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

In: M.G. Jaatun, D.S. Cruzes (eds.): Proceedings of the International Workshop on Secure Software Engineering in DevOps and Agile Development (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/

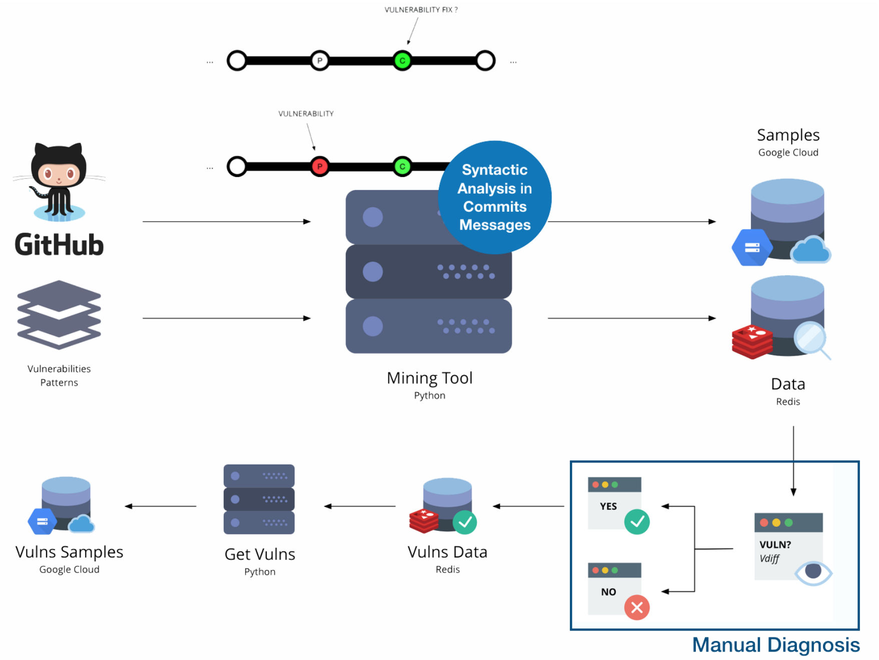


Fig. 1: Workflow to extract and identify real security vulnerabilities

- **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.

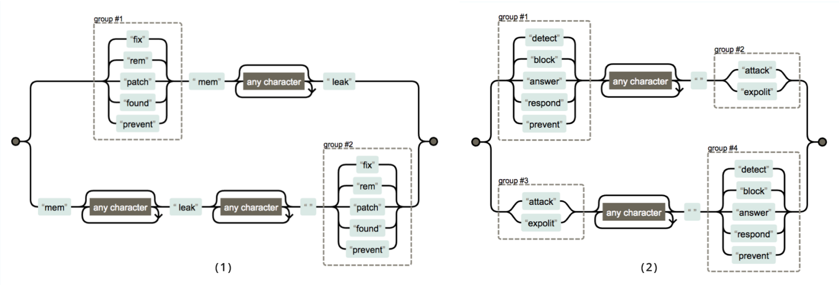


Fig. 2: Two examples of different regular expressions (Patterns)

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).

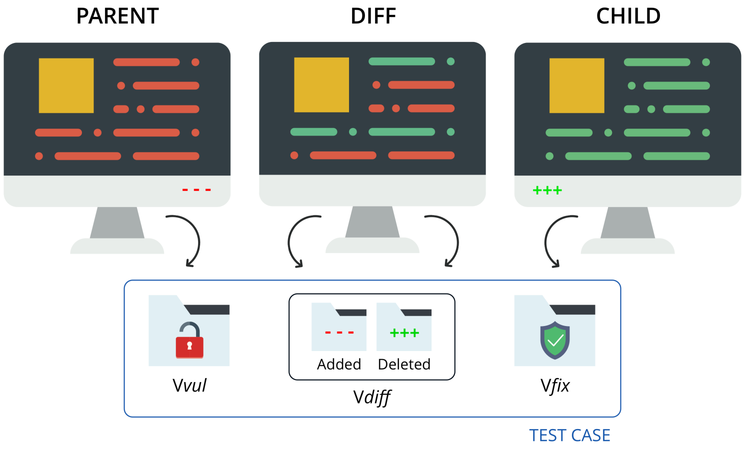
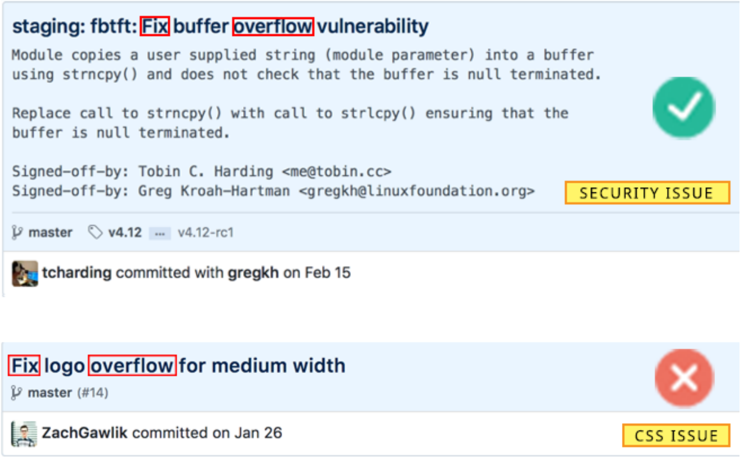


Fig. 3: Difference between V_{fix}, V_{vul} and V_{diff}

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.



staging: fbtft: Fix buffer overflow vulnerability

```
Module copies a user supplied string (module parameter) into a buffer
using strncpy() and does not check that the buffer is null terminated.

Replace call to strncpy() with call to strlcpy() ensuring that the
buffer is null terminated.

Signed-off-by: Tobin C. Harding <me@tobin.cc>
Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
```

✓ SECURITY ISSUE

master v4.12 ... v4.12-rc1

tcharding committed with gregkh on Feb 15

Fix logo overflow for medium width

master (#14)

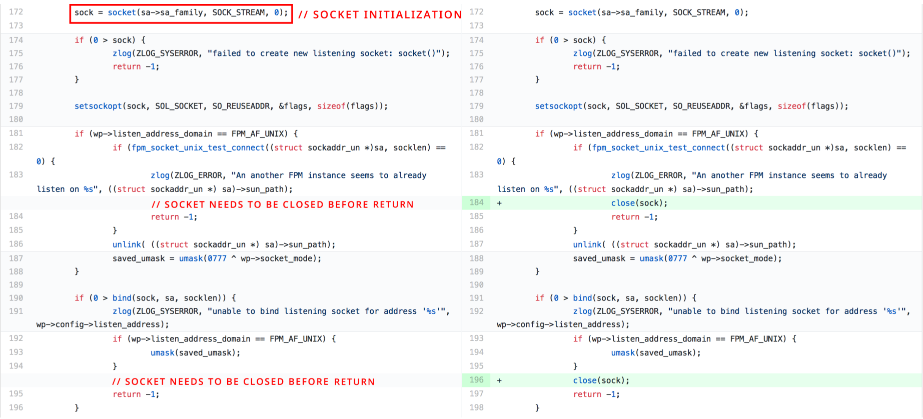
ZachGawlik committed on Jan 26

✗ CSS ISSUE

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.



```
        sock = socket(sa->sa_family, SOCK_STREAM, 0);  // SOCKET INITIALIZATION

        if (0 > sock) {
                zlog(ZLOG_SYSERROR, "failed to create new listening socket: socket()");
                return -1;
        }

        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &flags, sizeof(flags));

        if (wp->listen_address_domain == FPM_AF_UNIX) {
                if (fpm_socket_unix_test_connect((struct sockaddr_un *)sa, socklen) ==
0) {
                        zlog(ZLOG_ERROR, "An another FPM instance seems to already
listen on %s", ((struct sockaddr_un *) sa)->sun_path);
                        // SOCKET NEEDS TO BE CLOSED BEFORE RETURN
                        return -1;
                }
                unlink( ((struct sockaddr_un *) sa)->sun_path);
                saved_umask = umask(0777 ^ wp->socket_mode);
        }

        if (0 > bind(sock, sa, socklen)) {
                zlog(ZLOG_SYSERROR, "unable to bind listening socket for address '%s'",
wp->config->listen_address);
                if (wp->listen_address_domain == FPM_AF_UNIX) {
                        umask(saved_umask);
                }
                // SOCKET NEEDS TO BE CLOSED BEFORE RETURN
                return -1;
        }
```

```
        sock = socket(sa->sa_family, SOCK_STREAM, 0);

        if (0 > sock) {
                zlog(ZLOG_SYSERROR, "failed to create new listening socket: socket()");
                return -1;
        }

        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &flags, sizeof(flags));

        if (wp->listen_address_domain == FPM_AF_UNIX) {
                if (fpm_socket_unix_test_connect((struct sockaddr_un *)sa, socklen) ==
0) {
                        zlog(ZLOG_ERROR, "An another FPM instance seems to already
listen on %s", ((struct sockaddr_un *) sa)->sun_path);
+                       close(sock);
                        return -1;
                }
                unlink( ((struct sockaddr_un *) sa)->sun_path);
                saved_umask = umask(0777 ^ wp->socket_mode);
        }

        if (0 > bind(sock, sa, socklen)) {
                zlog(ZLOG_SYSERROR, "unable to bind listening socket for address '%s'",
wp->config->listen_address);
                if (wp->listen_address_domain == FPM_AF_UNIX) {
                        umask(saved_umask);
                }
+               close(sock);
                return -1;
        }
```

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.

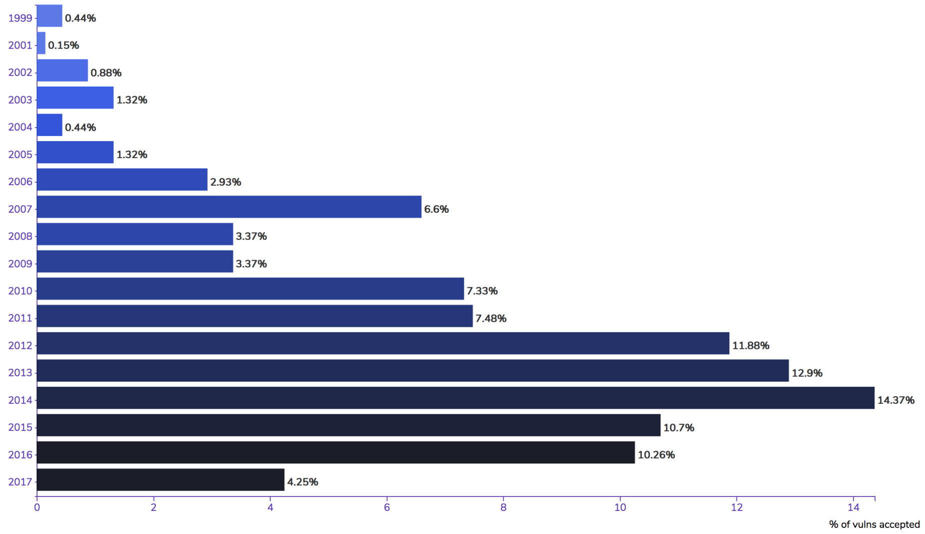


Fig. 6: Distribution of real security vulnerabilities per year

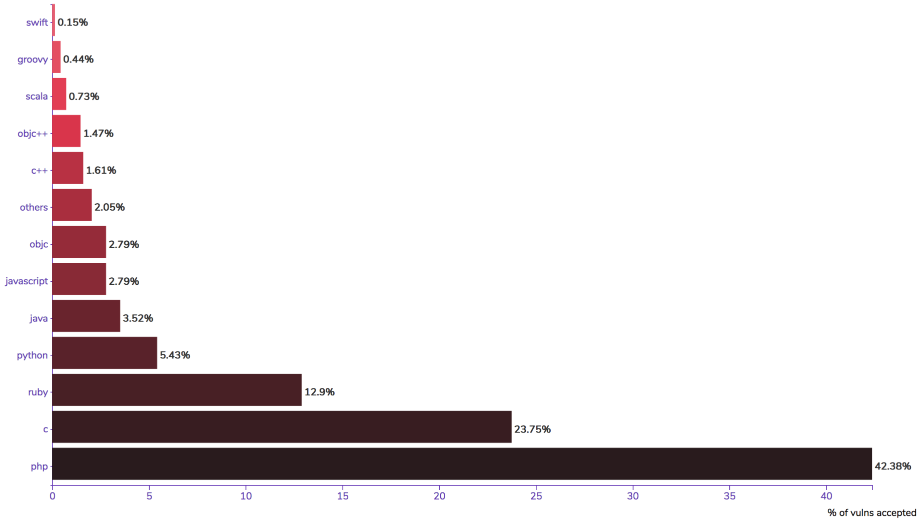


Fig. 7: Distribution of real security vulnerabilities per language

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 ≈ 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.

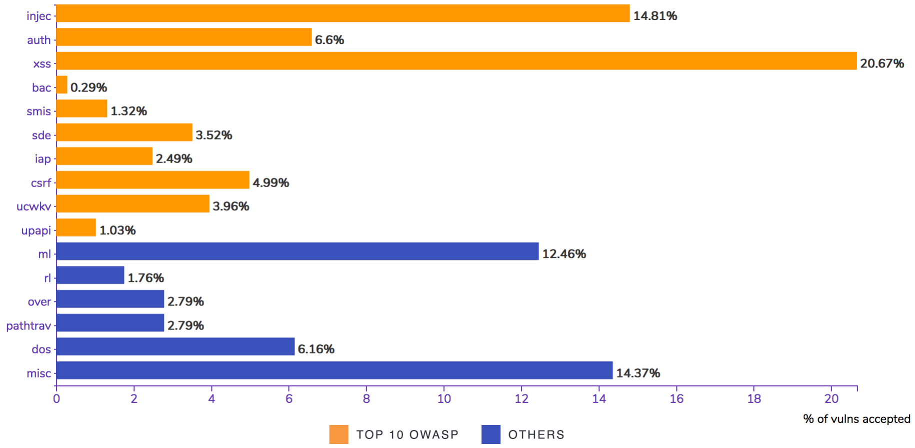


Fig. 8: Distribution of real security vulnerabilities by pattern

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. IBM: IBM X-Force threat intelligence index 2017. Technical report, IBM Security Department USA (March 2017)
2. OWASP: 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. ENISA: Enisa threat landscape report 2016. Technical report, European Union Agency for Network and Information Security (January 2017)