A Screening Test for Disclosed Vulnerabilities in FOSS Components

Stanislav Dashevskyi, Achim D. Brucker, and Fabio Massacci

Abstract—Free and Open Source Software (FOSS) components are ubiquitous in both proprietary and open source applications. Each time a vulnerability is disclosed in a FOSS component, a software vendor using this component in an application must decide whether to update the FOSS component, patch the application itself, or just do nothing as the vulnerability is not applicable to the older version of the FOSS component used. This is particularly challenging for enterprise software vendors that consume thousands of FOSS components and offer more than a decade of support and security fixes for their applications. Moreover, customers expect vendors to react quickly on disclosed vulnerabilities—in case of widely discussed vulnerabilities such as Heartbleed, within hours.
To address this challenge, we propose a *screening test*: a novel, automatic method based on thin slicing, for estimating quickly whether a given vulnerability is present in a consumed FOSS component by looking across its entire repository. We show that our screening test scales to large open source projects (e.g., Apache Tomcat, Spring Framework, Jenkins) that are routinely used by large software vendors, scanning thousands of commits and hundred thousands lines of code in a matter of minutes.
Further, we provide insights on the empirical probability that, on the above mentioned projects, a potentially vulnerable component might not actually be vulnerable after all.

Index Terms—Security maintenance; Security vulnerabilities; Patch Management; Free and Open Source Software

1 Introduction

According to a recent Black Duck study [1], more than 65 percent of proprietary applications leverage Free and Open Source Software (FOSS). This choice speeds up application development [2], as FOSS components are often used "as-is" without any modifications [3]. The price to pay is that vulnerabilities discovered in a FOSS component may affect the application that consumes it [4].

To avoid this problem a vendor could frequently and automatically update one's product (and thus its FOSS components). Such a maintenance policy can be acceptable for some applications, e.g., a web browser, but hardly so for components used in business or industrial control systems which may require regression testing, re-certification of compliance, or users' training [5]. Therefore, economic theory dictates that many customers will keep old, but perfectly functioning versions of the main application [6], [7], [8]. A simple, 'poor man' example is updating one's free version of Java: the new free Java component may disrupt the (expensive) existing Matlab/Simulink toolkit which uses it and whose updates are usually not free.

As a result, when a vulnerability about the *current* version of the FOSS library is reported, the vendor must provide maintenance support for a software bundled with a FOSS release that was new then, but it is now several years older than the presently available FOSS version.

- *A. D. Brucker is with University of Sheffield, United Kingdom.*
- *S. Dashevskyi is with University of Luxembourg, Luxembourg (he performed part of this work while being with University of Trento, Italy).*
- *F. Massacci is with University of Trento, Italy.*

Different choices are possible: update, patch, or don't touch. The first choice may be appropriate if only few APIs of the library have changed. The last one may be the rational choice when the vulnerability doesn't apply to the actually deployed version and the cost of changes is massive. Such choice should be *made quickly and early*, as it requires to mobilize expertise in different areas: in the first case, functional experts must adapt the main application (just using the fixed release of the FOSS component); in the last scenario, security experts should check that there really is no problem.

Unfortunately, the strong economic advantage of using a FOSS component as a "black box" turns into a severe disadvantage when a software vendor that integrates this component into its products must check it for defects [3], [9]. A vendor may try to test its application against a working exploit, but for many vulnerabilities there are no public exploits [10]. Even if published exploits exist, they must be adapted to trigger the FOSS vulnerability in the context of the consuming application. An alternative is to apply static analysis tools against the FOSS component. Such analysis requires a solid understanding of the FOSS source code [3], as well as expertise in the tools [11], as they can generate thousands of potentially false warnings for large projects. Further, the analysis may require days for processing even a single 'FOSS-release', 'main-application' pair [12]. If several FOSS releases are used in many different products [13] the above solutions do not scale.

Alternatively, one could rely on publicly available vulnerability data, such as the information published in the US National Vulnerability Database[1] (NVD). Among other vulnerability characteristics, such data often provides the

1. https://nvd.nist.gov/

information about the versions affected by vulnerabilities, but, due to effort constraints, it focuses only on the latest supported releases—often containing statements such as ''and all previous versions.'' Our research, actually, shows that the NVD entries contain a substantial number of inaccuracies in both directions: first, listing versions as vulnerable that are in fact secure and, second, omitting versions that are vulnerable. While the fist inaccuracy may result "only" in unnecessary security maintenance, the second may result in not acting upon potentially severe vulnerabilities.

Our solution to the hurdles of identifying which older versions of FOSS components are likely to be affected by newly disclosed vulnerabilities lies in the direction of "soundiness" [14] and is in the same spirit of Hindle et al. [15]: "*semantic properties of software are manifest in artificial ways that are computationally cheap to detect automatically, in particular when compared to the cost [. . .] of determining these properties by sound (or complete) static analysis*".

We propose an automatic, scalable *screening test* for estimating if an older version of a component is affected by a newly disclosed vulnerability, using the vulnerability fix. The test is quick and can be used early in the process albeit it is approximate. Several screening criteria can be used with different level of precision. Static or dynamic tools can then be used later in the analysis.

The original contribution of this work is to combine an adaptation the *thin slicing* technique [16] for finding relevant vulnerable source code statements with the *SZZ* approach adapted from Śliwerski et al. [17] for tracking the vulnerable code back to its introduction. We also build upon Fonseca and Vieira [18] who classified typical source code changes that fix injection security vulnerabilities, as well as Renieris and Reiss [19] for their concept of code dependencies in fault localization. To understand the limitations of proposed screening test we analyzed, in collaboration with an international enterprise software vendor, several popular and large scale FOSS projects. We selected the FOSS projects based on the needs of our industrial partner, i.e., we selected the most often used components across a wide spectrum of enterprise applications and frameworks developed by our partner. First, we performed a manual validation to determine the potential error rate of different screening criteria, and, second, we did a large scale empirical analysis of the underlying assumptions (e.g., security fixes in our projects are mostly local as in [20], in contrast to what is found on normal bugs [21]), as well as the trade-offs between the likelihood that an older version is affected by a newly disclosed vulnerability and the maintenance effort required for upgrading, showing that the approach scales to thousands of revisions and MLoCs.

A side-effect of our validation is also an insight on the empirical probability that a potential vulnerable component might not actually be vulnerable if it is too old. Once again, the default rule used by many security databases ("X is vulnerable and all its previous versions") is surely safe, but might be not so effective.

The rest of the paper is structured as follows: we start with the problem statement and research questions (section 2), then continue with an overview of the related work (section 3). Next, we introduce the general terminology used throughout the paper (section 4), and introduce our vulnerability screening test (section 5). Further, we discuss the data selection (section 7) and validation of the screening test (section 8). Finally, we discuss the security maintenance decision support for FOSS components (section 9), threats to validity (section 10), and conclude (section 11).

TABLE 1: Maintenance Cycles of Enterprise Software

Maintenance cycles of ten years or more are common for software used in enterprises. During this period, vendors need to fix security issues without changing either functionality or dependencies of the software.

Product	Release	EoLife	ext. EoL
Microsoft Windows XP	2001	2009	2014
Microsoft Windows 8	2012	2018	2023
Apache Tomcat	2007	2016	n/a
Red Hat Ent. Linux	2012	2020	2023
SAP SRM 6.0	2006	2013	2016
Siemens WinCC V4.0	1997	2004	n/a
Symantec Altiris	2008	2013	2016

2 Problem Statement

If a new vulnerability is disclosed and fixed in the *current* version of a FOSS component, the vendor of the consuming application must assess *(i) which releases are vulnerable* and *(ii) what actions should be taken to resolve issues for its different customers*. This is far from trivial for most companies [3].

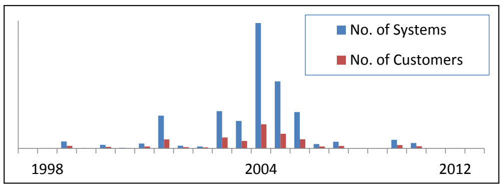
To illustrate the problem, Figure 1 shows the distribution in time (as of 2014) of the numbers of instances and customers for the past releases of *one* ERP application from a large industrial software vendor. Several customers used releases which were between five and nine years old. These releases included FOSS components that were "new" at the time but are now several years old. The vendor must now backtrack the FOSS release across its code base and for each of them take a decision.

Manually verifying whether a newly disclosed vulnerability applies to older versions of software applications is not only tedious, but is also expensive in terms of resources. For example, if we consider the software illustrated at Figure 1, this activity would require to manually examine many different versions of a third-party FOSS component for which the vulnerability was reported.

Long support lifecycles such as the one illustrated in Figure 1 is not a characteristic only of ERP software. Table 1 provides an illustrative example of the life-cycle of several products with respect to the maintenance, from operating systems to web servers, from industrial control software to security products. For example, Red Hat Enterprise Linux released in 2012 has an extended support for 11 years (until 2023). Siemens WinCC v4.0 (software for industrial control systems) had a lifetime of 7 years, and Symantec Altiris (service-oriented management software) released in 2008 has an extended lifetime of 8 years.

Depending on each vulnerability, different teams of experts may be needed to resolve it in a software product. For cases when the entire software product should be updated (e.g., for applying the security fixes provided by FOSS developers), it can be a team of software maintenance experts (for instance, the software developers that have the domain-specific knowledge about the software system and its components). For cases when it is more preferable to backport the security fixes provided by FOSS developers, or implement own security fixes for different versions of the product, a team of software security experts may be more appropriate.

However, security experts, developers and customers may, naturally, have different priorities when deciding whether a component should be upgraded, fixed or left alone: *security experts* want to minimize the attack surface and, thus, prefer upgrades of potential vulnerable components over staying with old versions. *Developers and customers* try to minimize maintenance and operational risks of changes and, thus, prefer staying with an old version if the security risk in doing so is low.



The distribution shows #systems (blue) and #customers (red) using the releases of one ERP application that are x-years old with respect to 2014 (values on the y-axis are omitted for confidentiality). In 2014, most business customers used releases (and corresponding FOSS components) that were between 9 and 11 years old. Each product version is most likely shipping different versions of different FOSS components. Hence, all these FOSS versions must be assessed when a new vulnerability in one them becomes publicly known.

Fig. 1: Single ERP application life time from 1998 to 2014, #systems and #customers using them in 2014.

Either way, we need to allocate resources to either port each application release, or audit their security. For example, developers could identify the vulnerable code fragment from the vulnerable release (by examining the source code with or without tools), thus focusing only on the relevant subset of the vulnerable component. This vulnerable code fragment could be later used in combination with running a static analysis security testing tool (SAST) on a potentially vulnerable version of a component to ascertain that the vulnerability is indeed not present. Unfortunately, precise SAST tools do not scale well to large programs: tools providing a precise analysis can take days for one version of a component [12] or can generate too many false alarms [22], [23] – the situation depicted in Figure 1, where we must assess several FOSS versions at once, would be unmanageable.

To focus our efforts on the actual vulnerable products, we must tentatively identify within minutes (not hours or days) all products that are likely affected by the vulnerability. We need the software equivalent of a clinical *screening test* [24]: something that may be neither (formally) sound, nor complete, but works well enough for practical problem instances and is fast and inexpensive.[2] Therefore, our first research question is as follows:

RQ1: *Given a disclosed security vulnerability in a FOSS component, what could be an accurate and efficient screening test to estimate its presence in the previous revisions of the component?*

2. A sound and complete solution is formally impossible to achieve: Rice's theorem [25, Proof 5.28, pp243] states that no recursive program can take a non-trivial set of programs (e.g., all past releases of a FOSS component) and produce the subset of programs satisfying a non-trivial property (e.g., containing a semantically equivalent fragment of the vulnerable code fragment).

Our main goal here is to identify and assess heuristics that would be "cheap" enough for screening through a large number of revisions of many FOSS components, allowing at the same time to improve over purely syntactic methods described in the literature (for example, Śliwerski et al. [17] and Nguyen et al. [20]).

Once we have such a screening test, we may use it for company-wise estimates to empirically assess the likelihood that an older version of a FOSS component may be affected by a newly disclosed vulnerability, as well as the potential maintenance effort required for upgrading or fixing that version. This raises the following question:

RQ2: *For how long the vulnerable coding is persistent in FOSS code bases since its introduction? What are the overall security maintenance recommendations for such components?*

3 Related Work

3.1 Identifying the Vulnerable Coding

As our **RQ1** is concerned with finding an appropriate technique for capturing a vulnerable code fragment using vulnerability fixes, we build upon Fonseca and Vieira [18] as the basis for our idea of using an intra-procedural fix dependency sphere that we introduce in Section 5.4. The authors of [18] compared a large sample of fixes for injection vulnerabilities to various types of software faults in order to identify whether security faults follow the same patterns as general software faults: their results show that only a small subset of software faults are related to injection vulnerabilities, also suggesting that faults that correspond to this vulnerability type are rather simple and do not require a complex fix.

While the work by Fonseca and Vieira [18] provides only the analysis of security patches for injection vulnerabilities, and does not consider tracking the presence of corresponding vulnerable code fragments in software repositories, we found the classification of changes for security fixes proposed by the authors [18] to be extremely useful for our purposes. However, as we were also interested in other vulnerability types (see Section 7 for the vulnerability demographics of FOSS components used by our industrial partner), we had to extend the classification of fixes by Fonseca and Vieira [18] (shown in Table 4), as well as to identify heuristics for tracking the vulnerable code fragments.

The work by Thome et al. [26] shows that sound program slices for the injection security vulnerabilities can be significantly smaller than traditional program slices, and that control flow statements should be included into slices. Therefore, we collect control-flow statements as well, in contrast to the original approach of *thin slicing* [16] on which we build our implementation for capturing vulnerable code fragments using vulnerability fixes.

Modern static analysis tools such as Wala [16] or Soot [27] can be used for extracting the vulnerable coding using security fixes. These tools implement different slicing algorithms that work over byte code, offering various features and trade-offs such as redefining the notion of relevance of statements to the seeds [16] or improving the precision of intermediate program representation [28]; simplifying the notion of inter-procedural dependencies for efficient slicing of concurrent programs [29]; and defining slices that capture specific classes of security vulnerabilities [26]. Acharya and Robinson [12] evaluated the applicability of static slicing to identifying the impact of software changes. Their findings suggest that for small programs (and changesets) static slicing can be used effectively, but it faces serious challenges when applied routinely against large systems: they report that the build time for the intermediate representation of one version of a project took about four days and observed that one must investigate and resolve various accuracy trade-offs in order to make large-scale analysis possible.

Thome et al. [26] implemented a lightweight program slicer that operates on the bytecode of a Java program and allows to extract all sources and sinks in the program for computing a program chop that would help software developers to perform faster audits of potential XML, XPath, and SQL injection vulnerabilities. It runs significantly faster than traditional slicing evaluated by Acharya and Robinson [12], however, still, it was close to impossible for our scenario (assessing thousands of revisions within seconds) to use precise tools based on byte code, as they require to build source code and resolve all dependencies as well. We found that for versions of Java projects which are older than five years from now, the latter could be very challenging. Moreover, we are interested in particular vulnerable code fragments that correspond to confirmed vulnerability fixes, but not in the whole set of slices that may contain all possible potentially vulnerable code fragments. Still, the approach by Thome et al. [26] can be used as a second-level test after our screening.

Considering the above, we have reverted to *thin slicing* [16] and modified the original algorithm to include the control flow statements, and limit the scope of slicing to the methods, where a security vulnerability was fixed.

To identify whether the library is called within the context of an application that consumes it, the approach by Plate et al. [30] can be also used as an additional test after our screening. However, the approach [30] cannot replace our own test as it requires to call a fully-fledged static analyzer to extract the call graph and fail our requirement of being inexpensive.

3.2 The SZZ Approach: Tracking the Origin of the Vulnerable Coding

It is well known that to manually identify when exactly a certain vulnerability is introduced into a software component is a long process. For example, Meneely et al. [31] studied properties of source code repository commits that introduce vulnerabilities – the authors manually explored 68 vulnerabilities of Apache HTTPD,[3] and they took six months to finish their analysis.

Many studies on mining software repositories aim at solving the problem of manual analysis [17], [20], [32], allowing to automate this tedious task. The seminal work by Śliwerski, Zimmermann, and Zeller, widely known as SZZ [17], provided an empirical study on the introduction of bugs in software repositories, showing how to locate bug fixes in commit logs and how to identify their root causes. Their method had inspired the work by Nguyen et al. [20] on which we also build our approach. Unfortunately, the original SZZ approach has several limitations [32]: for instance, SZZ identifies the origin of a line of code with the "annotate" feature of the version control system, therefore it could fail to identify the true origin of that line of code when the code base is massively refactored (e.g., the line of code is moved to another position within its containing method). In our case, such a limitation would be a problem, since the code of the projects that we considered has been massively changed over the course of time (for example, see Figure 6 in Section 7). Therefore, we adopted the heuristics by Kim et al. [32]: we perform cross-revision mapping of individual lines from the initial vulnerability evidence and associate them with their containing files and methods. This allows us to track the origin of lines of code even if they are moved, or their containing file or method is renamed, or they are moved to another location within the code base.

3.3 Empirical Studies on Trade-offs Between the Security Risk Posed by the Presence of the Vulnerable Coding and the Maintainability

Di Penta et al. [33] performed an empirical study analyzing the decay of vulnerabilities in the source code as detected by static analysis tools, using three open source software systems. The decay likelihood observed by the authors [33] showed that most of potential vulnerabilities tend to be removed from the system before major releases (shortly after their introduction), which implies that developers may prioritize security issues resolution over regular code changes. One of the questions that the authors in [33] aimed to answer is similar to the first part of our **RQ2**, however we use a different measure of the vulnerable coding: the lines of code relevant to a security fix as opposed to the lines of code relevant to a static analysis warning. Moreover, our main focus is on distinct vulnerabilities that already have evaded static analysis scans and testing by developers, therefore they will likely show different decay.

For assessing various "global" trade-offs between a vulnerability risk that a component (or a set of components) imposes and its maintainability, one feasible option is to employ various risk estimation models. Samoladas et al. [34] proposed a model that supports automated software evaluation, and specifically targets open source products. The set of metrics considered by the model is represented by various code quality metrics (including security), and community quality metrics (e.g., mailing lists, the quality of documentation and developer base). While this model takes security aspects into account, they are represented only by two source code metrics: "null dereferences" and "undefined values", which is largely insufficient to cover the vulnerability fixes in our sample (see Table 4).

Zhang et al. [35] proposed an approach for estimating the security risk for a software project by considering known security vulnerabilities in its dependencies, however the approach does not consider any evidence for the presence of a vulnerability. Dumitras et al. [36] discussed a risk model for managing software upgrades in enterprise systems. The model considers the number of bugs addressed by an update and the probability of breaking changes, but

3. We did not include this project to our sample as it is written in C, while our current implementation supports only Java.

cannot be applied to assess individual components. As such approaches would not allow to answer the second part of our **RQ2**, we resort to the code-base evidence for telling whether it is likely that a certain version of a component imposes security risk.

4 Terminology and Definitions

In this section we briefly introduce the terminology used in the rest of the paper:

Fixed revision r_1: the revision (commit) in which certain vulnerability was fixed.

Last vulnerable revision r_0: the revision in a source code repository that immediately precedes the fix r_1 when moving forward in time. Thus, r_0 is the *last* vulnerable revision when moving forward in time, and it is the *first* vulnerable revision, when moving backward.

Initial vulnerability evidence $E[r_0]$: the set of lines of code that correspond to the vulnerable source code fragment in r_0, obtained using changes between r_0 and r_1.

Vulnerability evidence $E[r_{-i}]$: the set of lines of code from the initial vulnerability evidence, that are still present in some revision r_{-i} that precedes r_0.

Repository difference $\text{diff}(r_{-i}, r_{-i+1})$: the set of lines of code changed (deleted and added) when changes from r_{-i} to r_{-i+1} were made.

Deleted lines $\text{del}(r_{-i}, r_{i+1})$: the set of lines of code deleted when changes from r_{-i} to r_{i+1} were made, s.t. $\text{del}(r_{-i}, r_{i+1}) \subseteq \text{diff}(r_{-i}, r_{i+1})$.

Added lines $\text{add}(r_{-i}, r_{i+1})$: the set of lines of code added when changes from r_{-i} to r_{i+1} were made, s.t. $\text{add}(r_{-i}, r_{i+1}) \subseteq \text{diff}(r_{-i}, r_{i+1})$.

Source code of a revision $\text{code}(r_{-i})$: the set of lines of code that belong to the source code of r_{-i}.

Set of relevant methods $\text{methods}(locs, \text{code}(r_{-i}))$: set of methods to which certain lines of code $locs \subseteq \text{code}(r_{-i})$ belong.

Set of lines of code relevant to a set of methods $\text{code}(\text{methods}_{-i})$: the set of lines of code that belong to the specified set of methods, s.t. $\text{code}(\text{methods}_{-i}) \subseteq \text{code}(r_{-i})$.

Set of defined variables $\text{def}(s)$: the function that returns a set of variables which values are defined or re-defined in a statement s.

Set of referenced variables $\text{ref}(s)$: the function that returns a set of variables which values are used in s.

Statement predicate $\text{isPredicateOf}(s1, s2)$: this function indicates whether a statement $s1$ is a conditional statement, and a statement $s2$ is statement which execution depends upon $s1$ (e.g., is a part of *then* or *else* branches of a conditional expression in Java).

5 Vulnerability Screening

In this section we start answering **RQ1** by discussing alternative techniques for performing screening tests for the likely presence of a vulnerability.

As the fixed version r_1 of a FOSS component is usually made available when a vulnerability is publicly disclosed, the information about source code modifications for implementing the fix transforming the last vulnerable version r_0 to r_1 can be used to understand where the vulnerable part in the source code is located [17], [37], [38]. Then, the approximate code fragment that is responsible for a vulnerability can be identified and tracked backwards in the source code repository history to identify a version that is not yet vulnerable [20].

Figure 2 illustrates an example for the vulnerability evolution in Apache Tomcat 6 (CVE-2014-0033): developers prohibited rewriting the URL string while handling session identifiers but a flag was not checked correctly and attackers could bypass the check and conduct session fixation attacks. The vulnerability was fixed on 16/01/2014 in revision r_1 (here: 1558822) by modifying the incorrect check (line 5) in r_0 (here: 1558788),[4] making it impossible to set a different session identifier and rewrite the URL (lines 6 and 7). Searching for the presence of these lines in previous versions, reveals that in r_{1-i} (here: 1149130) created on 21/07/2011 neither the check, nor the session fixation lines are present. At that point in time, the URL rewriting set-up was not yet introduced by developers, and hence the code base does not yet have this particular vulnerability.

Indeed, the absence of the vulnerable code fragment in some version r_{-i} that is older than the fixed r_1 is an ***evidence***, as opposed to a ***proof***, that this version is potentially not vulnerable: the vulnerable lines of code might be present in a different form or even in completely different, refactored files. If security fixes are rather local [20], these code lines constitute a *prima facie* evidence that we should allocate SAST, testing, or code auditing resources to analyze in depth the versions that correspond to the revisions where the vulnerable coding is still present, whilst having a more relaxed attitude on those versions preceding r_{-i}.

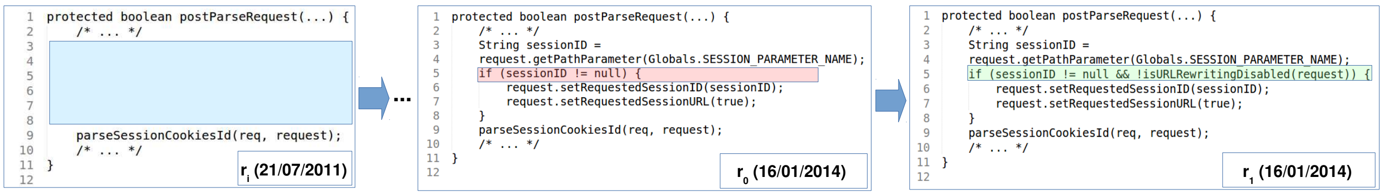
Let $\text{code}(r_0)$ be a source code fragment that represents a vulnerable version of software application r_0 that also contains a vulnerability $V \subseteq \text{code}(r_0)$, which is responsible for an unwanted behavior. What is currently known, is that r_0 contains the vulnerability, and the next revision of this program r_1 is fixed. It is unknown, however, whether an older variant of the program r_{-i} where $i \geq 1$ contains this vulnerability as well.

The goal of our vulnerability screening test is similar to screening tests used by clinicians to identify possible presence of a disease in individuals [24] – quickly separate the likely healthy portion of individuals from the portion of individuals that is likely to have a certain disease. In our case, we treat all revisions prior to r_0 (which is surely vulnerable) as those that potentially have the vulnerability, while different vulnerability evidences obtained from the fix are our metric that separates the vulnerable part of the population from the non-vulnerable one.

Algorithm 1 illustrates a generic screening test for the potential presence of the vulnerable coding:

1) $\text{Init}(r_0, r_1)$ is an abstract function that, using $\text{diff}(r_0, r_1)$ operation from the source code repository, retrieves the changes made during the fix and infers the code fragment responsible for the vulnerability – the initial vulnerability evidence $E[r_0]$. An example of such evidence can be the source code lines that were directly

4. Changes in one file may correspond to ordered but not necessarily consecutive revisions, because Subversion uses repository global commit IDs.



```
protected boolean postParseRequest(...) {
    /* ... */




    parseSessionCookiesId(req, request);
    /* ... */
}
```

r_i (21/07/2011)

```
protected boolean postParseRequest(...) {
    /* ... */
    String sessionID =
    request.getPathParameter(Globals.SESSION_PARAMETER_NAME);
    if (sessionID != null) {
        request.setRequestedSessionID(sessionID);
        request.setRequestedSessionURL(true);
    }
    parseSessionCookiesId(req, request);
    /* ... */
}
```

r_0 (16/01/2014)

```
protected boolean postParseRequest(...) {
    /* ... */
    String sessionID =
    request.getPathParameter(Globals.SESSION_PARAMETER_NAME);
    if (sessionID != null && !isURLRewritingDisabled(request)) {
        request.setRequestedSessionID(sessionID);
        request.setRequestedSessionURL(true);
    }
    parseSessionCookiesId(req, request);
    /* ... */
}
```

r_1 (16/01/2014)

In Apache Tomcat 6, CVE-2014-0033 is fixed at revision 1558822 (=r_0) on 16/01/2014. Revision 1558788 (=r_1) is the last vulnerable revision that lacks a check if URL rewriting is disabled. The revisions prior and including 1149130 (=r_i) from 21/07/2011 and earlier are not vulnerable to CVE-2014-0033, as the vulnerable feature is not present in these revisions.

Fig. 2: An example of the evolution of a vulnerable code fragment (CVE-2014-0033).

modified during a fix (such evidence is considered by the original SZZ approach by Śliwerski et al. [17], as well as by the method proposed by Nguyen et al. [20]). However, these modified lines of code may be not the ones actually responsible for the vulnerability, therefore we consider several other alternatives which we discuss in the next subsections.

2) for each revision r_{-i}, where $i \geq 1$, the current vulnerability evidence is represented by the lines of code from the initial vulnerability evidence that are still present in r_{-i}. We use the function Track$(r_{-i}, r_{-i+1}, E[r_{-i}])$ that keeps track of these lines of code individually, as suggested by Kim et al. [32].
3) for each revision r_{-i}, where $i \geq 1$, there is a test Test(r_{-i}) which is essentially a binary classifier that tells whether r_{-i} is likely vulnerable, based on whether the current vulnerability evidence given a reliability parameter δ. This parameter can be different for actual screening tests that use different vulnerability evidence extraction techniques.

Extract the vulnerability evidence using the last vulnerable revision r_0 and the fixed revision r_1:

$$E[r_0] \quad \leftarrow \quad \text{Init}(r_0, r_1) \tag{1}$$

For each revision r_{-i}, where $i \geq 1$, the evidence is computed as follows:

$$E[r_{-i}] \quad \leftarrow \quad \text{Track}(r_{-i}, r_{-i+1}, E[r_{-i+1}]) \tag{2}$$

Check, whether the source code of r_{-i} is still likely to be vulnerable:

$$\text{Test}(r_{-i}) = \begin{cases} r_{-i} \text{ is vuln.} & \text{if } \frac{|E[r_{-i}]|}{|E[r_0]|} > \delta \\ r_{-i} \text{ is not vuln.} & \text{otherwise} \end{cases} \tag{3}$$

Algorithm 1: Generic screening test using vulnerability evidence

The key question, however, is how to identify the right Init(r_0, r_1) function for the test? As this is the primary concern of our **RQ1**, we start off with describing several candidates and explaining how each of them works.

5.1 Deletion Screening Criterion

A prior work by Nguyen et al. [20] (inspired by the work of Śliwerski et al. [17]) has shown that the presence of the lines of code deleted during a security fix may be a good indicator on the likelihood that older software versions are still vulnerable: if at least one line of the initial evidence is present in a certain revision, this revision is considered to be still vulnerable. Also, the results in [20] suggest that the functionality (i.e., files and methods) where a vulnerability was fixed at some point in time may be not yet vulnerable at the point in time where it was introduced into the code base, and that there may be earlier versions in which most of the relevant functionality (files and methods) already existed but the vulnerability itself was not yet introduced.

The approach works as follows:

1) It starts by collecting the *deleted* lines of code from a vulnerability fix – *deletion vulnerability evidence;*
2) Then, it goes iteratively over older commits/revisions in the source code repository and checks for the presence of these lines;
3) Finally, it stops either when none of the lines from the initial evidence are present, or when all commits/revisions are processed. When a vulnerability is fixed by only adding lines of code, there will be no evidence to track, and the authors in [20] conservatively assume that in such cases the whole version prior the fix (namely, code(r_0)) is vulnerable. This screening test was appropriate for the empirical analysis of Vulnerability Discovery Models [39], which are typically based on the NVD and its cautious assumption "r_0 is vulnerable and so are all its previous versions" (see [20]), as this would create a consistent approximation of the NVD.

Essentially, the overall approach can be seen as an instance of the generic screening test that we defined in Algorithm 1. In this particular case, threshold $\delta = 0$, and our functions are instantiated as follows:

$$\text{Init}_d(r_0, r_1) \quad = \quad \begin{cases} \text{code}(r_0) & \text{if del}(r_0, r_1) = \emptyset \\ \text{del}(r_0, r_1) & \text{otherwise} \end{cases} \tag{4}$$

$$\text{Track}(r_{-i}, r_{-i+1}, E[r_{-i+1}]) \quad = \quad E[r_{-i+1}] \cap \text{code}(r_{-i}) \tag{5}$$

$$\text{Test}(r_{-i}) \quad = \quad |E[r_{-i}]| > \delta = 0 \tag{6}$$

For security management this may be at the same time too conservative, and too liberal as the presence of the deleted lines may not be necessary for the vulnerability to exist (see [20] for a discussion on such cases).

5.2 Method Screening Criterion

An alternative simple heuristic is the following one:[5] "if a method that was changed during a security fix is still present in an older version of a software product, this version is still vulnerable", under the conservative assumption the methods modified during the fix are responsible for a vulnerability. Again, this rule is likely imprecise but fast and inexpensive.

We instantiate the screening test for this heuristic as follows:

$$methods_1 \leftarrow \text{methods}(\text{add}(r_0, r_1), \text{code}(r_1)) \quad (7)$$
$$methods_0 \leftarrow \text{methods}(\text{del}(r_0, r_1), \text{code}(r_0)) \quad (8)$$

$$\text{Init}_m(r_0, r_1) = \quad (9)$$
$$(\text{code}(r_0) \cap \text{code}(\text{methods}_1)) \cup \text{code}(\text{methods}_0)$$

For *Track* and *Test* we use the same functions as for the deletion screening. However, tracking the presence/absence of vulnerable methods (or a change in their size) may be still overly conservative, because for cases when a method did not contain vulnerable code since it was first introduced, it may be still reported as vulnerable.

5.3 "Combined" Deletion Screening Criterion

For the original deletion screening test (see Section 5.1), if lines were only added during a fix, there are no cues on where vulnerable code could be located. Therefore, we can combine the original test with the method tracking: when a vulnerability was fixed only adding lines of code, we assume that the whole method (or methods) where these lines were added are responsible, otherwise, the technique works exactly as the original one (as before, $\delta = 0$)

$$\text{Init}_{ed}(r_0, r_1) = \begin{cases} \text{Init}_m(r_0, r_1) & \text{if del}(r_0, r_1) = \emptyset \\ \text{del}(r_0, r_1) & \text{otherwise} \end{cases} \quad (10)$$

$$\text{Track}(r_{-i}, r_{-i+1}, E[r_{-i+1}]) = E[r_{-i+1}] \cap \text{code}(r_{-i}) \quad (11)$$

$$\text{Test}(r_{-i}) = |E[r_{-i}]| > 0 \quad (12)$$

5.4 Fix Dependency Screening Criterion

Finally, we assume that not always the entire source code of fixed methods is responsible for the occurrence of a vulnerability. For instance, Fonseca and Vieira [18] empirically show that most of injection vulnerabilities may be due to a missing call to a sanitizer function, which is typically located at methods where user input is processed. Therefore, we need to devise a better approximation of the vulnerability evidence. This is our novel contribution.

Let F be the fixed lines of code obtained with diff(r_0, r_1). In order to fix the lines of code $F \subseteq \text{code}(r_0)$, a developer might need to consider several other lines of code related to F – the actual vulnerable code fragment F'. Such expansion from F to F' can be progressively scaled by a parameter k: an expansion $D_k(\text{code}(r_0), F)$ that, given a code fragment code(r_0) and the fixed lines of code F, returns the lines of code that F depends-on or that are dependent-on F for k steps according to some criteria for the notion of dependency. By $D_*(\text{code}(r_0), F)$ we identify the transitive closure of such dependencies such that $F' \subseteq D_*(\text{code}(r_0), F) \subseteq \text{code}(r_0)$: – the *fix dependency sphere*[6] of the code fragment F.

Therefore, we instantiate another screening test that considers the source code dependencies of the fixed source code fragment as follows:

$$\text{Init}_{fd}(r_0, r_1) = \quad (13)$$
$$D_*(\text{code}(r_1), \text{add}(r_0, r_1)) \cap D_*(\text{code}(r_0), \text{del}(r_0, r_1))$$

$$\text{Track}(r_{-i}, r_{-i+1}, E[r_{-i}]) = E[r_{-i+1}] \cap \text{code}(r_{-i}) \quad (14)$$

$$\text{Test}(r_{-i}) = \frac{|E[r_{-i}]|}{|E[r_0]|} > \delta \quad (15)$$

6 Implementing the fix Dependency Sphere

If we had used traditional program slicing [40] for extracting the vulnerable code, we would likely find out that (using the words by Sridharan et al. [16]) "*traditional slices often grow too large*" and "*unwieldy slices arise primarily from an overly broad definition of relevance, rather than from analysis imprecision; while a traditional slice includes all statements that may affect a point of interest, not all such statements appear equally relevant to a human*".

Therefore, we implemented the dependency expansion D_* as a generalized intra-procedural version of *thin slicing* introduced by Sridharan et al. [16], which improves over the notion of statement relevance of the original slicing algorithm by Weiser [40] to avoid collecting overly large slices. In particular, the thin slicing approach captures only the *producer statements* – the statements that either copy a value to the seed statements, or take part in computing that value. We further decided to limit the resulting slices to intra-procedural boundaries, since our analysis of vulnerability fixes (see Table 4) suggested that security fixes are rather "local": in all cases the vulnerable code is located closely to the fixed lines of code (in most cases, within the same method).

We do not perceive our adaptation of the original thin slicing as the original contribution of this work: we only use it as a tool for extracting the vulnerable coding from vulnerability fixes (as an implementation for the fix dependency screening criterion discussed in Section 5.4). However, we had to proceed with our own implementation due to the important technical difference with the original implementation: our version had to work directly on the source code.

In our case, the lines modified during a vulnerability fix are *seeds*, and, similarly to [16], a slice includes a set of *producer statements* for these seeds. To identify simple dependencies between statements we look for relevance

5. This heuristic was suggested by A. Sabetta from SAP Labs France in a private conversation. It is also suggested by several anonymous reviewers of a previous version of this paper.

6. This concept is similar to the notion of *k-dependency sphere* introduced by Renieris and Reiss [19] for dependencies in fault localization.

relations between variables in them. We also include as a set of *explainer statements* that are relevant to the *seeds*. These are the following types of statements:

1) **Producer statements**: "[...] statement s is a producer for statement t if s is a part of a chain of assignments that computes and copies a value to t" [16]. This is an assignment of a value to a certain variable.
2) We distinguish the following types of **explainer statements**:
   a) **Control flow statements**: the statements that represent the expressions in the condition branches under which a producer statement will be executed (this concept is taken from [16] as well). A statement s is control dependent on a conditional expression e if e can affect whether s is executed. A statement s is flow dependent on a statement t if it reads from some variable v that is defined or changed at t, or there exists a control flow path from t to s on which v is not re-defined.
   b) **Sink statements** – represents a statement that corresponds to a method call that has a parameter to which a value flows from a producer statement. Therefore, a statement s is a relevant sink of the statement t if s is a method call and s is flow-dependent upon t.

```
1) Set $Rel_f(s) \leftarrow$ def($s$) if any of the following holds
   a) $s \in Seeds \wedge$ def($s$) $\neq \emptyset$
   b) there exists a preceding $t$ such that:
      i) ref($s$) $\cap Rel_f(t) \neq \emptyset$, or
      ii) isPredicateOf($t, s$) $\wedge Rel_f(t) \neq \emptyset$
2) Set $Rel_f(s) \leftarrow$ ref($s$) if any of the following holds
   a) $s \in Seeds \wedge$ def($s$) $= \emptyset$
   b) there exists a preceding $t$ s.t. ref($s$) $\cap Rel_f(t) \neq \emptyset$
Otherwise $Rel_f(s) \leftarrow \emptyset$
```

Algorithm 2: Forward Slices of Relevant Variables

```
Set $Rel_b(s) \leftarrow$ ref($s$) if any of the following holds:
1) $s \in Seeds$
2) there exists a preceding line $t$ s.t.
   def($t$) $\cap Rel_b(s) \neq \emptyset$
   // conservative: ignore step (3) for "light" slicing
3) there exists a preceding line $t$ s.t.
   $t \in Sinks \wedge$ ref($t$) $\cap Rel_b(s) \neq \emptyset$
4) there exists a succeeding $t$ s.t.
   isPredicateOf($s, t$) $\wedge Rel_b(t) \neq \emptyset$
Otherwise set $Rel_b(s) \leftarrow \emptyset$
```

Algorithm 3: Backward Slices of Relevant Variables

We use Algorithm 2 for recursively finding a set of source code statements which are affected by the *seeds* (a forward slice). For instance, if a statement s is a seed and is an assignment, we collect all other statements that are located below s and are flow-dependent upon s (steps "1,a" and "2,b,i"). When s is a statement which execution depends upon another preceding statement t (step "1,b,ii"), and t is either a seed, or was collected because it is control- or flow-dependent upon a seed or another collected statement, we collect the statement s as well. When a statement s is not an assignment and it is a seed, we simply collect s (step "2,a"). When s is neither an assignment, nor a seed (i.e., it is an explainer statement) we collect s only if there is a preceding statement t that was collected because of the variables referenced in s.

In this way, if we collect s, and it is a control flow statement (e.g., an "if" statement), we also collect all other statements which execution can be affected by the values in s (e.g., statements inside of the "then" and "else" branches of an "if" statement). When s is a sink of the form $s(x, y, z)$, and we collect this statement because the parameter x is a variable that is relevant at some other preceding statement t, we conservatively consider that x becomes relevant at s. However, we also consider the parameters y and z to become relevant at s since x may be changed inside of s, as well as its value may be passed to y and/or z. Since this may be too conservative, as we may end up collecting too many statements that are not actually relevant to the seeds, we also implemented a *light* variant of this slicing that ignores the effect of the parameters: if a statement s of the form $s(x, y, z)$ is included into a slice because of the parameter x, then we assume that neither x, nor other parameters are changed inside of s, therefore their relevance will not be propagated further (we empirically compare these two variants in Section 8).

Similarly, we use Algorithm 3 for recursively finding a set of source code statements that affect the *seeds* (a backward slice).

The original slicing algorithm by Weiser [40] requires that the resulting slices are executable, however, by design the slices that we yield do not have this requirement. As we mentioned at the beginning of this Section, we have deliberately sacrificed the precision in favor of scalability, and expect our slices to be non-executable, as we consider only syntactic dependencies. Therefore, our implementation proceeds as follows:

1) We start with the set of seed statements $Seeds$ as the slicing criteria, where every criterion can be represented as a tuple $\langle s, V\rangle$ (similarly to Weiser's slicing criterion [40]) where s is the seed statement, and V is the set of variables of interest in that statement: for each seed statement s, the set V consists of the variables which values are used in s.
2) Then, for every seed statement s, we iteratively identify other statements that contain relevant variables that are dependent on (Algorithm 2) or influence (Algorithm 3) the set of relevant variables V in s (the statement sets Rel_f and Rel_b).
3) The final slice will include all statements in the method, for which there is at least one variable that is relevant to the seeds, thus, will contain a union of statements returned by Algorithm 2 and Algorithm 3 ($Rel_f \cup Rel_b$).

7 Data Selection

During our previous empirical study on the drivers for the security maintenance effort [13] we have collected data on a sample of FOSS components used in various products of our industrial partner – a large international software vendor. Our main criteria for including a FOSS component into the sample was its internal popularity: the number of internal products into which the component was integrated by the developers of our industrial partner for the last five years (as of 2016). We included all components for which this number was at least five, and this sample contained 166 FOSS components.

Figure 3 shows the descriptive statistics of the sample of these 166 FOSS components. Figure 3a illustrates the cumulative size of the code bases of all components broken down by different programming languages in which these components were implemented: this distribution suggests that the largest code base corresponds to the components implemented in Java. Further, Figure 3b shows the distributions of internal projects into which these FOSS components are integrated divided by Java an non-Java components: first, it shows that the number of internal projects that are using a FOSS component is fairly large, and second, it also suggests the prevalence of components implemented in Java.

To check whether this prevalence of Java projects was statistically significant, we applied non-parametric Wilcoxon test, since the data that we collected are not normally distributed (Shapiro-Wilk test returned $p < 0.5$), and contains unpaired samples. The results of the Wilcoxon test confirmed that indeed Java components prevail in comparison to other kinds of components used by the developers of our industrial partner: the two distributions shown on Figure 3b have small-to-medium and statistically significant difference ($p < 0.5$, Cohen's $d = 0.44$). Our follow-up discussions with the developers only confirmed this observation. Therefore, we focused our efforts specifically on the components implemented mainly in Java.

In order to stress-test our method, we have selected several large Java-based components used by our industrial partner with high internal popularity and high number of reported vulnerabilities. These components are of similar size, and have been actively maintained for several years (at least seven). Table 2 lists these components.

However, the successful application of the proposed screening test depends on the availability and the quality of the relevant information that can be found in vulnerability databases. Specifically, a vulnerability database entry has to be a real vulnerability, and the information about its fix must be available. For example, Śliwerski et al. [17] provide a set of heuristics based on regular expressions that the authors of [17] used for spotting commits that fixed bugs in source code repositories. From our experience, this method may fail (see also Bird et al. [41]), and when it fails, the information about security fixes can be extracted from the vulnerability description itself, found when examining a bug tracker of the corresponding software project, mentioned in third-party security bulletins or in the mailing list archives, as well as exist in different other sources which may vary from project to project [18].

We used all aforementioned heuristics for recovering the vulnerability fix commits across the entire sample of vulnerability entries (CVEs) reported for the selected components in the NVD (the "Total CVEs" column in Table 2). However, while doing that we found that it may be very difficult to establish the link between the CVEs and repository commits that fix them, especially when this information is not readily available in the NVD or other public information sources (Bird et al. [41] had made an observation that such linkage is often missing for regular bugs as well). Therefore, we were able to recover fixes for only 55 CVEs across the selected components (the "Processed CVEs column" in Table 2).

Figure 4 shows the distributions of vulnerability types from the entire sample of 166 FOSS components (dark bars), and the sample of 55 CVEs for the Java components selected for the evaluation (light bars). Both of these distributions suggest that the prevalent vulnerability types are "Injection" (including "Cross-Site Scripting", "Cross-Site Request Forgery", and "Command/Code execution"), "Denial of Service", and "Broken authentication/access control". The distribution of vulnerability types for 55 Java CVEs did not have the "Memory corruption/overflows" vulnerability type since these are not typical for Java components (and if found, such vulnerabilities are likely to happen due to implementation bugs in the Java Virtual Machine rather than the software that runs on it). From these two distributions we conclude that the sample of 55 CVEs that we used for evaluation is representative of the general distribution of vulnerabilities in FOSS components used by our industrial partner.

Figure 7 describes the software infrastructure that we set up for running the proposed vulnerability screening tests. After the vulnerability fix commit information is extracted,[7] the *Repository Utility* invokes the *Vulnerability Evidence Extractor* component that extracts the vulnerable code fragment according to a desired screening criterion (described through Sections 5.2–5.4): for instance, our lightweight slicer implementation (Section 6) is an instance of a fix dependency screening criterion (Section 5.4), and its purpose is to extract additional lines of code that have direct dependencies with the fixed lines of code. Different other implementations for inferring the relevant vulnerable code can be plugged into the *Vulnerability Evidence Extractor* component instead, however, the rest of the proposed screening test will remain the same: the *Repository Utility* tracks the evidence backwards in the software repository and stores it in the database for further analysis.

Existing program analysis frameworks (such as Wala and Soot[8]) that support various program analyses can be also used in place of the *Vulnerability Evidence Extractor* component (Figure 7), and provide a refined screening test criterion by helping to cover more programming languages and potentially improving the precision/recall ratio of the test. However, these frameworks do not provide other components necessary for running the test.

As our main goal was is to identify whether even simple heuristics that consider local data and control dependencies can make a significant difference with respect to the purely syntactic methods described in the literature (for example, by Śliwerski et al. [17] and Nguyen et al. [20]), we leave the evaluation of more precise (and more complex) program analyses in place of the *Vulnerability Evidence Extractor* component for the future work.

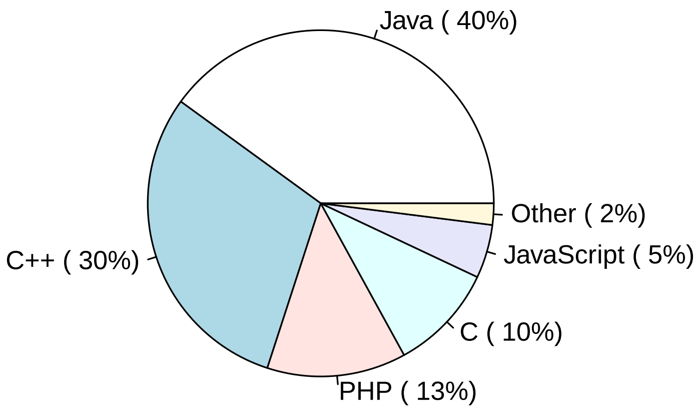
7. Currently, we perform this step manually as we must ensure that a commit is indeed fixing the relevant vulnerability. However, identifying vulnerability fix commits can be automated, see Levin and Yehudai [42].

8. Unfortunately, Soot does not support source code analysis out of the box. Therefore, the whole linkage to the source code will be another challenge that will complicate the approach significantly.

Fig. 3: Descriptive statistics of 166 FOSS components used by our industrial partner

We collected a sample of 166 FOSS components integrated into different internal projects: the figure on the left illustrates the sample in terms of the size of the cumulative code bases implemented in specific programming languages; the figure on the right shows the number of internal applications into which these components are integrated divided by Java and non-Java components.

(a) Relative size of cumulative code bases for different languages



(b) Internal projects relying on a component

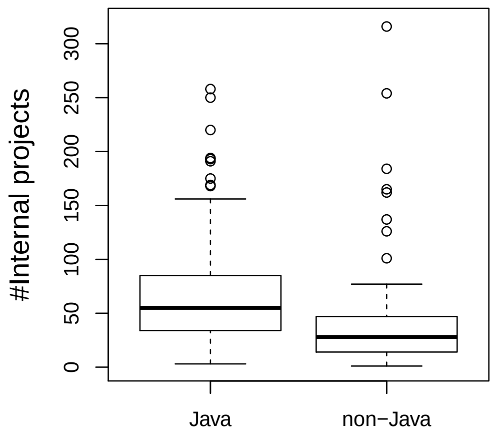
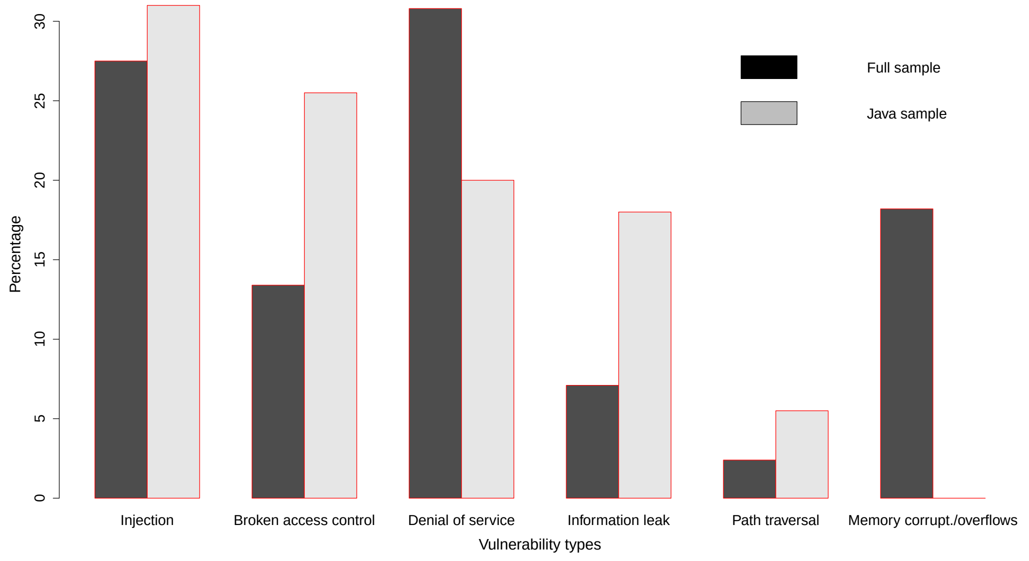


Fig. 4: Distributions of vulnerability types

We compare the distributions of vulnerability types from the sample of 166 FOSS components used by our industrial partner and the subset of 55 Java CVEs that we used for evaluation. Both distributions suggest that the most prevalent type of vulnerabilities are "Injection", "Denial of service" and "Broken access control" (Java distribution did not contain "Memory corruption/overflow" vulnerabilities).



During our study in [13], we also had informal interviews with software developers and maintenance experts of our industrial partner in order to better understand how the maintenance decisions about vulnerable FOSS components are typically made. As a result, we understood that such decisions are usually taken on ad-hoc, component-by-component basis: a component may be forked due to porting or changes to a subset of features; custom fixes for security bugs can be implemented and other technical modifications can be performed, if necessary [43], [44], [45].

Thus, the unlikely but not rare decision to down-port a security fix[9] that a software vendor that relies on FOSS components has to make may happen due to a combination of reasons:

1) The community that maintains the component may likely not provide the solution for the specific security problem with an outdated version;[10]
2) The newer version of a FOSS component that provides the fix is largely incompatible with the coding of the

9. An example of forking and long-term maintenance is SAP's decision to provide its own Java Virtual Machine for several years "because of end of support for the partner JDK 1.4.2 solutions". See http://docplayer.net/22056023-Sap-jvm-4-as-replacement-for-partner-jdks-1-4-2.html.

10. This could happen when the old version of a FOSS component is affected by a vulnerability but it is not supported by its developers (e.g., EOL of Tomcat 5.5), or it is not actively maintained at the moment.

TABLE 2: The sample of FOSS projects used in this paper

The FOSS components from our evaluation sample have been actively developed over several years (e.g., the commit speed is between 742 and 2551 commits per year). For each component, we also specify the total number of CVEs reported in the NVD, and the number of CVEs that we analyzed.

Project	**Total commits**	**Age (years)**	**Avg. commits (per year)**	**Total contributors**	**Current size (KLoC)**	**Total CVEs**	**Processed CVEs**	**μ files touched per fix**
Apache Tomcat (v6-9)	15730	10.0	1784	30	883	65	22	1.5
Apache ActiveMQ	9264	10.3	896	96	1151	15	3	1.5
Apache Camel	22815	9.0	2551	398	959	7	3	1.0
Apache Cxf	11965	8.0	1500	107	657	16	10	2.0
Spring Framework	12558	7.6	1646	416	997	8	5	1.6
Jenkins	23531	7.4	2493	1665	505	56	9	1.9
Apache Derby	7940	10.7	742	36	689	4	3	2.7

application that consumes it, thus there is significant effort involved in migrating the application;

3) The internal changes of the library are of limited concern for the developers of the consuming application unless the functionality has been changed – the latter change is often being captured by a change in the APIs (See [5], [46] for a discussion);

Considering the above, we understood that a simple metric for change, the number of changed API of a component, is considered to be more interesting by developers as their focus is to use the FOSS component as a (black box) library.

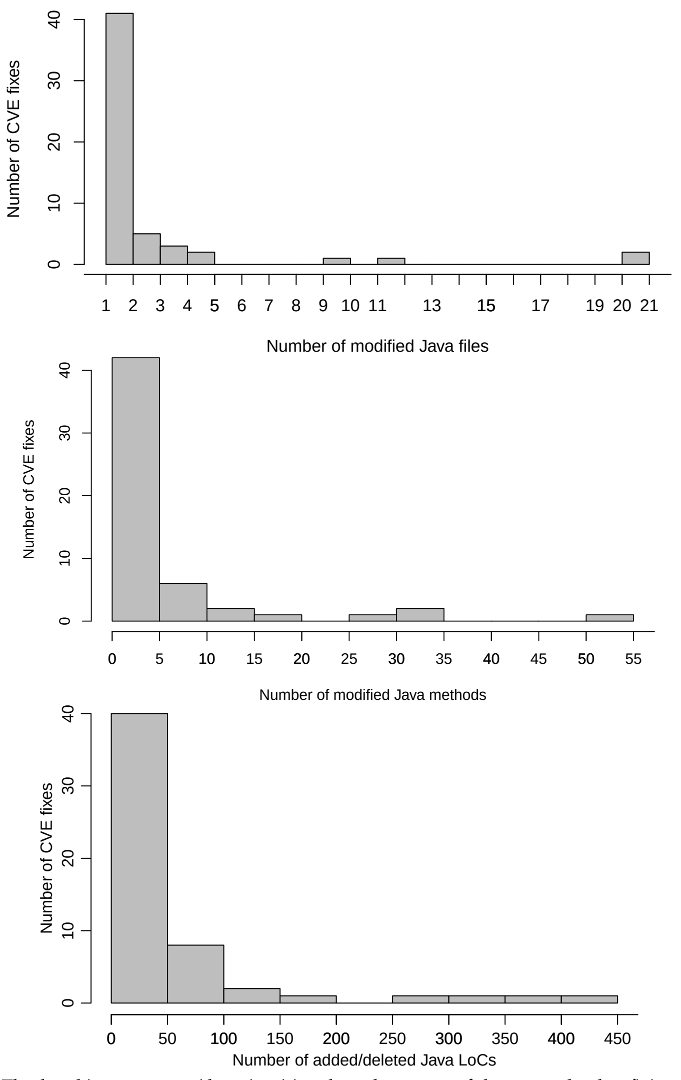
Therefore, to provide also an indication for the demographics of the "number of API changes" (shown on Figure 6) we include the *Change Extractor* component from Figure 7 for calculating the changes in the APIs for each commit in comparison to the fix:

1) For each commit, we identify all Java files; for each Java file, we count and sum the number of public methods;
2) Then, we take the difference between the current commit and the commit of the fix (difference in method signatures) and count the number of public methods that are not present in the fix, or could have been changed.
3) We record the number of changed methods in the current commit with respect to the fix using the above two numbers.

After the vulnerability data is processed and all evidences are extracted, they are aggregated and stored in a CSV file or a SQLite database which can be used for further analysis.

Figure 5 provides an intuition about the size of the changes made when fixing the CVEs from our sample. It combines the following three histograms: (1) the number of modified files;[11] (2) the number of modified Java methods; (3) the number of distinct added/deleted lines of code. In the majority of cases (51 out of 55), at most 5 files were modified, while in 29 cases it was only 1 file. Additionally, in 40 cases the number of added/deleted lines of code was at most 50. This gives us an intuition that the majority of fixes were rather "local" and not spanning across multiple files, methods, and commits (which would possibly require a more complex evidence extraction mechanism).

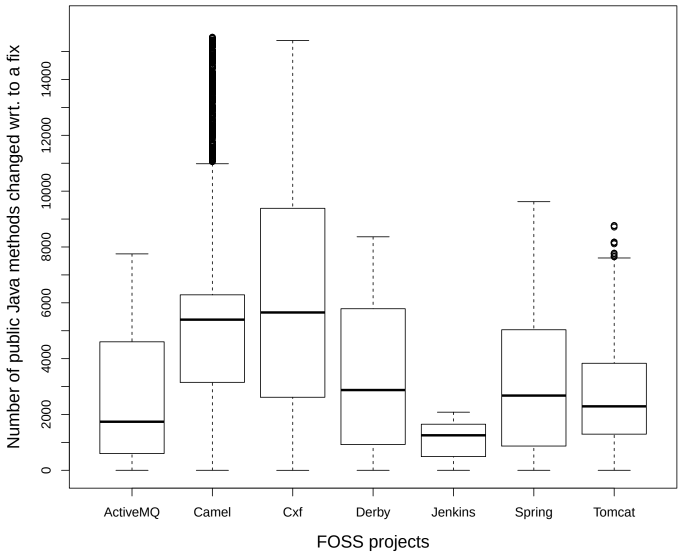
11. Here we count only Java files, excluding unit tests.



The three histograms provide an intuition about the amount of changes made when fixing CVEs from our sample. The majority of security fixes from our sample were rather "local", not spanning across many files, methods, and lines of code.

Fig. 5: The distribution of files and methods changed during a security fix

For 49 out of 55 CVEs a fix was performed with a single commit. For every of those 6 CVEs that were fixed with several commits we used an ad-hoc procedure: for each commit that was a part of the vulnerability fix, we calculated the vulnerability evidence (lines of code) independently, and then merged the sets of lines of code from these separate evidences into a single set; afterwards, we tracked this merged set as a single vulnerability evidence starting from the oldest fix commit (exactly as we did it for vulnerabilities fixed with only a single commit).



For every commit in which we tracked the vulnerable coding, we collected the number of public methods that were changed with respect to the public methods in corresponding fixes. The only exception was Jenkins – for this project we measured the number of changed methods in all commits (not only in those in which the vulnerable coding was present), as we found out that the repository history of this project was malformed (see Section 8 for an explanation). This distribution gives an intuition on the amount of such changes within each project.

Fig. 6: API changes statistics per project

8 Validation

In this section we describe the process of the validation that we performed to answer the part of **RQ1** about the accuracy and performance of the vulnerability screening test, and to assess the overall usefulness of the approach for the problems outlined in Section 2.

The empirical evaluation of the lightweight slicer for finding security features inherent for injection vulnerabilities by Thome et al. [26] reports the running time between 50 seconds to 2 minutes on a project that has 28 KLoC on average. Table 3 reports the runtime of our approach over the entire repository: while we cannot directly compare the running time of our implementation of extracting the fix dependency sphere and the slicer by Thome et al. [26][12], the running time of our entire approach is comparable, which shows that it is practical.

Next, we review the vulnerabilities in our data set, and analyze their fixes to understand whether the fix dependency sphere would possibly capture them. The results of this analysis for the conservative fix dependency screening are summarized in Table 4: it lists description of vulnerability types (taken from OWASP Top 10[13]), as well as description of typical fixes for these vulnerabilities. The "completeness" column describes the dependencies of a fix that will be captured by the fix dependency sphere $D_*(\text{code}(r_0), F)$. We claim that for these vulnerability types and fixes $D_*(\text{code}(r_0), F)$ includes the fragment of the code responsible for the vulnerability. Therefore, tracking the evolution of $D_*(\text{code}(r_0), F)$ from r_0 and downwards may be a satisfying indicator for the presence of a vulnerability.

TABLE 3: Runtime performance of fix dependency screening

The vulnerability screening test can provide an approximate evidence (based on actual code) about the presence of the newly discovered vulnerabilities by scanning the entire lifetime of a FOSS project in matter of minutes. Precise (but costly) static analyses can be deployed after that step, in surgical fashion.

Project	Analyzed Data		Time (in sec)	
	#Commits	#MLoCs	mean	(std)
Apache Tomcat	141016	186.331	35	(19)
Apache ActiveMQ	11598	27.904	28	(21)
Apache Camel	8892	4.706	16	(7)
Apace Cxf	53822	28.525	49	(33)
Spring Framework	17520	3.854	44	(35)
Jenkins	8039	9.416	16	(10)
Apache Derby	7588	5.597	17	(11)

For each of selected vulnerabilities, we identified the set of ground truth values as follows:

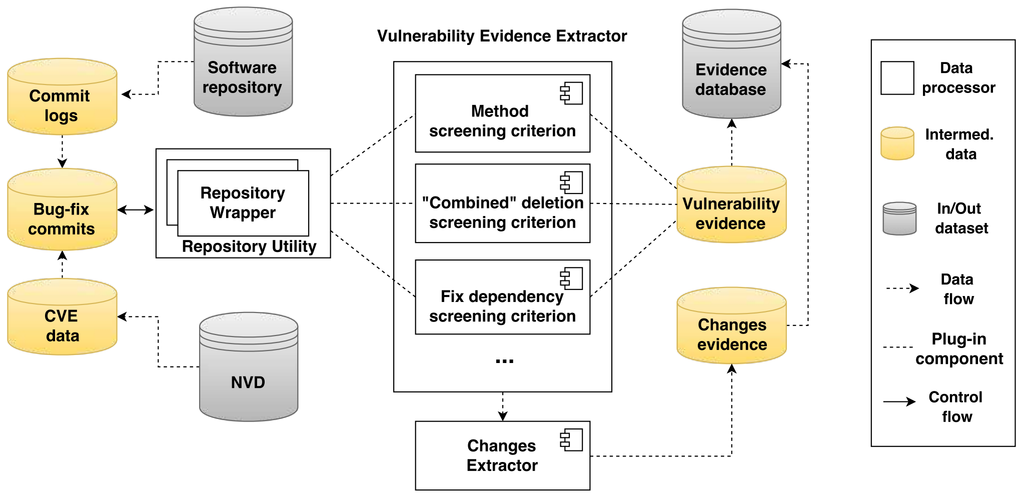
1) We performed source code audits starting from the last vulnerable revision r_0, moving backwards through the repository history.
2) When we observed that any of the files initially modified to fix a vulnerability had some changes in an earlier revision, we manually checked whether the vulnerability in that revision was still present.
3) We stopped the analysis either on a revision that we find to be not yet vulnerable (this implies that all earlier revisions are not vulnerable as well – we did several spot checks going further past the first non-vulnerable revision an that was indeed the case), or until we reached the initial revision of the repository.

Typically, when we go backwards in the repository history, we arrive at the initial commit that created the repository – here if we use the *diff* tool, we will see that no source code lines were deleted (because nothing existed before that point), but many new files were created (looking at the commit log we can confirm that it is indeed the initial commit). This is what typically should happen.

With Jenkins component, however, there exist two such points: the "true" initial commit where the repository was created, and the commit that deletes all original files (the very next commit adds them again) – it is unclear why Jenkins developers did that. So if we rely only on the *diff* tool to tell whether a certain file was created, our analysis will stop at this "fake" initial commit. We kept Jenkins in the full sample because the "true" and "fake" initial commits are not far away from each other in time, therefore, the analysis that we perform in Section 9 will still be valid. However, we decided to remove Jenkins from the manually checked sample as we wanted the ground truth to be accurate.

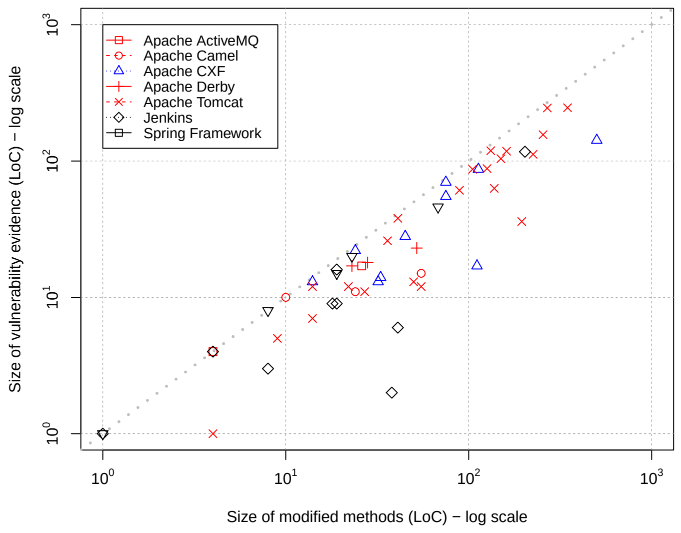
12. As we only extract the relevant code within a set of methods (it takes less than a millisecond), while the slicer by Thome et al. [26] extracts all potentially relevant sources and sinks.

13. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project



*The textual description of a vulnerability is taken from public vulnerability databases (e.g., the NVD) and then combined with a search for references to the actual commits fixing that vulnerability – this information can be present in the NVD, the commit logs of the project's software repository, or in various third-party sources. Once a vulnerability fix commit is obtained, the Repository Utility invokes the Vulnerability Evidence Extractor component that extracts the vulnerable code fragment according to a chosen screening test criterion. Once the vulnerable coding (evidence) is extracted, the Repository Utility tracks the evidence backwards in the software repository and stores it in the database. Please note that the Vulnerability Evidence Extractor component can be replaced by any heuristic or tool for extracting a vulnerable code fragment with different levels of granularity; in this work, to answer **RQ1** we used only the criteria described throughout Sections 5.2–5.4. Additionally we used the Changes Extractor component (described in Section 7) in order to identify a proxy for the update efforts for different software projects in different points in time.*

Fig. 7: Software infrastructure for vulnerability screening



Our implementation of fix dependency sphere (see Section 6) could have "saturated" the analysis by including the whole method where the vulnerable code was present. This phenomenon may appear for general bug fixes [21], [47] but it was not present in our projects, as the (vulnerable) security checks and the related fix are normally quite distinct from the code of the method that implements other functionalities.

Fig. 8: Comparing the initial amount of lines of code obtained with *conservative fix dependency screening* criterion versus the initial size of the entire fixed method

The final sample for the manual assessment consisted of 126 data points across the total of 126193 revisions that correspond to the histories of 21 CVEs (randomly selected across the projects shown in Table 2): we went backwards iteratively, and for many revisions the vulnerability evidence did not change. Therefore, we had to check only those points where it did actually change. In order to make sure that the sample size for the manual assessment was sufficient, we also approximated the potential error margin for the sample using Agresti–Coull confidence interval (see Section 8).

The manual assessment was carried out by three experts. At first, the experts had performed the assessment independently, and then compared their results and cross-checked them. The Kappa statistic for the inter-rater agreement between the experts has reached 0.87 (the experts did not reach perfect agreement for only 2 CVEs), which indicates that the agreement level was more than adequate.

In this way, we manually annotated every revision from r_0 and backwards with ground truth values, and obtained the *ground truth* binary classifier:

$$Test_{gt}(r_i) = \begin{cases} 1 & \text{if } r_i \text{ is still vulnerable} \\ 0 & \text{otherwise} \end{cases}$$

Then, we ran every variant of the vulnerability screening test described in Section 5, and compared the results with the ground truth classifier. For every revision $r_i < r_0$ (where $i < 0$), this comparison had the following result:

1) *True positive:* a revision was correctly classified as *vulnerable* (e.g., a test marks the revision as vulnerable, and we identified that it is indeed vulnerable with our ground truth analysis);
2) *False positive:* a revision was incorrectly classified as *vulnerable* (type I error of a classifier);
3) *True negative:* a revision was correctly classified as *non-vulnerable*;
4) *False negative:* a revision was incorrectly classified as *non-vulnerable* (type II error of a classifier).

As a part of the answer to **RQ1**, we wanted to understand whether our fix dependency variants of the vulnerability screening test (see Section 5.4) show results that are significantly different in comparison to the existing work of Nguyen et al. [20] and the simplest possible heuristic that can be expressed as "if the vulnerable piece of code (methods that were fixed) does not exist yet, the vulnerability does not exist as well".

TABLE 4: Construction of a fix dependency sphere

The fix dependency sphere is a type of vulnerability evidence that is constructed by taking the initial set of lines of code modified during a security fix, and collecting local dependencies of these lines (our implementation is intra-procedural, see Section 6). We illustrate how a typical security fix for a specific vulnerability type in the sample of vulnerabilities that we verified manually looks like, and how a conservative fix dependency sphere is collected for these security fixes.

Vuln. type	**Description**	**Fix**	**Completeness**
Injection (SQLi, XSS, code/command execution)	There exists a flow of data where a value coming from an insecure *source* (user input) can reach a secure *sink* (database, command interpreter, web browser) and be executed (e.g., CVE-2008-1947, CVE-2014-0075).	*The fix may:* 1) break such a flow (delete the source/sink statements), or insert a sanitizer between the source and the sink (add new method call inside of the method where the user input gets to the sensitive sink); 2) fix an incorrect sanitizer (the fix is inside the sanitizer method).	*The fix dependency sphere includes:* 1) statements that capture the faulty part of the flow between the source and the sink. 2) statements that capture the faulty part of the flow flow within the sanitizer.
Path traversal	There exists a flow of data from an insecure source, which is used for constructing path names intended to identify a resource located underneath a restricted parent location (e.g., CVE-2008-2370).	*The fix may:* 1) insert a sanitizer for the input used to construct a path (add new method call inside of the method where the path is constructed); 2) fix the sanitizer, e.g., add missing character encoding (the fix is inside the sanitizer method); 3) in case of URL construction, remove the query string from the path before the path is sanitized (re-arrange some of the statements inside of the method where the path is constructed);	*The fix dependency sphere includes:* 1) statements that capture the faulty flow between the source and the sink; 2) statements that capture the faulty flow within the sanitizer; 3) statements that capture faulty flow between the source and the sink as well as the statements that represent the altered control flow.
Info leak (configuration, usernames, passwords)	Incorrectly implemented or extensive error messages allow attackers to identify configuration details of an application and use it as a leverage for exploiting other vulnerabilities (e.g., CVE-2010-1157). Simple errors, such as passing a wrong variable into a method call (e.g., CVE-2014-0035) can lead to sending sensitive data over the network unencrypted. In case, when authentication functionality reveals too much context information, this can be used for enumeration of existing users and password guessing (e.g., CVE-2009-0580, CVE-2014-2064).	*The fix may:* 1) replace an incorrect message; 2) change the control flow upon which the message is shown; 3) neutralize an unhandled exception that is triggered under specific error conditions (add catch block); 4) replace an incorrect parameter passed to a method call.	*The fix dependency sphere includes:* 1) the modified error message, as well as control flow statements upon which the message is generated; 2) the altered control statements as well as the corresponding error message; 3) the whole "try" block that corresponds to the added "catch" block, as well as the relevant error message and it's control flow statements; 4) the faulty method call (sink) as well as all statements that capture the data flow of the replaced parameter (source).
Cross-site request forgery (CSRF)	An application accepts web requests from an authenticated user, but fails to verify whether requests are unique to the user's session (are actually sent from am user's browser).	*The fix may:* 1) implement a protection mechanism by adding specific tokens and cookies (place a token into a response body); 2) as most protection mechanisms for this vuln. can be bypassed if there exists related cross-site scripting vulnerability, a potential fix may actually be equivalent to *Injection* vulnerability fixes.	*The fix dependency sphere includes:* 1) statements that capture a control flow in which the protection mechanism is inserted and under which this protection mechanism was lacking (e.g., a token is placed into the response body); 2) same as the fix dependency sphere for *Injection* vulnerabilities.
Broken authentication, access control flaws	Application fails to ensure the access control of resources (e.g., CVE-2006-7216, CVE-2012-5633), or user accounts (e.g., CVE-2014-0033, CVE-2013-0239).	*The fix may:* 1) add (or replace) an ad-hoc access control rule to the resource (alter the control flow); 2) add new method that specifies explicit permissions of an object (e.g., for serialization and de-serialization); add corresponding method call to a method where corresponding object was created;	*The fix dependency sphere includes:* 1) the statements that correspond the resource referenced at the point where the new ad-hoc access control rule was added by the fix (control-dependency of the newly added rule), as well as some additional control and data flow dependencies; 2) the statements that correspond to the control/data flows under which the newly added statement is being called.
Denial of service	Application becomes unavailable to users due to errors in resource management that are exploited by an attacker. This vulnerability may exist either due to insufficient input validation (e.g., CVE-2011-0534), logical flaws (e.g., CVE-2014-0230, CVE-2012-2733), or the combination of both (e.g., CVE-2014-0095).	*The fix may:* 1) add or change existing conditions that control the "expensive" resource operation (e.g., buffer limits, thread numbers, etc.); 2) move the condition under which an "expensive" resource operation is invoked (e.g., under some conditions, an operation may be invoked before the check is performed), or add additional checks; 3) alternatively, add a sanitizer that does not allow user input to trigger the fault (e.g., size of the data to be cached, data validity checks, etc.).	*The fix dependency sphere includes:* 1) the statements that correspond to the resource operation itself, as well as control and data flows relevant to this operation; 2) same as the above; 3) same as the fix dependency sphere for *Injection* vulnerabilities.

TABLE 5: The versions affected by vulnerabilities

This table compares for all manually analyzed CVEs the range of vulnerable versions as published in the NVD with the version range identified by a manual expert analysis (ground truth). We observe three cases: 1) the version range given in the NVD is equal to the ground truth (✓), 2) the NVD specifies a larger set of versions, i.e., over approximates the vulnerable versions (🕒) compared to the ground truth, and 3) the NVD gives a smaller set of versions (⚠) compared to the ground truth. The second case (over approximation) results in unnecessary work created by "fixing" non-vulnerable consumption of components. In contrast, the third case (under approximation) results in not addressing, potentially severe, security vulnerabilities. Note that, for a specific CVE, the three cases are not disjoint; i.e., the version range published in the NVD might miss vulnerable versions as well as contain non-vulnerable versions at the same time.

Project	**CVE**	**NVD Versions**	**Ground Truth**	**Assessment**
Apache Tomcat	CVE-2008-1947	6.0.0 – 6.0.16	6.0.0 – 6.0.16	✓
	CVE-2010-1157	6.0.0 – 6.0.26	6.0.20 – 6.0.26	🕒
	CVE-2011-0013	6.0.0 – 6.0.29	6.0.0 – 6.0.29	✓
		7.0.0 – 7.0.5	7.0.0 – 7.0.5	✓
	CVE-2011-0534	6.0.0 – 6.0.30	6.0.15 – 6.0.30	🕒
		7.0.0 – 7.0.6	7.0.0 – 7.0.6	✓
	CVE-2012-2733	6.0.0 – 6.0.35	6.0.0 – 6.0.35	✓
		7.0.0 – 7.0.27	7.0.0 – 7.0.27	✓
	CVE-2013-2067	6.0.21 – 6.0.36	6.0.21 – 6.0.36	✓
		7.0.0 – 7.0.32	7.0.0 – 7.0.32	✓
	CVE-2014-0075	6.0.0 – 6.0.39	6.0.0 – 6.0.39	✓
		7.0.0 – 7.0.52	7.0.0 – 7.0.52	✓
		8.0.0 – 8.0.3	8.0.0 – 8.0.3	✓
	CVE-2014-0095		7.0.47 – 7.0.52	⚠
		8.0.0 – 8.0.3	8.0.0 – 8.0.3	✓
	CVE-2014-0099	6.0.0 – 6.0.39	6.0.0 – 6.0.39	✓
		7.0.0 – 7.0.52	7.0.0 – 7.0.52	✓
		8.0.0 – 8.0.3	8.0.0 – 8.0.3	✓
	CVE-2014-0230	6.0.0 – 6.0.33	6.0.0 – 6.0.33	✓
		7.0.0 – 7.0.54	7.0.0 – 7.0.54	✓
		8.0.0 – 8.0.8	8.0.0 – 8.0.8	✓
Spring Framework	CVE-2013-7315	3.0.0M1 – 3.2.3	3.2.0M2 – 3.2.3	🕒
		4.0.0M1 – 4.0.0M2	4.0.0M1 – 4.0.0M2	✓
	CVE-2014-1904	3.0.0M1 – 3.2.7	3.0.0M1 – 3.2.7	✓
		4.0.0 – 4.0.1	4.0.0 – 4.0.1	✓
Apache Camel	CVE-2013-4330	1.1.0 – 2.9.7		🕒
		2.10.0 – 2.10.6		🕒
		2.11.0 – 2.11.1		🕒
		2.12.0		🕒
			2.12.4 – 2.12.5	⚠
			2.13.0 – 2.13.4	⚠
			2.14.0	⚠
	CVE-2014-0002	1.1.0 – 2.11.3	2.8.3 – 2.8.6	🕒
			2.9.0 – 2.9.8	✓
			2.10.0 – 2.10.7	✓
			2.11.0 – 2.11.3	✓
		2.12.0 – 2.12.2	2.12.0 – 2.12.2	✓
	CVE-2015-0263	1.1.0 – 2.11.3		🕒
		2.14.0 – 2.14.1	2.14.1	🕒
Apache CXF	CVE-2014-0034	2.1.0 – 2.6.11	2.4.3 – 2.4.10	🕒
			2.5.0 – 2.5.11	✓
			2.6.0 – 2.6.11	✓
		2.7.0 – 2.7.8	2.7.0 – 2.7.8	✓
	CVE-2014-0035	2.1.0 – 2.6.12	2.1.5 – 2.1.10	🕒
			2.2.0 – 2.2.12	✓
			2.3.0 – 2.3.11	✓
			2.4.0 – 2.4.11	✓
			2.5.0 – 2.5.11	✓
			2.6.0 – 2.6.12	✓
		2.7.0 – 2.7.9	2.7.0 – 2.7.9	✓
	CVE-2014-0109	2.1.0 – 2.6.13	2.1.0 – 2.6.13	✓
		2.7.0 – 2.7.10	2.7.0 – 2.7.10	✓
	CVE-2014-0110	2.1.0 – 2.6.13	2.2.6 – 2.2.12	🕒
			2.3.0 – 2.3.11	✓
			2.4.0 – 2.4.11	✓
			2.5.0 – 2.5.11	✓
			2.6.0 – 2.6.13	✓
		2.7.0 – 2.7.10	2.7.0 – 2.7.10	✓
Apache Derby	CVE-2006-7216	10.0.2.1 – 10.2.1.6	10.0.2.1 – 10.2.1.6	✓
	CVE-2009-4269	10.0.2.1 – 10.6.1.0	10.0.2.1 – 10.6.1.0	✓

Figure 8 provides a comparison between the amount of lines of code of the entire vulnerable methods for the 55 CVEs in our sample versus the amount of potentially vulnerable lines of code extracted with *"conservative fix dependency screening"* criterion: a possible phenomenon that we feared is that this criterion will over-approximate the vulnerable lines of code and be no different from the *"method screening"* criterion. However, this is not happening for our sample of vulnerabilities: while in the left and the middle parts of the graph the size of the methods tends to be small (i.e., less than 100 lines of code), the amount of evidence is close to the size of the relevant methods; however, as the size of the relevant methods tend to grow, the difference with the amount of the lines of code from the evidence becomes more apparent. For example, the rightmost point (a blue triangle) represents a case for Apache CXF in which the difference between the size of the vulnerability evidence and the size of the vulnerable method is more than three hundred lines of code.

Figure 9 shows the performance of the variants of the vulnerability screening test in terms of true positive (Sensitivity) and false positive (1-Specificity) rates, when compared to the ground truth classifier. We discuss the results below.

The *"method screening"* test did not show very high performance with most values of the threshold δ. However, when δ is set to 0, the classifier is marking a revision vulnerable based on just the presence or the absence of these methods, which may be a good vulnerability indicator when security is the only factor that matters, as its Sensitivity is equal to 1. Still, this strategy may have too many false positives in case affected methods were not vulnerable right from the point when they were introduced (Specificity = 0.002). This may result in potentially high security maintenance effort.

The *"combined deletion screening"* test showed similar performance to the above variant of the test, however it has slightly smaller Sensitivity (which does not contradict with the false negative error rate reported by Nguyen et al. [20]), as in several cases the deleted lines disappear before the actual vulnerable part of a method is gone.

The *"light fix dependency screening"* test shows significantly better performance when the threshold δ is set to 0.5 and 0.2. With $\delta = 0.5$, Sensitivity = 0.863, with Specificity = 1.0 (no false positives); while with $\delta = 0.2$, Sensitivity equals to 1.0. However, in the latter there are much more false positives (Specificity = 0.218). The amount of false positive results may be not important for a security assurance team, as long as Sensitivity is close to 1.0 [11]. On the other hand, for making quick estimates, significantly cutting down the number of false positives may be more preferable. Thus, the above threshold values may represent the trade-offs between the two conflicting goals: (1) the limited amount of development resources that dictates to prioritize only the work that is necessary, and (2) the requirement to provide maximum security assurance regardless the cost. In the first case, most of vulnerable revisions will be recognized correctly so that the appropriate action can be taken immediately, but there is still a small chance that some significantly older vulnerable revisions will be marked as safe. In the second case, no revisions will be incorrectly classified as non-vulnerable, but developers may spend a lot of additional work on false positives – this case is still better than looking at the presence of a vulnerable method, as it provides the same level of assurance with significantly smaller number of false positives.

On the other hand, the *"conservative fix dependency screening"* test yields more false positives after $\delta > 0.5$, however, for $\delta > 0.2$ it is the same as the *light* test. This is because for some of the vulnerabilities from our manual sample, the *conservative* test yields a larger initial vulnerability evidence fragment capturing more lines of code within a method that are not relevant to the vulnerable code fragment. Therefore, in such cases initial vulnerability evidence decays slower than the initial vulnerability evidence for the *light* test, showing different results at certain thresholds.

TABLE 6: Performance of the screening tests

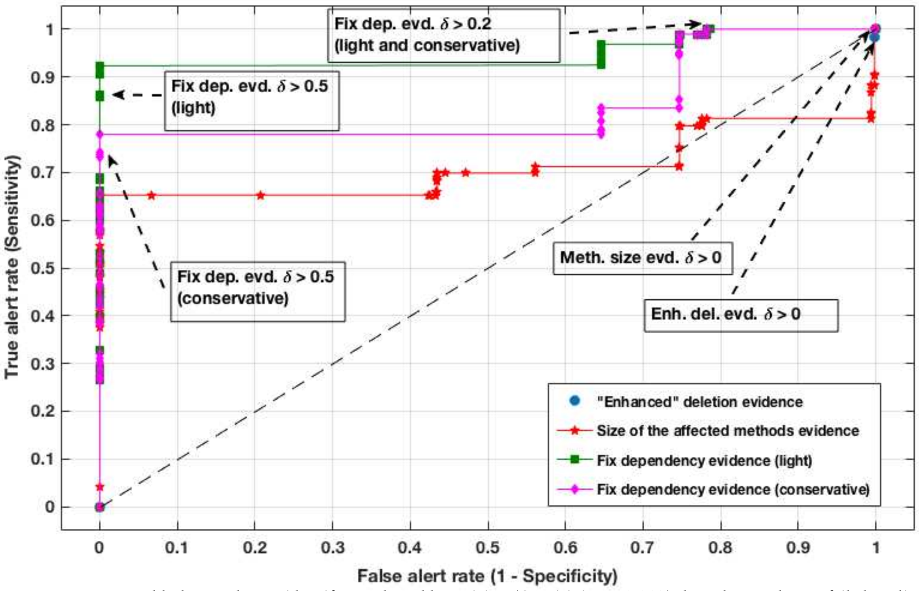
The Positive Predictive Value (PPV) for each test reflects the likelihood that a reported vulnerable version is actually vulnerable, while the Negative Predictive Value (NVP) suggests the opposite – the likelihood of a revision reported to be non-vulnerable is correctly a non-vulnerable revision. The results show that either variant of the fix dependency screening has better discriminative capabilities than the variants of the test based on the presence of deleted lines, or the size the affected methods.

Screening criterion	Threshold	Sens.	Spec.	PPV	NPV
Method screening (Section 5.2)	$\delta > 0.0$	1.000	0.002	0.927	1.000
	$\delta > 0.2$	0.905	0.002	0.920	0.002
	$\delta > 0.5$	0.801	0.224	0.929	0.082
	$\delta > 0.8$	0.653	1.000	1.000	0.186
"Combined" deletion screening (Section 5.3)	$\delta > 0.0$	0.982	0.002	0.925	0.010
Light fix dependency screening (Section 5.4)	$\delta > 0.2$	1.000	0.218	0.941	1.000
	$\delta > 0.5$	0.863	1.000	1.000	0.367
	$\delta > 0.8$	0.457	1.000	1.000	0.128
Conservative fix dependency screening (Section 5.4)	$\delta > 0.2$	1.000	0.218	0.941	1.000
	$\delta > 0.5$	0.742	1.000	1.000	0.235
	$\delta > 0.8$	0.458	1.000	1.000	0.128

However, Sensitivity and Specificity as general characteristics of a test are particularly informative in presence of a large prevalence of true positives in the population but might be significantly hindered in other scenarios (See [24] for a discussion on their limit). To discount for the prevalence [24] of the vulnerable revisions we also calculate the Positive Predictive Value (PPV, also called Precision) and the Negative Predictive Value (NPV) of the tests, that account for the test predictive capabilities. These values are shown in Table 6 alongside Specificity and Sensitivity. From these metrics we see that the fix dependency screening variants of the screening test have better discriminative capabilities than other variants of the test we tried.

As can be seen from Table 6, the *light fix dependency* test ($\delta > 0.5$) had no false positives, but had false negatives; in contrast, the *conservative fix dependency* test ($\delta > 0.2$) had no false negatives, but had false positives. We approximate the potential error rates for both tests – we use the Agresti–Coull confidence interval [48], that requires to solve for p the following formula:

$$|\hat{p} - p| = z \cdot \sqrt{p \cdot (1 - p)/n}, \tag{16}$$



The ***"combined" deletion screening*** *test could almost always identify a vulnerable revision (Sensitivity = 0.982), but almost always failed to distinguish a revision that is not yet vulnerable. The* ***method screening*** *test with $\delta = 0$ (a revision is classified as vulnerable when affected methods are present) could always identify a vulnerable revision (Sensitivity = 1.0), but had the same problem as the deletion screening (Specificity = 0.002). At the same time, both light and conservative fix dependency screening tests show significantly better performance than just looking at the deleted lines or the method(s) size: both in terms of true positive and false positive rates.*

Fig. 9: ROC curves for different variants of the vulnerability screening test

where p – is the estimated proportion of vulnerable (non-vulnerable) revisions; $\hat{p}$ – is the sample size proportion of vulnerable (non-vulnerable) revisions over the total sample of revisions n; and $z = 1.96$ – is the coefficient for the 95% confidence interval. We have chosen a large sample of CVEs for manual verification since it corresponds to a large sample of revisions n, which ensures small margin of error. Thus, we have a potential error rate for the tests as follows:

- The *light fix dependency* test with $\delta > 0.5$ had the 0% error rate when classifying *non-vulnerable* revisions (no false positives), and $13.7\% \pm 0.2\%$ error rate when classifying vulnerable revisions (few false negatives);
- The *conservative fix dependency* test with $\delta > 0.2$ had the $78.3\% \pm 0.8\%$ error rate when classifying *non-vulnerable* revisions (significant number of false positives), and 0% error rate when classifying *vulnerable* revisions (no false negatives).

As can be seen from Table 6 and Figure 9, the *method* and *deletion* screening criteria are less effective, therefore we do not report their error rates. Additionally, we compared the vulnerable version ranges taken from the NVD with the ranges identified by the expert analysis for the subset of 21 CVEs that were manually analyzed (the ground truth). This comparison is shown in Table 5 Already, on this subset of CVEs we observed the following three cases: (1) a vulnerable version range given by the NVD is correct (in comparison to the ground truth); (2) the NVD specifies a larger set of vulnerable versions, i.e., gives an over-approximation; (3) the NVD specifies a smaller set of vulnerable versions, i.e., gives an under-approximation. The cases in which the NVD over-approximates the range of vulnerable versions may result in unnecessary work that is created by resolving vulnerabilities for non-vulnerable versions of the consumed FOSS components (increased maintenance efforts). On the other hand, the cases in which the NVD provides under-approximations may result in not addressing potentially severe security vulnerabilities (increased security risks).

Therefore, a screening test that is using the fix dependency screening criterion with different threshold values can be used to mitigate both under- and over-approximation problems of the NVD, and to get better estimates about potential maintenance efforts as well as security risks connected with large consumption of FOSS components (we provide more discussion in Section 9).

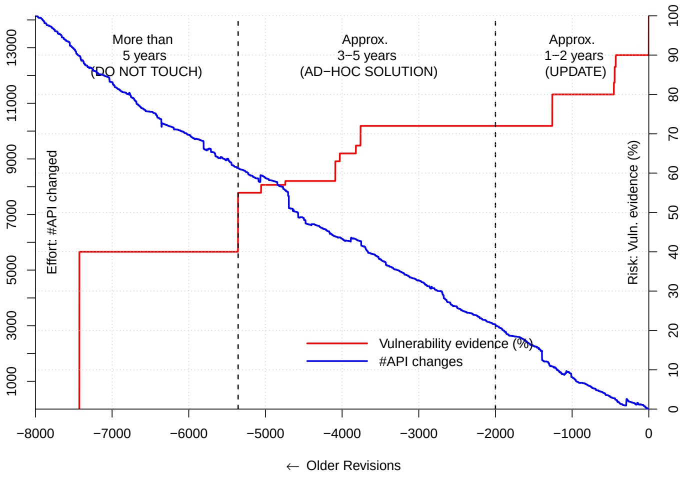
These results allow us to provide an answer to **RQ1**: tracking the presence/absence of the vulnerable methods or lines of code removed during a security fix may be not sufficient from the security maintenance effort management perspective. Still, fairly simple heuristics that capture lines of code that are potentially relevant to the vulnerable part of a method can be more beneficial for this task.

9 Decision Support for Security Maintenance

For those FOSS components, where upgrading to the latest version is likely a low effort, we just might want to update them – even if the risk is comparatively low. For components where the upgrade (or fixing) effort is high, we still can do a more expensive and more precise analysis. Still, getting an immediate estimate on the trade-offs between the upgrade effort and the likelihood of the security risk is the key for *not* wasting the (limited) available resources on FOSS components that are unlikely to be vulnerable, or are likely easy to upgrade.

Therefore, to answer **RQ2**, and provide an insight on whether developers could extract quick indicators for security maintenance decisions on FOSS components they consume, we performed an empirical analysis of the persistence of potentially vulnerable coding in source code repositories of the chosen projects. We also extracted the amount of changes between each revision and the fix in terms of changed public API, which we use as a proxy for the overall changes that may complicate component updates, increasing maintenance costs (see Section 7).

First of all, upon disclosure of a new vulnerability, developers could use a "local" decision support that would allow them to identify the vulnerability risk of for a version of a FOSS component in question, as well as the likelihood that the component can be updated without any major efforts. If an easy update is not possible (and for considerably older versions of software components this is rarely the case), the value of the vulnerability risk indicated by the presence of the vulnerable coding may be a useful indicator for the maintenance planning. With Figure 10, we illustrate such a decision support for developers: this information is generated by running the *conservative fix dependency screening* test for CVE-2014-0035 (Apache CXF). We take the value of the vulnerability evidence as the potential security risk, and measure the changes in the API between each revision and the fix for this CVE as a proxy for the upgrade effort. If a version of a FOSS component is not older than 2000 revisions back from the fix (approx. 1-2 years), it may be preferable to update the component, as most of the vulnerable coding is present, and difference in the API with respect to the fix is only starting to accumulate. On the other hand, if it is older than 5000 revisions back from the fix (more than 5 years), it may be more preferable to take no action, as most of the potentially vulnerable coding is gone, and changes accumulated between that point in time and the fix are too many. For cases when the version of interest lies somewhere between these two areas, a custom fix may be implemented.



As we move backward from the fix in the revision history, the coding that is responsible for a vulnerability possibly disappears (red curve, shows the value of evidence in LoC), whereas other changes in the code base start to accumulate (blue curve is the amount of API that changed in a certain revision with respect to the fix that represents the effort of upgrading from that point to the fix). A very old version may require to change 13000+ public methods for a vulnerability that may be very unlikely to be there (85% chances, see Figure 11). Thus, the position of the revision of interest in this diagram provides developers with a good insight on what decision to make.

Fig. 10: Trade-off curves for one vulnerability of Apache CXF (CVE-2014-0035)

To sketch a trade-off model that would allow to perform a retrospective analysis for "global" security maintenance of the whole FOSS component, we attempt to generalize the above "local" decision support. Similarly to Nappa et al. [49], who employed *survival analysis* to analyze the time after a security patch is applied to a vulnerable host, we used it to analyze the persistence of vulnerable coding that we extracted from the sample of FOSS projects (shown in Table 2) with our screening tests. Survival analysis is the field of statistics that analyzes the expected duration of time before an event of interest occurs [50], and is being widely used in biological and medical studies.

In our scenario, time goes backwards (from the fix), and we identify the following event affecting every pair of (CVE, FOSS) as an individual entity, depending on one's considerations:

Security Risk: the event whose probability we measure is "the ratio of the vulnerability evidence $E[r_{i-1}]/E[r_0]$ in a screening test falls below δ".

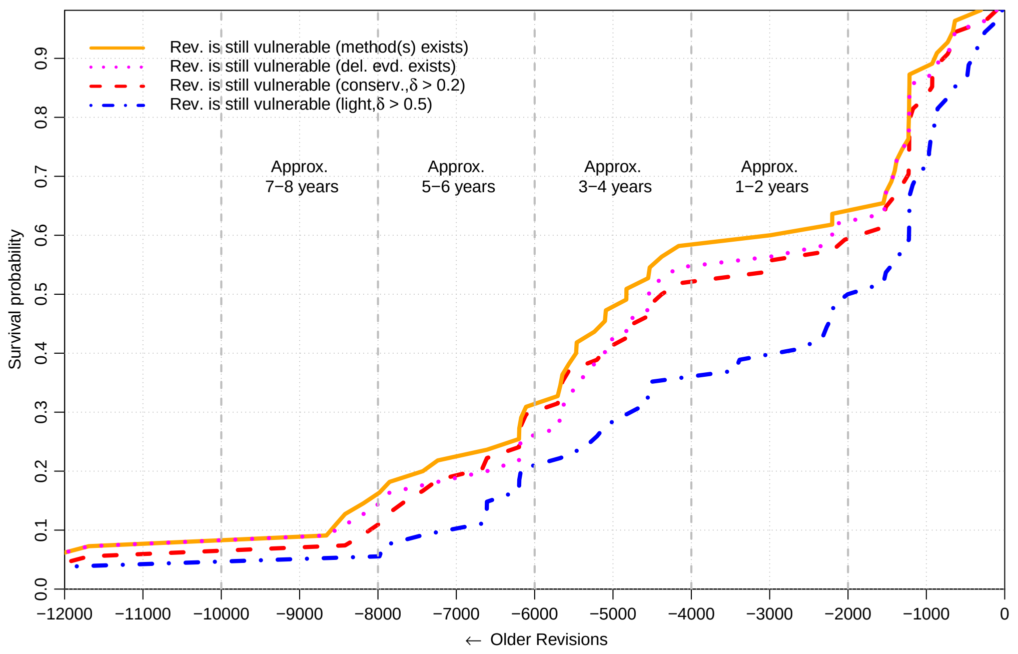
This event corresponds to the likelihood of the presence of the coding that is responsible for the vulnerability. To identify how this security risk may change over time, which is the concern of our **RQ2**, we computed the survival probabilities of vulnerable code fragments using the *light fix dependency screening* with $\delta > 0.5$, *conservative fix dependency screening* with $\delta > 0.2$, *method screening* with $\delta > 0$, and *"combined" deletion screening* tests (the variants of the screening test which performance we show in Figure 9). We performed survival analysis using the *survfit*[14] package in R, fitting the Kaplan-Meier non-parametric model (The Nelson–Aalen model gives the same qualitative result).

Figure 11 shows these survival probabilities: the vulnerable coding tends to start disappearing after 1000 commits (approximately 1 year preceding the fix), as already at 2000 revisions back there are 60% chances that the vulnerable coding is still there according to the evidence collected by *conservative fix dependency screening* (red curve). At 6000 revisions back (approx. 4 years) there is only 30% chance that the vulnerable coding survived, according to the same evidence. The curve that represents the probability of being vulnerable according to the evidence obtained with *light fix dependency screening* (blue curve) decays even faster. While the difference between the *conservative fix dependency screening* and method/deletion evidence presence is not that obvious on this figure, it is still significant (recall Figure 9).

Finally, we sketch the "global" decision support that represents the trade-offs that can be considered for the security maintenance of a FOSS project (**RQ2**), we further combine the survival curves for vulnerability evidences obtained with *light* and *conservative fix screening* tests over the set of vulnerabilities for the Apache Tomcat project, using the average values of API changes per project. Figure 12 represents the "global" trade-off decision support for the Apache Tomcat project, that consists of the following elements:

1) The dashed red line corresponds to the *conservative* probability that the vulnerable coding has survived at a certain point in time – this is based on the *conservative fix dependency screening* with $\delta > 0.2$ (our manual assessment for this test in Section 8 showed no false negatives, but a considerable amount of false positives).

14. https://cran.r-project.org/web/packages/survival/survival.pdf



The vulnerable coding tends to start disappearing after 1000 commits ($\leq$ 1 year preceding the fix), as already at 2000 revisions back there are 60% chances that the vulnerable coding is still there according to the evidence collected by conservative fix dependency screening (red curve). Further back (after approx. 6 years), there is only a small probability that a component is vulnerable.

Fig. 11: Survival probabilities of the vulnerable coding with respect to different variants of the screening test

2) The solid red line corresponds to the *lightweight* probability that the vulnerable coding is still there – this is based on the *light fix dependency screening* with $\delta > 0.5$ (our manual assessment for this test in Section 8 showed no false positives and a low number of false negatives).
3) Each point on the solid blue line corresponds the number of the API changed in a certain revision in comparison to the fix: these are the aggregated average numbers taken for the whole project sample (the two dashed lines are the 0.95% confidence interval).

Figure 12 gives a recommendation to developers to update their versions of a component on a yearly basis, as after that time the vulnerability risk is likely to be still high, and the API changes tend to grow fast. The average amount of API changes,[15] as well as both risk values, suggest that the security assessment should be performed when a version of interest lags for around 3-4 years behind the fix (between 4000 and 6000 commits). Here the down-port decision could be evaluated, considering that the *conservative* risk estimate is still high at this point. Alternatively, if the *lightweight* risk estimate is tolerable, developers may already prefer to take no action at this point. Looking at both *conservative* and *lightweight* probabilities for the vulnerability risk and the average amount of the API changes, the point after 8000 commits could be the one at which the “do not touch” decision might be the only reasonable choice.

The analysis presented in this section is based on the conservative assumption that a software vendor has an up-to-date inventory of FOSS components that at least contains the information about which FOSS components are used in specific software products, as well as the information about the versions of these components (e.g., Black Duck [1]).

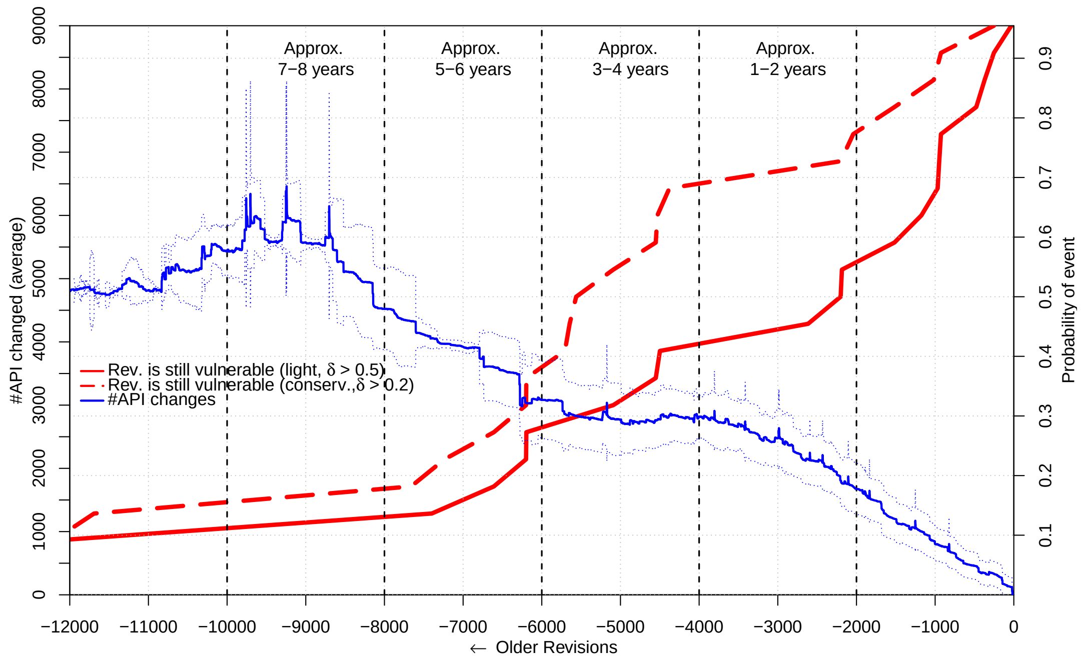
When such an inventory is outdated or does not exist, tracking the usage of FOSS components across the entire software portfolio of large software vendors is a very challenging task [51]. Yet, the information about software and its versions can be also extended by indicating the exact functionalities of FOSS components that are being used – this can further reduce the security risk as well as the associated maintenance effort.

10 Threats to Validity

In our approach the *construct validity* may potentially be affected by the means of data collection and preparation, the selected sample of FOSS projects, and the accuracy of the information about security fixes in them:

- *Misleading commit messages.* As pointed by Bird et al. [41] (and from our own experience), linking CVE identifiers to specific commits in source code repositories is not trivial: developers may not mention fix commits in the NVD and security notes, and they may not mention CVE identifiers within commit logs. Also, automatic extraction of bug fix commits may introduce bias due to misclassification (e.g., a developer mentions a CVE identifier in a commit that is not fixing this CVE). To minimize such bias, we collected this data manually, using several data sources, including third-party sources that do not belong to the actual projects. Manual data collection allowed us to additionally verify that every vulnerability fix commit that we collected is indeed a fix for a particular CVE, therefore we do not have the latter bias of misclassification.

15. A certain older revision r_y may actually have less API changes with respect to the fix than a certain newer r_x for a simple reason, that r_y has less functionality than r_x – this may be the reason why the amount of API changes that we observe in Figure 12 is not as “linear” as in Figure 10.



As we move back from the fix in the revision history, the probability that a revision is still vulnerable (red solid and dashed curves) holds high within the first 2 years before the fix (around 4000 revisions back). At the same time, the average amount of API changes (blue curve, the two dashed blue curves are the 0.95% confidence interval) accumulates fast – this may be the right time for an update. Further back, between approx. 3 and 4 years before the fix, the amount of changes does not grow significantly, but the vulnerability risk is still relatively high – this may be the time frame for a thorough security assessment of a version in question. Further back (after approx. 4 years before the fix), the vulnerability risk falls down, and changes begin to accumulate even more – here the "do not touch" decision might be the only reasonable choice.

Fig. 12: "Global" trade-off curves for 22 vulnerabilities of Apache Tomcat

- *Tangled code changes in vulnerability fixes.* There is a potential bias in bug-fix commits, such that along with fixing relevant parts of the functionality, developers may introduce irrelevant changes (e.g., refactor some unrelated code). Kawrykow and Robillard [47], and Herzig et al. [21] explored to what extent bug-fix commits may include changes irrelevant to the original purpose of the fix: while they show that there may be significant amount of irrelevant changes for general bugs, Nguyen et al. [20] observed that for the majority of security fixes this was not the case – this is also supported by our findings of very "local" changes (Figure 5). The subset of vulnerabilities that we checked manually did not contain such refactorings.
- *Incomplete or broken histories of source code repositories*. The commit history of FOSS projects may be incomplete (e.g., migrating to different types of version control systems, archiving or refactoring), limiting the analysis capabilities. We checked the repository histories of all seven projects in our sample finding them all to be complete, except for Jenkins. In case of Jenkins, at one point in time the whole repository layout was deleted, and then re-created again. Our current implementation does not handle such cases, as it works under the assumption that repositories are complete and well-structured. Still, such cases (and similar ones) can be handled automatically, extending the current implementation with more heuristics.
- *Existence of complex "architectural" vulnerabilities*. We improved over the work by Nguyen et al. [20] by using slicing over the source code albeit limiting the scope of the slice to distinct Java methods. This may be not adequate for sophisticated, "architectural", vulnerabilities. Nguyen et al. [20] have reported that less than 30% of security fixes for Firefox and Chrome browsers (out of the total of approximately 9,800 vulnerabilities that the authors processed) involved more than 50 lines of code, which implies that in most cases the changes to the code base were rather small (which is also supported by our sample of vulnerabilities). Hence, a *prima facie* evidence is that complex vulnerability fixes can be considered as outliers from the perspective of our methodology. In such complex cases, additional analysis would be anyhow needed.
- *The lack of representativeness of the collected sample of vulnerabilities on typical vulnerabilities found in software projects*. We believe that this threat is minimized, since we selected large and popular software projects, as well as checked whether the distribution of vulnerability types in these projects corresponds to the vulnerability type distribution of the large sample of FOSS components integrated by our industrial partner.
- *Human error*. Our manual validation of the screening tests over the subset of vulnerabilities might be biased due to human errors and wrong judgment. In order to minimize such bias, manual checks were performed by three different experts, who were cross-checking and discussing the results of each other.

The *internal validity* of the results depends on our interpretation of the collected data with respect to the analysis that we performed. We carefully vetted the data to minimize the risk of wrong interpretations.

Additionally, like any other approach that is purely static, the proposed screening test may be over- or under-approximating the results, since the actual vulnerable behavior is an estimation. While we did not create exploits to test the actual vulnerable behavior, of various versions against selected vulnerabilities (as it is a very labor-intensive task), we performed manual code audits that confirmed the adequacy of the results. Moreover, the error margin of the test that we calculated in Section 8 suggests that it is satisfactory for the primary purpose of the screening test – providing a quick and scalable method for estimating the vulnerability status of a large amount of FOSS components used by software vendors.

The *external validity* of our analysis lies in generalizing to other FOSS components. It depends on the representativeness of our sample of FOSS applications from Table 2, and the corresponding CVEs. As the FOSS projects that we considered are widely popular, have been developed for several years, and have a significant number of CVEs, those threats are limited for FOSS using the same language (Java), and having the same popularity. Generalization to other languages (such as C/C++) should be done with care, looking at Table 4.

11 Conclusions and Future Work

We presented an automated, effective, and scalable approach for screening vulnerabilities and upgrade effort for large FOSS components consumed by proprietary applications.

The main conclusion of our work is that the proposed screening test can be useful for mitigating the over- and under-approximation problems in the information about vulnerable versions from public vulnerability databases such as the NVD, which is of great help for large software vendors that integrate many FOSS components into their products in making security maintenance decisions when new vulnerabilities in these components are published.

We empirically confirm our intuition that purely syntactic heuristics for tracking the presence of vulnerable coding in source code repositories proposed in earlier literature can be further improved with the source code slicing that extracts local dependencies of the source code lines modified during a security fix. Moreover, we find that there can be different thresholds for the screening test that can represent different trade-offs between the desired security risk and the amount of maintenance effort that a software vendor can tolerate.

Our approach represents an enhancement of the original SZZ approach by Śliwerski et al. [17] and its successors (e.g., Nguyen et al. [20]), and can be applied to identify changes inducing generic software bugs. However, the fixes of such bugs should have similar properties as the security vulnerabilities that we discuss in this paper (see Table 4), and should be "local". Otherwise, different heuristics for extracting the evidence may be needed.

While our current prototype is limited to vulnerabilities in Java source code, the approach can be easily extended to other programming languages and configurations. In practice, it depends on the availability of a program slicer for a particular programming language.

We see several lines of future work to better understand the quality/speed trade-offs as well as to extend the scope of our approach:

- Improve the quality of vulnerability evidence tracking by handling changes across multiple files as well as investigating the impact of more precise slicing algorithms. We do not expect a significant improvement in this direction as, in our experience, vulnerabilities were mostly fixed locally by touching few lines.
- Improve the quality of estimation of the update effort by including changes in build dependencies (direct and transitive), which might influence the estimate.[16]
- Extend the approach to more programming languages and test the approach on different types of projects.

Currently, our experimental validation has focused on a selection of software components motivated by the needs of the security team at large enterprise software vendor. It can be easily adapted to support other scenarios: e.g., by development teams to assess whether a vulnerable functionality is actually invoked by a consuming application (as in Plate et al. [30]), or by security researchers to improve the quality of vulnerability database entries (as in Nguyen et al. [20]).

As the results of the empirical evaluation suggest, the screening test approach did not have a significant false positive problem, but, as we mentioned in Section 7, the precision/recall can be further refined with different versions of the *Vulnerability Evidence Extractor* component (shown in Figure 7).

Acknowledgments

We would like to thank M. Bernhardt, H. Plate, S. Ponta, A. Sabetta, and U. Sodan for many discussions on the challenges faced by software vendors, and E. Blanzieri for suggesting the name of "screening test" for our method.

This work has been partly supported by the European Union under the grant 317387 SECENTIS (FP7-PEOPLE-2012-IT), EU project VAMOSS (EIT/EIT DIGITAL/SGA2016-16367), and CISCO Country Digitalization Grant (Filiera Sicura).

References

[1] "The tenth annual future of open source survey," https://www.blackducksoftware.com/2016-future-of-open-source, Black Duck Software, 2016, last accessed 2016-05-22.

[2] M. Höst and A. Oručević-Alagić, "A systematic review of research on open source software in commercial software product development," *Inf. and Softw. Tech. Journ.*, vol. 53, no. 6, pp. 616–624, 2011.

[3] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio, "Development with off-the-shelf components: 10 facts," *IEEE Softw. Journ.*, vol. 26, no. 2, p. 80, 2009.

[4] Forrester Consulting, "Software integrity risk report. the critical link between business risk and development risk." Tech. Rep., 2011.

[5] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: the case of a Smalltalk ecosystem," in *Proc. of FSE'12*, 2012.

16. For example, consider an application that only runs on Java 1.4 and a FOSS upgrade that would require Java 1.8: to resolve this dependency conflict, either the fixes for the FOSS components need to be back-ported to Java 1.4 or the whole applications needs to be ported to Java 1.8.

[6] G. Ellison and D. Fudenberg, "The neo-luddite's lament: Excessive upgrades in the software industry," *RAND Journ. of Econ.*, vol. 31, no. 2, pp. 253–272, 2000.

[7] C. Ioannidis, D. Pym, and J. Williams, "Information security trade-offs and optimal patching policies," *Eu. Journ. of Op. Res.*, vol. 216, no. 2, pp. 434 – 444, 2012.

[8] N. Mukherji, B. Rajagopalan, and M. Tanniru, "A decision support model for optimal timing of investments in information technology upgrades," *Deci. Supp. Sys. Journ.*, vol. 42, no. 3, pp. 1684–1696, 2006.

[9] I. Sahin and F. M. Zahedi, "Policy analysis for warranty, maintenance, and upgrade of software systems," *Journ. of Softw. Maint. and Evol.*, vol. 13, no. 6, pp. 469–493, 2001.

[10] L. Allodi and F. Massacci, "Comparing vulnerability severity and exploits using case-control studies," *ACM Trans. on Inf. and Sys. Sec. Journ.*, vol. 17, no. 1, pp. 1–20, 2014.

[11] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities-does experience matter?" in *Proc. of ARES'09*, 2009.

[12] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proc. of ICSE'11*, 2011.

[13] S. Dashevskyi, A. D. Brucker, and F. Massacci, "On the security cost of using a free and open source component in a proprietary product," in *Proc. of ESSoS'16*, 2016.

[14] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto." *Comm. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. of ICSE'12*, 2012.

[16] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 112–122, 2007.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT SEN*, vol. 30, no. 4, pp. 1–5, 2005.

[18] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *Proc. of DSN'08*, 2008.

[19] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. of ASE'03*, 2003.

[20] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Emp. Soft. Eng. Journ.*, vol. 21, no. 6, pp. 2268âĂŞ–2297, 2015.

[21] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Emp. Soft. Eng. Journ.*, vol. 21, no. 2, pp. 303–336, 2016.

[22] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *Proc. of ICSE'08*, 2008.

[23] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proc. of PASTE'07*, 2007.

[24] D. A. Grimes and K. F. Schulz, "Uses and abuses of screening tests," *The Lancet*, vol. 359, no. 9309, pp. 881 – 884, 2002.

[25] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[26] J. Thome, L. K. Shar, and L. Briand, "Security slicing for auditing XML, XPath, and SQL injection vulnerabilities," in *Proc. of ISSRE'15*, 2015.

[27] A. Treffer and M. Uflacker, "Dynamic slicing with Soot," in *Proc. of SOAP'14*, 2014.

[28] J. Graf, "Speeding up context-, object- and field-sensitive SDG generation," in *Proc. of SCAM'10*, 2010.

[29] V. P. Ranganath and J. Hatcliff, "Slicing concurrent Java programs using Indus and Kaveri," *Inter. Journ. on Softw. Tools for Tech. Transf.*, vol. 9, no. 5-6, pp. 489–504, 2007.

[30] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *Proc. of ICSME'15*, 2015.

[31] A. Meneely, H. Srinivasan, A. Musa, A. Rodriguez Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *Proc. of ESEM'13*, 2013.

[32] S. Kim, T. Zimmermann, K. Pan, and J. E. Whitehead Jr., "Automatic identification of bug-introducing changes," in *Proc. of ASE'06*, 2006.

[33] M. Di Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Inf. and Softw. Tech. Journ.*, vol. 51, no. 10, pp. 1469–1484, 2009.

[34] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, "The SQO-OSS quality model: measurement based open source software evaluation," in *Proc. of IFIP OSS'08*, 2008.

[35] S. Zhang, X. Zhang, X. Ou, L. Chen, N. Edwards, and J. Jin, "Assessing attack surface with component-based package dependency," in *Proc. of NSS'15*, 2015.

[36] T. Dumitras, P. Narasimhan, and E. Tilevich, "To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains," *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 865–876, 2010.

[37] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *Proc. of PROMISE'07*, 2007.

[38] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. of CCS'07*, 2007.

[39] F. Massacci and V. H. Nguyen, "An empirical methodology to evaluate vulnerability discovery models," *IEEE Trans. on Softw. Eng. Journ.*, vol. 40, no. 12, pp. 1147–1162, 2014.

[40] M. Weiser, "Program slicing," in *Proc. of ICSE'81*, 1981.

[41] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proc. of ESEC/FSE'09*, 2009.

[42] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proc. of PROMISE'17*, 2017.

[43] L. Nyman and T. Mikkonen, "To fork or not to fork: Fork motivations in sourceforge projects," in *Proc. of IFIP OSS'11*, 2011.

[44] S. Stanciulescu, S. Schulze, and A. Wasowski, "Forked and integrated variants in an open-source firmware project," in *Proc. of ICSME'15*, 2015.

[45] G. Robles and J. M. González-Barahona, "A comprehensive study of software forks: Dates, reasons and outcomes," in *Proc. of IFIP OSS'11*, 2012.

[46] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study," *Emp. Soft. Eng. Journ.*, vol. 20, no. 5, pp. 1275–1317, 2015.

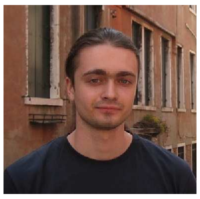
[47] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proc. of ICSE'11*, 2011.

[48] A. Agresti and C. A. Franklin, *Statistics: the art and science of learning from data*. Pearson Higher Education, 2012.

[49] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: a study of the impact of shared code on vulnerability patching," in *Proc. of SSP'15*, 2015.

[50] R. G. Miller Jr., *Survival analysis*. John Wiley & Sons, 2011, vol. 66.

[51] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: Finding the provenance of an entity," in *Proc. of MSR'05*, 2011.



Stanislav Dashevskyi received his Ph.D. in ICT at University of Trento in 2017, under the supervision of Prof. Fabio Massacci. He worked as a Quality Assurance Engineer in a software development company before joining University of Trento. Currently, he is a member of the SaToSS research group at the University of Luxembourg, working as a Research Associate under the supervision of Prof. Sjouke Mauw. His interests include software security, software vulnerability analysis and security certification of third-party software. Contact him at *stanislav.dashevskyi@uni.lu*.

Achim D. Brucker is a Senior Lecturer (Associate Professor) and Consultant (for software security) at the Computer Science Department of The University of Sheffield, UK. He has a Ph.D. in Computer Science from ETH Zurich in Switzerland. Until December 2015, he was a Research Expert (Architect), Security Testing Strategist, and Project Lead in the Global Security Team of SAP SE. He was involved in rolling out static and dynamic application security testing tools to the world-wide development organization of SAP. He represented SAP in OCL standardization process of the OMG. Contact him at *a.brucker@sheffield.ac.uk*.

Fabio Massacci is a full professor at the University of Trento. He has a Ph.D. in Computing from the University of Rome La Sapienza in 1998. He has been in Cambridge (UK), Toulouse (FR) and Siena (IT). Since 2001 he is in Trento. He has published more than 250 articles on security and his current research interest is in empirical methods for security. He participates to the FIRST SIG on CVSS and was the European Coordinator of the multi-disciplinary research project SECONOMICS on socio-economic aspects of security. Contact him at *fabio.massacci@unitn.it*.