Noname manuscript No.
(will be inserted by the editor)

An Automatic Method for Assessing the Versions Affected by a Vulnerability

Viet Hung Nguyen · Stanislav Dashevskyi · Fabio Massacci

Resubmitted with Revision: 4th September 2015

Abstract Vulnerability data sources are used by academics to build models, and by industry and governments to assess compliance. Errors in such data sources are not just threats to validity in scientific studies, but might cause organizations which rely on old versions of a software to lose compliance. In this work, we propose an automated method to determine whether there exists some code evidence that a vulnerability known to occur in the present version of a software is also present in its past versions. The method scans the code base and identifies the lines of code that were changed to fix the vulnerability. If an earlier version contains some lines from the changed vulnerable footprint, it is likely that this version is vulnerable. To show the scalability of the method we performed a large scale experiments on Chrome and Firefox (spanning $7,236$ vulnerable files and approximately $9,800$ vulnerabilities) on the National Vulnerability Database (NVD). As an example on the impact of our method, we show how the elimination of spurious vulnerability claims (i.e. entries in a vulnerability database) may change the conclusions of studies on the prevalence of foundational vulnerabilities.

Keywords Software Security · Empirical Validation · Vulnerability Analysis · National Vulnerability Database (NVD) · Browsers

V. H. Nguyen
Department of Information Engineering and Computer Science (DISI),
University of Trento, Italy
E-mail: vhnguyen@disi.unitn.it

S. Dashevskyi
Department of Information Engineering and Computer Science (DISI),
University of Trento, Italy
E-mail: stanislav.dashevskyi@unitn.it

F. Massacci (✉)
Department of Information Engineering and Computer Science (DISI),
University of Trento, Italy
E-mail: fabio.massacci@unitn.it

1 Introduction

Public vulnerability databases such as the National Vulnerability Database (NVD, web.nvd.nist.gov), the Open Source Vulnerability Database (OSVDB, www.osvdb.org), and Bugtraq (www.securityfocus.com) have been used by both academics, (Massacci *et al.* 2014; Neuhaus *et al.* 2007), and industry, (Younan 2013), to study and assess software security. NVD keeps tracts of tens of thousands of vulnerabilities, which are distinguished by a unique identifier, called CVE. For each entry, NVD lists its alleged vulnerable software configurations (or versions).

Besides academic interest, if NVD claims a software vulnerable, the presence of a vulnerable software in a company's deployed IT system has a major impact on a company's compliance with regulations. For example, the US Government has mandated the use of NVD to determine the compliance of its procured software (Quinn *et al.* 2010). To keep compliance, companies have to allocate their own resources to fix vulnerabilities of possibly out-of-support open source software in their products. They might need to upgrade software with no problems, just because the NVD says so.

Another example, a credit card merchant needs to be compliant with the Payment Card Industry Data Security Standard (PCI DSS) (Williams *et al.* 2012) even if it is not a regulation or a law[1]. One PCI DSS specific requirement is the following one: "fix all medium- and high-severity vulnerabilities". If the merchant has a vulnerable software, as reported by NVD, embedded in its products, then it might lose PCI DSS compliance. This may lead to fines raking hundreds of thousands of euros. This problem also affects proprietary software systems that use open source components and are often operational for several years. See for example the list[2] of open source components used by SAP, the world leader in proprietary enterprise resource planning products.

Yet, the information about vulnerable software in NVD is not always reliable. A folk knowledge, confirmed by our experience, is that NVD applies a conservative rule: "If version X is vulnerable , then so are all its previous versions". Some retro versions of software may therefore be marked as vulnerable because of this bias. For example, NVD claimed that over 99% of vulnerabilities in Chrome v2–v12 were originated from Chrome v1. Yet, Chrome v12 is significantly different from Chrome v1 in terms of code base: more than 100% new components were introduced. If this was true, then all new components would have been almost vulnerability free. A more likely explanation, supported by our analysis, could instead be that this conservative rule makes the NVD an inaccurate representation for past versions of software.

If NVD may be biased, how do we check that an old software version that a company has embedded in its shipped products is actually vulnerable? If

[1] Some states in US (*i.e.* Nevada and Minnesota) have adopted PCI DSS as a actual law for some business operating in these states (Williams *et al.* 2012, Chap.3)

[2] http://www.sap.com/corporate-en/about/our-company/policies/sybase/third-party-legal.html

the software is open source, the code is available but manually checking it is simply impossible when the number of vulnerabilities is large.

1.1 Contribution

Our goal is to answer such question. The major contributions of this work are as follows:

1. We propose an automatic method to empirically assess the reliability of claims about the vulnerable status of retro software versions, *i.e. vulnerability claims*. We build upon the work by Sliwerski *et al.* (2005), who attempted to detect source lines of code that are responsible for generic bugs. However, there are important differences between generic bugs and vulnerabilities (Needham 2002): vulnerabilities are mostly discovered and exploited by people who are not the software's own developers. Therefore, users of vulnerable software may need to upgrade to a new version that might break existing functionalities. In this respect, understanding whether a version is "really" vulnerable is paramount. As a result, our approach and Sliwerski *et al.* (2005) are different in two points:
   (a) Our approach could accept false positives (*i.e.* a version is claimed to be vulnerable, while it is not), but tries to avoid false negatives (*i.e.* a version is claimed to be clean, while it is vulnerable) as much as possible (the descriptions of potential false negative errors are given in Section 4 and Section 6). In contrast, the approach by (Sliwerski *et al.* 2005) tried to minimize false positives.
   (b) Our approach focuses on the question: *"which versions are truly affected by which vulnerabilities?"*, or *"is a vulnerability claim valid?"*. This was not the concern of the approach by (Sliwerski *et al.* 2005). Therefore, they could not answer such a question.
2. To show the applicability of the method, we perform a large scale, experiment to assess the validity of the vulnerability claims by NVD for 33 major versions of Chrome and Firefox, covering $7,236$ vulnerable files and $\sim 9,800$ vulnerabilities claims. The experiment revealed systematic spurious vulnerability claims (NVD says vulnerable, but there is no code evidence for the presence of the vulnerability).
3. We also performed a manual validation of the approach in order to identify potential false negatives that might be present. Out of a random sample of 80 manually assessed vulnerabilities in Firefox, 6 false negatives were due to a mis-alignment among different repositories (CVS and Mercurial) and only 1 was found to be an actual false negative (due to a code reversal). By using a the score confidence interval calculation we can estimate a possibility of error for our method between 1.3% and 3.9% with a 95% confidence interval.
4. We show that spurious claims can significantly bias empirical analysis about foundational/inherited vulnerabilities. For example, NVD data im-

plies that most of vulnerabilities originated from the very first version. Once spurious claims are removed, this is no longer the case.

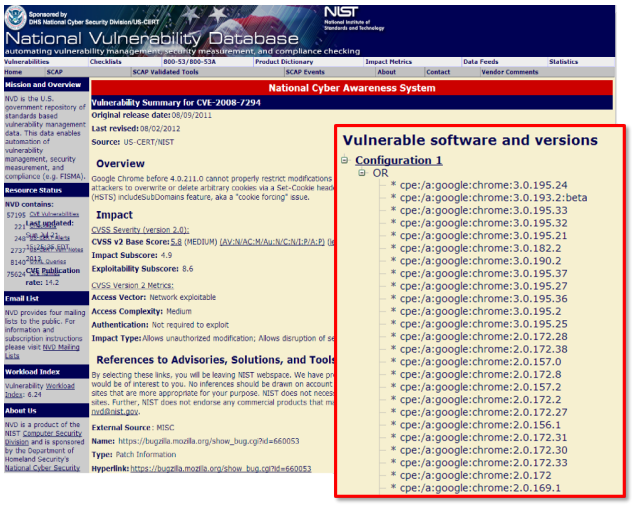
This work extends our earlier conference paper (Nguyen *et al.* 2013), where we have assessed the vulnerable version data of NVD on Chrome on a limited data set of about one-third of vulnerabilities. In this paper, we extend our previous work by revising the assessment method and extend the experiment on both Chrome and Firefox. We run our method on more than 70% of total vulnerabilities of these browsers.

In the next section (§2) we present the terminology. Then we describe our research questions (§3) and the details our proposed assessment method (§4). We describe an application of the proposed method to assess the vulnerability claims of Chrome and Firefox (§5) and present an independent manual analysis to assess the validity of the method (§6). Then we show how spurious claims can impact the analysis of vulnerability studies on those browsers (§7), address potential threats to validity (§8), and discuss related studies in the field (§9). Section 10 concludes this work.

2 Terminology

- A *software vulnerability* is an instance of a mistake in specification, configuration, and development such that its execution violates a security policy (Krsul 1998; Ozment 2007). In this work, we focus on vulnerabilities which could be classified as security programming bugs.
- A *vulnerability entry* is an entry reporting security problem(s) in a vulnerability data source, *e.g.,* NVD entries (a.k.a CVE) in the NVD data source.
- A *vulnerability claim* is a statement by a data source that a particular software version is vulnerable to a particular vulnerability entry. Fig. 1 shows an example of the claims of the entry CVE-2008-7294.
- A *commit* is a unit of changes in source code, managed by the code base repository.
- A *bug-fix commit* is a commit that contains changes to fix a (security) bug.
- A *vulnerable code footprint* is a set of lines of code (LoCs) which are changed, or removed to fix a security bug. The intuition to identify such vulnerable code footprints is to compare the revision where a security bug is fixed (*i.e.* bug-fix commit) to its immediately preceding revision. LoCs that appear in the preceding revision, but not in the bug-fixed revision are considered vulnerable code footprints[3]. If the fix was done by just adding code, then conservatively we consider the entire component as vulnerable.
- A *component* in a code base is a compilation unit. In C/C++, a component is a body file (*e.g.,* `.c`, `.cpp` file) plus its (optional) header file (*e.g.,* `.h`,

[3] While this is sufficient for non-critical applications, it is possible that we might miss important information that is not a part of the vulnerable code footprint (See Section 4.4 and Section 6 for a discussion).



A vulnerability claim is a statement by a data source that a particular software version is affected by a vulnerability.

Fig. 1 The vulnerability claims of the CVE-2008-7294, reported in its *vulnerable software and versions* feature.

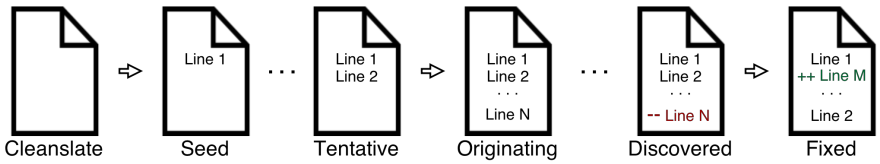


Fig. 2 An example of revision history that contains a security vulnerability and the corresponding fix.

`.hpp` file). The body file and the header file share the file name, but have different extension. Example: both `media_bench.cc` and `media_bench.hh` are treated as a single component `media_bench`.

Fig. 2 shows an example of a software component revision history where a security vulnerability was found and fixed. We distinguish the following typical revisions that might be present in such history (given in a reverse order):

- *Fixed* - is a revision that was created with a bug-fix commit. It might contain an *addition evidence* ("Line M") that corresponds to the lines of code added for producing a fix.
- *Discovered* - is a revision before the fixed one, where a security vulnerability was discovered. It might contain a vulnerability code footprint (*deletion evidence* - "Line N") that corresponds to the lines of code deleted for producing a fix.
- *Originating* revision is identified by our vulnerability assessment method, as the first version where a vulnerability was *initially* introduced.

- *Tentative* - is a revision that has common lines of code with the *Originating* revision, except that is does not contain a *deletion evidence*.
- *Seed* - is the earliest *Tentative* revision.
- *Cleanslate* - is a revision that has no lines of code in common with any of consequent revisions. A vulnerability claim about the cleanslate revision is clearly potentially spurious.

3 Research Questions and Method Overview

In this work, we consider the following research questions:

RQ1 *How could we estimate the validity of a vulnerability claim?*

RQ2 *To what extent do potentially spurious vulnerability claims impact the conclusions of a vulnerability analysis?*

The hard evidence that is showing whether a vulnerability claim is spurious, *i.e.* answering **RQ1**, can be obtained by reproducing (or failing to reproduce) the exploit in the corresponding software environment. Such verification can be easily done only for vulnerabilities for which a proof-of-concept exploit is available. If it does not exist, it must be created by carefully examining the source code, which requires a significant effort. For instance, the *Malware-Lab*(Allodi *et al.* 2013) was used for exploit kit verification, and included only few hundreds of vulnerabilities.

The above strategy is not feasible for large-scale verifications for thousands of claims, since NVD does not reveal enough information for quickly building exploits. We therefore can benefit from an automatic approach that will help us to estimate whether a vulnerability claim is spurious by trading off the precision for scalability.

To address **RQ1** we propose a method that automatically identifies the code evidence of a vulnerability entry in the code base of a particular version. From this evidence, we could estimate the validity of a vulnerability claim of this particular vulnerability entry. This limits the application of our method to software for which code is available.

Table 1 summarizes the claim assessment steps of our method. The method takes a list of entries of a vulnerability data source and their corresponding vulnerability claims as input of the first step. The output of each step is piped to the next consecutive one. The output of the final step, which is also the output of the method, is the assessment of each vulnerability claim plus its corresponding code evidence (if any). The method details are elaborated in Section 4.

The intuition behind our algorithm is the following:

1. If there is no deletion evidence, we consider all revision from the *Seed* to the *Discovered* revision as vulnerable. The *Cleanslate* is considered as not vulnerable.

Table 1 Overview of the proposed assessment method.

INPUT	Vulnerability entries of a vulnerability data source and their vulnerability claims	
STEP 1	*Link Vulnerability Claim to Bug Identifier*	Establish the trace from vulnerability claims to responsible security bug identifiers from the vendor.
STEP 2	*Locate Bug-Fix Commit*	Locate the security bug identifiers in the source code repository (change log).
STEP 3	*Identify Vulnerable Code Footprint*	Identify LoCs that are modified to fix the vulnerability fixed by the particular bug-fix commit.
STEP 4	*Determine the Validity of Vulnerability Claim*	Scan through the code base of every software version to determine the presence (or absence) of the vulnerable code footprint.
OUTPUT	The validity of each vulnerability claim plus its code evidence (if any)	

2. If there is some deletion evidence, then the algorithm stops at the *Tentative* revision and considers the *Seed* and other intermediate revisions as not vulnerable. Changed lines would count as the deletion evidence.

In Scenario 1 we have little evidence, therefore we consider *every* line of code as the source of a problem. In Scenario 2 we assume that the changed lines of code were the source of a problem. The approach might have both false positive (1,2) and false negative (2) errors, but we try to minimize the latter.

We illustrate the issue with some real examples in Section 6 where we performed a manual validation of the approach to check its correctness. The vulnerability sample selected for the manual verification is large enough to ensure some confidence in the result. To address **RQ2** we conduct an experiment to explore the effect of spurious vulnerability claims about the majority of foundational/inherited vulnerabilities (Ozment *et al.* 2006). Another study about the quarterly trends of the foundational vulnerability discovery is reported in (Nguyen 2014). We analyze the difference between the conclusions made in two different settings. The first setting uses all vulnerability claims by NVD. The second setting also uses vulnerability claims by NVD, but without the spurious claims identified by our proposed method.

An important assumption in our approach is the following:

ASS1 *The revision system of the repository is incremental, so that the notion of "preceding revision" is correctly captured by the historical notion of revisions recorded by the repository.*

Two phenomena violate this assumption: massive restructuring of the repository (e.g. passing from CVS to Mercurial), and the presence of many instances of code reversions. The false negatives we have identified in our manual analysis belong to these classes. We discuss them in more detail in Section 6, but the intuition is that in those cases the notion of "preceding revision" as provided by the repository becomes imprecise. Hence, our algorithm, as any algorithm

relying on the repository to identify incremental changes, may be misled and terminate the search earlier than it is necessary.

4 The Method Details

4.1 Step 1: Link Vulnerability Claim to Bug Identifier

We need to establish a trace from a vulnerability entry (*e.g.,* CVE) to the code base in order to carry out the assessment. Normally, a vulnerability entry provides only a general description of a security flaw and its impact. Therefore, we establish the code trace by passing through the security bug identifiers (IDs) responsible for a vulnerability entry as reported in the technical bug reports maintained by the software vendor.

To obtain these security bug IDs, we look at the details of this entry, which might have references to its responsible security bug IDs. Alternatively, we can also look at the manufactures' security advisories, which might report references to other vulnerability reports as well as references to the responsible security bug IDs.

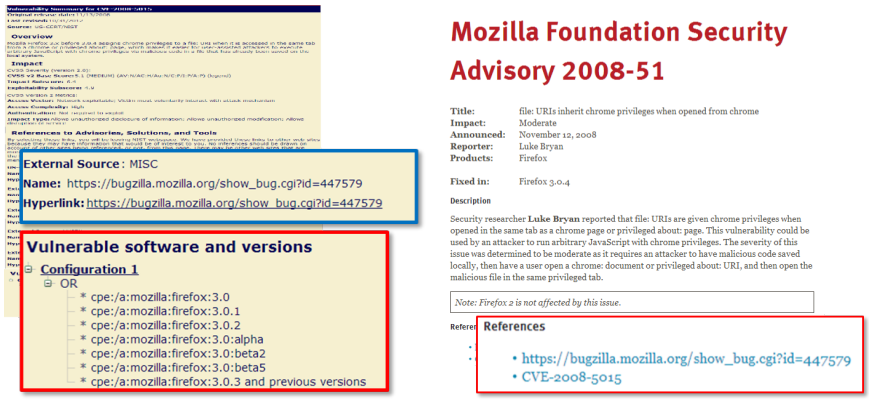
Example 1 Fig. 3(a) shows an example where the security bug IDs responsible for CVE-2008-5015 are reported in the *references* feature of the CVE entry. It is a URL to the report of a security bug ID 447579 of Firefox. For Firefox, bug report hyperlinks usually have two forms: `https://bugzilla.mozilla.org/show_bug.cgi?id=`n (single bug), or `https://bugzilla.mozilla.org/bug_list.cgi?=`n,...,n (bug list). For Chrome, the hyperlink to a bug report is usually `http://code.google.com/p/chromium/issues/detail?id=`n, where n is an integer number indicating the bug ID.

Fig. 3(b) illustrates another example where the CVE and its responsible security bug are reported by manufactures' advisory reports, *e.g.,* Mozilla Foundation Security Advisories (MFSA) for Firefox. The figure shows a snapshot of an MFSA entry: MFSA-2008-51. The *References* section of this MFSA reports two hyperlinks: one for a security bug ID (`https://bugzilla.mozilla.org/show_bug.cgi?id=447579`), and another one for a CVE (`CVE-2008-5015`). So we heuristically assume that this security bug is responsible for that CVE.

■

4.2 Step 2: Locate Bug-Fix Commit

This step takes the list of vulnerability entries and their responsible security bug IDs determined in the previous step to locate their corresponding bug-fix commits in the code base. We are using two popular techniques for locating bug-fix commits: *repository mining* and *bug-report mining*.



(a) Report detail for CVE-2008-5015 (b) Report detail for MFSA-2008-51

Fig. 3 Selected features from a CVE (a), and from an MFSA report (b). The security bug IDs relevant to a CVE are reported either in the *references* feature of the CVE entry, or by manufacturers' advisory reports (*e.g.,* MFSA for Firefox).

- The *repository mining* technique was introduced by Sliwerski *et al.* (2005) and adopted by many other studies in the literature *e.g.,* (Neuhaus *et al.* 2007; Shin *et al.* 2011). This technique parses commit logs of the code base repository for the security bug IDs according to predefined patterns. In our case, the commit logs mentioning security bug ID(s) are bug-fix commits.
- The *bug-report mining* technique, adopted by Chowdhury *et al.* (2011), parses a security bug report for bug-fix information. Such information includes links to bug-fix commits (*i.e.* the identifier of the commits).

Example 2 Fig. 4 depicts two bug-fix commits for Chrome and Firefox. For each bug-fix commit, we highlight information about the revision ID of the commit, the list of changed source files, and the ID of the bug fixed by this commit.

■

The aforementioned mining techniques complement each other since the advantage of one technique is the disadvantage of the other one. The *repository mining* technique requires access to commit logs which might not be publicly available. It could locate bug-fix commits for undisclosed bug reports (the *advantage*). However, it might skip bug-fix commits which do not mention explicitly security bug IDs, *e.g.,* when merging from external repositories (the *disadvantage*). On the other hand, the *bug-report mining* technique could parse the bug report for bug-fix commits even if the bug IDs were not mentioned in bug-fix commits (the *advantage*). Yet, it could not locate bug- fix commits for undisclosed bug reports due to the security policy of software vendors (the *disadvantage*).

Occasionally we could not find evidence either in favor or against a vulnerability claim. This could happen because *1)* the corresponding vulnerability

Fig. 4 Examples of bug-fix commits for Chrome (a) and Firefox (b).

In Chrome, a bug-fix commit is referring to bug IDs following patterns `BUG=n(,n)` or `BUG=http://crbug.com/n`, where n is the bug ID. In Firefox, bug IDs usually follow keywords such as "bug".

```
r41106 | inferno@chromium.org | 2010-03-10 02:03:43 +0100
(Wed, 10 Mar 2010) | 10 lines
Changed paths:
   M /branches/249/src/chrome/browser/views/login_view.cc
Merge 40708 - This patch fixes [... text truncated for saving space]
BUG=36772
TEST=Try a hostname url longer than 42 chars to see that
it wraps correctly and wraps to the next line.
```

(a) A Chrome bug-fix commit

```
changeset: 127761:88cee54b26e0
author: Justin Lebar <justin.lebar@gmail.com>
date: 2013-01-07 09:44 +0100
files: +|-|*dom/ipc/ProcessPriorityManager.cpp
desc: Bug 827217 - Fix null-pointer crash with webgl.can-
lose-context-in-foreground=false.
```

(b) A Firefox bug-fix commit

entry does not have any responsible security bug ID (incompleteness of the data source), or *2)* the bug-fix commit for the responsible bug ID cannot be located (incompleteness of the mining techniques).

In this step, we rely on the consistency of the development process of software manufactures and make use of the following assumptions:

ASS2 *Developers either mention the bug ID(s) responsible for a vulnerability in the description of the commit that contains the bug fix, or mention the commit ID that fixes the bug in the bug's report.*

ASS3 *A bug-fix commit is fixing a single bug.*

By making the above assumptions, we might face with a number of threats to validity. Indeed, a difficulty faced in Sliwerski *et al.* (2005) was to introduce appropriate heuristics to track bugs when these assumptions are violated. Antoniol *et al.* (2008) and Bird *et al.* (2009) discussed biases which violate **ASS2**: developers do not mention bug ID(s) in a bug-fix commit; and, developers mention bug ID(s) in a non-bug-fix commit. The former bias makes some vulnerabilities *unverifiable*. We know they have been fixed but do not know how. We could consider the entire code base as responsible, but this would be too large a footprint. The latter bias might introduce more false positives in our method, but it is still acceptable from a compliance perspective.

When performing an actual experiment one must check whether **ASS2** is reasonably justified by the data. In our analysis, we verified this assumption in Section 5.2 for both Chrome and Firefox.

Example 3 Bug-fixes for the WebKit module (the HTML renderer) in Chrome usually do not contain the bug IDs because they are merged from another repository.

■

Assumption **ASS3** is also important because if a bug-fix commit contains changes for fixing multiple bugs (*i.e. multiple-bug-fix commit*), we cannot distinguish which lines of code (LoCs) were touched to fix which bugs. So we should consider all changed lines as responsible for each individual vulnerability. This is a gross, albeit conservative, overestimation. If the number of multiple-bug-fix commit is too high, this will call into question the precision of the outcome: a large number of spurious claims will eschew detection.

In order to check whether **ASS3** is violated, one must check the distribution of numbers of bugs fixed per bug-fix commit, and check that multiple-bug-fix commits do not take a significant fraction of the total commits. We have verified that this threat was not applicable to our experimental data: 99% (for Chrome) and 91% (for Firefox) of the total bug-fix commits were single bug-fix (see Fig. 7 in Section 5.2).

4.3 Step 3: Identify Vulnerable Code Footprint

This step takes bug-fix commits and annotates source files in the commits to determine vulnerable code footprints. Let r_{fix} be the revision ID of a bug-fix commit. We compare every file f changed in this revision to its immediate preceding revision[4]. We employ the `diff` command supported by the repository to do the comparison. By parsing the comparison output, we can identify the vulnerable code footprints. We ignore formatting changes such as empty lines, or lines which contain only punctuation marks. Such changes occur frequently in all source files, and therefore they do not characterize changes to fix security bugs.

Example 4 Fig. 5(a) illustrates an excerpt of `diff` command applied to revisions $r95730$ and $r95731$ of the file `url_fixer_upper.cc`. The output is shown in the *Unify Diff* format where changes are organized in "hunks".

■

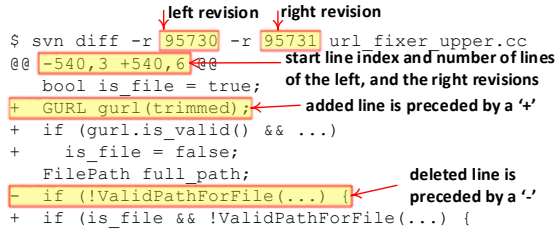
Example 4 shows a revision comparison "hunk" that begins with a header which is surrounded by double at-signs (@@). It contains the start line index and total number of lines in compared revisions. Added and deleted lines are respectively preceded by a plus sign and a minus sign. The vulnerable code footprint in this example is represented by changed and deleted lines(*e.g.,* line #542). The added lines are not considered because they were added precisely for *fixing* a bug.

Next, we identify the origin of vulnerable code footprints *i.e.* the revision where the potential bad code is introduced. Such origins are achieved by annotating the source file f at immediately preceding revision of the bug-fix commit. In an annotation of a source file, every individual line is annotated

[4] In an SVN repository, the immediately preceding of the revision r_{fix} is $r_{fix} - 1$. In some other repository such as Mercurial, the immediately preceding revision is determined by performing the command `parent`

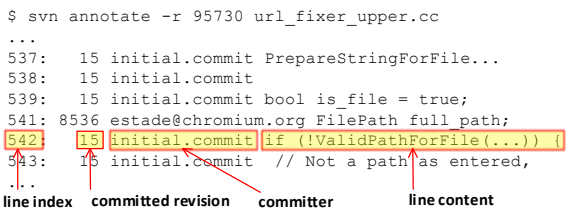
Fig. 5 Excerpts of the output of the `diff` command (a), and of the output of `annotate` (b) command.

For every line of code, the *annotate* function provides the original revision where the vulnerable code footprint was first inserted.




```
$ svn diff -r 95730 -r 95731 url_fixer_upper.cc
@@ -540,3 +540,6 @@
   bool is_file = true;
+  GURL gurl(trimmed);
+  if (gurl.is_valid() && ...)
+    is_file = false;
   FilePath full_path;
-  if (!ValidPathForFile(...)) {
+  if (is_file && !ValidPathForFile(...)) {
```

(a) An excerpt of `diff`




```
$ svn annotate -r 95730 url_fixer_upper.cc
...
537:    15 initial.commit PrepareStringForFile...
538:    15 initial.commit
539:    15 initial.commit bool is_file = true;
541: 8536 estade@chromium.org FilePath full_path;
542:    15 initial.commit if (!ValidPathForFile(...)) {
543:    15 initial.commit  // Not a path as entered,
...
```

(b) An excerpt of `annotate`

with meta-information such as the origin revision, the committed time, and the author. The annotation is done by the `annotate` command of the repository

Example 5 Fig. 5(b) shows that the line #542 in the file `url_fixer_upper.cc` is the vulnerable code footprint from revision $r95730$ that has been originally introduced in revision $r15$.

■

4.4 Step 4: Determine the Validity of Vulnerability Claim

This step scans through the code base of all versions claimed to be vulnerable for the existence of vulnerable code footprints. Such existence is the supported evidence for the claim that a version is actually vulnerable to a vulnerability. A special case is present when a CVE was fixed by only adding new code: no vulnerable code footprint is detected, and we assume conservatively that all original LoCs of the file where a vulnerability is fixed are all vulnerable code footprints. If a version contains some LoCs of the file, then it is considered vulnerable. This solution might not introduce false negatives, but may let false positives survive. From the compliance perspective, this is acceptable since the version has been already claimed to be vulnerable.

Example 6 Table 2 shows the the result of processing three distinct vulnerability claims for Chrome with our approach. The corresponding CVE entries claim that different version ranges are affected by the corresponding vulnerabilities. In case of CVE-2011-2822 and CVE-2011-4080, the approach identified the vulnerable code footprints, that showed the earliest versions where a security

Table 2 The example of three Chrome vulnerabilities processed by our approach.

CVE	Reported Versions	Bug ID	Bug-fix Commits	Footprint	Versions w/ Evidence
2011-2822	v1–v13	72492	url_fixer_upper.cc[1] ($r95731$)	$\langle r15, 542\rangle$	v1–v13
2011-4080	v1–v8	68115	media_bench.cc[2] ($r70413$)	$\langle r26072, 352\rangle$, $\langle r53193, 353\rangle$	v3–v8
2012-1521	v1–v18	117110	–	–	–

[1]: `chrome/browser/net/url_fixer_upper.cc` [2]: `media/tools/media_bench/media_bench.cc`

bug was originally introduced (v1 and v3 respectively). For CVE-2011-4080 we were unable to find the footprint and determine the appropriate vulnerable version, because this is in fact a vulnerability in WebKit (a third-party library used in Chrome). Since the bug-fix commit for that vulnerability is not present in Chrome repository, our current set up will be unable to locate it. We consider such vulnerabilities to be *unverifiable* as well.

■

Considering the above example, we also use the following assumption:

ASS4 *A vulnerability claim about a software version is **evidence-supported** if the version code base contains at least a single line from the vulnerable code footprint of the vulnerability. Otherwise, a vulnerability claim is spurious.*

Changed LoCs are not necessarily vulnerable, albeit they might have helped to remove the vulnerability. For example, a vulnerability that could lead to SQL injection attacks could be fixed by inserting a sanitizer around a user's input in another module. We will then consider all versions where this sanitizer is missing as vulnerable. This is a conservative assumption; for example, previous versions might have used another module for input and the older module had proper input sanitization. So we would consider as vulnerable a version that was not so. This assumption is acceptable in our context: minimizing false negatives, while accepting false positives (vulnerability claims for software that is not vulnerable).

Example 7 This example illustrates when the strategy *"X and previous versions are vulnerable"* was applied. The description of CVE-2012-4185 states:

> *"Buffer overflow in the nsCharTraits::length function in **Firefox before 16.0** allows remote attackers to execute arbitrary code or cause a denial of service (heap memory corruption) via unspecified vectors."*

By applying the above strategy, all versions from v1.0 – v15.0 are considered to be vulnerable. The corresponding bug ID for this CVE is 785753. This bug indicated that it was a global-buffer-overflow in the `nsCharTraits::length` function. This bug was fixed by changing the file `netwerk/base/src/nsUnicharStreamLoader.cpp` as follows.

```
222:     if (NS_FAILED(rv)) {
223: -      NS_ASSERTION(0 < capacity - haveRead,
224: -                   "Decoder returned an error but filled the output buffer! "
225: -                   "Should not happen.");
226: +      if (haveRead >= capacity) {
227: +        // Make room for writing the 0xFFFD below (bug 785753).
228: +        if (!self->mBuffer.SetCapacity(haveRead + 1, fallible_t())) {
229: +          return NS_ERROR_OUT_OF_MEMORY;
230: +        }
231: +      }
232:        self->mBuffer.BeginWriting()[haveRead++] = 0xFFFD;
233:        ++consumed;
```

The plus (+) shows the added lines, and the minus (-) indicates deleted lines. In this particular case, the buffer overflow vulnerability is happening at the line #232, but incomplete condition checks (lines #223–#225) are making this vulnerability actually exploitable. The fix was produced by changing the incomplete check into a better one (lines #226–#231), however the statement that may potentially cause a buffer overflow is still there. We scanned through the code base of Firefox (v15 and downward) to find the deleted lines that would show in which version the vulnerability was initially introduced. We found that the deletion evidence appeared in v6.0 for the first time and propagated until v15.0. The line #232, where the actual vulnerability occurs, was also introduced together with the deletion evidence in v6.0, therefore we can clearly see that versions from v5.0 and downward are not vulnerable.

We must point out that there is also a possibility of getting false negative errors in similar scenarios: if the line #232 has been introduced earlier than the deletion evidence, the approach could get a false negative error. For example, this vulnerability could be fixed by changing the condition on lines #223–#225 (and not touching the line #232), the approach would not be able to correctly identify the original version, because a part of the code that is related to a vulnerability is not a part of the deletion evidence (a fix commit).

Line #232 could possibly lead to even bigger problems in the future if the developers did not check what happened and how the returned value was handled. However, this would be a different vulnerability claim (i.e. a different CVE entry). ■

In Section 6, we perform a manual validation of the approach in order to observe the impact of such scenarios on the results and find other potential weak spots that might also lead to false negatives.

5 Empirical Validation

We apply the proposed method to assess the trustworthiness of vulnerability claims by NVD on Chrome and Firefox. For Firefox we obtain bug-fix commits by using the repository mining technique because it discovered bug-fix commits for most security bugs. For Chrome, we also use the repository mining technique for undisclosed security bug reports, and the bug-report mining

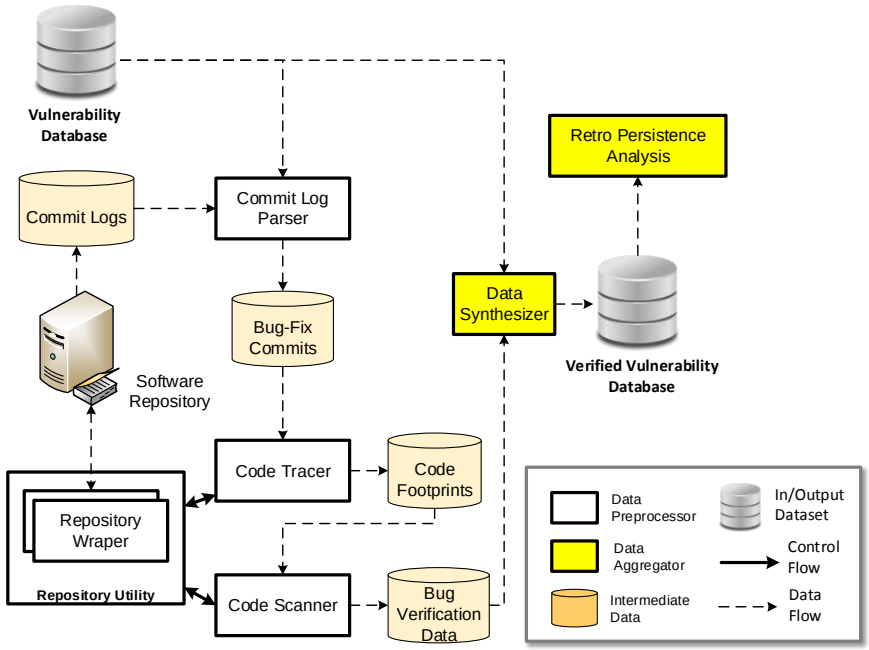


Fig. 6 The software infrastructure for assessing the vulnerabilities retro persistence.

technique for security bugs fixed in imported projects (*e.g.,* WebKit). These security bugs, as discussed, cannot be located by mining the repository as their bug IDs are not mentioned in the bug-fix commits.

5.1 Software Infrastructure

Fig. 6 presents the software infrastructure used for assessing the retro persistence of vulnerabilities. Rectangles denote data preprocessors (white) and data aggregators (yellow/gray); cans represent intermediate data repositories; and stacks of cans indicate input and output data sets. The control and data flows are respectively solid and dashed arrows. The infrastructure body includes the following scripts:

- *Commit Log Parser*. This script takes the commit logs of the source code repositories, and vulnerability data as inputs to produce the *Bug-Fix Commit* data set.
- *Code Tracer*. This script consumes the *Bug-Fix Commit* data set to produce the *Code Footprint* data set, which maintains the links between a security bug ID to particular LoCs that potentially contribute to the existence of this bug. For this purpose, this script search for the original version of the modified LoCs of the source components in *Bug-Fix Commit*. This is done

via the *Repository Utility* scripts that wrap basic commands provided by the software repository, such as `diff, annotate`.
- *Code Scanner*. This script scans LoCs in *Code Footprint* in the code bases of all versions of a software to determine the earliest version affected by a security bug. This is also done with the aid of the *Repository Utility* scripts. The outcome of this script is the *Bug Verification Data* data set.
- *Data Synthesizer*. This script aggregates vulnerability data plus the *Bug Verification Data* data set to produce the *Verified Vulnerability Database* data set. This data set is basically the same as the input *Vulnerability Database* plus extra information about the earliest version affected by each vulnerability. The aggregated data set is then consumed by analysis scripts to produce desired outputs for answering research questions.

We instantiate the framework described in Fig. 6 for the experiment: the repository of Chrome is Subversion (SVN); the repository of Firefox was migrated from Concurrent Version System (CVS) to Mercurial (HG) since version v3.5. Thus we have three different kinds of commit logs: *Firefox CVS Commit Logs*, *Firefox HG Commit Logs*, *Chrome SVN Commit Logs*. These commit logs are text files. So the *Commit Log Parser* script relies on a particular parsing script for each type of commit logs (*i.e. CVS Log Parser*, *HG Log Parser*, and *SVN Log Parser*).

We have analyzed approximately 9,800 vulnerability claims in 7,232 vulnerable files of Chrome and Firefox. We have scanned over 104,798 revisions of these vulnerable files for code evidence. Those revisions were extracted from the repositories of Chrome and Firefox. For this purpose, we made a local copy of Firefox repository, and extracted file revisions from it. However, we could not do the similar thing for Chrome. Instead, we had to use the public repository of Chrome for extracting file revisions during the experiment.

The implementation scripts of the method take approximately 14 days (*i.e.* 336 hours) for an automatic process on a 2 x quad-core 2.83GHz Linux machine with 4GB of RAM.

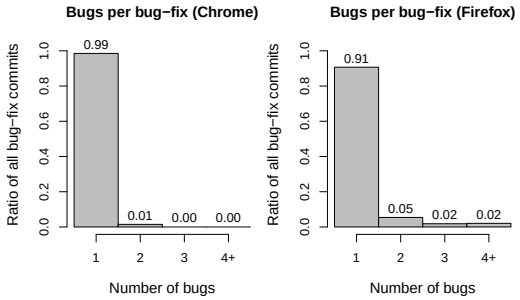
5.2 Descriptive Statistics

Table 3 presents the numbers of CVEs affecting Chrome and Firefox by April 2013. These CVEs make a total of 9,800 vulnerability claims of 18 major versions of Chrome (v1–v18), and 15 major versions of Firefox (v1.0–v12.0). The overwhelming majority of of them have at least one responsible security bug ID: 94.8% of Chrome CVEs and 83.3% of Firefox CVEs.

As discussed in Step 2, we investigate the distribution of fixed bugs per bug-fix commit to understand the unwanted effect of multiple-bug-fix commits that potentially biases the experiment's outcomes. Fig. 7 shows these distributions for Chrome (left) and Firefox (right). In almost all cases for Chrome and in over 90% of cases for Firefox, a bug-fix commit only contain fixes for one bug. The developers of both Chrome and Firefox eventually fixed each security bug individually, and committed the bug-fix to the repository when a bug was

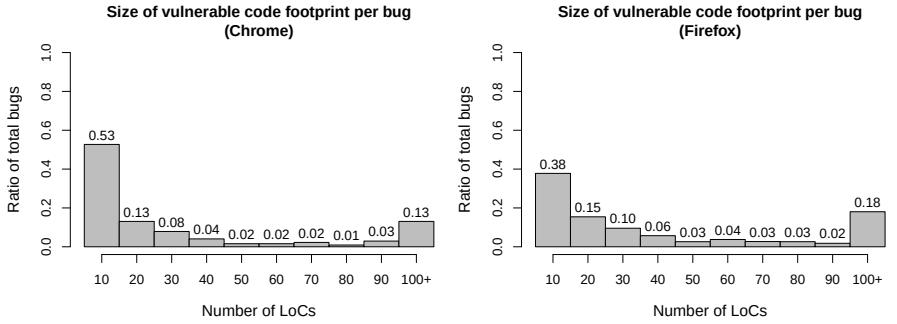
Table 3 Descriptive statistics for vulnerabilities of Chrome and Firefox

	Number of CVEs			Total
	verifiable	w/o resp. bugs	w/o bug-fix	
Chrome	554 (72.1%)	40(5.2%)	174(22.7%)	**768**
Firefox	681 (77.7%)	146(16.7%)	49(5.6%)	**876**



(a) Chrome (b) Firefox

Fig. 7 The number of security bugs per bug-fix commit in Chrome (a) and in Firefox (b).



Vulnerable code footprint is a set of LoC which are changed or removed to fix a security bug. Small footprints with less than 50 LoCs were the overwhelming majority of security bug-fixes (over 70%). When the ratio reaches 1% it is equivalent to 6 bugs for Chrome, and 14 bugs for Firefox. The percentage of footprints whose size is greater than 90 LoCs is the aggregation of a very long tail of few bugs.

Fig. 8 The size of vulnerable code footprint of individual bugs of Chrome and Firefox.

fixed. This consistent behavior makes the unwanted effect of multiple-bug-fix commit negligible for the purposes of our experiment.

We further investigate the distribution of sizes of vulnerable code footprints. The intuition behind this investigation is that a small size of vulnerable code footprint will generate less error in the experimental outcome.

Example 8 Fig. 8 reports the size of vulnerable code footprints of individual security bugs for Chrome (left) and Firefox (right). In Chrome, 53% of vulner-

ability code footprints have 10 LoCs (or less), 27% included between 10 and 50 LoCs. These numbers for Firefox are 38% and 34%.

■

To put the above data into perspective: on average, the size of vulnerable components of Chrome and Firefox are 754 and 2,939 LoCs, respectively. It means that more than two-third of the fixes affect less than 10% of the lines of code of a vulnerable component. Firefox has 18% (*i.e.* 257 over 1431 bugs) of code footprints with more than 100 LoCs, in contrast Chrome only has 13% (*i.e.* 81 over 613 bugs) of such footprints. We cluster footprints whose size is greater than 90 LoCs to save space, these footprints are distributed in a very long tail of few bugs. A possible explanation is that the average size of vulnerable components of Firefox is nearly four times larger than vulnerable components in Chrome. The small size vulnerable code footprints in Chrome and Firefox helps to reduce the potential bias in our proposed method.

These preliminary data analysis suggests that **ASS2**, **ASS3**, and **ASS4** are not a significant threat to the validity of the proposed method in the cases of Chrome and Firefox.

5.3 Spurious Vulnerability Claims

Table 4 reports the spurious vulnerability claims for individual versions of Chrome and Firefox. On average, each version of Chrome has about 309 vulnerability claims. Of these, approximately 28.6% are unverifiable, and 35.9% are found spurious by our method. The average number of vulnerability claims per each version of Firefox is slightly less than Chrome: about 283. In comparison to Chrome, vulnerability claims per each Firefox version are less unverifiable (10.3%), and slightly less spurious (31.5%). The number of unverifiable vulnerability claims in Chrome v18 is significantly higher than in other versions (5.7% without responsible bugs, 48.6% without bug- fix commits). This happens because the number of vulnerability claims reported at the data collection time is small. Out of 19 unverifiable entries, 2 have no responsible bugs, and 10 are from a separated repository (WebKit).

Table 4 shows that approximately 25% of Chrome and 10% of Firefox claims are unverifiable. This could be due to some numerous reasons as discussed in (Antoniol *et al.* 2008). In our opinion the major explanation is that Chrome developers made use of external libraries (*e.g.,* WebKit, Java Script Engine V8). Information about bug fixes in these libraries is not necessarily reported in Chrome's repository. To minimize the impact of these phenomena, we assume that these unverifiable claims are not spurious (to avoid false negative). We define the *error rate* (ER) as the ratio of the spurious vulnerability claims to the total ones.

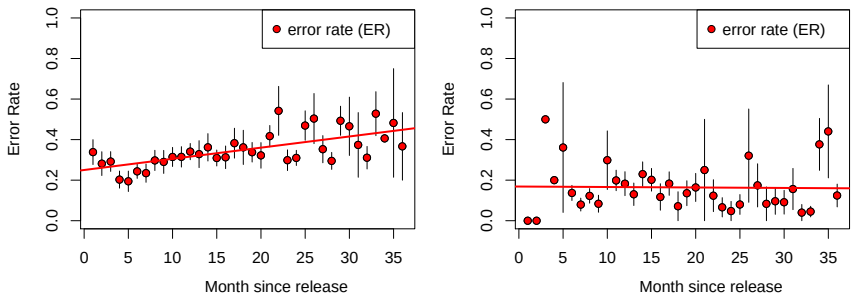
$$ER(v) = \frac{|spurious(v)|}{|vulnerabilities(v)|} \quad (1)$$

Table 4 Distribution of vulnerability claims for Chrome and Firefox.

		Verifiable (*e.g.*, vulns. within a code base)		Unverifiable (*e.g.*, vulns. in third-party libraries)	
Release	Total	w/ evidence	spurious	w/o resp. bug	w/o bug-fix
Chrome v1	526	34.8%	37.3%	5.9%	22.1%
Chrome v2	512	36.1%	37.3%	3.9%	22.7%
Chrome v3	504	38.7%	34.9%	3.6%	22.8%
Chrome v4	498	43.0%	31.1%	3.4%	22.5%
Chrome v5	455	38.9%	37.8%	1.1%	22.2%
Chrome v6	405	43.0%	31.9%	1.2%	24.0%
Chrome v7	391	42.2%	32.0%	1.3%	24.6%
Chrome v8	371	43.7%	30.5%	1.3%	24.5%
Chrome v9	337	42.7%	29.4%	1.5%	26.4%
Chrome v10	304	40.1%	30.6%	1.3%	28.0%
Chrome v11	273	38.1%	34.1%	1.8%	26.0%
Chrome v12	239	38.1%	34.3%	2.1%	25.5%
Chrome v13	217	36.9%	35.9%	1.8%	25.3%
Chrome v14	177	32.2%	42.9%	2.3%	22.6%
Chrome v15	121	38.0%	33.1%	4.1%	24.8%
Chrome v16	112	26.8%	44.6%	3.6%	25.0%
Chrome v17	88	20.5%	47.7%	4.5%	27.3%
Chrome v18	35	5.7%	40.0%	5.7%	48.6%
Mean(std.dev)	**309** (155)	**35.5%** (9.3%)	**35.9%**(5.1%)	**2.8%** (1.5%)	**25.8%** (5.8%)
Firefox v1.0	454	32.6%	48.9%	12.3%	6.2%
Firefox v1.5	461	38.8%	42.3%	12.6%	6.3%
Firefox v2.0	469	45.6%	41.8%	7.0%	5.5%
Firefox v3.0	388	45.1%	43.8%	4.6%	6.4%
Firefox v3.5	336	53.0%	37.5%	3.0%	6.5%
Firefox v3.6	310	52.3%	37.4%	2.9%	7.4%
Firefox v4.0	253	62.8%	28.1%	0.8%	8.3%
Firefox v5.0	230	67.4%	24.3%	0.0%	8.3%
Firefox v6.0	220	68.2%	24.1%	0.0%	7.7%
Firefox v7.0	210	68.1%	24.3%	0.0%	7.6%
Firefox v8.0	203	66.5%	25.1%	0.5%	7.9%
Firefox v9.0	196	66.8%	25.5%	0.0%	7.7%
Firefox v10.0	176	67.0%	25.0%	0.0%	8.0%
Firefox v11.0	175	70.3%	21.7%	0.0%	8.0%
Firefox v12.0	165	69.7%	21.8%	0.0%	8.5%
Mean(std.dev)	**283** (108)	**58.3%** (12.2%)	**31.4%** (9.0%)	**2.9%** (4.3%)	**7.4%** (0.9%)

where v is a version; $|spurious(v)|$, and $|vulnerabilities(v)|$ are the number of spurious, and total vulnerabilities of v. Given our inclusion of unverifiable vulnerabilities in the total count, the error rate is a lower bound of the actual value.

Fig. 9 illustrates the evolution of the ER along the number of months since the release date of individual versions, for Chrome, and Firefox. Circles in the figure indicate the mean values; and the vertical bars show the standard errors. The lines interpolate the global trends. For Chrome, Fig. 9(a), the magnitude of the ER mean is mostly around 0.2 to 0.4. The error bars are quite small



(a) Chrome (b) Firefox

Version age is the number of months since the release of a particular version. A red (dark) circle is the average ER of all versions in their particular age. The bars are the standard errors.

Fig. 9 The average error rates (with error bars) of Chrome (a) and Firefox (b) along the lifetime of individual versions.

in most of the cases, which means that there is less variance among ER of the individual versions, and the number of spurious vulnerability claims in Chrome is remarkable. For Firefox, Fig. 9(b), the magnitude of the ER mean is smaller than Chrome's, mostly from 0 to 0.2 with a few outliners. The small error bars presented in most cases also indicate the small variance of ER of individual Firefox versions.

We only report the trend of the error rates for the first 36 months of each version because the interval between two consecutive major versions is quite short for recent versions of Firefox (Wikipedia 2013) and all versions of Chromes (Chromium Developers 2013).

Clearly the vulnerability claims of Firefox are more reliable than those of Chrome since the ratio of spurious vulnerability claims for Firefox is smaller than Chrome's. Moreover, the error rate of Firefox tends to decrease over time, whereas the error rate of Chrome tends to move up, so that vulnerability claims for older versions are less reliable.

The different trends could be potentially explained by the process that generates vulnerability claims. According to an archive document[5], the information reported in the "vulnerable version" feature is *"obtained from various public and private sources. Much of this information is obtained (with permission) from CERT, Security Focus and ISS/X-Force"*. Our private communications with the NVD team and and software vendors have revealed an inconsistency: the National Institute of Standards & Technology (NIST) claimed vulnerable versions were taken from software vendors (NIST 2012); whereas, software vendors claimed they did not know about this information (Mozilla Security 2011). In another conversation, the NVD team said they do not perform any tests to determine which versions are affected by which CVEs. The vulnerabil-

[5] This page has been removed, but can be accessed by URL http://web.archive.org/web/20021201184650/http://icat.nist.gov/icat_documentation.htm

ity claims are derived from the CVE description by MITRE (www.mitre.org), release notes by software vendors, and additional data by third-party security researchers. Therefore, to ensure completeness, all versions before a version X are claimed vulnerable to a CVE if its description says something like: *"version X and before"* or *"X and 'previous' versions"*. Apparently, in this case the NVD analysis team received better information for Firefox than for Chrome. This may be due by the fact that Firefox has a proper security advisory, while Chrome has none.

6 Independent manual validation

In this section we present the results of the independent manual validation process for our vulnerability assessment approach. Below we describe the process itself and discuss the results.

We verified only the results obtained from the Firefox repository because (1) their bug tracking system is publicly available, so it is easier to understand each particular fix; (2) the whole repository can be downloaded and analyzed locally; (3) to avoid potential bias by mixing results obtained from different repositories that are present in Chrome.

We manually verified whether each vulnerability claim from this sample is spurious and compared the outcome with the results of the automatic approach. Additionally, we carefully checked whether the approach identified all key revisions correctly. It has already been shown that the manual inspection of software repositories can take a long time. For instance, Meneely et al. (Meneely *et al.* 2013) mention that it took them hundreds of man-hours over several months to manually collect and check the data on vulnerable-contributing code commits. We also found out that the manual labor behind the analysis of source code repositories and bug tracking systems is very time consuming. It took us around two weeks to manually inspect the selected sample of vulnerabilities, notwithstanding that we already had the information about the approximate location of vulnerable and fixed revisions.

We selected a random sample of 80 vulnerabilities for the manual processing, and divided the results into the following categories:

1. *True positives*: a vulnerability claim was correctly identified as *not spurious* (i.e. the approach marks versions as vulnerable, and they indeed are vulnerable);
2. *False positives:* a certain version was identified as vulnerable by the approach, but the manual inspection showed that it is not vulnerable (we consider this case as acceptable);
3. *True negatives:* a vulnerability claim was correctly identified as *spurious* (i.e. the approach marks versions as not vulnerable, and they are indeed not vulnerable).
4. *False negatives:* a vulnerability claim was incorrectly marked as *spurious* (i.e. the approach marks versions as not vulnerable, but the manual inspection shows that they are vulnerable).

We found out 35 true positives, 13 false positives, 29 true negatives, 6 false negatives due to changes in the repository structure, and 1 false negative due to a code reversal. Both cases violate the assumption that changes are incremental between revision *N*, revision *N-1*, and revision *N-2*, etc.

Example 9 The found false negative is a realization of the following scenario:

1. A developer takes a revision of the code $rev1$ and makes some changes to add new features to the program - a revision $rev2$ is produced.
2. Along with these new features, a developer introduces a security bug.
3. Later on, the development team decides that a proper security fix requires reverting all changes made by the developer to the original revision $rev1$. A "new" revision $rev3$, actually equivalent to $rev1$, will be created.
4. Later on there could be another incorrect change by another developer that will re-introduce the same code and the same vulnerability. This generates another "new" revision $rev4$.
5. To fix the bug the relevant part of the code will be again reverted to its initial state from $rev1$, creating yet another revision $rev5$.

■

After the third step, the approach will consider the deletion evidence from $rev2$, and correctly identify $rev2$ as the earliest vulnerable revision, and $rev3$ as the corresponding fix. The revision $rev1$ will be discarded, because it does not contain any deletion evidence from $rev2$.

After the second reversal at step 4, our approach will be able to identify only $rev4$ as vulnerable. All earlier revisions, including vulnerable $rev2$, will be discarded because there will be no deletion evidence that lasts from $rev3$ to $rev4$. If revisions $rev2$ and $rev4$ belong to different released versions of a program, i.e. v1.0 and v2.0, the approach will make a false negative error by concluding that only v2.0 is vulnerable. This scenario makes inaccurate the notion of "preceding revision" as reported from the history commands of the repository. Indeed, from a syntactic view point both $rev4 \approx rev2$ are "preceding revisions" of $rev3 \approx rev1$. However, our algorithm will look at the temporal history as provided by the repository. It is possible to change the algorithm, so that it will look in the past for the evidence of code reversions, but it is not clear whether the added complication is worth the effort to cover such corner cases. We believe such scenario of code reversal is very rare (as it is an example of truly bad programming) and a longer manual analysis will likely reveal its "black swan"[6] nature.

The Firefox source code repository was migrated from CVS subversion system to Mercurial. The migration process caused false negative errors on the boundary versions of Firefox. For instance, the approach may correctly match specific revisions to specific versions until Firefox 3.6, but there may be difficulties for versions 3.5 and below. We therefore eliminate from consideration

[6] Unexpected and extremely rare events hat can have major impact but can be only explained in retrospection (Taleb 2010)

the 6 false negatives due to repository migration as they can be eliminated by a systematic mapping between the revisions in the repositories.

To calculate the error rate of the procedure for false negatives we used Agresti and Coull's score confidence interval (Agresti *et al.* 2012), which requires to solve for p the following formula:

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

where p - is the proportion estimate of false negatives; $\hat{p}$ - is the sample size proportion of false negatives over the total sample of negatives n; and $z = 1.96$ is the critical coefficient for a 95% confidence interval.

If we limit the analysis to the population of negatives (TN=29,FN=1) we have a potential error rate between 8.3% and 13.8% within a 95% confidence interval. If we compare the overall population of results (Acceptable=73,Not Acceptable=1) we have margin of error between 1.3% and 3.9%.

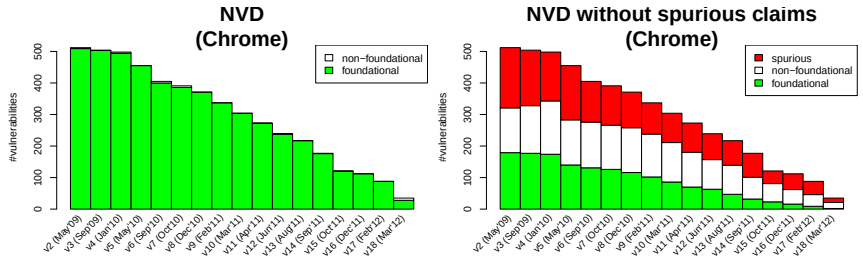
Another scenario that we have not found but could potentially be present takes place when a vulnerability consists of two components: the vulnerable code itself, and some additional code that makes it exploitable. To fix the problem, a developer may issue a quick patch that changes only that additional code, making the vulnerability unexploitable. Later on, another developer changes the code and occasionally uncovers the path to vulnerability. Further, this vulnerability will be re-discovered again and fixed. Our approach will be able to find only the latest "vulnerable - fixed" revision pair, since there is no deletion evidence that can be tracked in between (this is similar to the case described in Example 9). If these pairs belong to different versions of a program, the approach will yield a false negative error.

7 The impact of Spurious Vulnerability Claims

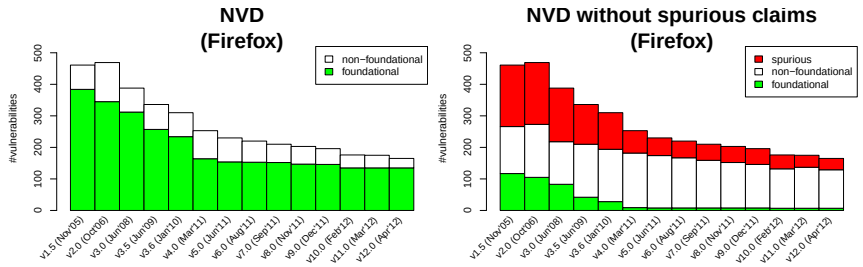
In summary, both the automatic and the manual experiment has provided evidence for the non-negligible error in the vulnerability claims made by NVD to retrospective versions of Chrome and Firefox.

This section investigates the potential impact of spurious vulnerability claims (**RQ2**) to scientific or compliance analyses based on such data, and in particular the analysis of foundational vulnerabilities. We specify a number of hypotheses as they are derived from the literature. We then proceed to verify them empirically by using two datasets: (1) the original NVD dataset; (2) the NVD dataset, where the spurious vulnerability claims have been eliminated. For each dataset we run a statistical test to check whether a hypothesis is supported. If the results are different, then we have highlighted a problem: the usage of a wrong dataset may lead to a result that might be "statistically significant" but "practically wrong".

A foundational vulnerability is a vulnerability which is introduced in the very first version (*i.e.* v1.0), but continues to survive in later versions. We assess the claim suggested by Ozment *et al.* (2006) for other type of software:



(a) Chrome foundational vulnerabilities



(b) Firefox foundational vulnerabilities

The dates in the X axis are the release dates.

Fig. 10 Foundational vulnerabilities by NVD: all vulnerability claims (left) and spurious vulnerability claims highlighted (right).

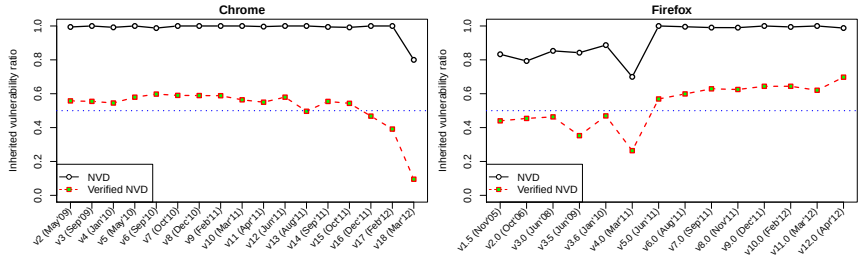
"Foundational vulnerabilities are the majority in each version of both Chrome and Firefox". The claim could be formulated into the following hypothesis.

H3.1$_{A+}$ *More than* 50% *vulnerabilities in a version are foundational.*

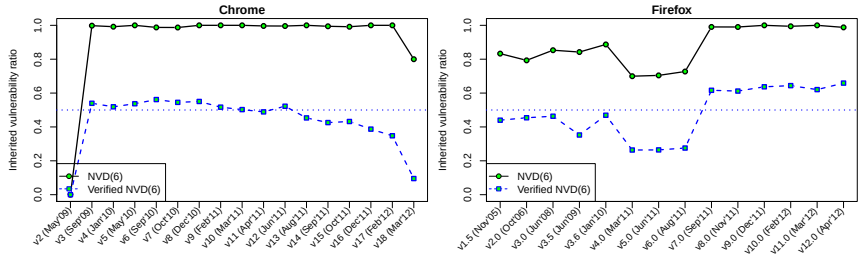
The subscript $A+$ indicates **H3.1**$_{A+}$ is an alternative hypothesis. It means that if the returned *p-value* of a statistic test is less than the significance level 0.05, we could reject the null hypothesis and accept the alternative one.

Fig. 10 reports the distributions of foundational vulnerability claims for Chrome (a) and Firefox (a), reported by both NVD (left) and Verified-NVD (right). In the unverified source, foundational vulnerability claims are dominant, approximately 99% and 75% per each version in Chrome and Firefox, respectively. The verified dataset has a much smaller number of vulnerabilities claims in total, and an even smaller fraction of foundational vulnerabilities.

There is a natural explanation for this phenomenon: modern versions of software are often completely different from the initial one. In the course of time the code has changed. Therefore, foundational vulnerabilities are unlikely to be found in more recent versions. Thus, the claim about the majority of foundational vulnerabilities is eventually false. Therefore we relax the concept of foundation vulnerability to *inherited vulnerability*, as suggested by (Massacci *et al.* 2011). An *inherited vulnerability* of a version X is a vulnerability affecting X and its preceding versions. The claim about the majority of



(a) Inherited vulnerabilities from preceding versions



(b) Inherited vulnerabilities from preceding versions which are at least 6 month older

Circles and squares denote the ratios of inherited vulnerabilities counted from NVD and Verified-NVD data sets, respectively. Inherited vulnerabilities seem to be the totality in NVD but most of them are spurious. Around half of actual vulnerabilities are freshly introduced by developers.

Fig. 11 The ratios of inherited vulnerabilities in Chrome and Firefox.

"foundational" vulnerabilities is relaxed to the claim about the majority of "inherited" vulnerabilities: *"Inherited vulnerabilities are the majority"*.

H3.2$_{A+}$ *More than* 50% *vulnerabilities in version X also exist in version X-1.*

H3.3$_{A+}$ *More than* 50% *vulnerabilities in version X also exist in version Y which is older than X by at least 6 months.*

The hypothesis **H3.2**$_{A+}$ follows exactly the definition of inherited vulnerabilities from (Massacci *et al.* 2011). Meanwhile, the hypothesis **H3.3**$_{A+}$ considers the short-cycle release policy where software vendors try to ship a new version in a relative short period (*e.g.,* less than 2 months per version). Due to this policy, two consecutive versions might not have enough significant difference in their code base, and therefore the hypothesis **H3.2**$_{A+}$ may be just true by default. We assume that two versions which are different apart by 6 month would have significant changes in their code base.

Fig. 11 reports the ratios of inherited vulnerabilities in individual versions of Chrome and Firefox. In Fig. 11(a), we count the inherited vulnerabilities of a version X from all its preceding versions, see **H3.2**$_{A+}$. Meanwhile, in Fig. 11(b), we only count inherited vulnerabilities from preceding versions

Table 5 The Wilcoxon signed-rank test results for the majority of foundational and inherited vulnerabilities.

Numbers in parentheses are *p-values* returned by the Wilcoxon signed-rank test for the corresponding null hypothesis. "Verified-NVD" is the NVD where spurious vulnerability claims identified by our methods are eliminated. Most statistical claims "supported" by the analysis based on the NVD are in reality spurious ones.

	Alternate hypothesis	NVD		Verified-NVD	
		Chrome	Firefox	Chrome	Firefox
H3.1$_{A+}$	More than 50% vulnerabilities in a version are foundational	Accept $(0.01 \cdot 10^{-2})$	Accept $(0.06 \cdot 10^{-3})$	Reject (1.00)	Reject (1.00)
H3.2$_{A+}$	More than 50% vulnerabilities in version X also exist in version X-1	Accept $(0.01 \cdot 10^{-2})$	Accept $(0.05 \cdot 10^{-2})$	Accept (0.03)	Reject (0.16)
H3.3$_{A+}$	More than 50% vulnerabilities in version X also exist in version Y which is older than X by at least 6 months	Accept $(0.01 \cdot 10^{-1})$	Accept $(0.05 \cdot 10^{-2})$	Reject (0.85)	Reject (0.67)

which are at least 6 months older, see **H3.3**$_{A+}$. In this figure, circles and squares denote the ratios of inherited vulnerabilities counted from NVD and Verified-NVD data sets, respectively.

From Fig. 11 we can observe a phenomenon in the inherited vulnerabilities similar to the foundational vulnerabilities in Fig. 10. In the NVD data set, inherited vulnerabilities are dominant in both ways of counting. However, they might be no longer dominant in Verified-NVD data set. There is an anomalous data point in Chrome's: the initial number of inherited vulnerabilities in Fig. 11(b) is 0. It is because the interval between Chrome v1.0 and v2.0 was less than 6 months. By applying the counting method described in **H3.3**$_{A+}$, Chrome v2.0 does not have any inherited vulnerabilities.

We test these hypotheses in two data sets: one for all vulnerability claims by NVD, and another one for vulnerability claims, excluding ones which are found spurious by the proposed method. We shortly refer to the former as *NVD* data set, and refer to the latter as *Verified-NVD* data set. We could use either one-sided t-test or one-sided Wilcoxon signed-rank test. The former tests on the mean, but requires the ratios to be normally distributed. The latter does not require normality, but tests on the median instead. We run the Shapiro-normality test on vulnerability data. It rejects the null hypothesis for Chrome (*i.e.* data is not normal, $p\text{-}value = 0.07 \cdot 10^{-6}$), but not for Firefox (*i.e.* data is normal, $p\text{-}value = 0.95$). Therefore, we use the Wilcoxon signed-rank test to check the hypotheses.

The outcomes of hypothesis tests are reported in Table 5. The reported *p-values* are for rejecting the null hypothesis and accepting the alternative hypothesis when $p\text{-}value < 0.05$. It shows that if we rely on the NVD data set, all hypotheses are accepted with strong evidence (*i.e. p-values* are almost zero). However, if we use the Verified-NVD data set, which is the NVD data set excluding the spurious vulnerability claims, we obtain the opposite conclusions in most cases. In other words, one would be badly mislead to use the NVD to make any inference on the ratio of foundational vulnerabilities (in Chrome and

Firefox). Researchers must check for the existence of spurious vulnerabilities before drawing conclusion from unverified data sources. See (Nguyen 2014) for further discussion on the impact on the trends of vulnerability discovery.

8 Threats to Validity

Construct validity may be affected by the means we use to collect and verify vulnerabilities:

- *Bias in bug-to-CVE linking scheme.* While collecting data for Firefox, we employ heuristic rules to link a security bug ID to a CVE based on their relative positions in an MFSA report (Massacci *et al.* 2010). We manually checked many links for the relevant connection. All checked links were found to be consistent with the result of the automatic algorithm.
- *Bias in bug-fix commit data.* There are two potential biases on the bug-fix commit data (Antoniol *et al.* 2008; Bird *et al.* 2009): the developers do not mention the bug ID in a bug-fix commit; and the developers mention a bug ID in a non-bug-fix commit. To evaluate the impact of the latter bias, we performed a qualitative analysis on some bug-fix commits and found that all are actually bug fixes. This confirms the finding in (Bird *et al.* 2009) for this type of bias. As for the former bias, we check the completeness of the bug-fix commits for vulnerabilities. As discussed, about one fourth of vulnerabilities are unverifiable (see also Table 3). We conservatively assumed that these vulnerabilities are not spurious.
- *Bias in the history of commits.* Branches are usually used for maintenance (e.g. working on a single bug in isolation), and they are eventually merged back into the trunk, thus, from the historical perspective, these are changes to the trunk. Therefore, if branches are properly merged, the history indeed looks linear. Our algorithm can also be used to analyze branches that are not merged. One issue in our current implementation that makes history reconstruction difficult is the migration to a new repository. Indeed, as we discussed in Section 6, six out of seven false negatives were present due to Firefox's migration from CVS to Mercurial.
- *Bias in the assessment method.* The method assumption **ASS4** is syntactical and might not cover all cases of bug fixes since it is extremely hard to automatically understand the root cause of vulnerabilities. Therefore, the method classifies an NVD-reported vulnerability of a particular version as correct while it may be a false-positive. Again it only makes our conclusion a lower bound of the real errors.
- *Bias in the method implementation.* It is possible that the implementation of the assessment method has some bugs, causing bias in the identification of vulnerability evidence. To minimize such problem, we employ a multi-round test-and-fix approach where we ran the program on some vulnerabilities, then we manually checked the output, and. fixed found bugs. We repeated this until no bug was found. Finally, we randomly checked the output again to ensure that there was no mistake.

Internal validity concerns the causal relationship between the collected data and the conclusion. Our conclusions are based on statistical tests, and we carefully analyzed the assumptions of the tests. For instance, we did not apply any tests with normality assumption since the distribution of vulnerabilities (at least for Chrome) is not normal. We did not create proof-of-concept exploits to verify vulnerable behaviors due to limited resources, instead we have manually checked a subset of vulnerabilities (Section 6) - this may lead to errors induced by human factors and lack of dynamic execution evidence.

External validity is the extent to which our conclusion could be generalized to other applications. We tested the proposed method on two applications of the same kind (web browsers) that share a number other common characteristics: these are large, well-maintained open source applications that are written in C, and are prone to memory corruption vulnerabilities. Some projects with different characteristics might produce different results.

Additionally, projects' maintainers have no obligations, and may have no possibility, to keep their NVD records accurate, so it is fair to expect discrepancies between the actual vulnerable versions and versions stated in corresponding CVE entries. Yet some projects may spend additional effort on making precise NVD records, and overall precision of NVD data may be significantly increased in the future.

In general, the proposed method, which is our main contribution, can be used to replicate the experiment on other types of software.

9 Related Work

Sliwerski *et al.* (2005) proposed a technique that automatically locates fix-inducing changes. This technique first locates changes for bug fixes in the commit log, then determines earlier changes at these locations. These earlier changes are considered as the cause of the later fixes, and are called fix-inducing. This technique has been employed in several studies (Sliwerski *et al.* 2005; Zimmermann *et al.* 2007) to construct bug-fix data sets. However, none of these studies mention how to address bug fixes for which earlier changes could not be determined. These bug fixes were ignored and maybe a source of bias in their work.

Bird *et al.* (2009) studied the bias in bug-fix data set in the code base. The authors have gathered a data set linking bugs and fixes in the code base of five open source projects, and manual checked for biases of bug features in their data set. They have found strong evidence of systematic bias of bug features in their data set. In their data set, the linkage between bugs and bug-fix depended on bug features *e.g.,* bug severity (higher severity bugs have less chance to link to bug-fixes), and experience of fixers (more experience fixers are likely to link bugs to bug-fixes). Such bias might be also existed in other bug-fix data set, and could be a critical problem with any studies relying on bug features in bug-fix data sets.

Antoniol *et al.* (2008) showed another kind of bias that bug-fixes data sets might suffer from. Many issues reported in tracking systems are not actual bug reports, but feature or improvement requests. Therefore, this might lead to inaccurate bug counts. However, such bias rarely happens for security bug reports. Furthermore, Nguyen *et al.* (2010), in an empirical study about bug-fix data sets, showed that the properties of the linkage between bugs and fixes is mostly the results of the software development process, rather than the used technique. Additionally, the linking bias has a stronger effect than the *bug-report-is-not-a-bug* bias.

Meneely *et al.* (2013) studied the properties of vulnerability-contributing commits (VCCs) which are similar to the commits containing vulnerability code footprints. Meneely *et al.* applied a similar method to identify security bug-fix commits, but then they used an ad-hoc approach to identify VCCs. Thus, the identification of VCCs for 68 vulnerabilities of Apache HTTP server took them hundreds of man-hours over six months. Here, we have to deal with thousands of vulnerabilities

10 Conclusion

Vulnerability data sources are not only employed in scientific studies, but also for software compliance assessment. Potential biases in such data sources can be serious validity threats to scientific studies, and costly and unnecessary change requests resulting from compliance assessment. To address these issues, we have proposed a method to identify code evidence for vulnerability claims and validate it experimentally on Firefox and Chrome code bases.

Method. The method is inspired by the work of Sliwerski *et al.* (2005) which aims to identify code inducing fixes for generic bugs, whereas our proposed method focuses on the evidence of the presence of vulnerabilities. Such objectives result in the significant difference between the two methods: the method by Sliwerski *et al.* (2005) tries to reduce false positives, while our proposed method aims to reduce false negatives, and could accept false positives. The proposed method takes a list of vulnerabilities and their corresponding security bug IDs as input. Then it traces the commits in the source code repository for the commits that fixed security bugs. From these commits, it determines the so-called vulnerable code footprints which are changed LoCs to fix security bugs. Then it looks for the origin revision where vulnerable code footprints were introduced. Finally it scans the code base of individual versions to determine the vulnerable status of these versions.

Experiment. We have conducted an experiment applying the proposed method to assess the retro persistence validity of vulnerability claims by NVD for Firefox and Chrome. The experiment has assessed $1,235$ out of $1,644$ CVEs ($\sim 75\%$) in 33 versions of Chrome and Firefox. Performing the analysis takes

around 14 days of automatic processing on 104,798 revisions of 7,236 vulnerable files. It is an evidence that the proposed method is not only effective (*i.e.* the majority of vulnerabilities have been assessed), but also efficient (*i.e.* in comparison with 6 months for 68 vulnerabilities by using an ad-hoc method (Meneely *et al.* 2013)).

The experiment showed that there is a significant presence of spurious vulnerability claims in NVD (30% for Chrome and Firefox). The experiment also showed that if we only rely on the information about vulnerable software by NVD, the spurious claims might significantly mislead our conclusions. In particular, by using an unverified NVD we might wrongly make a "statistically significant" claim about the prevalence of foundational vulnerabilities. Such claim would fall when one removes the spurious vulnerability claims identified by our method.

Limitations. Our method works under the assumptions that *i)* it is possible to trace a bug-fix commit in the code base from a vulnerability entry through the security bulletin or bug tracking system, *ii)* there is prevalently a single bug- fix-per-bug commit entry, and *iii)* a vulnerability gets fixed by removing LoCs (change is removal and addition) or otherwise by only adding new LoCs. An underlying assumption is that changes are incremental and the software repositories keep track of them. These assumptions are mostly satisfied by the code bases we considered, but may not hold in general. If they do not hold, our method will conservatively identify less spurious vulnerability claims that there may actually be.

One important caveat is that our algorithm is purely syntactic in nature. The code footprint may be actually incorrect as it may mix vulnerability related changes and other unrelated changes. The syntactic nature makes our algorithm also sensible to large scale changes (e.g. migration from CVS to Mercurial) in the repository or code reversions as the notion of "preceding revision" becomes imprecise. In both cases our method may introduce false negatives at the boundaries.

Our manual validation of the approach showed that in same cases this might happen. Out of a random sample of 80 manually assessed vulnerabilities in Firefox, 6 false negatives were due to a mis-alignment among different repositories (CVS and Mercurial) and only 1 was found to be an actual false negative (due to a code reversal, i.e,. a piece of code repeatedly eliminated and re-inserted in a component, so that changes were looping). The former type of false negatives can be entirely eliminated by an accurate mapping between repositories, the latter may escape detection. By using a the score confidence interval calculation we can estimate a possibility of error for our method between 1.3% and 3.9% with a 95% confidence interval (between 8.3% and 13.8% if we limit our attention to negative claims). In particular for known repository restructuring, it is possible to overcome such errors by adapting the algorithm to such corner cases.

Since our focus is "vulnerabilities as software bugs" we cannot obviously verify vulnerabilities due to misconfigured applications.

Summary. Our proposed method is the first automatic approach to find code evidence for vulnerability claims in past versions of a software when it becomes known that the current version is vulnerable. It accepts false positives (claims that an old version is vulnerable while it is not) to minimize false negatives (claims that a version is not vulnerable while it is) while still eliminating several spurious claims. It is a convenient alternative to manual analysis for an initial assessment to be followed by in-depth analysis which may be important for safety- and security-critical applications.

The experiment on Firefox and Chrome not only shows the scalability of our method, but also confirms the folk knowledge of the application of a very conservative rule in NVD vulnerability claims: "if version X is vulnerable, then so are all its previous versions". Just using the NVD off the shelves to draw conclusions about trends of vulnerabilities as done in (Roumani *et al.* 2015; Shahzad *et al.* 2012) may be misleading.

For a comprehensive evaluation, it is interesting to replicate our experiment on other OSS projects, e.g. Linux RedHat, which have code repository for tracking development, and whose vulnerabilities are also reported by the NVD.

Acknowledgments

This work has been partly supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 256980 (NESSOS), and agreement no. 285223 (SECONOMICS), and grant agreement no. 317387 (SECENTIS), and the Italian Project MIUR-PRIN-TENACE.

References

Agresti, Alan and Christine A Franklin (2012). *Statistics: the art and science of learning from data*. Pearson Higher Ed.

Allodi, Luca, Vadim Kotov, and Fabio Massacci (2013). "MalwareLab: Experimentation with Cybercrime attack tools". In: *Proceedings of the 2013 6th USENIX Workshop on Cybersecurity Security and Test (CSET' 13)*.

Antoniol, Giuliano, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc (2008). "Is it a bug or an enhancement? a text-based approach to classify change requests". In: *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, pp. 304–318.

Bird, Christian, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu (2009). "Fair and balanced? bias in bug-fix datasets". In: *Proceedings of the 7th European Software Engineering Conference*. ACM, pp. 121–130.

Chowdhury, Istehad and Mohammad Zulkernine (2011). "Using Complexity, Coupling, and Cohesion Metrics as Early Predictors of Vulnerabilities". In: *Journal of System Architecture* 57(3), pp. 294–313.

Chromium Developers (2013). *Chrome Stable Releases History*. http://omahaproxy.appspot.com/history?channel=stable, visited in April 2013.

Krsul, Ivan Victor (1998). "Software Vulnerability Analysis". PhD thesis. Purdue University.

Massacci, F. and Viet Hung Nguyen (2014). "An Empirical Methodology to Evaluate Vulnerability Discovery Models". In: *Software Engineering, IEEE Transactions on* 40(12), pp. 1147–1162.

Massacci, Fabio, Stephan Neuhaus, and Viet Hung Nguyen (2011). "After-Life Vulnerabilities: A Study on Firefox Evolution, its Vulnerabilities and Fixes". In: *Proceedings of the 2011 Engineering Secure Software and Systems Conference (ESSoS'11)*.

Massacci, Fabio and Viet Hung Nguyen (2010). "Which is the Right Source for Vulnerabilities Studies? An Empirical Analysis on Mozilla Firefox". In: *Proceedings of the International ACM Workshop on Security Measurement and Metrics (MetriSec'10)*.

Meneely, Andrew, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodrguez Tejeda, Matthew Mokary, and Brian Spates (2013). "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits". In: *Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement*.

Mozilla Security (2011). *Missing CVEs in MFSA?* Private Communication.

Needham, Ross (2002). "Security and Open Source". In: *Open Source Software Economics*. Available at http://idei.fr/doc/conf/sic/papers_2002/needham.pdf.

Neuhaus, Stephan, Thomas Zimmermann, Christian Holler, and Andreas Zeller (2007). "Predicting Vulnerable Software Components". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pp. 529–540.

Nguyen, Thanh, Bram Adams, and Ahmed E. Hassan (2010). "A Case Study of Bias in Bug-Fix Datasets". In: *Proceedings of 17th Working Conference on Reverse Engineering (WCRE'10)*.

Nguyen, Viet Hung (2014). "Empirical Methods for Evaluating Empirical Vulnerability Models". PhD thesis. University of Trento.

Nguyen, Viet Hung and Fabio Massacci (2013). "The (Un) Reliability of NVD Vulnerable Versions Data: an Empirical Experiment on Google Chrome Vulnerabilities". In: *Proceeding of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13)*.

NIST (2012). *Question on the data source of vulnerable configurations in an NVD entry*. Private Communication.

Ozment, Andy (2007). "Vulnerability Discovery and Software Security". PhD thesis. University of Cambridge. Cambridge, UK.

Ozment, Andy and Stuart E. Schechter (2006). "Milk or Wine: Does Software Security Improve with Age?" In: *Proceedings of the 15th USENIX Security Symposium*.

Quinn, Stephen D., Karen A. Scarfone, Matthew Barrett, and Christopher S. Johnson (2010). *SP 800-117. Guide to Adopting and Using the Security*

Content Automation Protocol (SCAP) Version 1.0. Tech. rep. National Institute of Standards & Technology.

Roumani, Yaman, Joseph K. Nwankpa, and Yazan F. Roumani (2015). "Time series modeling of vulnerabilities". In: *Computers & Security* 51, pp. 32 –40.

Shahzad, Muhammad, Muhammad Zubair Shafiq, and Alex X. Liu (2012). "A large scale exploratory analysis of software vulnerability life cycles". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, pp. 771–781.

Shin, Yonghee, Andrew Meneely, Laurie Williams, and Jason A. Osborne (2011). "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". In: *IEEE Transactions on Software Engineering* 37(6), pp. 772–787.

Sliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller (2005). "When do Changes Induce Fixes?" In: *Proceedings of the 2nd International Working Conference on Mining Software Repositories MSR('05)*, pp. 24–28.

Taleb, Nassim Nicholas (2010). *The black swan: the impact of the highly improbable*. Random House.

Wikipedia (2013). *Firefox Release History*. http://en.wikipedia.org/wiki/Firefox_release_history, visited in April 2013.

Williams, Branden R. and Anton A. Chuvakin (2012). *PCI Compliance, Thrid Edition: Understand and Implement Effective PCI Data Security Standard Compliance*. Ed. by Dereck Milroy. 3rd. Syngress, Elsevier.

Younan, Yves (2013). *25 Years of Vulnerabilities:1988-2012*. Tech. rep. Source Fire.

Zimmermann, Thomas, Rahul Premraj, and Andreas Zeller (2007). "Predicting Defects for Eclipse". In: *Proceedings of the 3th International Workshop on Predictor models in Software Engineering (PROMISE'07)*. IEEE Computer Society, pp. 9–15.