Which is the Right Source for Vulnerability Studies? An Empirical Analysis on Mozilla Firefox*

Fabio Massacci
University of Trento, Italy
fabio.massacci@disi.unitn.it

Viet Hung Nguyen
University of Trento, Italy
vhnguyen@disi.unitn.it

ABSTRACT

Recent years have seen a trend towards the notion of quantitative security assessment and the use of empirical methods to analyze or predict vulnerable components. Many papers focused on vulnerability discovery models based upon either a public vulnerability databases (*e.g.,* CVE, NVD), or vendor ones (*e.g.,* MFSA). Some combine these databases. Most of these works address a knowledge problem: can we understand the empirical causes of vulnerabilities? Can we predict them? Still, if the data sources do not completely capture the phenomenon we are interested in predicting, then our predictor might be optimal with respect to the data we have but unsatisfactory in practice.

In our work, we focus on a more fundamental question: the quality of vulnerability database. We provide an analytical comparison of different security metric papers and the relative data sources. We also show, based on experimental data for Mozilla Firefox, how using different data sources might lead to completely different results.

1. INTRODUCTION

Recent years have seen a growing interest in a *Quantitative Security Assessment*: a number of books on security metrics [11] and economics of security [10], a workshop on security and economics (WEIS), a practitioner workshop at USENIX (MetriCon), a more scientific one at CCS and ESEM (QoP, now Metrisec) and a number of papers that analyze vulnerability trends and correlate them with the evolution and characteristics of software components.

Most of these works address a knowledge problem: *can we predict the presence of vulnerabilities?* For example, Meneely and Williams provide some preliminary evidence that unfocussed contributions are a potential cause for the introduction of vulnerabilities in Linux [14].

If we could attain this knowledge, then developers and testers could concentrate their efforts on components predicted to have vulnerabilities in order to eliminate them.

*This research is partly supported by the EU-FET-IP-SECURE CHANGE project `www.securechange.eu`.

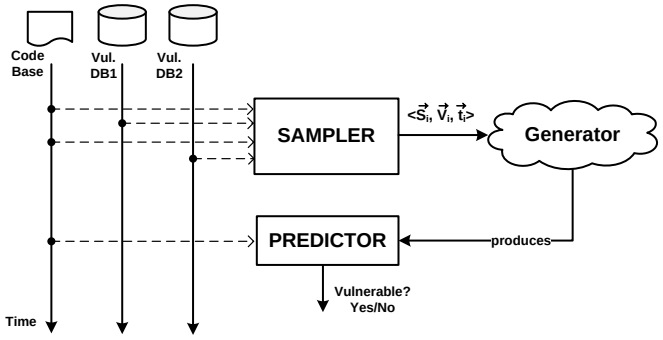


Figure 1: A Vulnerability Predictor Set-Up

Figure 1 illustrates the common schema for most research papers. The experimenter analyzes the code for source code metrics, $\vec{S}$, and then samples one or more vulnerability databases to determine information of vulnerable code entities, $\vec{V}$. The $\vec{S}$ and $\vec{V}$ vectors are associated with timestamps, $\vec{t}$. Most economics of security papers stops here and plots data of correlating information. Security metric papers go a step further and feed this information to a black box (*i.e.,* statistical correlation, machine learning trainer) that produces a "law of vulnerability" *i.e.,* a predictor. The final software engineering step is then feeding other fragments of the code base to the predictor to check whether the predictor is actually accurate on new data.

Almost all studies focus on predicting or understanding vulnerabilities, but only few provide some claims about the quality of the data source they used in their experiments.

1.1 The Contribution of this paper

It is clear that if the data sources do not completely capture the phenomenon, the predictor may be optimal with respect to the data, but unsatisfactory in practice.

In this work we analyze the data sources used in past studies in order to understand which databases and database features are used to address key research questions. We further make a deeper analysis on the Mozilla Firefox vulnerability databases to see the data quality of past papers. The experiment shows many papers proposing vulnerability prediction methods might have used an incomplete data source.

In the next section we discuss our research question more in details and how its answer can impact the "traditional" research questions in quantitative security assessment. Then

we present a comprehensive classification of the public Data Sources for vulnerabilities (§3). This information is the stepping stone to understand on which data a number of related works in quantitative security assessment has addressed their own research questions (§4). We then discuss an experimental set-up that integrates these data sources (§5) in order to see whether a broader picture can show the (un)suitability of different data sources for the analysis (6). It turns out that this is indeed the case. Finally we discuss the threats to the validity of our experimental findings (§7) and conclude (§8).

2. RESEARCH QUESTIONS

Below we present some of the most popular topics.

Fact Finding (RQ1). *Describe the state of practice in the field* [19, 21, 22]. They provide data and aggregate statistics but not models for prediction. Some research questions picked from prior studies are *"What is the median lifetime of a vulnerability?"* [21], *"Are reporting rate declining?"* [21, 22].

Modeling (RQ2). *Find mathematical models for vulnerabilities propose* [1–4, 22]. Working on this topic, researchers raise mathematical descriptions of the evolution of vulnerability, and collect facts to validate them.

Prediction (RQ3). *Predict defected/vulnerable component* [6, 9, 12, 15, 17, 19, 23, 24, 28, 29]. The main concern of these papers is to find a metric or a set of metrics that correlate with vulnerabilities in order to predict vulnerable components.

If we look at the issue of the median lifetime of vulnerability, papers in the first group will produce statistics on various software and the related vulnerability lifetime. Papers in the second group will identify a mathematical law that describes the lifetime of a vulnerability *e.g.,* a thermodynamic model [4], or a logistics model [1]. The good papers in the group will provide experimental evidences that support the model, *e.g.,* [1–3]. Studies on this topic aim to increase the goodness-of-fit of their models *i.e.,* try to answer the question *"How good does our model fit the fact?"*.

The last group will identify a software characteristics (or metrics) that correlate with the existence of the vulnerability, and then use this metrics to predict whether a software component will exhibit a vulnerability during its lifetime. These papers usually use statistics and machine learning methods and back up their claim with some empirical evidence. These studies focus on the attribute and the quality of prediction, and they aim to answer the question *"How good we are at predicting?"*

Most papers gave no or little space for the data sources that their researches are built on. In fact, the quality of data sources directly affects the goodness-of-fit as well as the quality of predictions.

The preliminary study that we have done on Mozilla Firefox shows a number of phenomena in the evolution of code and vulnerabilities that make the choice of the data source critical: the natural data source might turn out to be good for answering a research question (*e.g.,* the time-to-discovery-time-to-fix) but totally inadequate to answer another closely related question (*e.g.,* the total vulnerabilities of a component). This motivates our research question:

Data Quality(RQ0) : *The discussion on the quality of vulnerability sources.* On other words, we want to answer "Which features of vulnerability databases are needed (useful) to answer some research questions?", and "Is their used data adequate for their purpose?".

3. DATA SOURCES

There are hundreds of databases that keep track of security related issues for different software applications: on SecurityTracker (www.securitytracker.com), we found over 157 databases and security advisories. Unfortunately, many of them lack detailed information and are out of date, which significantly restrict the choice of data sources to assess. Hereafter, we describe some of the most popular (and usable) vulnerability databases and some of the less popular ones that are related to our study.

We classify databases in three classes based on their target products: *multi-vendor databases*, *vendor databases*, and *others*. Here, we just briefly present these databases.

Multi-vendor databases include:

- *Bugtraq* is an electronic mailing list about computer security, which publishes vulnerability information of many products, regardless of the vendor response.
- *Common Vulnerabilities Exposure* (CVE) is just a global identifier dictionary for vulnerability.
- *National Vulnerability Database* (NVD) provides expanded information and references to vulnerable software for CVE vulnerability.
- *Open Source Vulnerability Database* (OSVDB) is another public open vulnerability database created by and for the security community.
- *ISS/XForce* is another multi-target vulnerability database run by IBM. Each entry of ISS/XForce contains almost the same information as `Bugtraq`.

The second class includes databases maintained by software vendors, in which they announce bug and security information about their product:

- *OpenBSD errata bulletin* for OpenBSD.
- *Microsoft Security bulletin* for many products of Microsoft *e.g.,* Windows, IE.
- *Mozilla Foundation Security Advisories* (MFSA) is the vulnerability report for Mozilla products
- *Bugzilla* keeps track of programming bugs. In this work, we consider two instances of Bugzilla, which are Mozilla Bugzilla and RedHat Linux Bugzilla. The former is a defect database of all Mozilla products, and the later includes RedHat Linux.

The third class includes all other databases. Mostly, they are sanitized defected data of anonymous applications for testing purpose. These databases include:

- *Predictor Models In Software Engineering* (PROMISE) is a public repository that hosts many sanitized data sets used in many prediction models.

	ID	Feature	Multi-vendor DB						Vendor DB				Misc. DB			
			Bugtraq	CERT	CVE	OSVDB	NVD	ISS/Xforce	OpenBSD	MS Sec. Bulletin	MFSA	Bugzilla	PROMISE	MDP	NVDB	Our DB
	#	ID/Title	x	x	x	x	x	x	x	x	x	x		x	x	x
Ref.	R1	Reporter	x			x				x		x			x	x
	R2	CVE ID	x	x	x	x	x	x		x					x	x
	R3	References Link	x	x		x	x	x			x	x			x	x
Time	T1	Injection date													x	x
	T2	Discovery date				x						x	x		x	x
	T3	Fixed date				x			x						x	x
	T4	Published Date	x	x		x	x	x	x	x	x				x	x
	T5	Exploit publish date				x										
	T6	Updated Date	x	x			x			x		x				
Impact	I1	Description	x	x	x	x	x	x	x	x	x	x			x	x
	I2	Classifcation/Category				x	x								x	x
	I3	Impact/Severity/CVSS Score				x	x	x		x	x			x	x	x
	I4	Solution		x		x				x						
Code	C1	Version at discovery							x		x	x*			x	x
	C2	First vulnerable version													x	x
	C3	Other affected versions	x	x		x	x	x		x					x	x
	C4	Non Vulnerable/Fixed versions	x	x							x				x	x
	C5	Reference to codebase										x			x	x
	C6	Source code metrics											x	x		

*: this information is depend on software vendor *e.g.,* in RedHat Bugzilla, this feature is the software version. Meanwhile, in Mozilla Bugzilla, this feature is the branch version in source control.

Table 1: The common features of vulnerability databases

- *NASA IV&V Facility Metrics Data Program* (MDP) is a sanitized repository that stores defected data and metrics data for several products.
- *OpendBSD Vulnerability Database* (NVDB): is a vulnerability database of OpenBSDB constructed for studying vulnerability discovery process.

To sum up this section, a comparison of databases' features is shown in Table 1. The features fall into four groups, except the very basic *ID/Title* one.

The first one, *Reference*, describes the source of the vulnerability or cross references to other databases. All databases credit vulnerability reporters and some have a reference to the global dictionary, CVE.

The second group, *Time*, includes several features describing life-cycle of vulnerabilities. It starts with the time when the vulnerable code is inserted into code base (T1, a.k.a. birthday), the time when the vulnerability is first discovered (T2), the time when the vulnerability exploit is reported (T5), and the time when it is fixed (T3). We also denote the time when the vulnerability is publicly announce (T4), and the time when its information is updated (T6) *e.g.,* new affected version is added.

Among these time features, the *Published date* is available in most databases. This time is dependent on the database maintainers and do not provide meaningful information about the vulnerability. More interesting features such as T1, T2, T3 only appear in some databases. There are two databases: OSVDB and NVDB that claim to provide many time features. However, one only provides little data (OSVDB) and one is no longer supported (NVDB).

The third group, *Impact*, depicts the impact of a vulnerability, which are the short description (I1), the classification (I2), the severity (I3) and the solution (I4) for the vulnerability. Currently, I3 is described in both qualitative assessment (*i.e.,* low, moderate, high, critical), and quantitative assessment (*e.g.,* Common Vulnerability Scoring System).

The last group, *Code*, describes the software versions and code base related to a vulnerability. Software version data includes version where a vulnerability is discovered (C1); the earliest vulnerable version (C2), and the list of versions which are/are not affected (C3, C4). The references to source modules (C5) *e.g.,* classes, files containing a fix, and static code metrics of defected code (C6). None of multi-vendor databases like `Bugtraq`, NVD provides this data since it requires the permission to access code base. Observably, vendor and multi-vendor databases only focus on C1 and C3 (vendor databases provide C1, and multi-vendor ones provide C3), and almost say nothing about others. None of them supports the connection between vulnerabilities and code base (C5, C6).

4. DATA USAGE BY RESEARCHERS

The work by Frei and others [7] can be easily described as the representative of the security and economics fields. It offers a detailed landscape of which security vulnerabilities affect which systems but does not provide a concrete answer to any of the research questions we have listed in Section 2.

The two first research topics (RQ1, RQ2) have a close relationship. Normally, researchers observe the world (finding facts) then introduce models describing the observed phenomena and predicting future trends.

Among the papers in these area, Rescorla [22] focuses on the discovery of vulnerability. Although Rescorla points out many shortcomings of NVD, his study heavily relies on it. By studying vulnerability reports of several applications in NVD, Rescorla introduces two mathematical models, called *Linear model* and *Exponential model*, to identify trends in vulnerability discovery.

Alhazmi *et al.* [1, 3] observe vulnerabilities of Windows and Linux systems from different sources. For Windows systems, the data sources are mostly from NVD, other papers, and private sources. For Linux systems, data come from CVE and Bugzilla for Linux. The authors try to model the cumulative vulnerabilities of these system into two models: the *logistic model* and the *linear model*. Based on the goodness of fit on each model, the authors give a forecast about the number of undiscovered vulnerabilities, and emphasize the applicability of the new metric called *vulnerability density* obtained by dividing the total of vulnerabilities by the size of the software systems. Also based on these vulnerability data, in [2] Alhazmi *et al.* compare their proposed models with Rescorla's [22] and Anderson's [4]. The result shows that their logistic model has a better goodness of fit than others. It is an interesting topic for applying these two approaches to various vulnerability databases and see if conclusion changes by changing databases.

Ozment [21] points out many problems that NVD database suffered, which are *chronological inconsistency*, *inclusion*, *separation of events* and *documentation*. The chronological inconsistency refers to the inaccuracy in the versions affected by a vulnerability. The second problem is that NVD does not cover every vulnerability detected in a software system. In fact, only vulnerabilities that are discovered after 1999 and assigned CVE identifiers are included. The third problem refers to the duplication of vulnerabilities. The last problem is the lack of documentation. Many data fields of NVD are not well documented, particularly, the meaning of the data field and how the data is collected or calculated. The papers discuss technique to address the first problem, in which the actual bug's birth date is obtained by analyzing the log of the source version control. As a demonstration, the authors set up a vulnerability database of OpenBSD. Based on this data, Ozment [19] conducts an experiment to test the fitness of various vulnerability discovery models.

Works focusing on prediction capability (RQ3) are the most frequent ones. Hereafter, we briefly review studies on this area after 2005. For older studies, interested readers can find more detail in the review of Cata and Diri [5].

Nagappan and Ball [16] present a prediction model using code churn for system defect density. The experiment data come from source version control log and the defected data of Windows 2003.

Neuhaus *et al.* [17] construct a tool called Vulture to predict vulnerable components for Mozilla products with the accuracy of 50%. Vulture uses a vulnerability database for Mozilla products for training its predictor. This database is compiled upon three main different sources: MFSA, Mozilla Bugzilla and CVS archive. Vulture collects the import patterns and function-call patterns in many known vulnerable modules and then applies a machine learning technique (*i.e.,* Support Vector Machine) to classify new modules.

Menzies *et al.* [15] claim that choosing attribute metrics is less significant than choosing how to use these metric values. In the experiment on MDP data sets, they rank different metrics by using *InformationGain* values to select the metrics for the predictor. The ranking value of a metric is different across projects. The accuracy of the predictor is evaluated by a *probability of prediction* (*pd*), and a *probability of false alarm* (*pf*). However, Zhang *et al.* [26] point out that assessment using the IR notion of *precision* and *recall rate* is better than *pd,pf*. Zhang *et al.* [27] replicate the work in [15], with a combination of three function-level complexity metrics to do the prediction.

Olague *et al.* [18] make the comparison of the prediction power of three different metric suites: CK, MOOD and QMOOD. Their experiment is conducted on six versions of Rhino, an open-source JavaScript implementation of Mozilla. The defect data are collected from Mozilla Bugzilla. The authors use logistic regression methods to perform the prediction for each metric suite. In the result, the CK suite is the superior prediction metrics for Rhino.

Zimmerman *et al.* [29] build a logistic regression model to predict post-release defects of Eclipse using several metrics on different levels of code base *i.e.,* methods, classes, files and packages. The defect data are obtained by analyzing the log of the source version control. This method is detailed in [30], and used by [17]. The final dataset is put in the PROMISE repository. In an other work, Zimmerman and Nagappan [28] exploit program dependencies as metrics for their predictor. However, they did not state clearly where the defect data come from in their study.

Shin and Williams [23, 24] raise a research question about the correlation between complexity and software security. In order to validate these hypotheses, the authors conduct an experiment on the JavaScript Engine (JSE) component of the Mozilla Firefox browser. They mine the code base of four JSE's versions for complexity metric values. Meanwhile, faults and vulnerabilities for these versions are collected from MFSA and Bugzilla. Their prediction model is based on the *nesting level* metric and logistic regression methods. Although the overall accuracy is quite high, their experiment still misses a large portion of vulnerabilities.

Jiang *et al.* [12], in their work, study the prediction power of machine learning based vulnerability discovery models. Their experiments are based on the MDP data sets, using many metrics belonging to three categories: *code level*, *design level* and *combination of code and design level*, as well as several machine learning methods. The experiment results show that the metrics strongly impact the power of the models, while, there is not much difference among learning methods. Also, the most powerful metric is the combination of both code and design level metrics, and the design metrics are the most inferior ones.

Gegick *et al.* [8, 9] employ automatic source analysis tools (ASA) warnings, code churn and total line of code to implement their prediction model. However, the conducted experiments are based on private defected data sources and they thus are difficult to reproduce.

Chowdhury and Zulkernine [6] combine complexity, coupling and cohesion metrics for a vulnerability prediction model. These values are fed to a trained classifier to determine whether the source code is vulnerable. In their experiment, the authors used a vulnerability dataset for Firefox, which is assembled from MFSA and Bugzilla.

5. INTEGRATING EMPIRICAL DATA

We first retrieve all advisories for Mozilla Firefox to ex-

tract their features. Since these advisories are written manually, one HTML page for each advisory entry, they are not organized. Therefore, information extraction is done by a crawler application that parses and extracts data from web pages. An MFSA entry itself does not provide much value information, but the announced date when the vulnerability goes to public and the references to Bugzilla and CVE. Next, more detail are extracted from the bug detail in the Bugzilla as in [17,30]. Here, the interesting features are bug identifier, CVE identifier (optional), status, resolution and reported date. The reported date is the discovery date in the bug life cycle.

Neither MFSA nor Bugzilla provides a full list of vulnerable versions. These information can be obtained by exploiting the CVE identifiers cited in each MFSA entry. Using these CVE, we can collect the missing information on the NVD. However, to find out affected version list of each bug in each MFSA entry, we need to determine which NVD entry is about which bug. We call this task *CVE-to-bug mapping*.

This mapping is obtained in two ways: *automatic mapping* and *manual mapping*. The automatic mapping is relied on the explicit CVE references in bug details, or implicit one described in the MFSA which contains only one bug and one CVE closed together. The manual mapping requires human effort to understand the links between bug and CVE, and map them together if relevant. These cases happen to the MFSA entries that have more than one bug and CVE.

In the following, we describe common layouts in which bugs and CVEs usually appear in an MFSA entry. To clarify, we use the term `BUG` to denote a link to bugzilla, and term `CVE` for link to CVE. If more than one similar term appears in sequence, we use the star symbol (*) *e.g.,* `BUG*`.

- `BUG*`: there is no reference to CVE, so we could not make the mapping. This only appears in Firefox 1.0 MFSAs.
- `BUG CVE`: only one bug and one CVE. It is truly the case that this bug refers to this CVE, and the mapping is done automatically. This occurs in most MFSA entries.
- `(CVE BUG*)*`: a CVE entry is followed by many bugs, and then other CVE and bugs. In most cases, the CVE refers to the immediately bugs following it, some manual efforts are needed to check the details of CVE and relevant bugs. This layout is common to MFSAs reported before March 2008 (mostly for Firefox 1.5).
- `(BUG* CVE)*`: there are many bugs followed by a CVE, and other bugs and other CVE entries. Similar to the prior case, the CVE is usually referred by preceding bugs, but a manual check is necessary. This layout occurs in MFSA entries reported after March 2008 (mostly for Firfox 2.0 and higher).

The vulnerable modules in code base can be located by mining the code archive repository, CVS - in case of Mozilla. CVS is a source control which records all versions (a.k.a revisions) of source files. Each source file revision (revision for short) is annotated with a *committer name*, *date time* when the revision is committed to the repository, *tags* and *branches* the revision belongs to, and a short *description* describing the difference between this revision and its nearest

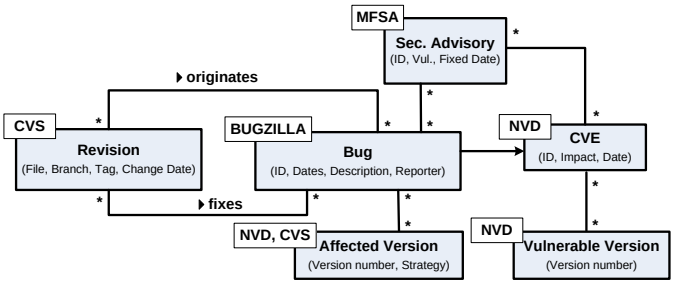


Figure 2: Simplified schema of our vulnerability database. Rectangles denote tables; icons at the top left corner of tables denote the source database.

ancestor. In case of bug fix, the description usually mentions the bug identifier with some special keyword such as *Fixes, Fixed*, or *Bug*. Thank to this meaningful and serious description style, we are able to construct the bridge from a reported bug to source modules. The approach mapping source modules and bugs by parsing the commitment description is detailed in [25] and used in [17] to predict vulnerable software components.

The next cumbersome task is to identify original revisions introduced the bugs, and the bugs' birthdays. In our database, we gathered data according to a number of different strategies. We maintain separated fields for birthdays, since we do not know which strategy is more appropriate than others. In addition, collecting all of them might help increasing the data accuracy.

By finding the bug's birthday, we also found the original revisions that contain vulnerable code. Further tracking on software release tags, we are able to determine vulnerable versions of the software. For example, considering a source file `jsapi.c`, the CVS reports that revision 3.214 of this file has the tag `FIREFOX_2_0_RELEASE`, and revision 3.220 has the tag `FIREFOX_2_0_0_1_RELEASE`. We can then assume that revisions 3.x where $x = 215..220$ belong to version 2.0.0.1.

In practice, a bug is often fixed in many source files. Each of these files could have many original revisions as discussed above. So, these files may be scattered over many versions, and one may belong to a version which is different from others. If it is truly the case, the similar birthday-detection strategies can be employed to deal with the conflict. We then obtain a list of affected software versions, a.k.a list of vulnerable versions, or CVS-reported vulnerable list.

This CVS-reported list is then compared with one reported in NVD, the NVD-reported list. If there is any vulnerable version reported in NVD, but not in the CVS-reported list but it exists in the *potential CVS-reported list*, then we include this version into the final vulnerability list. The potential CVS-reported list is exactly the CVS-reported list in which the pessimistic strategy is applied.

To sum up, Figure 2 illustrates the general schema of our vulnerability database for the specific case of Firefox.

6. DATA ANALYSIS

In order to understand the impact of a low data quality on the research claims we first compare how data has been used in Table 2. The Table 2 illustrates the favorite databases and feature of past papers depended on their research problem.

Papers	Research Problem			Databases														
	Fact Finding	Modeling	Prediction	Bugtraq	CERT	CVE	OSVDB	NVD	ISS/Xforce	OpenBSD	MS Sec. Bulletin	MFSA	Bugzilla	PROMISE	MDP	NVDB	Other sources
Alhazmi *et al.* [3]		x						T4, C1, C3					T2, C1				?
Ozment *et al.* [19] †	x			R3, T4, C3		#		R3, T4, C3		T4, C1	R1, T4					T1, C2	?
Nagappan *et al.* [16]			x														?
Rescorla [22] †	x	x						T4, C3									
Manadhata *et al.* [13]				#	#	#											
Menzies *et al.* [15]			x												C6		
Neuhaus *et al.* [17]			x									R3	#				
Olague *et al.* [18]			x										T2, C1				
Ozment *et al.* [20, 21]	x			R3, T4, C3			R3, T4, C3	R3, T4, C3	R3, T4, C3	T4, C1						T1,T2,T4, C2,C4	
Zhang *et al.* [27]			x												C6		
Zimmerman *et al.* [29]			x											C6			
Alhazmi *et al.* [2]		x						T4, C1, C3					T2, C1				?
Jiang *et al.* [12]			x												C6		
Shin *et al.* [23, 24]			x									R3, C1	C5				
Gegick *et al.* [8, 9]			x														?
Zimmerman *et al.* [28]			x														?
Chowhury *et al.* [6]			x									R3, C1	C5				
Ours								C3				T4, C4	R3, T2, C5				

†: Papers that explicitly discuss about which features are used. #: Counting of total number of vulnerability reports. ?: unknown features.

Table 2: Databases as used in recent works.

The table has two parts: the research question of each paper, and the features of which database are used in each paper. The used features are listed at the junction between database and paper. Most of the papers only discuss the few of the database features they use.

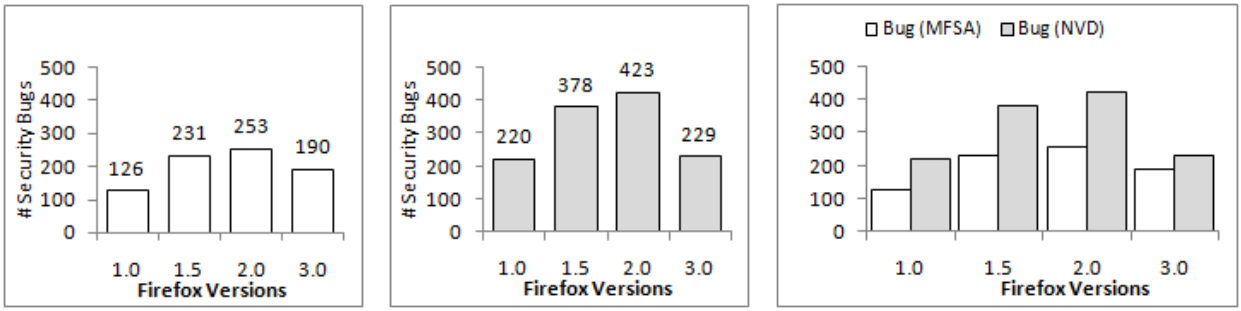
It is easy to see that, on the one hand, papers focusing on *fact finding, modeling* prefer to time and code features, particular published time (T4) and affected versions (C3). And most of them retrieve this data from NVD. Obviously, these studies wish to have the vulnerability discovery time (T2), but it is not supported by NVD, except some other vendor databases like Bugzilla. As discussed, OSVDB has T2, but this feature still remains empty in the majority of database entry. This is the reason that papers in the field reluctantly use T4 as a surrogate. Unfortunately, the connection between T2 and T4 is very weak, because T4 is completely depended on the database maintainers. Therefore the same vulnerability but reported in two different databases *e.g.,* NVD, `Bugtraq`, would have different value of T4. This might be the case that if we replay the experiments in past studies in different databases, we would receive different conclusions which may contradict each other.

Most papers focusing on vulnerability prediction, are interested in vendor-databases. These studies need the references from the security bugs to the corresponding code base (feature C5) which are not supplied in any multi-vendor database. The Table 2 shows that the data sources used in prediction mostly come from two different groups. Databases in the first group come from vendor bug-tracking database [6, 17, 18, 23, 24], some analyze the source version control to detect source files that contain bug fixes, [17, 18]. The vulnerability databases constructed in this way could be reproduced by others, and thus they are able to be cross-validated. At opposite side, the databases of the second group are obtained from private sources *i.e.,* closed source like Microsoft [16], Cisco or other private companies [8, 9], or public repository (*e.g.,* NASA MDP, PROMISE). We have no access to these sources and therefore we cannot say whether each type of information was present or not.

None of papers in our survey has seriously used any impact features (I1-I4). We speculate that this has two reasons. First, existing research studies actually do not need these features. Second, even if researchers need impact features, they do not want to use them because their quality is unknown and depended on human judgement.

In order to see the impact that a wrong data model can have on researches, we consider here the data used for the studies focusing on Mozilla Firefox [6,17,23]. Figure 3 shows the three different pictures of the vulnerability landscape for Firefox. We can easily concludes that:

- Figure 3(a) shows that the security of Firefox has significantly worsened from version 1.0 and has *moderately* improved from 2.0. In order words, Mozilla developers are not doing enough to secure their product.
- Figure 3(b) leads to a different conclusion that: Fire-



(a) Fixed bugs according to MFSA (b) Present bugs according to NVD (c) Bugs reported by integrated data

Figure 3: The number of security bugs of different Firefox versions reported by MFSA and our database.

fox has significantly worsened from 1.0 and has *significantly* improved from version 2.0 drop of. This is rather the opposite of the previous one.

- Figure 3(c) again brings up a new conclusion that: a large share of vulnerabilities of 1.5 and 2.0 has not been fixed while only a small share of 3.0 has not been fixed[1]. So not only 3.0 has less vulnerabilities but most of them have been fixed.

Now, if we used the data of Figure 3(a) to identify code metrics that correlate with vulnerabilities we could say that: all data used for the training have ignored between 17% to 43% of the vulnerabilities present in the software. Particularly, [6, 17, 23] might suffer from this issue if they use the code base of 2.0 by the modest recall rates for the missing portion of vulnerabilities. Also, the work in [18] only uses Bugzilla as its source, it may be suffered from a similar issue. Then its conclusion might be unsound.

Bugzilla keeps track of all vulnerabilities even if they are not fixed, but only authorized people are able to access to them. When a vulnerability is fixed and patch is released, an entry about this vulnerability is appeared in MFSA. Therefore only vulnerabilities discovered during support phase and fixed in a patch are recorded in MFSA, other ones will not be published and are considered to be vulnerabilities of future versions (if any). This causes only 253 security bugs are reported in version 2.0, meanwhile the number is actual 423 (more than 67%) and may increase.

If we use these biased data to construct predictors that could identify unknown vulnerability we could even say that they have been trained on the completely wrong dataset: the vulnerabilities that have been already fixed.

7. VALIDITY

In this section we discuss about both internal and external threats that can affect the validity of our data.

Bug in data collection. We collect data from various sources. Some of them, MFSA and CVS, requires parsing. The code that downloaded and parsed MFSA pages and the code that read the CVS log, parsed for history data, could contain bugs and thus might produce errors. However, these risks were mitigated by manually checking for a small amount of data and then correct the code. After collecting all data, a random check was carried out to validate the data. If the random check found an error, the code was then fixed. And then the collection and random check were repeated until there was no error.

Missing information in CVS. The mapping between bugs and code base relies upon conventional patterns in CVS committed messages. These patterns might be missed in some messages due to developers' mistakes. However, we believe this phenomenon, if exists, rarely happens and can be ignored since we were able to locate the corresponding code base for every fixed bug.

Generality. The combination of multi-vendor databases (*e.g.,* NVD, `Bugtraq`) and software vendor's databases (*e.g.,* MFSA, Bugzilla) only works for products for which the vendor maintains a vulnerability database and is willing to publish it. Also, the source control log mining approach only works if the software vendor grant community access to the source control, and developers commit changes that fix vulnerability in a consistent, meaningful fashion *i.e.,* independent vulnerabilities are fixed in different commits and each of these commits is associated with a message that refers to a vulnerability identifier. These constraints eventually limit the application of the proposed approach.

Mapping between CVE and Bugzilla. The Bugzilla and MFSA themselves do not contain enough information about affected versions (both "retrospective" and "prospective"). We found this data in NVD, but not all bugs refer explicitly to CVEs. The missing data is filled by manually looking at the MFSA. This task, as discussed in section 5, are laborious, time consuming and may contain mistakes.

8. CONCLUSION

In this work, we analyzed different research problems in the emerging trend towards to the quantitative security assessment and the use of empirical methods to analyze or predict vulnerable components. We identified the databases used by recent studies, and identified which database features are often used to answer the key research questions in vulnerability studies.

[1]The number of unfixed bugs in each version are determined by the difference between MFSA and NVD.

We setup an experiment in which we integrated numerous data sources on Mozilla Firefox. This database was a stepping stone to analyze prior work. Based on it, we noticed that many papers focusing on Mozilla Firefox might based their analysis on a data set that missed a large portion of vulnerabilities. This weakens the prediction power of their models.

9. REFERENCES

[1] O. Alhazmi and Y. Malaiya. Modeling the vulnerability discovery process. In *Proc. of ISSRE'05*, pages 129–138, 2005.

[2] O. Alhazmi and Y. Malaiya. Application of vulnerability discovery models to major operating systems. *IEEE Trans. on Reliab.*, 57(1):14–22, 2008.

[3] O. Alhazmi, Y. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Comp. & Sec.*, 26(3):219–228, 2007.

[4] R. Anderson. Security in open versus closed systems - the dance of Boltzmann, Coase and Moore. In *Proc. of Open Source Soft.: Economics, Law and Policy*, 2002.

[5] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Sys. with App.*, 36(4):7346–7354, 2009.

[6] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early predictors of vul. *J. of Soft. Arch.*, 2010.

[7] S. Frei, T. Duebendorfer, and B. Plattner. Firefox (in) security update dynamics exposed. *ACM SIGCOMM Comp. Comm. Rev.*, 39(1):16–22, 2009.

[8] M. Gegick, P. Rotella, and L. Williams. Toward non-security failures as a predictor of security faults and failures. *Eng. Secure Soft. and Sys.*, 5429:135–149, 2009.

[9] M. Gegick, P. Rotella, and L. A. Williams. Predicting attack-prone components. In *Proc. of IEEE ICST'09*, pages 181–190, 2009.

[10] L. A. Gordon and M. P. Loeb. *Managing Cybersecurity Resources: a Cost-Benefit Analysis.* McGraw Hill, 2006.

[11] A. Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt.* Addison-Wesley Professional, 2007.

[12] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proc. of PROMISE'08*, pages 11–18. ACM, 2008.

[13] P. Manadhata, J. Wing, M. Flynn, and M. McQueen. Measuring the attack surfaces of two ftp daemons. In *Proc. of QoP'06*, 2006.

[14] A. Meneely and L. Williams. Secure open source collaboration: An empirical study of linus' law. In *Proc. of CCS'09*, 2009.

[15] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *TSE*, 33(9):2–13, 2007.

[16] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of ICSE'05*, pages 284–292, 2005.

[17] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proc. of CCS'07*, pages 529–540, October 2007.

[18] H. M. Olague, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *TSE*, 33(6):402–419, 2007.

[19] A. Ozment. The likelihood of vulnerability rediscovery and the social utility of vulnerability hunting. In *Proc. of 4th Annual Workshop on Economics and Inform. Sec. (WEIS'05)*, 2005.

[20] A. Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In *Proc. of QoP'06*, 2006.

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

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

[23] Y. Shin and L. Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proc. of ESEM'08*, 2008.

[24] Y. Shin and L. Williams. Is complexity really the enemy of software security? In *Proc. of QoP'08*, pages 47–50, 2008.

[25] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. of the 2nd Int. Working Conf. on Mining Soft. Repo. MSR('05)*, pages 24–28, May 2005.

[26] H. Zhang and X. Zhang. Comments on data mining static code attributes to learn defect predictors. *TSE*, 33(9):635–637, 2007.

[27] H. Zhang, X. Zhang, and M. Gu. Predicting defective software components from code complexity measures. In *Procc. of PRDC'07*, pages 93–96, 2007.

[28] T. Zimmermann and N. Nagappan. Predicting defects with program dependencies. In *Proc. of ESEM'09*, 2009.

[29] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proc. of PROMISE'07*, page 9. IEEE Computer Society, 2007.

[30] T. Zimmermann and P. WeiSSgerber. Preprocessing cvs data for fine-grained analysis. In *Proc. of the 1st Int. Working Conf. on Mining Soft. Repo. MSR('04)*, pages 2–6, May 2004.