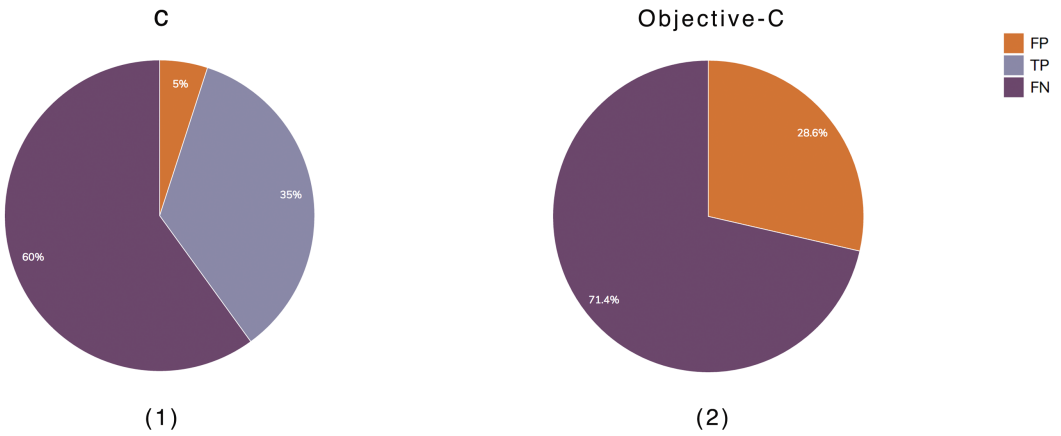

Figure 5.9: Distribution of FP, TP and FN on detecting memory leaks by `Infer` for C (1) and Objective-C (2)

In order to study the tool, we were only able to use 4 of 15 test cases for memory leaks in Objective-C and 6 of 33 test cases for memory leaks in C. We were not able to detect TP for memory leaks in Objective-C.

When identifying memory leaks on C source code, the percentage of FP and FN overcomes the percentage of TP (Fig. 5.9) which reflects the bad performance reported by other studies before.

Language	Precision (pr)	Recall (r)	F-Score(fs)	#Vulns
C	0.875	0.368	0.519	20
Objective-C	0	0	0	7

Table 5.4: `Infer`: Resulting metrics for memory leaks

Since 0 TPs were identified by `Infer` for memory leaks in Objective-C, its precision, recall and f-score are zero. This could mean that the tool does not perform good (value far from 1) when looking for memory leaks in Objective-C but 7 vulnerabilities it is not a considerable amount of vulnerabilities to make such conclusion.

Although `Infer` has a precision of 0.875 for memory leaks identification that shows a considerable amount of reliability, it also shows a small capability of correctly identifying vulnerabilities (0.368). Thus, based on its F-Score we can conclude that `Infer` is an average tool for memory leaks identification in C.

5.2.1.2 Resource Leaks

19 vulnerabilities were extracted from 8 test cases (4 for C and 4 for Java). `Infer` only identifies resource leaks in C when resources are entities such as files, sockets and connections that need to be closed after being used. Resource Leaks for Java are, normally, caught for exceptions skipping past close() statements.

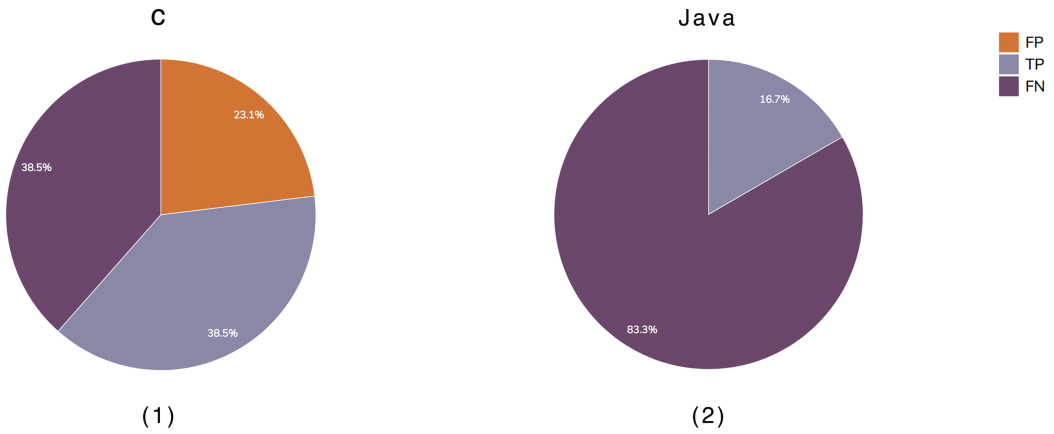

Figure 5.10: Distribution of FP, TP and FN on detecting resource leaks by `Infer` for C (1) and Java (2)

We were only able to use 4 of 22 test cases for resource leaks in C and 4 of 8 test cases for resource leaks in Java. We were able to catch TP for the two samples but 0 FP for resource leaks in Java.

When identifying resource leaks on C and Java source code, the percentage of FP and FN overcomes the percentage of TP (Fig. 5.10) which again reflects the bad performance reported by other studies before.

Language	Precision (pr)	Recall (r)	F-Score(fs)	#Vulns
C	0.625	0.5	0.556	13
Java	1.0	0.167	0.286	6

Table 5.5: `Infer`: Resulting metrics for resource leaks

It seems that `Infer` is less reliable (0.625) on identifying resource leaks than memory leaks on C source code. However, the number of studied vulnerabilities for resource leaks (13) was lower than memory leaks (20). At the same time, its capability of correctly identifying vulnerabilities resource leaks for C (0.5) is higher than in for memory leaks (0.368). The resulting F-score reflects a tool with a medium performance on resource leaks identification (0.556) with is almost the same as for memory leaks.

Since 0 FPs were caught for resource leaks in Java, the metrics reflect a precision of 1.0, but at the same time, they also reflect a recall of 0.167 which means a really bad sensibility on Java resource leaks identification. Overall, the performance of `Infer` capability on detecting Java resource leaks in bad (0.286). But as mentioned before the number of used vulnerabilities is low, so this may not be a good reflection even though it reflects studies done before saying that SATs are still far from their higher level of maturity.

5.2.1.3 Overall

In general and now with a higher considerable amount of samples we can have a better look at the performance of `Infer` on memory leaks and resource leaks identification.

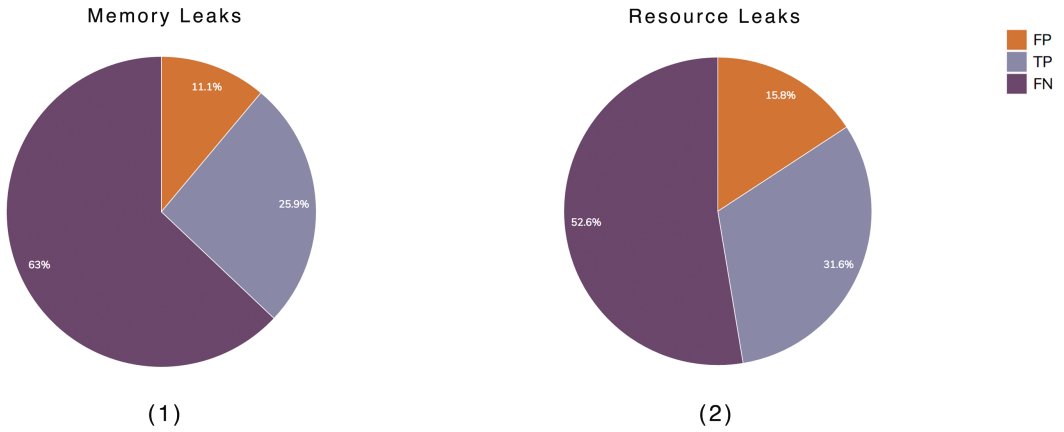

Figure 5.11: Distribution of FP, TP and FN on detecting memory leaks (1) and resource leaks (2) by `Infer`

It was possible to identify FPs for the two patterns and the sum of FP and FN percentages is higher than the TPs percentage. There is a higher percentage of FPs and TPs on resource leaks than in memory leaks (Fig. 5.11), i.e., resource leaks have a lower percentage of FNs.

Pattern	Precision (pr)	Recall (r)	F-Score(fs)	#Vulns
ml	0.7	0.292	0.412	27
rl	0.667	0.375	0.480	19

Table 5.6: `Infer`: Resulting metrics for memory leaks and resource leaks

Based on the metrics results, it is possible to conclude that the precision, recall and resulting f-score from both patterns are pretty close, being resource leaks the pattern identified with more success by `Infer`. However, due to the considerable difference of studied vulnerabilities between both patterns, it may not be correct to make this conclusion since the final f-scores are very close.

Tool	Precision (pr)	Recall (r)	F-Score(fs)	#Vulns
`Infer`	0.684	0.325	0.441	46

Table 5.7: Resulting `Infer` metrics

Overall, `Infer`'s performance is medium on identifying resource leaks and memory leaks with an f-score of 0.441 and worst sensibility (0.325) level compared with reliability (0.684).

5.2.2 SAT: Find-Sec-Bugs

`Find-Sec-Bugs` is a SAT for detecting Java security vulnerabilities. It identifies 113 different patterns which are more difficult to find than memory leaks and resource leaks. A higher

percentage of the vulnerabilities in our database for Java or Scala are not covered by the patterns due, mainly, to the type of libraries and APIs used by them. After a few research on Github Big Query and Github search engine, searching for a few combinations of more specific words representing `Find-Sec-Bugs` patterns, it was concluded that based only on commits messages it won't be easy to find test cases to study this tool. Scala patterns will be tough to find since Scala it is not even one of the most popular languages on GitHub. Several patterns are identified by CWE but the outcome of the *SecBench* research is that developers don't use much this kind of identification. So, looking for CWE on messages won't be a good approach.

The approach for this tool needs to be different. First, it may be a good idea to do a collection of Java web applications on Github to find test cases more efficiently. And then, the mining tool may be adapted to scan the imports of each file involved in the V_{diff} sample and evaluate if the libraries and APIs are present.

The results for this tool aim to show that we need to change things in our methodology to be successful with tools that support patterns with higher specificity.

5.3 Conclusions

According to the performed empirical evaluation, we can conclude that:

- Github has security vulnerabilities for a wide range of different years.
- The top 5 of languages with more percentage of accepted vulnerabilities on our database reflects the top 5 on Github.
- It is possible to find a good amount of identified vulnerabilities (CVE) with this approach.
- According to the correlations presented, repositories with higher size and development time can be a better bet.
- Although it is difficult to find viable test cases that reflect the models and whose complexity is understood by the tools, we can find a considerable amount of vulnerabilities inside of each test case.
- With the test cases hosted by our database (the result of 2-3 months of mining and evaluation), we were able to find samples to successfully study a SAT using security vulnerabilities mined from Github.
- Tools identifying security patterns with higher specificity need different approaches from the one used until now.

Chapter 6

SAT Problems

In this chapter, an overview of the results of a few test cases studied with `Infer` are provided and discussed, in order to prove that the tool still needs modernization.

6.1 Infer

`Infer` is a static analysis tool used to identify memory leaks on Objective-C and C code; and, resource leaks on Java and C code.

Resource leaks in C are reported when resources like files, sockets and connections are opened and never closed. Resource leaks in Java include input streams, output streams, readers, writers, sockets, HTTP connections, cursors, and JSON parsers. `Infer` focuses mainly on exceptions skipping past close() statements when scanning for resource leaks in Java.

`Infer` identifies memory leaks in C if variables are initialized with *malloc*, *realloc* and *calloc*. Objective-C memory leaks are reported for objects created using Core Foundation or Core Graphics (iOS) and not released.

It also identifies null dereference vulnerabilities, but we were not able to study it due to the lack of test cases.

6.2 Infer: How does it work?

For a given piece of code, `Infer` [CD11] synthesises pre and post specifications (represented by Hoare's triples) of the form below inferring suitable P and Q.

$$\{P\}\,C\,\{Q\}$$

Instead of expressing functional correctness, specifications express memory safety. `Infer` uses bi-abductive inference to synthesise pre and post-conditions in specifications. Bi-abductive inference consists in solving or validate:

$$H * A \vdash H' * F$$

where H and H' are the separation logic formulaes describing a heap configuration and **F**rame and **A**nti-frame which need to be inferred [CD11].

The tool automatically discovers only the specifications for the functions that do not leak memory (memory safe).

`Infer` does a compositional shape analysis [CDOY11] using Bi-Abduction, which makes possible the handling of procedure calls (interprocedural analysis).

6.3 Results and Source Code Comparison

Here, seven different cases studied for `Infer` will be presented. We try to discuss why the tool did not work based on the models and type of analysis performed. However, further research must be done to get to the root of the problems and understand how exactly we can improve the tool. The main goal of the next points is to show that `Infer` still needs modernization and that we can pinpoint it somehow.

6.3.1 Case 1

The first case presents 2 FPs for memory leaks caught by the tool. These results can be seen in the second part of figure 6.1. `Infer` detected two memory leaks which do not exist since the variable *reachability* is released/ownership is transferred to ARC[1] on line 157.

This is not a vulnerability, but we thought it would be interesting to the study since it is a real false positive that we caught through the mining process.

From the report, `Infer` thinks that *reachability* is being allocated again through *initWithReachability* after being allocated using *SCNetworkReachabilityRef* from *Core Foundation*. But the ownership of the object is transferred to ARC, so it can release the memory allocated when the object is not needed anymore.

`Infer` has the possibility of tracking all the analysis path which allowed to easily pinpoint the problem. The tool does not recognize the function *CFBridgingRelease* (Fig. 6.2) because it is not a method from the project (it is from *Core Foundation*) and it is not implemented on `infer`'s models. Thus, when `Infer` tries to create the specifications of function *initWithReachability* it is not successful and throws the memory leaks errors.

[1] https://developer.apple.com/library/content/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html

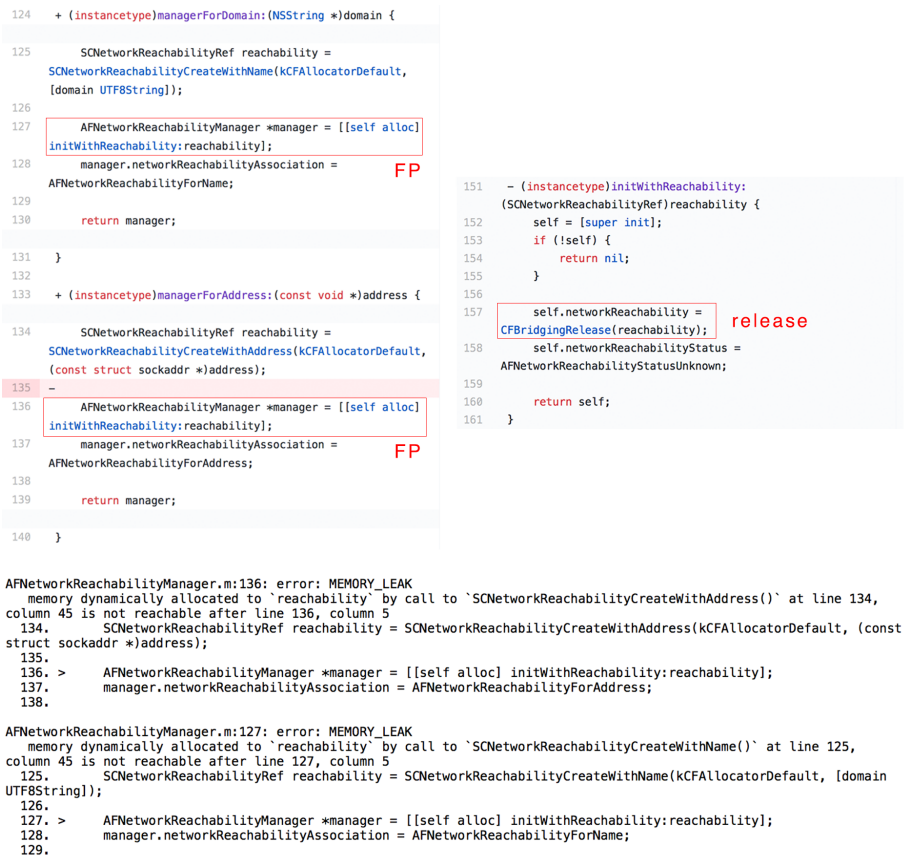

Figure 6.1: Infer Case 1: 2 FP for memory leaks in Objective-C

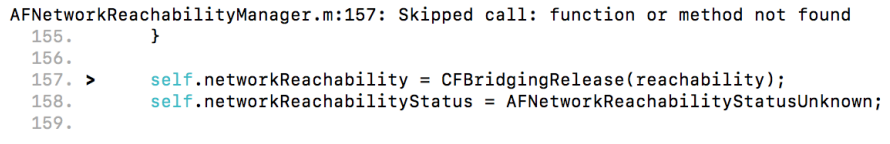

Figure 6.2: Infer Case 1: Trace bugs

6.3.2 Case 2

The second case presents 2 FPs for resource leaks caught by the tool. These results can be seen in the second part of figure 6.3. `Infer` detected two resource leaks which do not exist since the variable *fd* is closed for the two different conditions on line 1481 e 1492.

This vulnerability was caught scanning the source code for memory leaks in on of the test cases.

After analyzing the models for *open* and *close* on `Infer` repository, we were not able to totally understand why `Infer` returned these results. A resource leak is launched if *close* model returns -1 after `open` returns a result 0 or higher. The *close* model calculates an arbitrary non

deterministic int value which is the return value of the function. The problem can be here but we are not 100% sure.

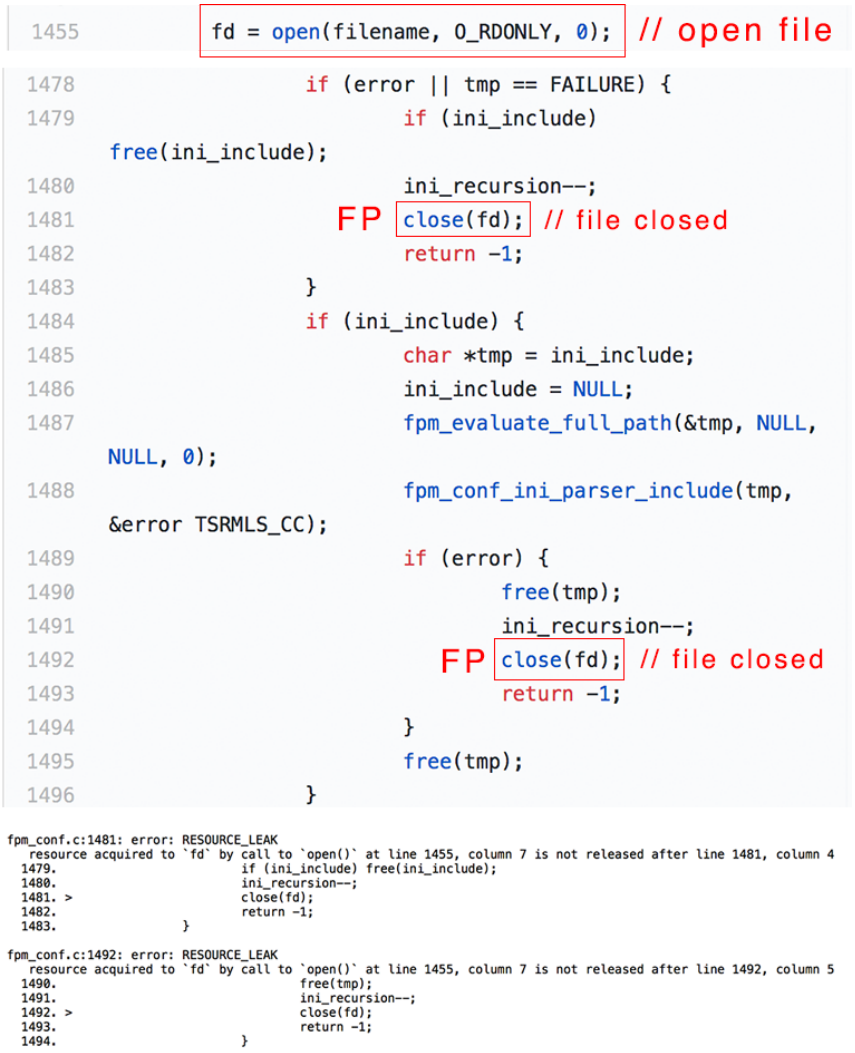

Figure 6.3: Infer Case 2: 2 FP for resource leaks in C

6.3.3 Case 3

This case is an FN for resource leaks for C. If the condition on line 35 is true, the function returns. Before the return, *fd* must be closed. `Infer` probably does not identify the vulnerability because it has no model representing *dup2*, so it is not capable of understanding the condition value. The condition is skipped because the tool is not able to make the comparisons.

```
28          int fd = open("/dev/null", O_RDWR);

35          if (0 > dup2(fd, STDIN_FILENO) || 0 > dup2(fd,
STDOUT_FILENO)) {
36                  zlog(ZLOG_SYSERROR, "failed to init stdio:
dup2()");
                    // missing close fd before return
37                  return -1;
38          }
39          close(fd);
40          return 0;
41  }
```

Figure 6.4: Infer Case 3: 1 FN for resource leaks on C

6.3.4 Case 4

This is an FN for resource leaks for C. In this case, if the creation, binding or the starting of a socket fails, the function needs to return, and the socket needs to be closed. It is almost the same things as case 3. `Infer` only recognizes *socket* as a pattern. Thus, the tool is not able to make the other comparisons to obtain the condition value, and the resource leak is not identified.

```
753         /* Create, bind socket and start listen on it */
754         if ((listen_socket = socket(sa.sa.sa_family,
SOCK_STREAM, 0)) < 0 ||
755  #ifdef SO_REUSEADDR
756             setsockopt(listen_socket, SOL_SOCKET,
SO_REUSEADDR, (char*)&reuse, sizeof(reuse)) < 0 ||
757  #endif
758             bind(listen_socket, (struct sockaddr *) &sa,
sock_len) < 0 ||
759             listen(listen_socket, backlog) < 0) {
760
                // listen_socket is not released
761                 fcgi_log(FCGI_ERROR, "Cannot bind/listen
socket - [%d] %s.\n",errno, strerror(errno));
762                 return -1;
763         }
```

Figure 6.5: Infer Case 4: 1 FN for resource leaks in C

6.4 Conclusions

These are a few examples of test cases we were able to study for `Infer`. The main objective of this chapter is to expose the concerns the researchers need to have in order to understand how the tool works. Since this kind of tools does not have much documentation, the only way of knowing the models and the type of analysis made by the tool is studying the tools source code.

Under this methodology, we were able to find points of modernization of the tool. Although not pointed yet, the majority of static analysis tools have really bad reports and lack of information. It is general for all OSS tools.

Chapter 7

Conclusions and Future Work

As a result of this thesis, a database containing real security vulnerabilities is proposed. In particular, `SecBench` is composed of 682 real security vulnerabilities, which was the outcome of mining 248 projects - accounting to almost $2M$ commits - for 16 different vulnerability patterns.

The importance of this database is the potential to help researchers and practitioners alike improve and evaluate software security testing techniques.

We have demonstrated that there is considerable information on open-source repositories to create a database of real security vulnerability for different languages and patterns. And thus, we can contribute to considerably reduce the lack of real security vulnerabilities databases. On Github, besides several languages and patterns, we can also find a large variety of different developers with different programming skills. Thus, the test cases quality may not be the best, since we do not have automated ways of checking the quality of a repository.

Overall, this methodology has proven itself as being very valuable since we collected a considerable number of security vulnerabilities from a very small group of repositories (248 repositories from $61M$). However, there are several points of possible improvements, not only in the mining tool but also in the evaluation and identification process which can be costly and time-consuming.

SATs can be studied using this methodology. Yet, the patterns identified by the tool can be pretty specific which may not work with the approach of the mining tool now.

Finding test cases that match with the tools models, it is not easy and some samples can even be so complex that we were almost sure that the tool would not identify anything.

A positive point is the fact that we were able to find more than one vulnerability for more than 80% of the studied test cases. Although we were using a small group of test cases, we were able to almost duplicate the number of results.

We were also able to show that `Infer` needs to be modernized not only through the values obtained from the empirical evaluation but also in the previous chapter where we try to pinpoint a few problems. Thus, we were able to prove our initial hypothesis even though it needs to be improved.

7.1 Answer Research Questions

To prove our hypothesis, we tried to answer a few research questions focused on the data and the results obtained under this methodology.

RQ1.1: *s there enough information available on OSS repositories to create a benchmark of software security vulnerabilities?*

From 248 repositories (the equivalent to almost 2*M* of commits), we were successfully able to collect 682 vulnerabilities for 16 different patterns.

15.4% of our database are identified vulnerabilities (CVE). We were able to obtain results from the mining tool for more than 60% of the repositories where almost 50% contained real security vulnerabilities.

In the end, we were able to extract vulnerabilities with an existence ratio of 2.75 per repository which can lead to a potential database of 168 billions of test cases. Even if it does not lead to such a high number, it is already satisfactory having an initial database of 682 vulnerabilities from only 248 repositories and with such variety since only one database of real security vulnerabilities existed until now (only for Python).

Due to the dimension of the domain under study (Github) and the number of mined repositories we can answer this question with only 99% of confidence and with a margin of error equal to 8.18%.

> There are enough vulnerabilities available on open-source repositories to create a database of real security vulnerabilities.

RQ1.2: *What are the most prevalent security patterns on OSS repositories?*

After mining and evaluating the samples, we obtained results for 16 different patterns. Although we are using a considerable amount of projects for these statistics, it is possible that the most prevalent patterns change as we continue the study.

From mining 16 patterns, we can conlude that *xss* (20.67%), *injec* (14.81%) and *ml* (12.46%) are the trendiest patterns on OSS which is curious since *injec* takes the first place on Top 10 OWASP 2017 and *xss* the second. *ml* does not integrate the top because it is not a vulnerability normally found on web applications.

Although, *misc* has a higher percentage than *ml* we do not considerate it for the answer since it contains several different patterns in smaller sizes.

> The most prevalent security patterns are Injection, Cross-Site Scripting and Memory Leaks.

RQ2.1: *Is it viable to study SATs with real security vulnerabilities?*

Using this methodology we were able to collect a few test cases to study the tool. However, it is a difficult task since many of the samples would not match the tool models. To collect test cases for patterns with higher levels of specificity, the mining tool must be improved. Checking only the commits messages is not a good methodology.

More than 80% of the test cases contain more than one vulnerability which ends up satisfying the lack of test cases. Two different vulnerabilities were successfully studied (memory leaks and resource leaks), each of them for two different languages and overall of three different languages (Java, C and Objective-C).

Based on the results reported by the tool for each vulnerability, we were able to make conclusions about the tool's performance regarding reliability and sensibility. In the end, we conclude that `Infer` has a precision of 0.684, a recall of 0.325 and a final score of 0.441 which reflects a medium performance tool.

> It is viable to find samples on Github to study SATs but it also is important to have in mind the tool's patterns specificity.

RQ2.2: *Can we understand where SATs can be improved using this methodology?*

The tool study results allowed us to pinpoint a few problems with the tool. We are not still sure of how to solve the issues but based on the models and the technology under the tool we were able to somehow understand the problems.

Through the metrics results, we were also able to understand if the tool is good or not. Despite presenting just 4 cases of examples that represent problems on the SAT, we successfully identified problems in the tool.

> Yes, it is possible to pinpoint problems in a SAT using this methodology.

7.2 Limitations

Although the resulting number of vulnerabilities is considerable, it is also too much time-consuming, and the vulnerabilities evaluation are highly dependent on humans inserting a high percentage of errors in the results.

The mining tool needs to be improved, in order to, overcome the limitations related to retrieving samples for more specific patterns.

The sample under study may not be representative of all the Github domain, and the number of test cases used to study the tool may not reflect the real tool's performance. Thus, the values obtained in this study may not be considered.

7.3 Contributions

This thesis makes the following main contributions:

- SecBench, the first benchmark with a considerable amount of security vulnerabilities test cases and, mainly, focused on the study of static analysis tools.
- Empirical study of Github with a few interesting outcomes that could indeed help future researchers on improving the mining approach.
- Although still limited, a methodology which showed its value proving the thesis-hypothesis.
- A good comparison between more than 25 static analysis tool that can help future researchers to choose the next tool to be studied.
- A set of OSS tools and applications to mine, validate and manage data.

7.4 Future Work

There are many ways for improvement in this study and opportunities to future work:

- Extend the scope to more source code hosting websites (e.g., bitbucket, svn, etc).
- Augment the amount of security vulnerabilities, patterns and languages support.
- Augment the number of tools studied and, hopefully, provide information to help companies choosing the most reliable tools.
- Evaluate if we can obtain information about security vulnerabilities from pull requests and possibly classify repositories with a level of maturity based on the number of pull requests related to the patch of vulnerabilities.
- We can use mutants to represent real security vulnerabilities under specific conditions to increase the variety of test cases[JJI+14].
- Contribute with improvements for these tools based on our results.
- Use natural processing languages to improve the mining tool, in order to minimize the garbage collected through the mining process.

7.5 Publications

The first version of Secbench was submitted and accepted on the *28th International Workshop on Principles of Diagnosis (DX'17)*, 2017 for a poster session. The paper outlines the mining process, data validation and the results obtained after mining 238 repositories for 16 different patterns.

Another paper was submitted and accepted on the *Workshop on Secure Software Engineering in DevOps and Agile Development 2017* for a paper publication and presentation. This paper was invited to an extension for the *International Journal of Secure Software Engineering (IJSSE)*.

References

[ABE+15] Rui Abreu, Danny Bobrow, Hoda Eldardiry, Alexander Feldman, John Hanley, Tomonori Honda, Johan Kleer, and Alexandre Perez. Diagnosing advanced persistent threats: A position paper. In *Proceedings of the 26th International Workshop on Principles of Diagnosis*, pages 193–200, 2015.

[BL04] Lionel Briand and Yvan Labiche. Empirical studies of software testing techniques: Challenges, practical strategies, and future research. *SIGSOFT Softw. Eng. Notes*, 29(5):1–3, September 2004.

[Bri07] L. C. Briand. A critical analysis of empirical research in software testing. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 1–8, Sept 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[CD11] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *Proceedings of the Third International Conference on NASA Formal Methods*, pages 459–465, Berlin, Heidelberg, 2011. Springer-Verlag.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. OHearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, December 2011.

[CIS07] CISCO. Cisco 2007 annual security report. Technical report, CISCO, March 2007.

[Dav15] Amanda Davis. A history of hacking. *The institute IEEE*, Março 2015.

[DER05] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[Dun17] John E Dunn. 21 of the most infamous data breaches affecting the uk, May 2017. [Online; posted 18-May-2017].

[fAS11] Center for Assured Software. Static analysis tool study. Technical report, National Security Agency, December 2011.

[fNS17] European Union Agency for Network and Information Security. Enisa threat landscape report 2016. Technical report, January 2017.

[Fou17] The OWASP Foundation. Oswap top 10 - 2017: The ten most critical web application security risks. Technical report, The OWASP Foundation, February 2017. Release Candidate.

[GPP15] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.*, 68(C):18–33, December 2015.

[Gua14] The Guardian. Apple's ssl iphone vulnerability: how did it happen, and what next?, February 2014. [Online; posted 25-February-2014].

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[JJI+14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[Kha17a] Swati Khandelwal. Wannacry ransomware that's hitting world right now uses nsa windows, May 2017. [Online; posted 12-May-2017].

[Kha17b] Swati Khandelwal. Warning! don't click that google docs link you just received in your email, May 2017. [Online; posted 03-May-2017].

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[NZHZ07] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, New York, NY, USA, 2007. ACM.

[PCJ+17] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*, Buenos Aires, Argentina, May 2017.

[PR10] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[Sec17] Help Net Security. The cost of iot hacks: Up to 132017. [Online; posted 5-May-2017].

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[USA17] IBM Security Department USA. Ibm x-force threat intelligence index 2017. Technical report, IBM, March 2017.

[VBKM00] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, ACSAC '00, pages 257–, Washington, DC, USA, 2000. IEEE Computer Society.

[Woo16] Nicky Woolf. Ddos attack that disrupted internet was largest of its kind in history, experts say. *The Guardian*, October 2016.

[YM13] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, Oct 2013.

[YT88] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, Oct 1988.

REFERENCES

Appendix A

Patterns

This appendix includes the visualization of the regular expressions created to extract the primitve data from Github. We based our patterns, mainly, on the top 10 OWASP of 2017 (Sec. A.1). Then, we added more patterns that are trendy out of web applications (Sec. A.2). For each pattern, the words and combinations used to extract information are presented using `regexper`[1].

[1] https://regexper.com/

A.1 Top 10 OWASP 2017

The top 10 OWASP has 10 different types of patterns usually found on web applications:

- A1: Injection (Sec. A.1.1)
- A2: Broken Authentication and Session Management (Sec. A.1.2)
- A3: Cross-Site Scripting (XSS) (Sec. A.1.3)
- A4: Broken Access Control (Sec. A.1.4)
- A5: Security Misconfiguration (Sec. A.1.5)
- A6: Sensitive Data Exposure (Sec. A.1.6)
- A7: Insufficient Attack Protection (Sec. A.1.7)
- A8: Cross-Site Request Forgery (CSRF) (Sec. A.1.8)
- A9: Using Components with Known Vulnerabilities (Sec. A.1.9)
- A10: Underprotected APIs (Sec. A.1.10)

A.1.1 A1 - Injection

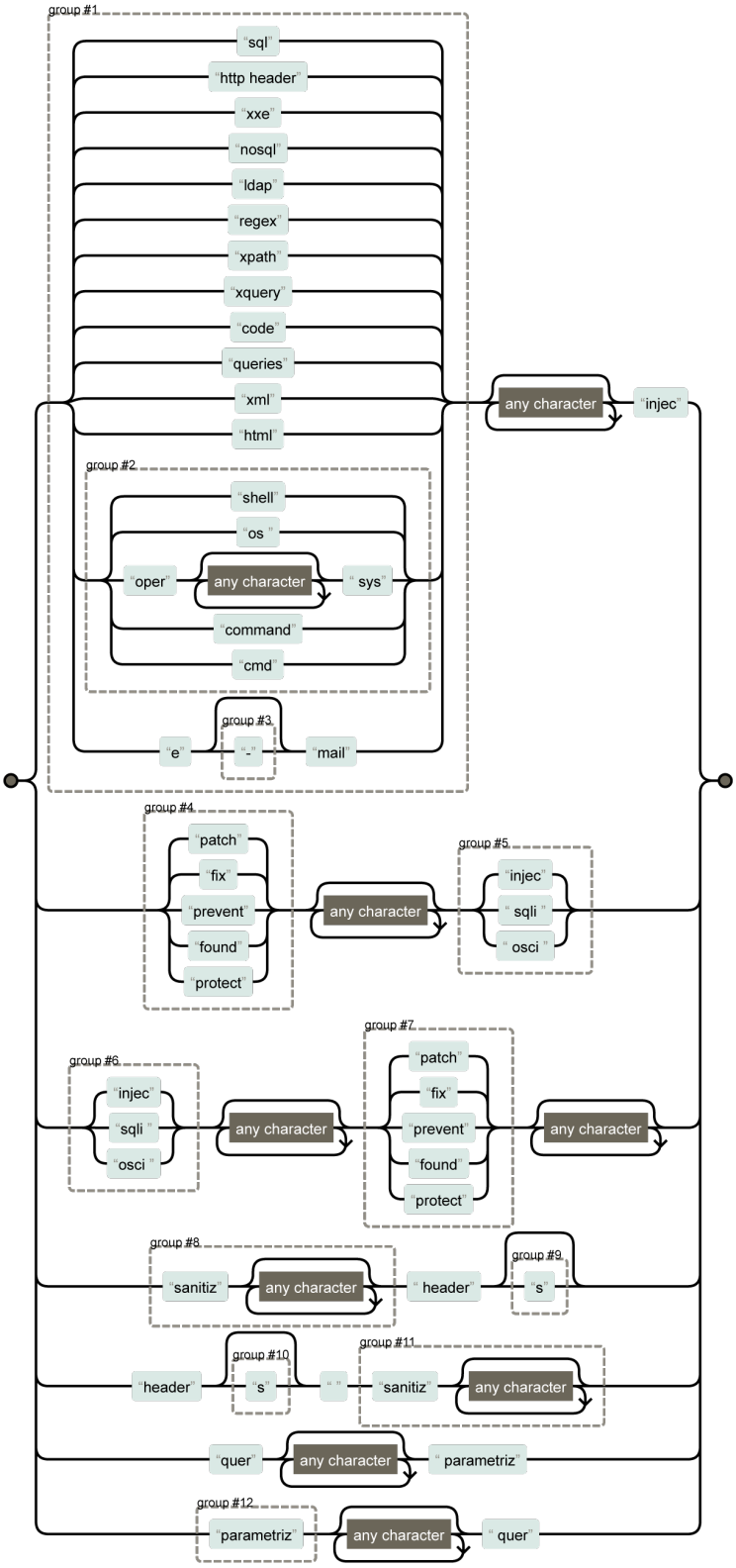

Figure A.1: Injection pattern

A.1.2 A2 - Broken Authentication and Session Management

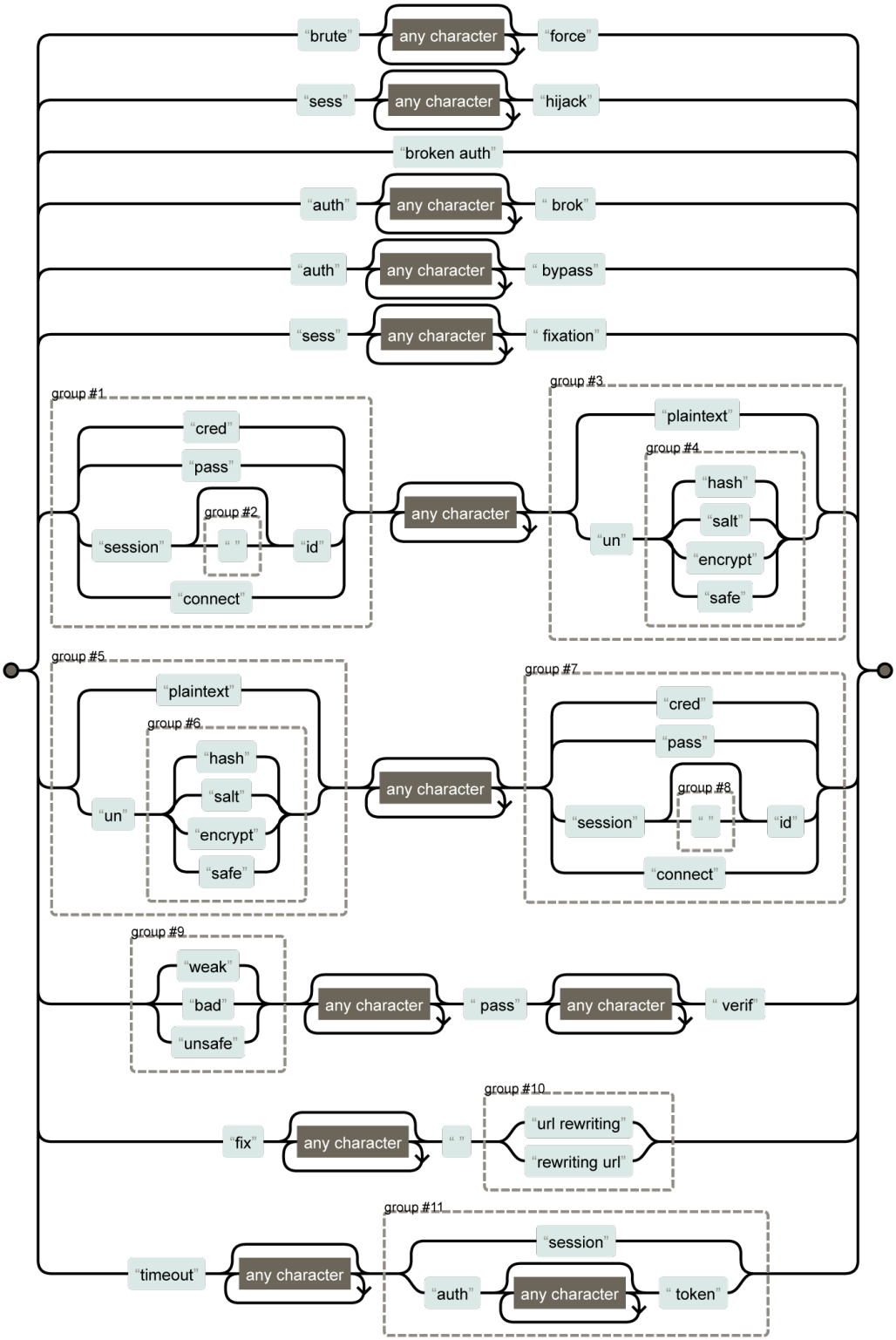

Figure A.2: Broken Authentication and Session Management pattern

A.1.3 A3 - Cross-Site Scripting (XSS)

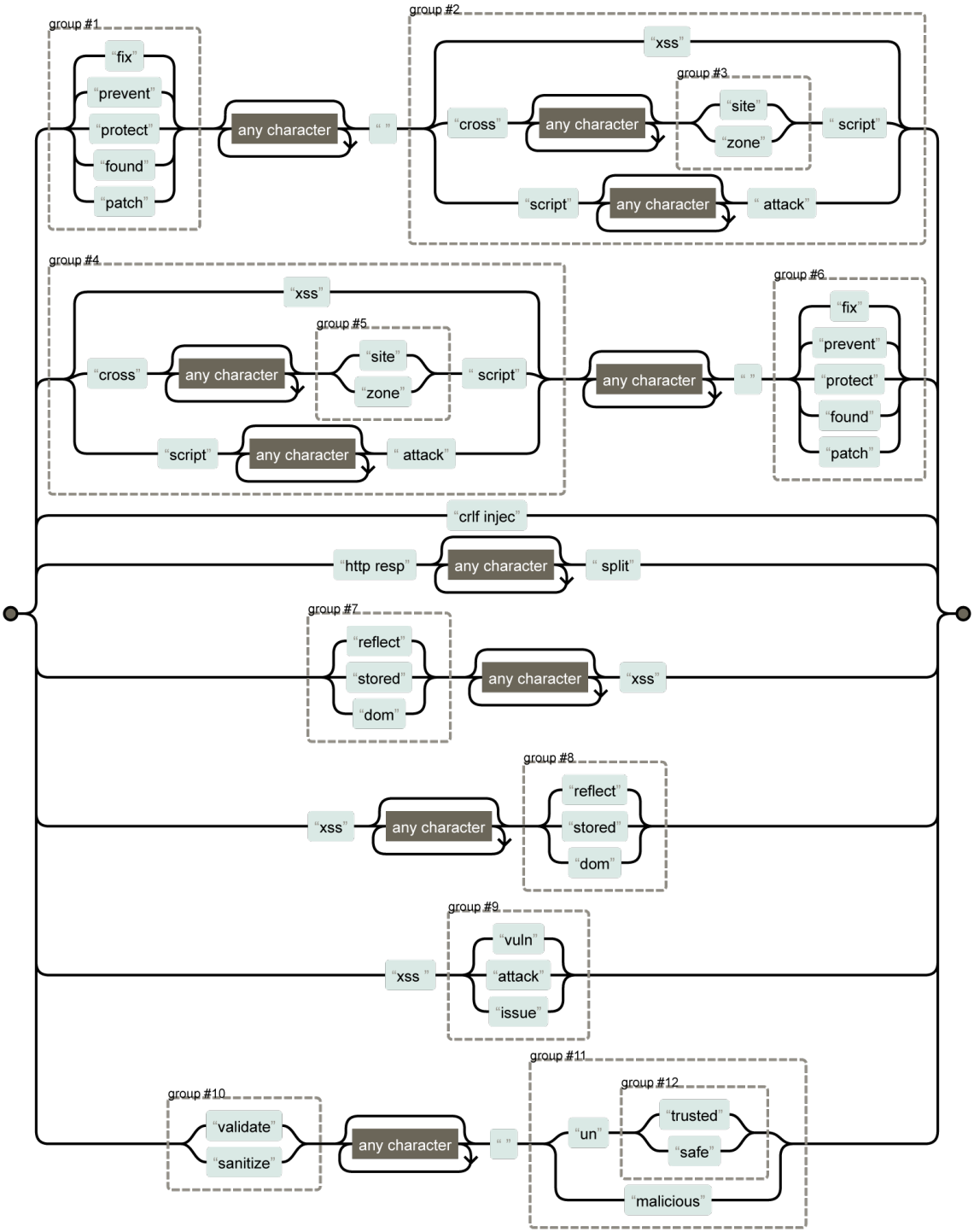

Figure A.3: Cross-Site Scripting pattern

A.1.4 A4 - Broken Access Control

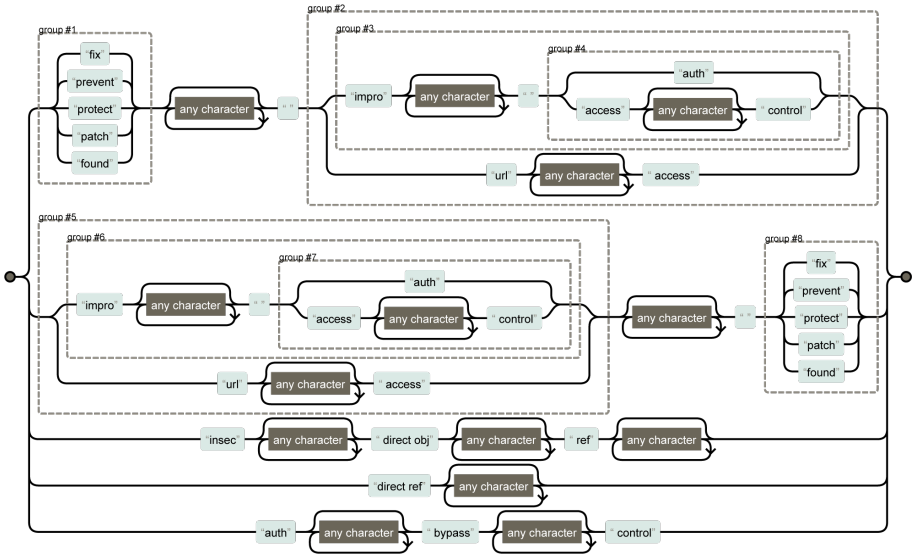

Figure A.4: Broken Access Control pattern

A.1.5 A5 - Security Misconfiguration

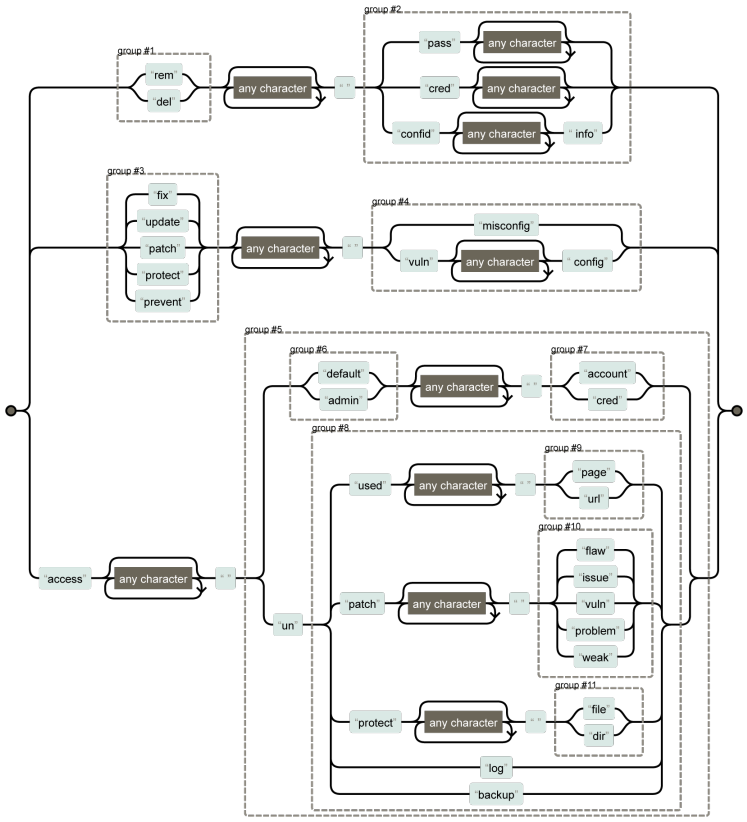

Figure A.5: Security Misconfiguration pattern

A.1.6 A6 - Sensitive Data Exposure

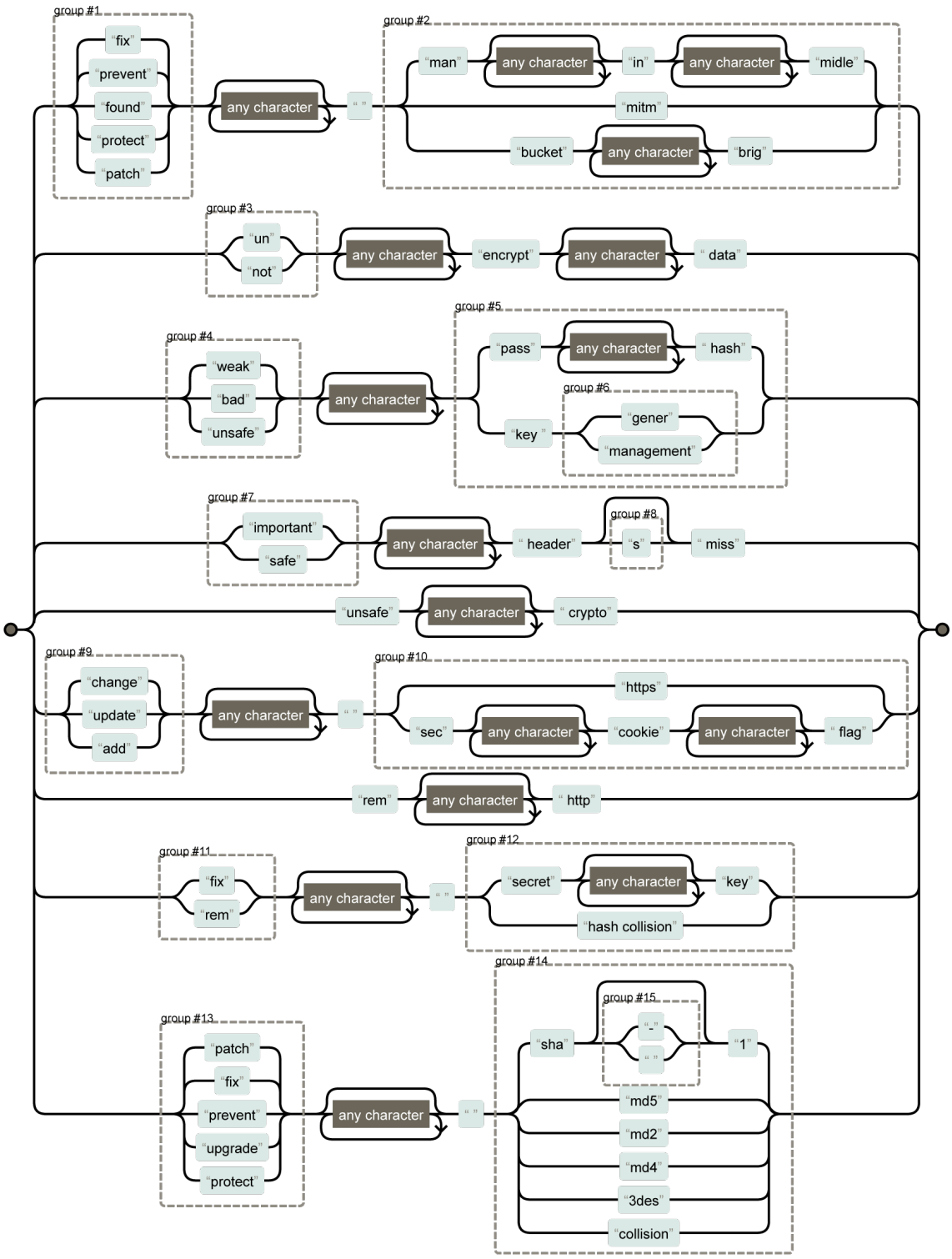

Figure A.6: Sensitive Data Exposure pattern

A.1.7 A7 - Insufficient Attack Protection

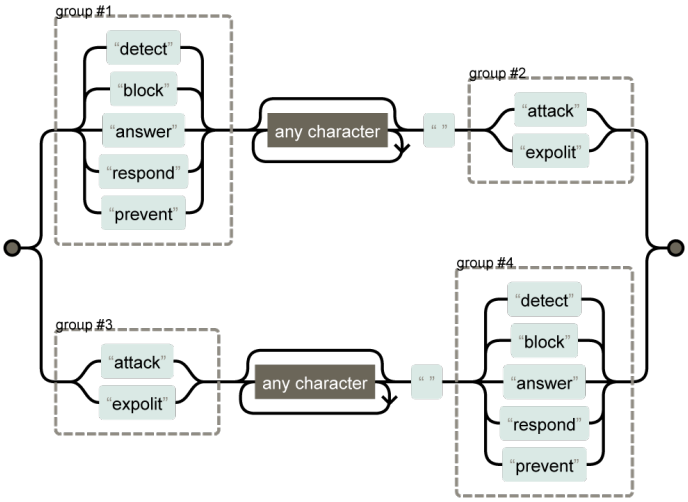

Figure A.7: Insufficient Attack Protection pattern

A.1.8 A8 - Cross-Site Request Forgery (CSRF)

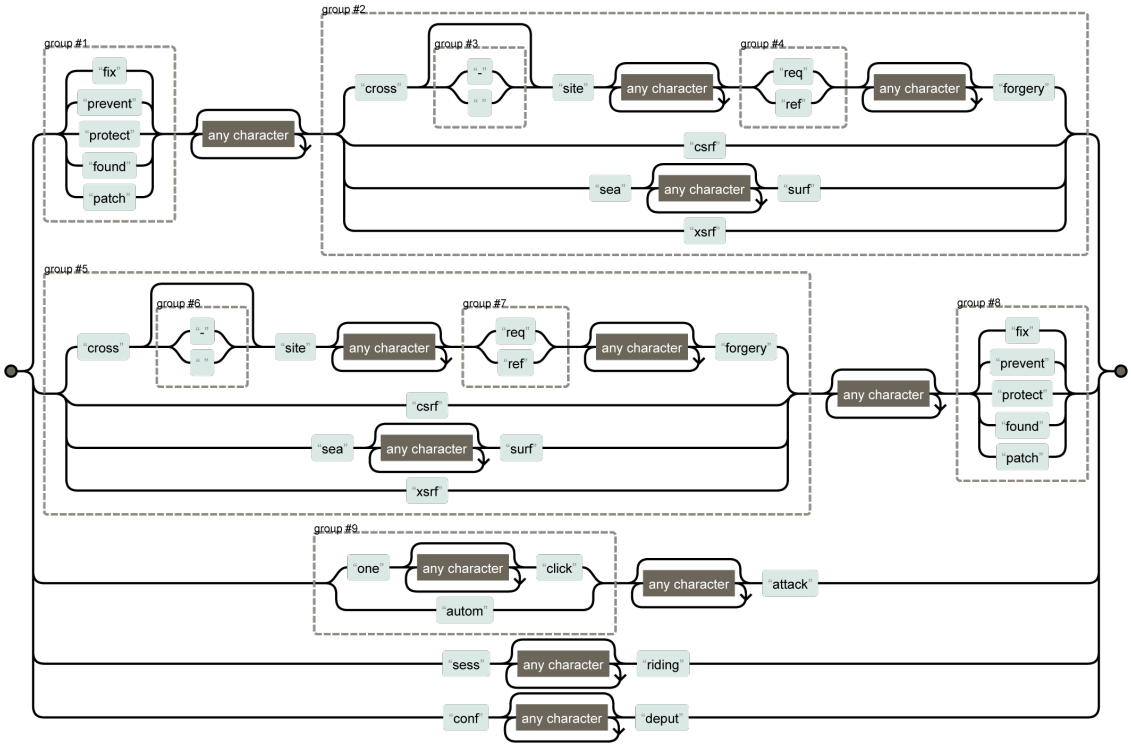

Figure A.8: Cross-Site Request Forgery pattern

A.1.9 A9 - Using Components with Known Vulnerabilities

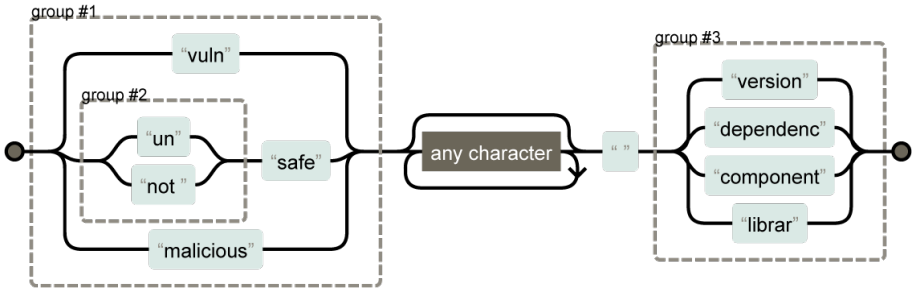

Figure A.9: Using Components with Known Vulnerabilities pattern

A.1.10 A10 - Underprotected APIs

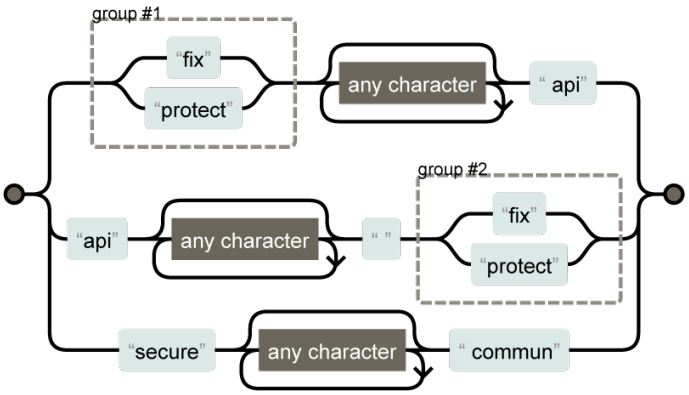

Figure A.10: Underprotected APIs pattern

A.2 Others

Other security patterns were created:

- Memory Leaks (Sec. A.2.1)
- Resource Leaks (Sec. A.2.2)
- Context Leaks (Sec. A.2.3)
- Path Traversal (Sec. A.2.4)
- Denial-of-Service (Sec. A.2.5)
- Overflow (Sec. A.2.6)
- Miscellaneous (Sec. A.2.7)

A.2.1 Memory Leaks

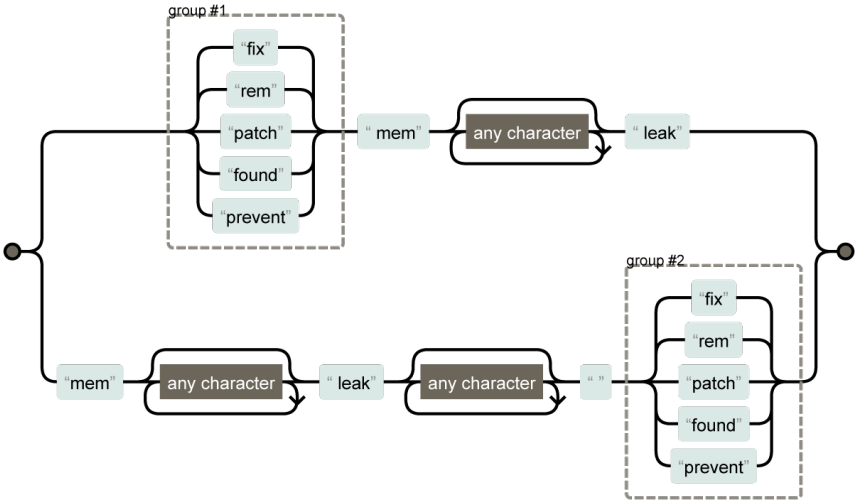

Figure A.11: Memory Leaks pattern

A.2.2 Resource Leaks

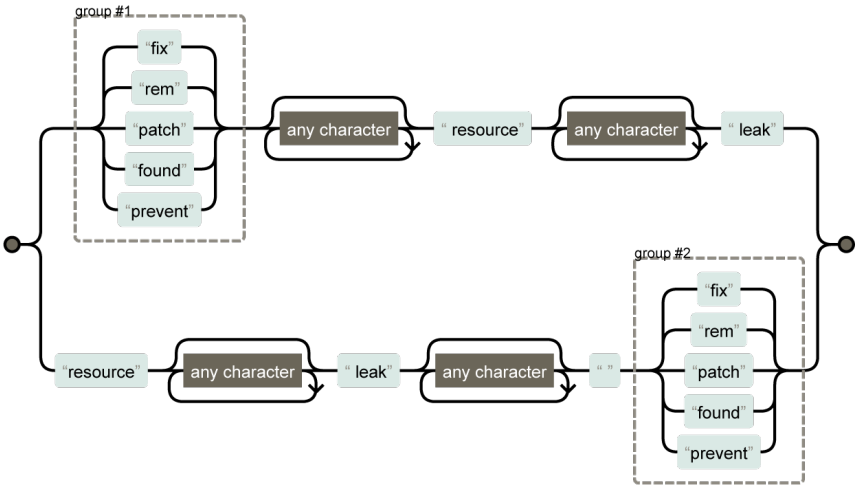

Figure A.12: Resource Leaks pattern

A.2.3 Context Leaks

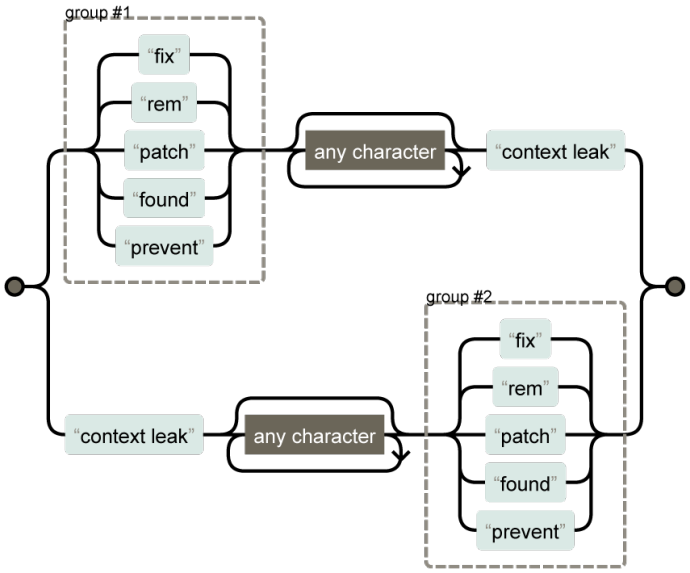

Figure A.13: Context Leaks pattern

A.2.4 Path Traversal

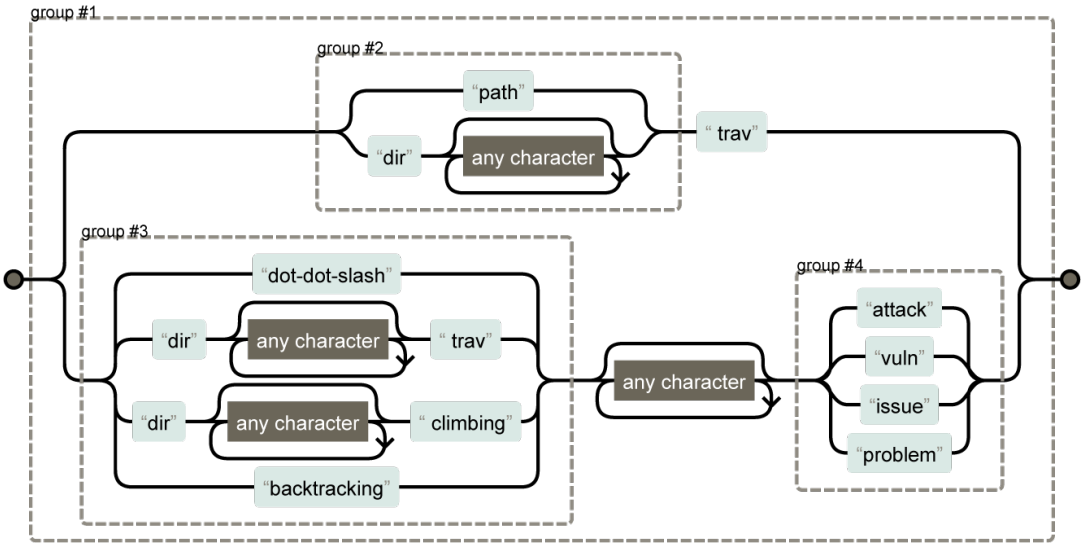

Figure A.14: Path Traversal pattern

A.2.5 Denial-of-Service

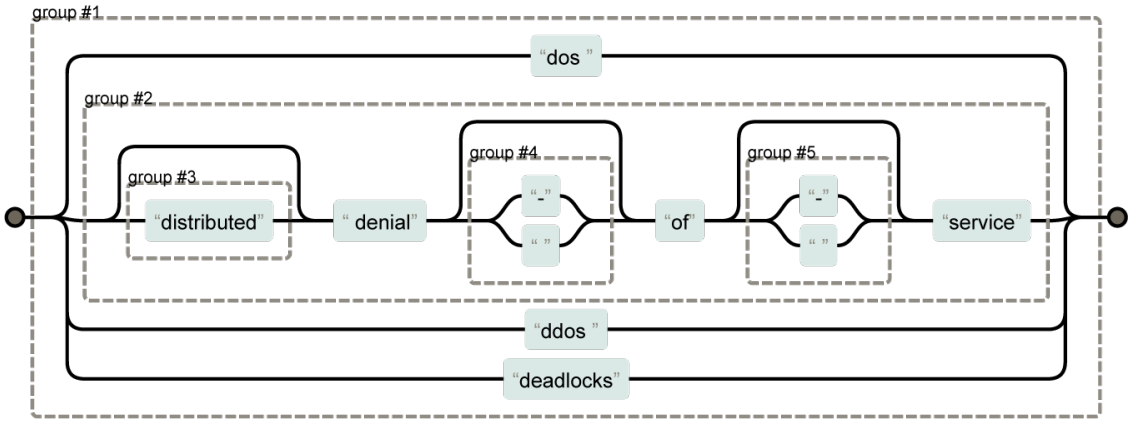

Figure A.15: Denial-of-Service pattern

A.2.6 Overflow

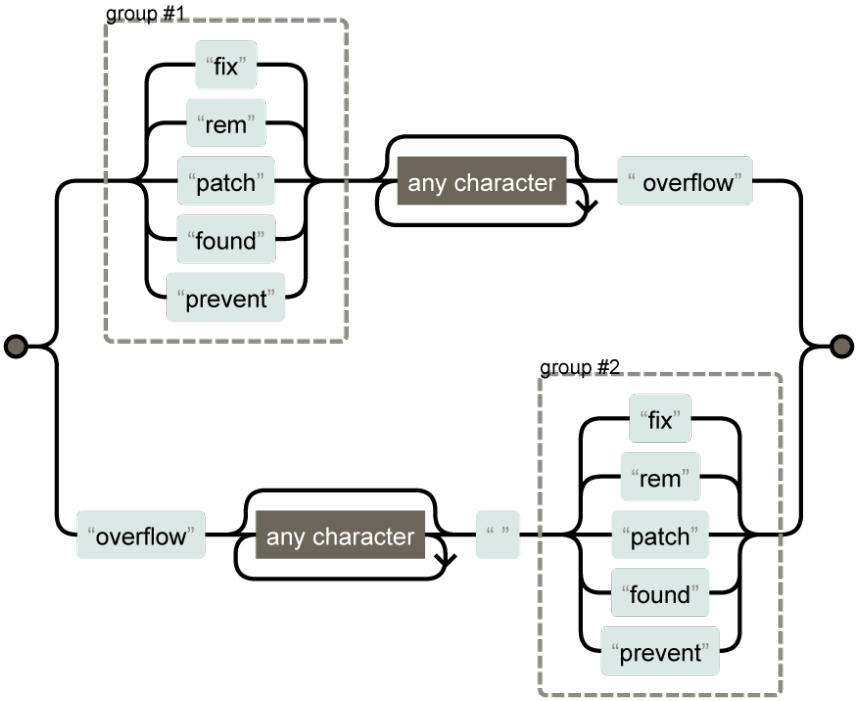

Figure A.16: Overflow pattern

A.2.7 Miscellaneous

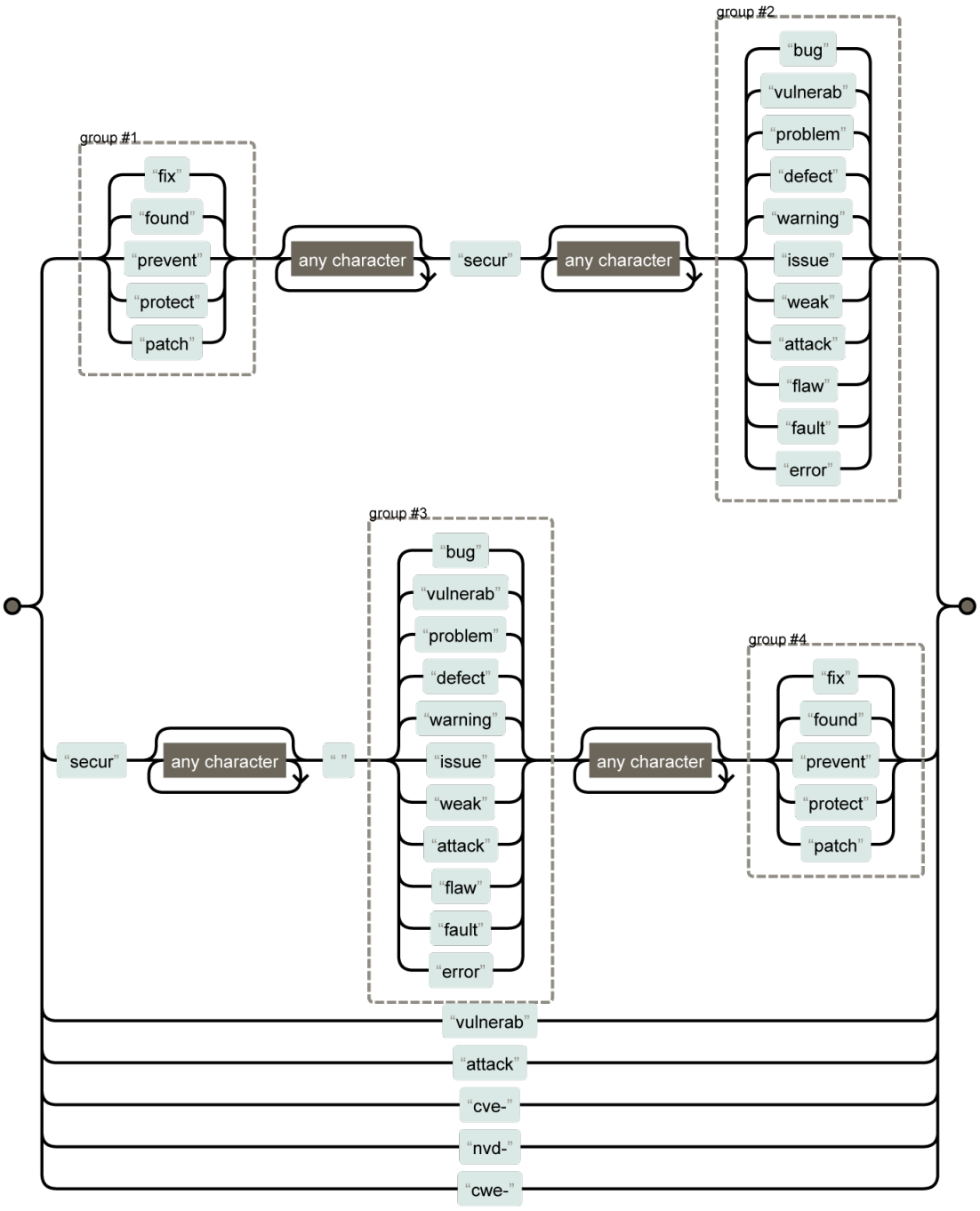

Figure A.17: Miscellaneous pattern

Appendix B

Publications

This appendix presents the scientific publications submitted and accepted in different international conferences and journal. IJSSE paper is still under production and it will be available here[1].

[1] https://www.igi-global.com/journal/international-journal-secure-software-engineering/1159

Using Github to Create a Dataset of Natural Occurring Vulnerabilities

Sofia Reis[1] and **Rui Abreu**[2]
[1]Faculty of Engineering of University of Porto, Portugal
e-mail: sofia.reis@fe.up.pt
[2]IST, University of Lisbon & INESC-ID, Portugal
e-mail: rui@computer.org

Abstract

Currently, to satisfy the potential high number of system requirements, complex software is crafted which turns its development cost-intensive and more susceptible to security vulnerabilities. According to IBM's X-Force Threat Intelligence 2017 Report, the number of vulnerabilities per year has been significantly increasing over the past years. In software security testing, performing empirical studies is challenging due to the lack of widely accepted and easy-to-use databases of real vulnerabilities as well as the fact that it requires both human effort and CPU time. Consequently, researchers tend to use databases of hand-seeded vulnerabilities, which may differ inadvertently from real vulnerabilities and thus might lead to misleading assessments of the capabilities of the tools. Although there are databases targeting security vulnerabilities test cases, only one database contains real vulnerabilities, the other ones are a mix of real and artificial or even only artificial samples. This paper explains our efforts to create a vulnerability database, *Secbench*, by mining 238 repositories from GitHub. GitHub is particularly interesting since it hosts millions of open-source projects carrying a considerable number of security vulnerabilities. More than $1M$ of commits were mined for 16 different patterns which yielded 602 security vulnerabilities. The study described in this paper provides a methodology to mining security vulnerabilities from open-source software. Our methodology has proven itself as being valuable since we were able to collect a considerable number of security vulnerabilities from a small group of repositories. However, there is still much work to do in order to improve not only the mining process but also the vulnerabilities diagnosis. All the information related to *Secbench* is available at/through `https://tqrg.github.io/secbench/`.

SECBENCH: A Database of Real Security Vulnerabilities

Sofia Reis[1] and Rui Abreu[2]

[1] Faculty of Engineering of University of Porto, Portugal
[2] IST, University of Lisbon & INESC-ID, Portugal

Abstract. Currently, to satisfy the high number of system requirements, complex software is created which turns its development cost-intensive and more susceptible to security vulnerabilities. In software security testing, empirical studies typically use artificial faulty programs because of the challenges involved in the extraction or reproduction of real security vulnerabilities. Thus, researchers tend to use hand-seeded faults or mutations to overcome these issues which might not be suitable for software testing techniques since the two approaches can create samples that inadvertently differ from the real vulnerabilities and thus might lead to misleading assessments of the capabilities of the tools. Although there are databases targeting security vulnerabilities test cases, one database contains only real vulnerabilities, the other ones are a mix of real and artificial or even only artificial samples. *Secbench* is a database of real security vulnerabilities mined from Github which hosts millions of open-source projects carrying a considerable number of security vulnerabilities. We mined 248 projects - accounting to almost $2M$ commits - for 16 different vulnerability patterns, yielding a Database with 682 real security vulnerabilities.

Keywords: Security, Real Vulnerabilities, Database, Open-Source Software, Software Testing

1 Introduction

According to IBM's X-Force Threat Intelligence 2017 Report [1], the number of vulnerabilities per year has been significantly increasing over the past 6 years. IBM's database counts with more than $10K$ vulnerabilities in 2016 alone. The most common ones are cross-site scripting and SQL injection vulnerabilities – these are two of the main classes that incorporate the Open Web Application

Copyright ©2017 by the paper's authors. Copying permitted for private and academic purposes.

In: M.G. Jaatun, D.S. Cruzes(eds.): Proceedings of the International Workshop on Security in DevOps and Agile Secure Software Engineering (SecSE 2017), published at http://ceur-ws.org

Security Project (OWASP)'s [2] 2017 Top-10 security risks. The past years have been flooded by news from the cybersecurity world: exposure of large amounts of sensitive data (e.g., $17M$ of zomato accounts stolen in 2015 which were put up for sale on a dark web marketplace only now in 2017), phishing attacks (e.g., Google Docs in 2017), denial-of-service attacks such as the one experienced last year by Twitter, The Guardian, Netflix, CNN and many other companies around the world; or, the one that possibly stamped the year, the ransomware attack which is still very fresh and kept hostage many companies, industries and hospitals information. All of these attacks were able to succeed due to the presence of security vulnerabilities in the software that were not tackled before someone exploit them. Another interesting point reported by IBM is the large number of unknown vulnerabilities (the so-called zero-day vulnerabilities), i.e., vulnerabilities that do not belong to any known attack type/surface or class which can be harmful since developers have been struggling already with the known ones.

Most software development costs are spent on identifying and correcting defects [3]. Several static analysis tools (e.g., Infer, Find Security Bugs, Symbolic PathFinder, WAP, Brakeman, Dawnscanner and more) are able to detect security vulnerabilities through a source code scan which may help to reduce the time spent on those two activities. Unfortunately, their detection capability is not the best yet (i.e., the number of false-negatives and false-positives is still high) and sometimes even comparable to random guessing [4].

Testing is one of the most important activities of software development lifecycle since it is responsible for ensuring software's quality through the detection of the conditions which may lead to software failures. In order to study and improve these software testing techniques, empirical studies using real security vulnerabilities are crucial [5] to gain a better understanding of what tools are able to detect [6]. Yet, performing empirical studies in software testing research is challenging due to the lack of widely accepted and easy-to-use databases of real bugs [7,8] as well as the fact that it requires human effort and CPU time [5]. Consequently, researchers tend to use databases of hand-seeded vulnerabilities which differ inadvertently from real vulnerabilities and thus might not work with the testing techniques under evaluation [9,10]. Although there are databases targeting security vulnerabilities test cases, only one of them contains real vulnerabilities (`Safety-db`), the other ones are a mix of real and artificial or even only artificial samples.

This paper reflects the results from mining 248 projects from Github for 16 different patterns of security vulnerabilities and attacks which led to the creation of *Secbench*, a database of real security vulnerabilities for several languages that is being used to study a few static analysis tools. The main idea is to use our database to test static analysis tools, determine the ones that perform better and possibly identify points of improvement on them. Thus, developers may be able to use the tools on the Continuous Integration and Continuous Delivery (CI/CD) pipeline which will help decrease the amount of time and money spent on vulnerabilities' correction and identification. With this study, we aim

to provide a methodology to guide mining security vulnerabilities and provide database to help studying and improving software testing techniques. Our study answers the next questions:

- **RQ1** *Is there enough information available on open-source repositories to create a database of software security vulnerabilities?*
- **RQ2** *What are the most prevalent security patterns on open-source repositories?*

More information related to *Secbench* is available at `https://tqrg.github.io/secbench/`. Our database will be publicly available with the vulnerable version and the non-vulnerable version of each security vulnerability (i.e., the fix of the vulnerability).

The paper is organized as follows: in Section 2, we present the existing related work; in Section 3, we explain how we extracted and isolated security vulnerabilities from Github repositories; in Section 4, we provide statistical information about *Secbench*; in Section 5, we discuss results and answer the research questions. And, finally, in Section 6, we draw conclusions and discuss briefly the future work.

2 Related Work

This section mentions the existing related work in the field of databases created to perform empirical studies in the software testing research area.

The `Software-artifact Infrastructure Repository` (SIR) [8] provides both real and artificial real bugs. SIR provides artefacts in Java, C/C++ and C# but most of them are hand-seeded or generated using mutations. It is a repository meant to support experimentation in the software testing domain.

The Center for Assured Software (CAS) created artificial test cases - Juliet Test Suites - to study static analysis tools. These test suites are available through National Institute of Standards and Technology (NIST). The Java suite has $25,477$ test cases for 112 different Common Weakness Enumerations (CWEs) and the C/C++ suite has $61,387$ test cases for 118 different CWEs. Each test case has a non-flawed test which will not be caught by the tools and a flawed test which should be detected by the tools.

`CodeChecker` is a database of defects which was created by Ericsson with the goal of studying and improving a static analysis tool to possibly test their own code in the future. The `OWASP Benchmark` is a free and open Java test suite which was created to study the performance of automated vulnerability detection tools. It counts with more than 2500 test cases for 11 different CWEs.

`https://samate.nist.gov/SRD/testsuite.php`
`https://cwe.mitre.org/`
`https://github.com/Ericsson/codechecker`
`https://www.owasp.org/index.php/Benchmark#tab=Main`

`Defects4j`[7] is not only a database but also an extensible framework for Java programs which provides real bugs to enable studies in the software testing research area. They started with a small database containing 375 bugs from 5 open source repositories. The researchers allow the developers to build their framework on top of the program's version control system which adds more bugs to their database. `Safety-db` is a database of python security vulnerabilities collected from python dependencies. The developers can use continuous integration to check for security vulnerabilities in the dependencies of their projects. Data is to be analyzed by dependencies and their security vulnerabilities or by Common Vulnerabilities and Exposures (CVE) descriptions and URLs.

Secbench is a database of only real security vulnerabilities for several different languages which will help software testing researchers improving the tools' capability of detecting security issues. Instead of only mining the dependencies of a project, we mine security vulnerabilities patterns through all the commits of Github repositories. The test cases - result of the patterns mining - go through an evaluation process which tells if it will integrate the final database or not.

3 Extracting And Isolating Vulnerabilities From Github Repositories

This section describes the methodology used to obtain real security vulnerabilities, from the mining process to the samples evaluation and approval. The main goal of this approach is the identification and extraction of real security vulnerabilities fixed naturally by developers on their daily basis work. The research for new methodologies to retrieve primitive data in this field is really important due to the lack of databases with a considerable amount of test cases and lack of variety for different defects and languages to support static analysis tools studies.

The first step was the identification of a considerable amount of trending security patterns (Section 3.1). Initially, the main focus was the Top 10 OWASP 2017 and other trending security vulnerabilities such as memory leaks and buffer overflows which are not much prevalent between web applications. Thereafter, more patterns were added and we still have place for much more. For each pattern, there is a collection of words and acronyms which characterizes the security vulnerability. These words were mined on commits' messages (syntactic analysis), in order to find possible candidates to test cases. Every time the tool identified a pattern, the sample was saved on the cloud and the informations attached (e.g., sha, url, type of security vulnerability) on the database. The candidates' search to our database is performed automatically using a crawler in `Python` responsible for matching our patterns with commits' messages.

As seen in Figure 1, after saving the initial data, a manual diagnosis (Section 3.3) is performed on two different types of information retrieved by our tool:

`https://github.com/pyupio/safety-db`
`https://cve.mitre.org/`

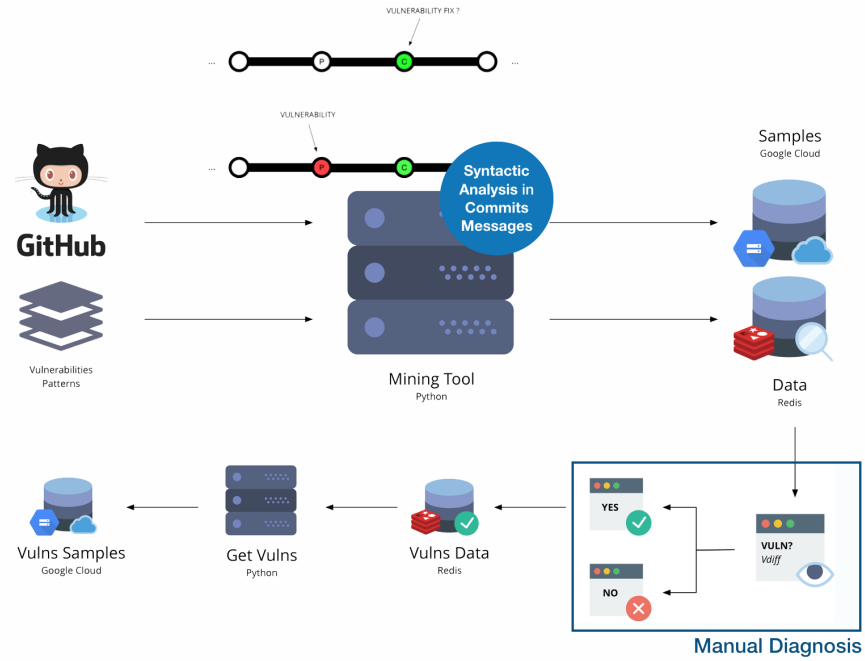

Fig. 1: Workflow to extract and identify real security vulnerabilities

– **Commit's message**, to validate if the message actually represents the fix of vulnerability or a false-positive;
– **Source code**, to identify if the pieces of code responsible for the potential vulnerability and its fix exist or not;

Both are validated manually in order to integrate the final database. If a sample is totally approved, then its information will be updated on the database and, consequently, the test case (Section 3.2) is added to the final database.

3.1 Patterns - Extracting/Detecting Vulnerabilities

The goal was mining for indications of a vulnerability fix or patch committed by a developer on a Github project. The first step was the identification of a considerable amount of trending security patterns (Section 3.1) based on annual security reports from IBM[1], OWASP[2] and ENISA[11]; cybersecurity news and sites where common security vulnerabilities are reported (e.g., CVE and CWE). In order to understand if the chosen patterns were prevalent on Github, Github BigQuery and Github searches through the search engine were used which led to a good perception of what patterns would be more difficult to collect.

For each pattern, a regular expression was created joining specific words from its own domain and words highlighting a tackle. In order to represent the tackling

of a fix, words such as *fix, patch, found, prevent* and *protect* were used (Figure 2, Example 1). In certain cases, such as the pattern *iap*, it was necessary to adjust this approach due to nature of the vulnerability. This pattern represents the lack of automated mechanisms for detecting and protecting applications. So, instead of the normal set, another words were used: *detect, block, answer* and *respond* (Figure 2, Example 2). It was necessary to adapt the words to each type of vulnerability. To really specify the patterns and distinguish between them more specific words were added. For example, to characterize cross-site scripting vulnerability tokens like *cross site scripting*, *xss*, *script attack* and many others were used.

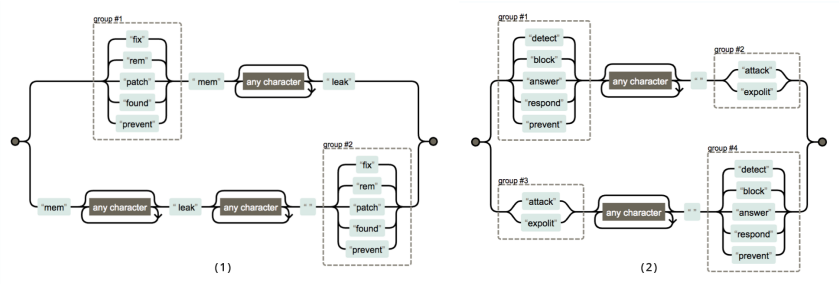

Fig. 2: Two examples of different regular expressions (Patterns)

First, we tried to create patterns for the Top 10 OWASP 2017 and then we extended the tool to others that can be found on our website: `https://tqrg.github.io/secbench/patterns.html`. Besides words related to each pattern, we added to the miscellaneous pattern (*misc*) the identification of dictionaries of common vulnerabilities or weaknesses (using regular expressions able to detect the IDs: CVE, NVD or CWE) or any cases where the message contains indications of a generic security vulnerability fix.

ID	**Pattern**
injec	Injection
auth	Broken Authentication and Session Management
xss	Cross-Site Scripting
bac	Broken Access Control
smis	Security Misconfiguration
sde	Sensitive Data Exposure
iap	Insufficient Attack Protection
csrf	Cross-Site Request Forgery
ucwkv	Using Components with Known Vulnerabilities
upapi	Underprotected APIs

Table 1: Top 10 2017 OWASP

ID	Pattern
ml	Memory Leaks
over	Overflow
rl	Resource Leaks
dos	Denial-of-Service
pathtrav	Path Traversal
misc	Miscellaneous

Table 2: Other Security Issues/Attacks

3.2 Test Cases Structure

Every time a pattern is found in a commit by the mining tool, a test case is created. The test case has 3 folders: V_{fix} with the non-vulnerable source code from the commit where the pattern was caught (child), V_{vul} with the vulnerable source code from the previous commit (parent) which we consider the real vulnerability; and, V_{diff} with two folders, added and deleted, where the added lines to fix the vulnerability and the deleted lines that represent the security vulnerability are stored (Figure 3).

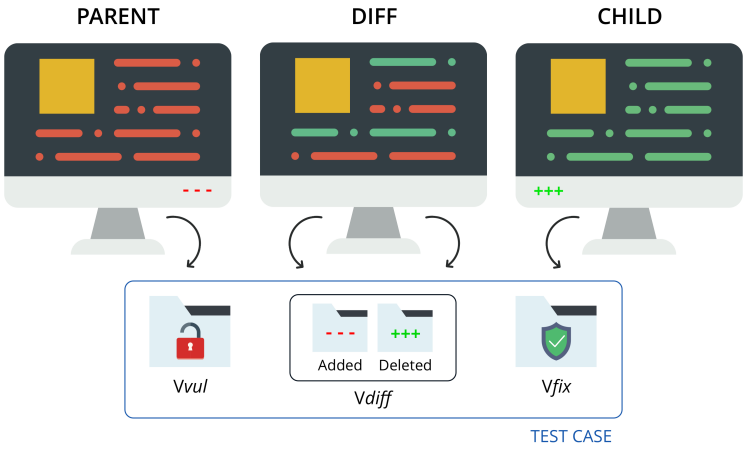

Fig. 3: Difference between V_{fix}, V_{vul} and V_{diff}

3.3 Sample Diagnosis

After obtaining the sample and its information, a manual diagnosis was performed on two different kinds of information retrieved from Github (commit's message and source code). For each single candidate, we evaluated if the message really reflected indications of a vulnerability fix because some of the combinations

represented by the regular expressions can lead to false positives, i.e., messages that do not represent the actual vulnerability fix. The example presented in Figure 4 shows not only how the mining tool finds two candidates matching the *over* pattern (red boxes) but also how those two samples were finally diagnosed. The first reflects a real security vulnerability (buffer overflow) but the second one represents a `CSS` issue (i.e., not a security vulnerability). Thus, the second example is pointed out as non-viable and automatically not considered for the final database.

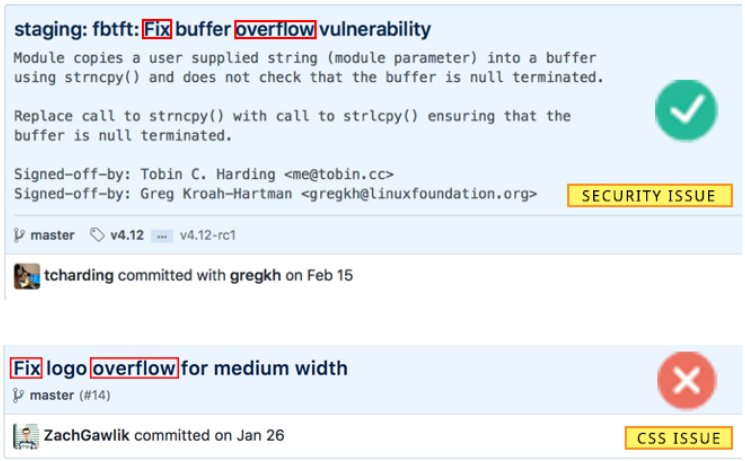

Fig. 4: Commits' messages diagnosis example

If the analysis succeeds (first message, Figure 4), then the code evaluation is performed through the *diff* source code analysis. Hopefully, the researcher is capable of isolating manually the functions or problems in the code responsible for the fix and the vulnerability. During the study, several cases were inconclusive, mainly due to the difficulties in understanding the code structure or when the source code did not reflect the message. Normally, these last cases were pointed out has non-viable, except when there was something that could be the fix but the researcher did not get it. In that case, they were put on hold as a *doubt* which means that the case needs more research.

To validate the source code much research was made on books, security cheatsheets online, vulnerabilities dictionary websites and many other sources of knowledge. Normally, the process would be giving a first look at the code trying to highlight a few functions or problems that could represent the vulnerability and then make a search on the internet based on the language, frameworks and information obtained by the *diff*. The example presented below is easy to identify because the socket initialized in the beginning needs to be released before the function returns on the two different conditions (line 282 and 292) otherwise

we have two resource leaks. It was not always like this, sometimes it was really difficult to understand where the issues were due to the source code complexity.

Fig. 5: Example of two resource leaks identification

Besides the validation, the set of requirements presented below needs to be fulfilled, in order to approve a test case as viable to the final test suite:

- **The vulnerability belongs to the class where it is being evaluated**
  If it does not belong to the class under evaluation the vulnerability its put on hold for later study except if the class under evaluation is the Miscellaneous class which was made to mine vulnerabilities that might not belong to the other patterns; or, to catch vulnerabilities that may skip in other patterns due to the limitations of using regular expressions in these cases.
- **The vulnerability is isolated**
  We accepted vulnerabilities which additionally include the implementation of other features, refactoring or even fixing of several security vulnerabilities. But the majority of security vulnerabilities founded are identified by the file names and lines where they are positioned. We assume all V_{fix} is necessary to fix the security vulnerability.
- **The vulnerability needs to really exist**
  Each sample was evaluated to see if it is a real vulnerability or not. During the analysis of several samples commits that were not related to security vulnerabilities and fixes of vulnerabilities, i.e., not real fixes were caught.

3.4 Challenges

These requirements were all evaluated manually, hence a threat to the validity as it can lead to human errors (e.g., bad evaluations of the security vulnerabilities

and adding replicated samples). However, we attempted to be really meticulous during the evaluation and when we were not sure about the security vulnerability nature we evaluated with a D (Doubt) and with R (Replica) when we detected a replication of another commit (e.g., merges or the same commit in other class). Sometimes it was hard to reassign the commits due to the similarity between patterns (e.g., *ucwkv* and *upapi*). Another challenge was the trash (i.e., commits that did not represent vulnerabilities) that came with the mining process due to the use of regular expressions.

4 Empirical Evaluation

In this section, we report the results that we obtained through our study and answer the research questions.

4.1 Database of Real Security Vulnerabilities

This section provides several interesting statistics about *Secbench* that were obtained during our study. Our database contains 682 real security vulnerabilities, mined from 248 projects - the equivalent to 1978482 commits - covering 16 different vulnerability patterns (Tables 1 and 2).

In order to obtain a sample which could be a good representative of the population under study, it was ensured that the top 5 of most popular programming languages on Github and different sizes of repositories would be covered. Due to the large amount of Github repositories ($61M$) and constant modification, it is very complicated to have an overall of the exact characteristics that the sample under study should have in order to, approximately, represent the domain under study. According to a few statistics collected from the Github blog and GitHut, some of the most popular programming languages on Github are JavaScript, Java, Python, Ruby, PHP, CSS, C, C++, C# and Objective-C. We tried to, mainly, satisfy the top 5 of most popular programming languages on Github (i.e., with higher number repositories): JavaScript ($979M$), Java ($790M$), Python ($510M$), Ruby ($498M$) and PHP ($458M$). Other than covering the top 5, we also tried to have a good variety of repositories sizes since Github has repositories from different dimensions. Our database has repositories with sizes between 2 commits and $700M$ commits. It would be expected that the result of mining larger repositories would easily lead to more primitive data. But since the goal is to have a good representation of the whole Github, it is necessary to also contain smaller repositories, in order to reach balanced conclusions and predictions. Github has a wide variety of developers whose programming skills can be good or bad. This can be a threat to test cases quality. But we are not able to identify repositories quality in an automated way yet. Also, due to the structure of Github, the main limitation that we were not able to tackle was dealing with

`https://github.com/blog/2047-language-trends-on-github`
`http://githut.info/`

samples with more than one parent. Sometimes in our manual diagnosis, we detected samples with 4 or 5 parents (e.g., merges). We tacked the issue analyzing each single parent to detect the one containing the potential vulnerability.

Throughout our diagnosis process, we were able to identify several CVE identifiers. Thus, 105 out of the 682 security vulnerabilities are identified using the CVE identification system. These 105 vulnerabilities belong to 98 different CVE classes for 12 different years (e.g., CVE-2013-0155 and CVE-2017-7620). The identifier for weaknesses (CWE) was never identified through our manual diagnosis which reflects the information retrieved by Github's search engine: only $12K$ of commits' messages containing CWE but $2M$ for CVE.

Year	2017	2016	2015	2014	2013	2012	2011	2010	2009	2008	2007	2006
#CVE	4	20	13	22	16	9	9	5	2	3	1	1

Table 3: Vulnerabilities identified with CVE

SecBench includes security vulnerabilities from 1999 to 2017, being the group of years between 2012 and 2016 the one with the highest value of accepted vulnerabilities (especially 2014 with a percentage of 14.37%). This supports the IBM's X-Force Threat Intelligence 2017 Report [1] where it was concluded that in the last 5 years the number of vulnerabilities per year had a significant increase compared with the other years. The decrease of security vulnerabilities in the last 2 years, it definitely does not reflect the news and security reports. However, these reports contain all kinds of software and the study is only performed on open-source software.

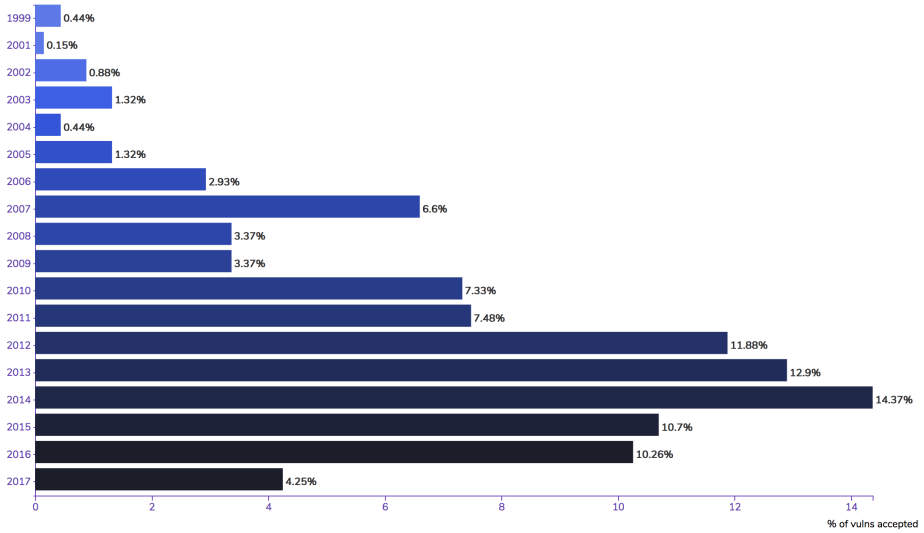

Fig. 6: Distribution of real security vulnerabilities per year

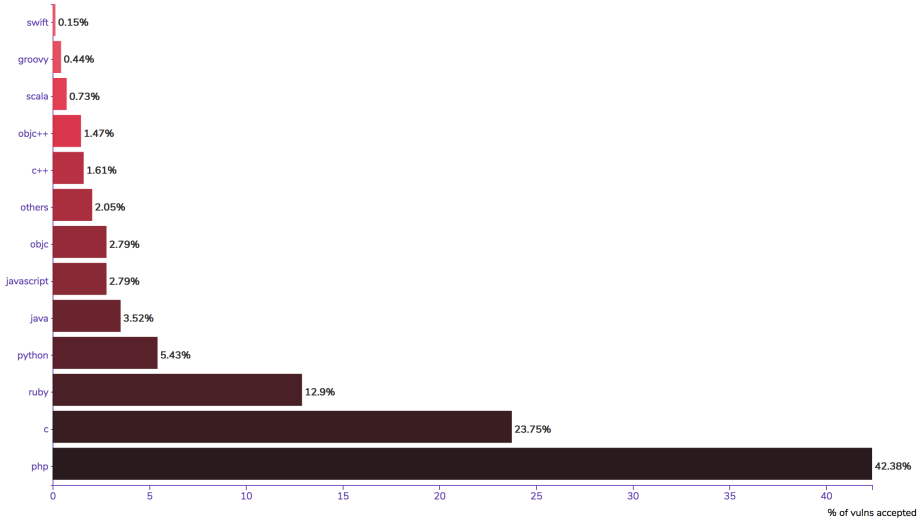

Fig. 7: Distribution of real security vulnerabilities per language

The decrease can reflect the concerns of the core developers within making the code public since the number of attacks is increasing and one of the potential causes can be the code availability. Except for 2000, we were able to collect test cases from 1999 to 2017. The last 5 years (excluding 2017) were the years with the higher percentage of vulnerabilities. The sample covers more than 12 different languages being PHP (42.38%), C (23.75%), and Ruby (12.9%) the languages with the higher number of test cases (Figure 7). This supports the higher percentage of security vulnerabilities caught for *injec* (16.1%), *xss* (23.4%) and *ml* (12.8%) - Figure 6 - since C is a language where memory leaks are predominant and Ruby and PHP are scripting languages where Injection and Cross-Site Scripting are popular vulnerabilities. Although the database contains 94 different languages, it was only possible to collect viable information for 12 different languages.

4.2 Research Questions

As mentioned before, there are several automated tools that can scan security vulnerabilities on source code. Yet, their performance is still far from an acceptable level of maturity. To find points of improvement it is necessary to study them using real security vulnerabilities. The primary data for this kind of studies is scarce as we discussed on Section 2, so we decided to first evaluate if there is enough information on Github repositories to create a database of real security vulnerabilities (**RQ1**). And if yes, what are the security patterns we can most easily find on open source repositories (**RQ2**).

- RQ1: Is there enough information available on open-source repositories to create a database of software security vulnerabilities?

To answer this question, it was necessary to analyze the distribution of real security vulnerabilities across the 248 mined Github repositories. As a result of our mining process for the 16 different patterns (Table 4), 62.5% of the repositories contain vulnerabilities (*VRepositories*) and 37.5% contained 0 vulnerabilities.

#Vulns	**#Repositories**	**Repositories(%)**
> 0	155	62.5%
$= 0$	93	37.5%
Total	248	100%

Table 4: Mined Vulnerabilities Distribution

After mining the repositories, the manual evaluation was performed where each candidate had to fulfil a group of requirements (Section 3.3). As we can see on Table 5, the percentage of success, i.e., repositories containing vulnerabilities, decreases to 54.19%. The approximate difference of 8% is due to the cleaning process made through the evaluation process where a human tries to understand if the actual code fixes and represents a security vulnerability or not. Although the decrease from one process to another, we can still obtain a considerable percentage ($> 50\%$) of *VRepositories* containing real vulnerabilities.

#AVulns	**#VRepositories**	**VRepositories(%)**
> 0	84	54.19%
$= 0$	71	45.81%
Total	155	100%

Table 5: Accepted Vulnerabilities (AVulns) Distribution

In the end, we were able to extract vulnerabilities with an existence ratio of ≈ 2.75 (682/248). The current number of repositories on Github is $61M$, so based on the previous ratio we can possibly obtain a database of ≈ 168 millions ($167750K$) of real security vulnerabilities which is $\approx$ 246 thousand (245968) times higher than the current database. Between 2 and 3 months, we were able to collect 682 real security vulnerabilities for 16 different patterns with a resulting success of 54.19% of vulnerabilities accepted. Thus, we can conclude that it is possible to extract a considerable amount of vulnerabilities from open source software to create a database of real security vulnerabilities that will highly contribute to the software security testing research area. Due to the constant change and dimension of Github, the lack of information about the domain and the small size of the sample under study, it may not be plausible to take this conclusion. However, based on results obtained we believe the answer to this question is indeed positive.

There are enough vulnerabilities available on open-source repositories to create a database of real security vulnerabilities.

- RQ2: What are the most prevalent security patterns on open-source repositories?

This research question attempts to identify the most prevalent security patterns on open-source repositories.

After mining and evaluating the samples, the results for 16 different patterns were obtained being the two main groups the ones presented on Figure 8, Top 10 OWASP and others. *xss* (20.67%), *injec* (14.81%) and *ml* (12.46%) are the trendiest patterns on OSS which is curious since *injec* takes the first place on Top 10 OWASP 2017 [2] and *xss* the second. *ml* does not integrate into the top ten because it is not a vulnerability normally found on web applications. Injection and Cross-Site Scripting are easy vulnerabilities to catch since the exploits are similar and exist, mainly, due to the lack of data sanitization which oftentimes is forgotten by the developers. The only difference between the two is the side from where the exploit is done (server or client). Memory leaks exist because developers do not manage memory allocations and deallocations correctly. These kind of issues are one of the main reasons of *dos* attacks and regularly appeared on the manual evaluations, even in the *misc* class. Although these three patterns are easy to fix, the protection against them is also typically forgotten. Another prevalent pattern that is not considered is *misc* because it contains all the other vulnerabilities and attacks found that do not belong to any of the chosen patterns or whose place was not yet well-defined. One example of vulnerabilities that you can find on *misc* (14.37%) are vulnerabilities that can lead to timing attacks where an attacker can retrieve information about the system through the analysis of the time taken to execute cryptographic algorithms. There is already material that can possibly result in new patterns through the *misc* class analysis.

Although *auth* (6.6%) is taking the second place on Top 10 OWASP 2017, it was not easy to find samples that resemble this pattern maybe because of the fact that highlighting these issues on Github can reveal other ones in their session management mechanisms and, consequently, leading to session hijacking attacks. The *csrf* (4.99%) and *dos* (6.16%) patterns are seen frequently among Github repositories: adding protection through unpredictable tokens and fixing several issues which lead to denial-of-service attacks. The most critical patterns to extract are definitely *bac* (0.29%), which detects unproved access to sensitive data without enforcements; *upapi* (1.03%), which detects the addition of mechanisms to handle and answer to automated attacks; and, *smis* (1.32%) involving default information on unprotected files or unused pages that can give unauthorized access to attackers. *rl* (1.76%) is another pattern whose extraction was hard. Although, memory leaks are resource leaks, here only the vulnerabilities related to the need of closing files, sockets, connections, etc, were considered.

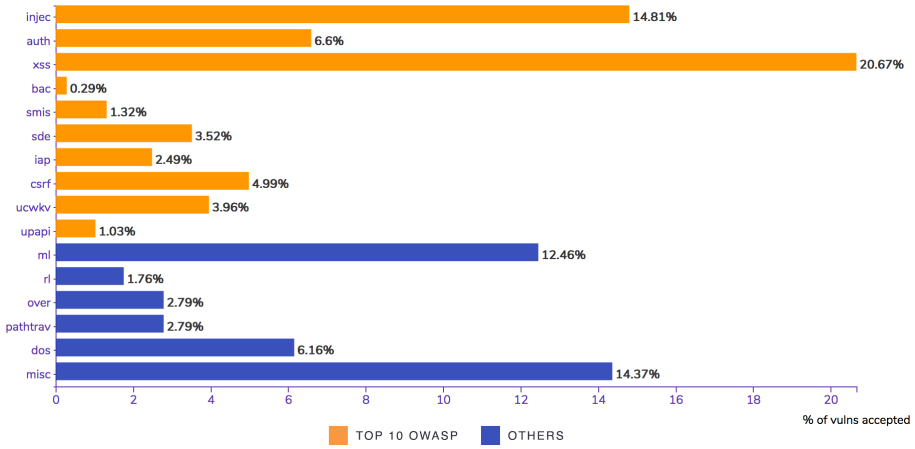

Fig. 8: Distribution of real security vulnerabilities by pattern

The other patterns (e.g., *sde, iap, ucwkv, over and pathtrav*) were pretty common during our evaluation process and also on our Github searches. The *over* pattern contains vulnerabilities for several types of overflow: heap, stack, integer and buffer. Another interesting point here is the considerable percentage of *iap* (2.49%), which normally is the addition of methods to detect attacks. This is the first time that *iap* makes part of the top 10 OWASP 2017 and still, we were able to detect more vulnerabilities for that pattern, than for *bac* which was already present in 2003 and 2004. From 248 projects, the methodology was able to collect 682 vulnerabilities distributed by 16 different patterns.

> The most prevalent security patterns are Injection, Cross-Site Scripting and Memory Leaks.

5 Conclusions & Future Work

This paper proposes a database, coined *Secbench*, containing real security vulnerabilities. In particular, *Secbench* is composed of 682 real security vulnerabilities, which was the outcome of mining 248 projects - accounting to almost $2M$ commits - for 16 different vulnerability patterns.

The importance of this database is the potential to help researchers and practitioners alike improve and evaluate software security testing techniques. We have demonstrated that there is enough information on open-source repositories to create a database of real security vulnerabilities for different languages and patterns. And thus, we can contribute to considerably reduce the lack of real security vulnerabilities databases. This methodology has proven itself as being

very valuable since we collected a considerable number of security vulnerabilities from a small group of repositories (248 repositories from $61M$). However, there are several points of possible improvements, not only in the mining tool but also in the evaluation and identification process which can be costly and time-consuming.

As future work, we plan to augment the amount of security vulnerabilities, patterns and languages support. We will continue studying and collecting patterns from Github repositories and possibly extend the study to other source code hosting websites (e.g., bitbucket, svn, etc). We will also explore natural processing languages, in order to introduce semantics and, hopefully, decrease the percentage of garbage associated with the mining process.

References

1. USA, I.S.D.: Ibm x-force threat intelligence index 2017. Technical report, IBM (March 2017)
2. Foundation, T.O.: Oswap top 10 - 2017: The ten most critical web application security risks. Technical report, The OWASP Foundation (February 2017) Release Candidate.
3. Tassey, G.: The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (May 2002)
4. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Inf. Softw. Technol. **68**(C) (December 2015) 18–33
5. Briand, L.C.: A critical analysis of empirical research in software testing. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). (Sept 2007) 1–8
6. Briand, L., Labiche, Y.: Empirical studies of software testing techniques: Challenges, practical strategies, and future research. SIGSOFT Softw. Eng. Notes **29**(5) (September 2004) 1–3
7. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ISSTA 2014, New York, NY, USA, ACM (2014) 437–440
8. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Softw. Engg. **10**(4) (October 2005) 405–435
9. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014, New York, NY, USA, ACM (2014) 654–665
10. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: ICSE 2017, Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, Argentina (May 2017)
11. for Network, E.U.A., Security, I.: Enisa threat landscape report 2016. Technical report (January 2017)