# The (Un)Reliability of NVD Vulnerable Versions Data: an Empirical Experiment on Google Chrome Vulnerabilities [*]

Viet Hung Nguyen and Fabio Massacci
DISI, University of Trento, Italy
{viethung.nguyen, fabio.massacci}@unitn.it

## ABSTRACT

NVD is one of the most popular databases used by researchers to conduct empirical research on data sets of vulnerabilities. Our recent analysis on Chrome vulnerability data reported by NVD has revealed an abnormally phenomenon in the data where almost vulnerabilities were originated from the first versions. This inspires our experiment to validate the reliability of the NVD vulnerable version data. In this experiment, we verify for each version of Chrome that NVD claims vulnerable is actually vulnerable. The experiment revealed several errors in the vulnerability data of Chrome. Furthermore, we have also analyzed how these errors might impact the conclusions of an empirical study on foundational vulnerability. Our results show that different conclusions could be obtained due to the data errors.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

## Keywords

Software security; vulnerability analysis; NVD reliability

## 1. INTRODUCTION

The last few years have seen a significant interest in empirical research on data sets of vulnerabilities. Public third-party vulnerability databases, *e.g.,* such as Bugtraq [12], ISS/XForce [3], National Vulnerability Database (NVD) [5], Open Source Vulnerability Database (OSVDB) [9], are mostly preferred by researchers due to their diversity, availability, and popularity. Among these, NVD is one of the most popular ones. The CVE-ID, *i.e.* the identifier of each NVD entry, is usually used as a common vulnerability identifier among other third-party data sources. In this type of research, the quality of data sources play a crucial role in empirical research on software vulnerabilities. If the data sources contain wrong data, any conclusion derived from these data sources may be potentially invalid.

Our research started from an abnormality in the data when we analyzed the NVD. According to our analysis of NVD data, all of vulnerabilities in Chrome v2–v12 were originated from version v1.0. To explain this, the following scenarios might occur: either yet more vulnerabilities in newer versions have not been detected, or there is a problem in the vulnerability data of Chrome, or both.

The analysis was based on an NVD data feature called *'vulnerable software and versions'* (or *vulnerable versions* for short). This feature remarks versions of particular applications that are vulnerable to the vulnerability described in the entry. For example, CVE-2008-7294 lists all Chrome versions before v3.0.195.24 in its *vulnerable versions*: this means that vulnerability affects Chrome v3.0 and all retrospective versions. According to an archive document[1], the information reported in this feature is *"obtained from various public and private sources. Much of this information is obtained (with permission) from CERT, Security Focus and ISSX-Force"*. Furthermore, our private communications with National Institute of Standards and Technology (NIST), host of NVD, and software vendors, have revealed a "paradox": NIST claimed vulnerable versions were taken from software vendors; whereas, software vendors claimed they did not know about this information. In other words, the original source of this feature is unknown to the public, and therefore its quality is unclear.

This raises a major threat to the validity of studies exploring this feature such as [4,8,10,11,14], and possibly others. We believe this may be a strong motivation to check for the reliability of NVD.

### 1.1 Contribution

The major contributions of this work are as follows:

- We present a replicable experiment to validate the reliability of "vulnerable software and versions" feature of NVD for Chrome. This experiment can be applied for other open source applications (*e.g.,* Firefox, Linux).
- We show that the error rates of vulnerabilities in Chrome versions are significant. The errors are both erroneously reporting vulnerabilities in past and future versions.

[*]This work is supported by the European Commission under the project EU-SEC-CP-SECONOMICS

[1]This page is removed, but can be accessed by url `http://web.archive.org/web/20021201184650/http://icat.nist.gov/icat_documentation.htm`

The rest of the paper is organized as follows. We present our research question and hypothesis (§2). After that we describe the validation method (§3) that we follow to conduct the experiment. Next, we report our result and perform analysis on collected data (§4). We also discuss threats to the validity and how to mitigate them (§5). Next we briefly review studies mostly related to our work (§6). Finally, we conclude our paper and discuss about the future work (§7).

## 2. RUNNING EXAMPLE AND RESEARCH QUESTION

We elaborate a running example on foundational vulnerabilities of Chrome to study the impact of the (un)reliability of NVD data. A *foundational vulnerability* [10] is one that was introduced in the very first version of a software (*i.e.* v1.0), but discovered later in newer versions. In theory, foundational vulnerabilities have higher chance to be exploited than others because they are exposed to attack longer than others. By finding these vulnerabilities in v1.0, attackers could use them to exploit recent versions (say, v20) at the release date. As the result, foundational vulnerabilities are a source for zero-day exploits.

By June 2012, NVD reported 539 vulnerabilities for 12 stable versions[2] of Chrome[3]. Out of these, 460 (85.3%) are reported as foundational. Figure 1 depicts the fraction of foundational vulnerabilities of Chrome. Clearly, each analyzed version of Chrome is rife with foundational vulnerabilities: 99.5% on average are foundational. We find unlikely that a lot of vulnerabilities were introduced in the first version, but none was introduced for the subsequent 11 versions. This motivates our research question as follows.

**RQ1** *To what extent is the 'vulnerable versions' feature of the data reported by NVD truthworthy?*

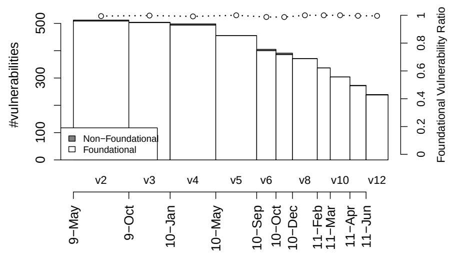
To have such knowledge, for each pair of NVD entry and software version listed in the *vulnerable versions* data feature, we verify whether the NVD entry impacts the corresponding version or not. If it is not, this pair is an *error*. The ratio of the number of error and the number of pairs is the *error rate* which we use as an indicator for the unreliability. In many cases, a small error rate is acceptable. Depending on the type of study the acceptable threshold of errors may vary. Here we choose the threshold of 5% which is normally considered a threshold for statistical significance. We consider the error rate as significant if the median of error rates in individual versions is significantly greater than 5%. We test the median of error rates, rather than the mean because a previous study [4] has shown that vulnerabilities do not follow the normal distribution. Hence, we test the following hypothesis:

**H1** *The median of error rates for vulnerabilities reported in Chrome versions is greater than* 5%.

We employ the non-parametric tests for the median to check for the significance.

[2]http://omahaproxy.appspot.com/about, visit on July 2012. This is a web application supported by Google team for tracking releases.

[3]We only consider 1+ year old versions as to allow their vulnerability data to mature.



Above it is the stack bars represent the fraction of foundational and non-foundational vulnerabilities, below it is the release date of the versions.

**Figure 1: Foundational vulnerabilities in Chrome.**

## 3. VALIDATION METHOD

To verify 539 vulnerabilities for 12 versions of Chrome, we need to check 5, 158 pairs[4] of vulnerability and version. Such huge amount of pairs is impossible for a manual verification. Additionally, the manual approach is not replicable. Thus, our proposed method is based on a repeatable and automatic approach [13] where security bug fixes are traced back to the code base to locate the vulnerable code responsible for vulnerabilities. Then we can determine whether a version claimed as vulnerable is actually vulnerable. Our method relies on the following assumptions.

**ASS1** When developers commit a bug-fix, they denote the bug ID in the commit message.

**ASS2** If the fragment of code responsible for a vulnerability is not there then the software is not vulnerable.

**ASS3** If a vulnerability is fixed by only adding code to a vulnerable file, all versions containing the non-fixed revision of the vulnerable file are vulnerable.

By *vulnerable files* we mean the files developers changed to fix a vulnerability. Also, by *code responsible* for a vulnerability we mean the code that developers changed to fix the vulnerability. In some cases, the changed code might not be the vulnerable one, but it helped to remove the vulnerability, despite the original buggy code not being edited. For example, a vulnerability that could lead to SQL injection attack could be fixed by inserting a sanitizer around the source in another module. However, missing of such sanitizer does not mean the application is vulnerable to the same SQL injection attack. Even though the changed code is not buggy in some cases, we still abuse the concept and call it *code responsible*.

Figure 2 sketches the steps of the proposed method. The input of the process is a list of vulnerabilities and the output is list of vulnerabilities annotated with vulnerable versions. The details are as follows:

STEP 1 *Repository mining.* This step takes the list of vulnerabilities and the commit log (*i.e.* list of commits) generated by the repository to produce commits of security bug fixes. A commit of security bug-fixes is one that mentions a security bug ID[5] in its commit

[4]To check the reliability of a vulnerability affecting 12 versions of Chrome, we need to check 12 pairs of vulnerability and version.

[5]Bugs appear in the input list of vulnerabilities.

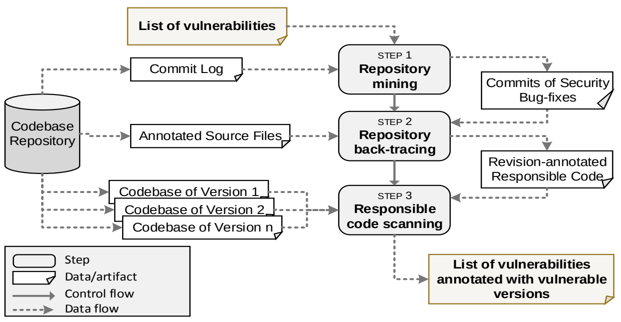
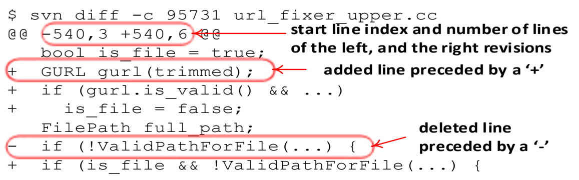


**Figure 2: Validation method overview.**



```
$ svn diff -c 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(...) {
```

start line index and number of lines of the left, and the right revisions
added line preceded by a '+'
deleted line preceded by a '-'

**Figure 3: An excerpt of the `diff` of two revisions.**

message and in some special patterns. These patterns may be vary in different software. For Chrome, they are `BUG=n(,n)*`, or `BUG=http://crbug.com/n` where $n$ is the bug ID.

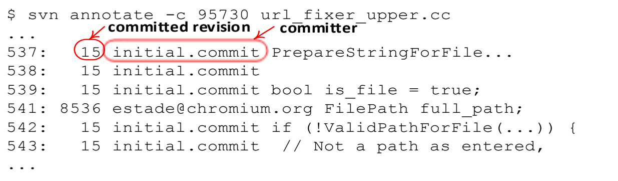
STEP 2 *Repository back-tracing* This step takes commits of security bug-fixes, and the annotated source files from the repository to produce revision-annotated responsible lines of code (LoC). For each source file $f$ in each commit, let $r_{fixed}$ be the revision of this commit. We compare revision $r_{fixed}$ to revision $r_{fixed} - 1$ of file $f$ using the `diff` command supported by the repository. The comparison output is in Unify Diff format, as exemplified by Figure 3, where we compare revision $r95730$ and $r95731$ of file `url_fixer_upper.cc`. By definition, responsible LoC appears in $r_{fixed} - 1$, but not in $r_{fixed}$. For instance, from Figure 3, the responsible LoC is $\{542\}$. We ignore trivial responsible LoC such as empty lines, or lines that contain only '{' or '}'.

Next, we execute `annotate` command for $r_{fixed} - 1$ of file $f$ to obtain the revisions of responsible LoC. Figure 4 presents an excerpt of the annotated file `url_fixer_upper.cc`. We see that the revision of LoC 542 is $r15$.

There is a special case where the comparison between $r_{fixed}$ and $r_{fixed} - 1$ contains no line preceded with the minus sign. It means developers fixed the vulnerability by adding code only (*e.g.,* security check). In this case we assume that all versions containing revision $r_{fixed} - 1$ and lower are vulnerable (see ASS3).

STEP 3 *Responsible code scanning.* This step looks for each revision-annotated responsible LoC in the code base of every version. If found, we append the corresponding version and the LoC to a list of version-annotated responsible LoC (see ASS2). From this list, we can identify vulnerable versions for each vulnerability.

Notice that there are *unverifiable* vulnerabilities for which the method can not verify the corresponding vulnerable versions. This could be due to a couple of reasons. First, there is no corresponding security bug for a vulnerability. Second, STEP 1 may not be always able to determine the commit for security bug-fixes of vulnerabilities.



```
$ svn annotate -c 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,
...
```

**Figure 4: An excerpt of the annotation.**

Table 1 shows a few examples of Chrome vulnerabilities where we apply the method to verify their vulnerable versions. The two first columns indicate the input of the method where we have list of NVD entries and their corresponding bug. We additionally annotate the vulnerable versions reported by NVD for each entry next to the CVE-ID. The next columns show the outputs of the steps. The dash line indicates the data is not available. It means the corresponding NVD entry is not verifiable.

For a better understanding, we describe how the NVD entry 2011-2822 is verified as in Table 1. This vulnerability is reported to affect Chrome v1 up to v13. Its corresponding bug is 72492. In STEP 1, by scanning the log, the bug fix is found at revision $r95731$ of file `url_fixer_upper.cc`. In STEP 2, we diff revision $r95730$ and $r95731$ of this file (see Figure 3). The responsible LoC is determined as $\{542\}$. Then we annotate $r95730$ of the file to get the revision of the responsible LoC, which is $\{r15\}$ (see Figure 4). In STEP 3, we scan for this line in the code base of all versions, and found it in v1 to v13. Finally, we identify the vulnerable versions for this vulnerability, which are v1–v13.

In additional, we manually checked the verification output of some NVD entries (*e.g.,* 2011-1120, 2011-3087, and 2011-1124) for the correctness of the method. The detail validation of these entries is provided in [7]

## 4. RESULTS AND EXAMPLE REVISED

We apply the proposed method to verify vulnerabilities of major versions of Chrome from v1 to v12. By June 2012, NVD reported 539 entries that allegedly affect these versions of Chrome. Out of these, 503 entries have links to 552 security bugs in Chrome Issue Tracker in their *references* section. The method took 16 hours on a 3 x quad-core 2.83GHz Linux machine with 4GB of RAM to complete.

As the result, 167 NVD entries (31%) are *verifiable*, and 372 (69%) are unverifiable. Among the verifiable ones, 134 (81%) have errors, *i.e.* their verified vulnerable versions are different than reported ones. Among the unverifiable, 36(10%) do not have corresponding bugs, and for 336(90%) we could not locate their commits of security bug fixes. We have done a qualitative analysis on these entries and found that they are bugs in external projects used in Chrome, *e.g.,* WebKit – the HTML rendering engine, V8 – the java script engine, and so on. Therefore, their commits of bug fixes do not exists in the repository of Chrome. Later we will discuss how to work around this problem as a part of future work.

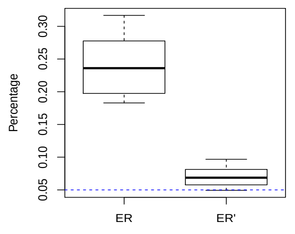
In the following, we analyze the difference of vulnerabilities in individual Chrome versions. Let *cve* be an NVD entry, and $v$ be the version in analyzed, we define:

- $V(cve)$: is a set of reported vulnerable versions of *cve*.

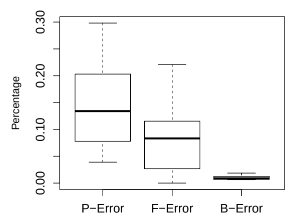
Table 1: The execution of the method on few Chrome vulnerabilities.

| INPUT | | Output of STEP 1 | Output of STEP 2 | Output of STEP 3 | |
|---|---|---|---|---|---|
| CVE-ID of NVD entry | Corresponding Bug | Commits for Bug-fixes | Revision-annotated responsible LoC | Version-annotated responsible LoC | Verified vulnerable versions |
| 2011-2822 (v1–v13) | 72492 | url_fixer_upper.cc[1] ($r95731$) | $\langle r15, 542\rangle$ | $\langle v1 - v13, 542\rangle$ | v1–v13 |
| 2011-4080 (v1–v8) | 68115 | media_bench.cc[2] ($r70413$) | $\langle r26072, 352\rangle$, $\langle r53193, 353\rangle$ | $\langle v3 - v8, 352\rangle$, $\langle v5 - v8, 353\rangle$ | v3–v8 |
| 2012-1521 (v1–v18) | 117110 | – | – | – | – |

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



(a) Error Rate



(b) Types of Error

In the whicker-box plot, the whickers represent the min and max value, the bold line in the middle is the median value, and the lower and upper part of the box are the quartile of the distribution. The blue dash line at 0.05 shows the threshold of error rate.

**Figure 5: The errors in vulnerable version data of NVD entries for Chrome.**

- $V'(cve)$: is a set of verified vulnerable versions of *cve*. If *cve* is unverifiable, $V'(cve) = \perp$.
- $\textit{verified}(v) = \{cve | v \in V'(cve)\}$: is the *cve* which responsible code is detected in version $v$.
- $\textit{erroneous}(v) = \{cve | v \in V(cve) \wedge v \notin V'(cve)\}$: is the set of verifiable *cve* which responsible code is not detected in version $v$.
- $\textit{unverifiable}(v) = \{cve | v \in V(cve) \wedge V'(cve) = \perp\}$: is the set of unverifiable *cve* of version $v$.

For example, according to Table 1, $V(\text{2011-4080}) = \{v1 - v8\}$, $V'(\text{2011-4080}) = \{v3 - v8\}$. Then, 2011-4080 is a verified *cve* in v3 (*i.e.* 2011-4080 $\in \textit{verified}(v3)$), but an erroneous *cve* in v1 (*i.e.* 2011-4080 $\in \textit{erroneous}(v1)$).

By ignoring the unverifiable vulnerabilities, the error rate of a version $v$ of Chrome is defined as the ratio of the number of erroneous vulnerabilities of $v$ by the number of verifiable vulnerabilities of $v$, as shown in the following formula:

$$ER(v) = \frac{|\textit{erroneous}(v)|}{|\textit{verified}(v)| + |\textit{erroneous}(v)|} \tag{1}$$

Being more optimistic, we assume all unverifiable vulnerabilities are all correct. The error rate is rewritten as follows:

$$ER'(v) = \frac{|\textit{erroneous}(v)|}{|\textit{unverifiable}(v)| + |\textit{verified}(v)| + |\textit{erroneous}(v)|} \tag{2}$$

Figure 5(a) shows the box plots for the distribution of the error rates in Chrome versions. In a box plot, the whickers represent the min and max values, the bold line in the middle is the median, and the lower and upper parts of the box are the quartiles of the distribution. According to the figure, since the median of both error rates $ER, ER'$ are much greater than 5%. It remarkably denotes that the number of erroneous vulnerabilities is not negligible. This is confirmed

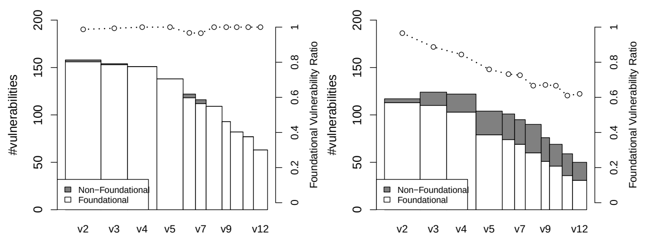


**Figure 6: Verifiable (left) vs. Verified (right) vulnerabilities.**

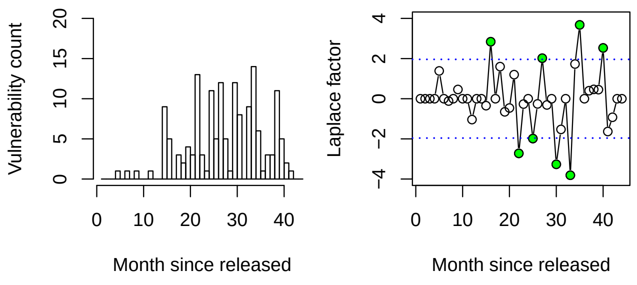
in the one-sided Wilcoxon rank-sign test where the null hypothesis is *"the median of error rates is* 5%*"*, and the alternative hypothesis is H1. The returned *p-value* for $ER$ and $ER'$ are almost zero ($2.44 \cdot 10^{-4}$, and $1.22 \cdot 10^{-4}$respectively). It means the error rates (both $ER$ and $ER'$) did not randomly happen and therefore are significantly greater than 5%.

We break down erroneous vulnerabilities into following categories:

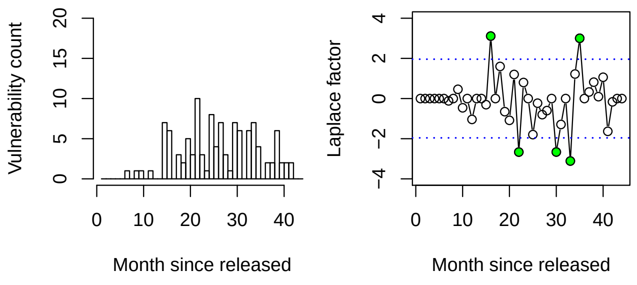
- *stretched-past error* (*P-error*): is the set of erroneous vulnerabilities whose version $v$ is older than all versions that the NVD entries are verified to impact to.
  $$\textit{P-error}(v) = \{cve \in \textit{erroneous}(v) | v < \min(V'(cve))\}$$
- *future-version error* (*F-error*): is the set of erroneous vulnerabilities whose version $v$ is newer than all versions that the NVD entries are verified to impact to.
  $$\textit{F-error}(v) = \{cve \in \textit{erroneous}(v) | v > \max(V'(cve))\}$$
- *beta error* (*B-error*): is the set of erroneous vulnerabilities whose corresponding NVD entries only impact non-official versions, *i.e.* $V'(cve) = \emptyset$.
  $$\textit{B-error}(v) = \{cve \in \textit{erroneous}(v) | V'(cve) = \emptyset\}$$

Similarly to (1), we calculate the *stretched-past error rate, future-error rate*, and *beta error rate.* Figure 5(b) reports the distributions of these rates. The *P-error* is slightly greater than *F-error*, and both of them are much greater than *B-error*. *B-error* seems to be negligible. We employ Wilcoxon rank-sum test to compare each pair of error categories. Since one category is compared with two other ones, the Bonferroni correction is applied *i.e.* the significance level is divided by 2: $\alpha = ^{0.05} /_2 = 0.025$. The test result confirms that both *P-error* and *F-error* are significantly greater than *B-error* since the returned *p-value*s are less than $\alpha$. The *p-value* $= 0.03 > \alpha$ of the comparison between *P-error* and *F-error* can be considered an evidence (even if not significant) that *P-error* is greater than *F-error*.

Hereafter we revise the running example about foundational vulnerability in Chrome. We assume that the same



(a) Verifiable vulnerabilities



(b) Verified vulnerabilities

**Figure 7: Trends in foundational vulnerability discovery.**

ratio of errors would be applied in the unverifiable vulnerabilities. Therefore, in following analysis we study the impact of error in verifiable vulnerabilities when we study the trend of foundational vulnerability.

We have two data sets: Verifiable and Verified. The former is the set of verifiable vulnerabilities that we could verify by the proposed method. Their vulnerable versions are reported by NVD. The latter is also the same set, but their vulnerable versions are verified.

Figure 6 illustrates the fraction of foundational vulnerabilities in each version. Left is based on Verifiable, and right is based on Verified. In this figure, The circles denote the percentage of foundational vulnerabilities. By looking at the left picture, even though the absolute number of foundational vulnerabilities decreased, their fractions are almost unchanged. Additionally, as aforementioned, it is strange that vulnerabilities are only introduced in v1.0, but none are introduced in later versions. However, by looking at the right side, this phenomenon disappears. Moreover, there is a decreasing trend in the foundational vulnerabilities fractions from v2.0 to v12.0. We additionally employ the Wilcoxon rank-sum test on the absolute number and the fraction of foundational vulnerabilities in each version. The test results show that the difference between the left and the right is indeed not random since the returned *p-values* are almost zero (*i.e.* $3.82 \cdot 10^{-3}$, and $3.84 \cdot 10^{-3}$ respectively).

Furthermore, we replicate the analysis on the trend of foundational vulnerability discovery as described in [10]. Figure 7 exhibits the analysis result on Verifiable (Figure 7(a)) and Verified(Figure 7(b)). In the figure, left is the discovery rate of foundational vulnerabilities discovered monthly since the release date, right is the Laplace test for trend in monthly discovered foundational vulnerabilities. Two dotted horizonal lines at value 1.96 and $-1.96$ indicate the range such that if a value of Laplace factor is out of this range, it is significant evidence for either an increasing ($> 1.96$) or a decreasing ($< -1.96$) trend in the data. Such values are indicated as green (gray) circles. Again we see a clearly difference between two discovery rates between two data sets. The *p-value* of the Wilcoxon rank-sum test is almost zero (0.0008): this indicates that the difference is significant. We can also see the difference in the trend of discovery. By using Verifiable, we might observe several significant trends (both increasing and decreasing) of foundational vulnerability discovery. Some of these trends, however, disappear in Verified.

In short, our experiment provides evidence that the error in the *vulnerable versions* feature of NVD entries for Chrome is not negligible. Among the errors, NVD tends to commit more stretch-past error than others. It is one of the reasons for the abnormality that 99.5% vulnerabilities of Chrome are foundational. This error in NVD has significantly impact to the analysis of foundational vulnerabilities where different conclusions can be drawn.

## 5. THREAT TO VALIDITY

**Construct validity** includes threats affecting the approach by means of which we collect and verify vulnerabilities. Threats in this category come from the assumptions as follows.

By making the assumption ASS1, we delegate the completeness of our method to the responsibility of developers and the quality control of the software vendor. According to [2], there are two types of mistakes: 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. Also in [2], the authors showed that the latter is negligible, while the former does exist. To evaluate the impact of the latter mistakes, we have done a qualitative analysis on bug-fix reports, and we found that all analyzed bug-fix commits are actually bug fixes. As for the former mistakes, we check the completeness of the bug-fix commits for vulnerabilities. As discussed, we found a large portion of vulnerabilities for which we could not locate the bug-fix commits. Our qualitative analysis on these vulnerabilities reveals that they originate from external projects used in Chrome. We discuss a potential solution addressing this threat in future work.

The second assumption ASS2 is apparently syntactical and might not cover all the cases of bug fixes since it is extremely hard to automatically understand the root of vulnerabilities. The assumption also means that if a version contains at least one line of responsible code, this version is vulnerable. Together with the assumption ASS3, our method might overestimate the vulnerable versions by classifying safety code as buggy (error type I, false positive). However, since most Chrome vulnerabilities are reported as foundational, if we overestimate the vulnerable version, the reported *error rate* is a lower bound of the actual one.

Besides, a technical threat to construct validity may be the buggy implementation of the method. To minimize such problem, we employ 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 there was no mistake.

**Internal validity** concerns the causal relationship between the collected data and the conclusion. Our conclusions are based on statistical tests. These tests have their own assumptions. Choosing tests which assumptions are violated might end up with wrong conclusions. To reduce the risk we

carefully analyzed the assumptions of the tests: for instance, we did not apply any tests with normality assumption since the distribution of vulnerabilities is not normal.
**External validity** is the extent to which our conclusion could be generalized to other applications. Our experiment is based on the vulnerability data of Chrome. So, to have a more generalized conclusion, a replication of this work on other applications should be done.

## 6. RELATED WORK

Śliwerski et al [13] 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 [13, 15] to construct bug-fix data sets. However, none of these studies mention how to address bug fixes which earlier changes could not be determined. These bug fixes were ignored and became a source of bias in their work.

Bird et al [2] conducted a study the level bias of techniques to locate bug fixes in code base. The authors have gathered a data set linking bugs and fixes in code base for five open source projects, and manual checked for the biases in their data set. They have found strong evidence of systematic bias in bug-fixes in their data set. Such bias might be also existed in other bug-fix data set, and could be a critical problem to any study relied on such biased data.

Antoniol et al [1] showed another kind of bias that the bug-fixes data set might suffer from. Many issues reported in many tracking system 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 [6], in an empirical study about bug-fix data sets, showed that the bias in linking bugs and fixes is the symptom of the software development process, not the issue of the used technique. Additionally, the linking bias has a stronger effect than the *bug-report-is-not-a-bug* bias.

## 7. CONCLUSION AND FUTURE WORK

In this paper we have conducted an experiment to verify the reliability of the *vulnerable versions* data of Chrome vulnerabilities reported by NVD. The experiment has revealed that the error in the *vulnerable versions* data is notable. Among verifiable vulnerabilities of individual Chrome version, approximately 25% of them are erroneous. If we assume that all unverifiable vulnerabilities are all correct, still more than 7% are erroneous in overall. We also demonstrated how these erroneous vulnerabilities could potentially impact the conclusion of foundational vulnerability study. Another study on the impact of erroneous vulnerabilities is further discussed in [7] . This experiment has shed a light into the (un)reliability of the NVD, and allows researchers to revisit the reliability of existing vulnerability databases.

However about two-third of Chrome vulnerabilities are unverifiable because they are vulnerabilities of the external projects used in Chrome. To be able to verify them, extra effort is required. First, we need to link the unverifiable vulnerabilities to the bug ID of the external projects. This could be done by parsing the Chrome bug report. Our qualitative study on several unverifiable vulnerability reports shows that all of them have links to bug reports of the external projects. Second, we apply the proposed method to identify vulnerable revisions of the external projects. Finally, we link these vulnerable revisions to the version of Chrome by looking at the repository of Chrome where the revisions of external projects used in individual versions of Chrome are mentioned. For example, Chrome v12.0 uses WebKit revision 80695, V8 revision 7138. A more detail discussion can be found in [7] .

Also as a part of future work, we plan to evaluate the robustness of the proposed method in identifying vulnerable revisions correctly. We also plan to repeat the experiment on other open source software like Firefox to have a better insight about the reliability of NVD.

## 8. REFERENCES

[1] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y. Guhneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proc. of CASCON'08*, pages 304–318, 2008.

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. pages 121–130. ACM, 2009.

[3] IBM. Internet Security System/X-Force, 2012. http://xforce.iss.net/.

[4] F. Massacci, S. Neuhaus, and V. H. Nguyen. After-life vulnerabilities: A study on firefox evolution, its vulnerabilities and fixes. In *Proc. of ESSoS'11*, 2011.

[5] National Institute of Standards and Technology. National Vulnerability Database, August 2012. http://web.nvd.nist.gov/.

[6] B. H. A. Nguyen, T.H.D.; Adams. A case study of bias in bug-fix datasets. In *Proc. of WCRE'10*, 2010.

[7] V. H. Nguyen and F. Massacci. The (un)reliability of nvd vulnerable versions data: an empirical experiment on google chrome vulnerabilities. *CoRR*, 2013. http://arxiv.org/abs/1302.4133.

[8] V. H. Nguyen and F. Massacci. An independent validation of vulnerability discovery models. In *Proc. of ASIACCS'12*, May 2012.

[9] OSVDB. The Open Source Vulnerability Database. http://www.osvdb.org.

[10] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *Proc. of USENIX'06*, 2006.

[11] E. Rescorla. Is finding security holes a good idea? *IEEE Sec. and Privacy*, 3(1):14–19, 2005.

[12] Security Focus. Bug Traq, 2012. http://www.securityfocus.com.

[13] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. of MSR'05*, pages 24–28, 2005.

[14] A. Younis, H. Joh, and Y. Malaiya. Modeling learningless vulnerability discovery using a folded distribution. In *Proc. of SAM'11*, pages 617–623, 2011.

[15] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. pages 9–15. IEEE Computer Society, 2007.