Large Scale Characterization of Software Vulnerability Life Cycles

Muhammad Shahzad, M. Zubair Shafiq, Alex X. Liu

Abstract—Software systems inherently contain vulnerabilities that have been exploited in the past resulting in significant revenue losses. The study of various aspects related to vulnerabilities such as their severity, rates of disclosure, exploit and patch release, and existence of common vulnerabilities in different products can help in improving the development, deployment, and maintenance process of software systems. It can also help in designing future security policies and conducting audits of past incidents. Furthermore, such an analysis can help customers to assess the security risks associated with software products of different vendors. In this paper, we conduct an exploratory measurement study of a large software vulnerability data set containing 56077 vulnerabilities disclosed since 1988 till 2013. We investigate vulnerabilities along following eight dimensions: (1) phases in the life cycle of vulnerabilities, (2) evolution of vulnerabilities over the years, (3) functionality of vulnerabilities, (4) access requirement for exploitation of vulnerabilities, (5) risk level of vulnerabilities, (6) software vendors, (7) software products, and (8) existence of common vulnerabilities in multiple software products. Our exploratory analysis uncovers several statistically significant findings that have important implications for software development and deployment.

Index Terms—vulnerability; disclosure; patch; exploit; diversity

1 INTRODUCTION

In computer software, a vulnerability is a loophole in the software code that enables an attacker to circumvent the deployed security measures [1]. Each software vulnerability has a life cycle that consists of distinct phases characterized by the events of its discovery, disclosure, exploitation, and patching. Each phase has a certain level of security risk associated with it. The first phase of the life cycle of a vulnerability starts when it is discovered by the vendor, a hacker, or any third-party software analyst. The risk associated with a vulnerability is particularly high if it is first discovered by malicious hackers. The next phase starts with the public disclosure of the vulnerability, which can be done by the vendor, a hacker, or any third-party software analyst. After disclosure, the information about a vulnerability is publicly available. Therefore, the level of risk increases further because the malicious hacker community is active in developing and releasing zero-day exploits [2]. The aim of the vendor is to release a patch for the vulnerability as soon as possible. The life cycle of a vulnerability ends when all users install the patch to fix it. A vulnerability can be exploited by hackers at any time during its entire life cycle.

The exploratory analysis of various aspects of vulnerabilities and their life cycles can uncover interesting patterns for vendors and software products that are helpful in following ways. First, a thorough analysis is helpful in the deployment of best practices in the software development processes. Second, such analysis is useful to develop the security policies that can handle future attacks and threats more effectively. Third, an exploratory analysis provides insights about the previous security incidents that are helpful in their audit. Fourth, it helps customers assess the risks associated with the software products of a particular vendor. Finally, it helps in implementing failure diversity by parallel use of multiple software that do not have common vulnerabilities.

To the best of our knowledge, no prior work exists that analyzes the evolution of life cycle of different types of vulnerabilities for different software products and vendors. The only work in this direction was from Frei et al. [3], [4]. In [3], Frei et al. studied the performance of the software industry as a whole but did not characterize individual vendors behavior. In [4], the authors only compared the vulnerability handling process of two vendors. Some researchers have focused on the modeling of vulnerability discovery process [2], [5], [6]. The goal of such work is to estimate the number of vulnerabilities in new software products. Another direction of work aims to study the changes in the patching behavior of vendors in response to vulnerability disclosures and the existence of competitors [7], [8]. These studies analyze only small vulnerability data sets and do not cover the behavior of individual vendors. Garcia et al. studied the presence of common vulnerabilities in operating systems (OSes) using a set of 1887 vulnerabilities [9]. They concluded that for some combinations of OSes, there are no common vulnerabilities while for others, there are. They, however, did not analyze the types of vulnerabilities that are usually common across different OSes.
In this paper we make following three contributions. (1) We have aggregated a large software vulnerability data set from three vulnerability repositories: (a) National Vulnerability Database (NVD) [10], (b) Open Source Vulnerability Database (OSVDB) [11], and (c) the vulnerability data collected by Frei et al. (FVDB) [3]. Our aggregated software vulnerability data set contains 56077 vulnerabilities between 1988 and 2013. (2) We have comprehensively analyzed software vulnerabilities along the eight dimensions mentioned in the abstract. Our observations are supported by statistical tests for significance. (3) To systematically analyze patterns in our vulnerability data set, we have utilized association rule mining to extract rules that represent exploitation behavior of hackers and the patching behavior of vendors.

Our study reveals several interesting observations about software vulnerabilities. For example, the trend of exponential increase in the vulnerability disclosure rate has discontinued since 2006. The types of vulnerabilities prevalent in early 2000s were mostly denial of service and executable code (such as buffer overflow); however, around 2008, cross site scripting, SQL injection, and PHP vulnerabilities became dominant. Vulnerabilities in open-source software are exploited more quickly and patched more slowly compared to closed-source software. Although the response of vendors to vulnerability disclosures has improved over time, the hackers are still faster than vendors in finding and exploiting vulnerabilities. OSes belonging to different classes such as Windows, Mac, Linux, and Solaris contain only a few common vulnerabilities, whereas OSes such as Windows XP, Windows Vista, and Windows 7 belonging to same class contain a large number of common vulnerabilities. These common vulnerabilities exist probably because vendors carry significant amounts of code from older versions to the newer versions of their OSes even when they claim that the newer version is completely redesigned.

**Paper Organization:** In Section 2, we explain the terminology and notations used in the paper and provide details about our vulnerability collection process and the aggregated data set. In Section 3, we analyze the evolution of vulnerability disclosure rates, access methodology for vulnerability exploitation, impact of the exploitation, risk associated with vulnerabilities, and evolution of different types of vulnerabilities. In Sections 4 and 5, we study the exploitation and patching behavior of hackers and vendors, respectively. In Section 6, we cross examine the exploitation behavior of hackers and the patching behavior of vendors. In Section 7, we study the vulnerabilities that are common among different OSes with a focus on failure diversity and code reuse. In Section 8, we present the implications of our work followed by the related work and conclusion.

## 2 Preliminaries

In this section, we first explain the terms and notations used in this paper and then present the data set used for analysis.

### 2.1 Terminology and Notations

**Vendor** is an entity (an individual, a group of individuals, or an organization) that develops a software product and is responsible to keep it secure. An ideal vendor would discover and patch all the vulnerabilities in its products before they are exploited.

**Hacker** is an entity that releases exploits for the vulnerabilities in the software products. Hackers can either be benign or malicious. Benign hackers write exploits to show how vulnerabilities can be exploited and usually do not make them publicly available without first informing the vendors. Malicious hackers write and release exploits with the objective of adversely affecting all or a group of people using the vulnerable software.

**Independent organization** is an entity that independently discovers and discloses vulnerabilities as well as their corresponding exploits and patches but is not involved in the development of patches or exploits.

**Disclosure Date** \(t_d\) refers to the date when information about a vulnerability is made publicly available after establishing that the vulnerability poses a potential risk. Vulnerability disclosure always takes place after the actual discovery of the vulnerability or at most on the day of actual discovery.

**Patch Date** \(t_p\) is the date when a vendor provides a solution (i.e. patch) for a vulnerability to neutralize the threat posed by it. We consider only those patches that are released by the corresponding vendor.

**Exploit Date** \(t_e\) is the earliest date when a hacker exploits a vulnerability. An exploit can be in the form of an automatic script, a virus, a tool, or any such thing that can breach the security of a software.

**Exploit – Disclosure** \(t_{ed}\) is the duration (in days) between the date an exploit for a given vulnerability was provided by hackers and the date the vulnerability was disclosed.

**Patch – Disclosure** \(t_{pd}\) is the duration (in days) between the date a patch for a vulnerability was released by the vendor and the date the vulnerability was disclosed.

**Risk Score** is assigned to a vulnerability by Common Vulnerability Scoring System (CVSS) [12] and establishes the magnitude of risk associated with that vulnerability. The numerical values of CVSS scores lie in the range \([0, 10]\).

Based on CVSS scores, we divide vulnerabilities into three categories. *Low*: \([0, 4]\); *Medium*: \([4, 7]\); *High*: \([7, 10]\).

**Access Vector** \((AV \in \{\text{Local}, \text{Adjacent Network}, \text{Network}\})\) indicates whether local access to the hardware is required to exploit the vulnerability or network access is sufficient.

**Access Complexity** \((AC \in \{\text{Low}, \text{Medium}, \text{High}\})\) is a measure of the complexity of the attack required to exploit the vulnerability.

**Integrity Impact** \((II \in \{\text{None, Partial, Complete}\})\) measures the potential impact of a successfully exploited vulnerability on the integrity of the system. Integrity refers to the trustworthiness of information.

## 2.2 Data Set

In this section, we provide details of our aggregate data and its basic statistics. We have collected vulnerability information from three sources: (1) NVD [10], (2) OSVDB [11], and (3) FVDB [3].

### 2.2.1 Data Aggregation

NVD and FVDB identify each vulnerability with Common Vulnerability and Exposures Identifier (CVE-ID) [13]. OSVDB also provides CVE-IDs of about 70% of vulnerabilities. We leverage the CVE-IDs to aggregate the vulnerability data from the three sources. We take CVSS scores, CVSS-vector...
metrics (i.e., access vector, access complexity, and integrity impact), vendor and product names, text description, and disclosure dates from NVD. From OSVDB and FVDB, we take disclosure dates, exploit dates, and patch dates.

The total number of vulnerabilities in our aggregate data set is 56077 and the number of vulnerabilities for which we have the disclosure dates, patch dates, and exploit dates are 56077, 9666, and 15363, respectively. Due to the sheer size of the data set, it is infeasible to find the unknown exploit and patch dates manually. To systematically conduct our study, we make following two subsets of our aggregate data set: ED-subset consists of 15363 vulnerabilities and contains those vulnerabilities for which both exploit and disclosure dates are known. PD-subset consists of 9666 vulnerabilities and contains those vulnerabilities for which we have both patch and disclosure dates. A total of 1424 vulnerabilities are common in both ED- and PD-subsets. Note that all dates in the two subsets are taken from OSVDB and FVDB.

2.2.2 Selection of Vendors and Products

The aggregate data set contains vulnerabilities from more than 12000 vendors and over 24000 software products. Figure 1 plots the number of vulnerabilities of each vendor in descending order. It can be seen that over 95% of the vendors have fewer than 10 vulnerabilities. To make statistically sound observations, we focus our attention on the top 8 vendors, each of which has at least 500 vulnerabilities. We select closed-source vendors including Microsoft, Apple, and Oracle, open-source vendors including Linux, Mozilla, and Red Hat, and hybrid vendors including Sun and Google. Linux is not a vendor rather it represents generic Linux kernel. We also study popular software products of these vendors that include Internet Explorer, Safari, Firefox, Chrome, Windows, MAC OS X, Solaris, and several Linux distributions.

3 GENERAL VULNERABILITY ANALYSIS

In this section, we study the trends in vulnerability disclosure and CVSS-vector metrics over the past two decades. We also categorize the vulnerabilities into groups and study their evolution.

3.1 Vulnerability Disclosure Trend

The rate of vulnerability disclosures experienced an exponential growth since 1997 and lasted till 2006 as can be seen in Figure 2. The vertical lines in the figure show the number of vulnerabilities disclosed every month since January 1990 and the dashed line shows the cumulative number of vulnerabilities. The number of vulnerability disclosures has not been increasing since 2006, rather it decreased from 2006 to 2011 despite the ever increasing use of software products. In 2012, the number of vulnerability disclosures saw a significant increase again.

3.2 General Trend of CVSS Scores

Recall from Section 2 that every vulnerability has an associated risk quantified by CVSS score. Figure 3 shows the box plots of CVSS scores for vulnerabilities in the products of the selected vendors. We note that CVSS scores of most vulnerabilities in our study lie in medium to high range. The median CVSS scores for closed-source vendors are greater than the median scores for open-source vendors.

3.3 Evolution of CVSS-Vector Metrics

Figures 4(a) to 4(c) show the evolution of three metrics of CVSS-vector. For each metric, we have calculated the percentage of vulnerabilities corresponding to each of its three values for every month since January 1990. We observe from Figure 4(a) that the percentage of remotely exploitable vulnerabilities has been increasing since 1998. The fact that most computer systems are connected to Internet has made it possible for hackers to exploit these systems remotely. Figure 4(b) shows the change in access complexity of vulnerabilities over the years. We observe that the percentage of low complexity vulnerabilities has decreased over time indicating that the hackers have to use more sophisticated techniques to exploit new vulnerabilities. From Figure 4(c), we observe a reduction in the percentage of vulnerabilities having complete integrity impact. This shows that modern software are more robust even when they are compromised.

3.4 Evolution of Different Types of Vulnerabilities

Our data sources do not specify the type of vulnerabilities in the data set. To determine the prevalent types of vulnerabilities and to study their evolution, we utilized unsupervised k-means clustering and agglomerative hierarchical clustering to cluster the vulnerabilities of different types into groups. For this, we leveraged the text description of vulnerabilities in NVD.

We compared words in the text descriptions with words in everyday non-technical news articles and removed the matching ones. This left us with keywords that describe the vulnerabilities. We manually went through them, removed the irrelevant ones, and used the remaining as attributes for clustering.

The keyword attributes are binary in nature, where an attribute value of 0 represents the absence of a keyword in description of a vulnerability and 1 represents the presence. To quantify similarity between text descriptions during clustering, we use Jaccard’s coefficient [14]. To define Jaccard’s coefficient between a given pair of vulnerabilities, let \( M_{xy} \) represent the number of attributes whose value for the first vulnerability is \( x \in \{0,1\} \) and for the second vulnerability is \( y \in \{0,1\} \). The Jaccard coefficient between a given pair of vulnerabilities is defined as the ratio of the number of attributes that are present in both vulnerabilities to the number of attributes that are present in at least one of them, i.e.,

\[
J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}
\]

The motivation behind using jaccard coefficient is that it ignores all those attributes that are not present (i.e., their values are 0) in either of the two vulnerabilities in the given pair of vulnerabilities. It is important to ignore all such attributes because such attributes do not provide any information about the “similarity” between the two vulnerabilities. This is also the reason behind not using Euclidean distance to calculate similarity because in Euclidean distance, if the values of a pair of corresponding attributes in two vulnerabilities are 0, that pair of attributes contributes as much to the similarity measure between the two vulnerabilities as a pair of corresponding attributes whose values are both 1.

It is well known that k-means clustering algorithm is well suited for large data sets with large number of attributes, where \( k \) represents the number of clusters it makes. The k-means clustering algorithm produces the best clusters
when a correct value of $k$ is used and the algorithm is initialized with appropriate values of the $k$ centroids. To determine the correct value of $k$, we performed agglomerative hierarchical clustering with Ward’s linkage [15] on 10 bootstrapped samples of the data set, sampled at a sampling rate of 20%. Hierarchical clustering with Ward’s linkage is the hierarchical equivalent of the $k$-means clustering and is often used to generate initial centroid values for the $k$-means algorithm [16]. We generated different number of clusters, ranging from 1 to 10, for each bootstrapped sample and extracted dominant keywords from resulting clusters to determine the type of vulnerabilities in each cluster. The dominant keywords of a cluster are the keywords that appear in the text description of over 80% of vulnerabilities in that cluster. We observed that when we increase the number of clusters from 7 to 8, the two new clusters that result from the splitting of one of the 7 clusters represent the same vulnerability type. Therefore, we chose $k = 7$. We used the average of centroid values from the 10 bootstrapped samples as the initial values of the $k$ centroids for the $k$-means algorithm.

Table 1 tabulates the dominant keywords extracted from the centroids of the 7 clusters. From the observed keywords, we label the clusters as PHP vulnerabilities (PHP), executable code (EXE), directory traversal (DT), SQL injection (SQL), cross-site scripting (XSS), and miscellaneous vulnerabilities (Misc). A major portion of EXE is constituted by buffer overflow vulnerabilities. Figure 5 shows the number of vulnerabilities belonging to each cluster disclosed each year since 1999.

To illustrate the usefulness of augmenting agglomerative hierarchical clustering with the $k$-means clustering, Table 2 tabulates the dominant keywords extracted from the centroids of the 7 clusters obtained using $k$-means clustering without augmenting it with agglomerative hierarchical clustering. On comparing Table 2 with Table 1, we see that out of the seven labels and their corresponding keywords shown in the two tables, six labels are the same while the seventh label is different: directory traversal (DT) in Table 1 and buffer overflow (BO) in Table 2. If we compare the keywords corresponding to BO in Table 2 with the keywords of EXE in the same table, we see that they both contain “execute” and “code”, i.e., all vulnerabilities clustered by simple $k$-means in BO cluster actually are EXE vulnerabilities. On the other hand, when we used the more sophisticated method, i.e., $k$-means augmented with hierarchical clustering, we obtained a cluster that kept all buffer overflow vulnerabilities inside the EXE cluster, and made a new cluster containing vulnerabilities of type directory traversal (DT). This identification of a new cluster and keeping the vulnerabilities of the same type in a single cluster demonstrates the supervisor performance of $k$-means augmented with hierarchical clustering compared to the simple $k$-means clustering.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Label</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>“php, parameter, execute, file, code, url”</td>
<td>PHP</td>
<td>8.32%</td>
</tr>
<tr>
<td>“execute, code”</td>
<td>EXE</td>
<td>14.1%</td>
</tr>
<tr>
<td>“buffer, execute, code, overflow”</td>
<td>BO</td>
<td>10.2%</td>
</tr>
<tr>
<td>“service, denial”</td>
<td>DOS</td>
<td>14.2%</td>
</tr>
<tr>
<td>“injection, sql, execute, commands”</td>
<td>SQL</td>
<td>11.2%</td>
</tr>
<tr>
<td>“cross, scripting, site, script, html, inject”</td>
<td>XSS</td>
<td>12.4%</td>
</tr>
<tr>
<td>“misc”</td>
<td>Misc</td>
<td>36.3%</td>
</tr>
</tbody>
</table>

TABLE 2

Results of vulnerability clustering: simple $k$-means
From Table 1, we observe that only DoS, EXE, and DT vulnerabilities were prevalent till 2001. DoS and EXE constitute a major portion of software vulnerabilities even today which indicates that the vendors have not been able to devise effective strategies to limit these types of vulnerabilities. Since 2002, we observe an increase in the XSS vulnerabilities, which peaked in 2006. PHP vulnerabilities were prevalent in 2006 and 2007 and SQL vulnerabilities were prevalent between 2005 and 2010. These trends highlight the shift in focus of hackers to exploit new services as they become popular.

In this section, we study the behavior of hackers in releasing exploits for vulnerabilities. We analyze trends in $t_{cd}$ values (defined in Section 2.1) of vulnerabilities. The analysis presented in this section is done on ED-subset. We study following three ranges of $t_{cd}$ values:

- $t_{cd} < 0$ shows that an exploit for a given vulnerability was released before its public disclosure. The vulnerabilities falling in this range represent a big threat to the security of end-users as the vendor could be oblivious about them. A total of 2.8% vulnerabilities fall into this range.

- $t_{cd} = 0$ refers to the case when an exploit for a given vulnerability was released on the day it was disclosed. A total of 88.2% vulnerabilities fall into this range. The exploits corresponding to vulnerabilities with $t_{cd} \leq 0$ are called zero-day exploits.

- $t_{cd} > 0$ means that the exploit for a vulnerability was released after its public disclosure. The vulnerabilities for which $t_{cd} > 0$ represent the case where a vulnerability is disclosed by the vendor or an independent organization and the hackers used this information to release an exploit in more than a day. 9.7% vulnerabilities fall in this range. To do more detailed analysis, we subdivide this range into three parts: (1) $0 < t_{cd} \leq 7$ gives us the percentage of exploits released within a week of disclosure, (2) $7 < t_{cd} \leq 30$ gives us the percentage of exploits released after a week and within a month of disclosure, and (3) $t_{cd} > 30$ gives us the percentage of exploits released more than a month after the disclosure.

4.1 Evolution of Exploitation

To extract the dominant trends in the exploitation behavior of hackers, we first divided the vulnerabilities in the ED-subset into groups, where each group contains vulnerabilities disclosed in a distinct year. Then we subdivided the vulnerabilities in each group into five subgroups corresponding to the five ranges of $t_{cd}$. We then calculated the percentage of vulnerabilities in each subgroup (called the percentage size of the subgroup) in its respective group and plotted the results in Figure 6 in the form of stacked bars where each bar corresponds to the group of vulnerabilities disclosed each year and each block in every bar represents the percentage size of the corresponding subgroup in its respective group. The number inside each block is the value of the percentage size of the corresponding subgroup. The number at the top of each bar represents the total number of vulnerabilities in the corresponding group. All figures in rest of the paper have been made using similar methodology.

4.2 Exploitation of Types of Vulnerability

We now consider the exploitation of different types of vulnerabilities. Figure 7 has been made in the same way as Figure 6 except that now the groups are the types of vulnerabilities. It can be seen that for over 80% of vulnerabilities of each type (except EXE), exploits are released on or before the day of disclosure. In case of EXE, a significant percentage of vulnerabilities is exploited several weeks after the disclosure. One possible reason for slow exploitation of EXE vulnerabilities can be that EXE vulnerabilities are one of the oldest types of vulnerabilities, as seen in Figure 5. The software industry has evolved to make the products more secure against exploitation of such vulnerabilities, making it harder for the hackers to find and exploit them.

4.3 Exploitation Trend for Vendors and Products

We study the behavior of hackers in releasing exploits for different vendors and their respective products. Figures 8 and 9 show the exploit data for the selected vendors and
products respectively. These figures have been made for vendors and products in the same way as Figure 7 was made for vulnerability types.

Let’s first compare the vulnerability exploitation in open vs. closed-source vendors. In comparison to closed-source vendors, for open-source vendors e.g., Linux, Red Hat etc., a comparatively larger percentage of vulnerabilities is exploited before the day of disclosure. This shows that the malicious hackers are faster in exploiting the vulnerabilities in open-source vendors. This is probably due to the access to the source code of open-source software products.

To make statistically significant observations, we do statistical hypothesis testing. As our samples for open-source and closed-source vendors contain large number of data points, the most appropriate statistical test for this scenario (and all the subsequent scenarios) is the standard one-tailed t-test [17]. The t-test is considered to be the most appropriate when the number of data points in the samples are large (>50) regardless of the distributions they come from.

To remove any bias in testing, we state the null hypothesis as: the mean value of \( t_{ed} \) for open-source vendors, \( \mu_{t_{ed}}(O) \), is equal to the mean value for closed source vendors, \( \mu_{t_{ed}}(C) \). The alternative hypothesis is: \( \mu_{t_{ed}}(C) \) is greater than \( \mu_{t_{ed}}(O) \). We apply the right tailed t-test to the null hypothesis. If the null hypothesis is rejected, it would be statistically sound to claim that the average time to exploit a vulnerability in closed-source software is larger compared to open-source software.

![Fig. 8. Exploited vulnerabilities w.r.t vendors](image)

![Fig. 9. Exploited vulnerabilities w.r.t products](image)

Let’s first compare the vulnerability exploitation in open vs. closed-source vendors. In comparison to closed-source vendors, for open-source vendors e.g., Linux, Red Hat etc., a comparatively larger percentage of vulnerabilities is exploited before the day of disclosure. This shows that the malicious hackers are faster in exploiting the vulnerabilities in open-source vendors. This is probably due to the access to the source code of open-source software products.

To make statistically significant observations, we do statistical hypothesis testing. As our samples for open-source and closed-source vendors contain large number of data points, the most appropriate statistical test for this scenario (and all the subsequent scenarios) is the standard one-tailed t-test [17]. The t-test is considered to be the most appropriate when the number of data points in the samples are large (>50) regardless of the distributions they come from.

To remove any bias in testing, we state the null hypothesis as: the mean value of \( t_{ed} \) for open-source vendors, \( \mu_{t_{ed}}(O) \), is equal to the mean value for closed source vendors, \( \mu_{t_{ed}}(C) \). The alternative hypothesis is: \( \mu_{t_{ed}}(C) \) is greater than \( \mu_{t_{ed}}(O) \). We apply the right tailed t-test to the null hypothesis. If the null hypothesis is rejected, it would be statistically sound to claim that the average time to exploit a vulnerability in closed-source software is larger compared to open-source software.

We give a general equation for hypothesis testing that will be used for all the subsequent tests:

\[ H_0 : \mu_A(X) = \mu_B(Y) \]
\[ H_1 : \mu_A(X) > \mu_B(Y) \]

where \( X=C \) represents closed-source vendors, \( Y=O \) represents open-source vendors, and \( A=B=t_{ed} \) represents that the data points of \( t_{ed} \) are being considered. We do the hypothesis testing for a 95% confidence interval i.e., \( \alpha=0.05 \).

Our test resulted in a p-value of 0.003 which is much smaller than \( \alpha \), thus we reject \( H_0 \) to accept \( H_1 \). Thus, it is statistically sound to state that the exploitation of vulnerabilities (by malicious and benign hackers collectively) in closed-source software is slower compared to open-source software.

Figure 8 shows that hackers release most exploits on or before the disclosure dates for Microsoft and Apple. This is primarily because malicious hackers find it more rewarding to exploit the products that have wider market capitalization. Similarly, benign hackers consider it more important to apprise such vendors about potential exploitation because malicious exploitation of their products can adversely affect a large number of users.

For the selected products, we see the similar trend in Figure 9 as for vendors in Figure 8 except for Windows. The percentage of exploited vulnerabilities for Windows till disclosure date is less than that of OS X but at the same time the percentage of exploited vulnerabilities for Windows before disclosure is greater than that for OS X. In fact, the mean value of \( t_{ed} \) for Windows is negative while that for OS X is positive. The t-test with \( X=\text{"OS X"}, Y=\text{"Windows"}, \text{and } A=B=t_{ed} \) yields \( p=0.031 \) proving that the exploitation in Windows is quicker compared to OS X.

Among web browsers, Firefox has the smallest percentage of vulnerabilities exploited till disclosure date compared to Internet Explorer and Safari but at the same time has the highest percentage of vulnerabilities exploited before the disclosure. The t-test with \( X=\text{"Safari"} \) and \( Y=\text{"Internet Explorer"} \) yields \( p=0.05 \) showing that exploitation in Internet Explorer is quicker compared to Safari. The t-test with \( X=\text{"Safari"} \) and \( Y=\text{"Firefox"} \) yields \( p=0.09 \), and therefore, fails to reject the null hypothesis.

### 4.4 Exploitation Behavior: CVSS Scores

Recall from Section 2.1 that we divide vulnerabilities into three risk categories based on CVSS scores: low, medium, and high.

Figure 10 has been generated in the same way as Figure 8 except that we plotted the vulnerabilities belonging to low, medium, and high risk categories separately. The white lines with round markers represent the percentage of total vulnerabilities belonging to low, medium, or high categories.

It is intuitive to think that hackers would be less interested in exploiting low risk vulnerabilities because such vulnerabilities cause less damage. This is exactly what the markers for low risk vulnerabilities show in Figure 10. The bars in Figure 10 show that the percentage of medium risk vulnerabilities for which exploits are released on or before the disclosure date is greater than that for high risk vulnerabilities for all closed-source vendors and some open-source vendors.
4.5 Interesting Exploitation Rules

Now we present some interesting association rules about the exploitation behavior in the products of the short-listed vendors. We used the implementation of the Apriori association rule mining algorithm [18] in WEKA [19] to extract the rules with confidence greater than 95%. For association rule mining, we used following attributes of each vulnerability: Vendor Name \(<vnd>\), Product Name \(<prd>\), Vulnerability Type \(<typ>\), Severity \(<sev>\), \(t_{ed}\), and \(t_{pd}\). For the rules presented in this section, we used \(t_{ed}\) as class attribute.

We found that in case of Microsoft, for the majority of the vulnerabilities including DoS, XSS, and EXE, exploits are released on the day the vulnerabilities are disclosed. One such rule obtained from association rule mining is:

\[
\text{vnd} = \text{Microsoft} \quad \text{typ} = \text{XSS} \quad \text{sev} = \text{H} \rightarrow t_{ed} = \text{0-day.}
\]

In case of Apple, the exploits for the majority of the vulnerabilities are released on or before their disclosure dates. For example, as shown in the following rule, vulnerabilities in Safari browser are mostly exploited on the day of disclosure: vnd = Apple prod = Safari typ = EXE sev = H \(\rightarrow t_{ed} = \text{0-day.}\)

For Solaris, association rules show that high risk vulnerabilities are exploited on the day of disclosure while medium risk vulnerabilities are mostly exploited within a week after their disclosure. The latter trend is shown by the following rule: vnd = Sun prod = Solaris sev = M \(\rightarrow 0 < t_{ed} \leq +1 \text{ week.}\)

For Mozilla, we get interesting rules showing that hackers do not exploit a vulnerability that has already been patched while they quickly exploit those that have not been patched. Two rules stating this observation are: (1) vnd = Mozilla prod = Firefox typ = EXE \(t_{pd} = \text{0-day} \rightarrow t_{ed} > +1 \text{ month.}\) (2) vnd = Mozilla prod = Firefox typ = EXE \(t_{pd} < +1 \text{ month} \rightarrow t_{ed} = \text{0-day.}\)

5 Patching Behavior

Now we study the behavior of vendors in providing patches for vulnerabilities in their products. For this, we study the trends in \(t_{pd}\) values of vulnerabilities. The analysis presented in this section is based upon PD-subset. The three ranges for \(t_{pd}\) that we study are described below.

\(t_{pd} < 0\) shows that the patch for a given vulnerability was released before its public disclosure. A total of 10.1\% vulnerabilities have \(t_{pd} < 0\) which is greater than the corresponding value for \(t_{ed} < 0\). A possible reason is that the independent organizations inform the vendors about the vulnerabilities they discover and give them enough time to release a patch before disclosing the vulnerabilities. Another possible reason is that the vendors discover the vulnerabilities themselves and release patches without disclosing them.

\(t_{pd} = 0\) means that the patch for a vulnerability was released on the disclosure day. A total of 62.2\% vulnerabilities have \(t_{pd} = 0\). The patches corresponding to \(t_{pd} \leq 0\) are called zero-day patches.

\(t_{pd} > 0\) refers to the case where the patch for a given vulnerability was released after its public disclosure. In our PD-subset, 27.7\% of all the vulnerabilities are patched more than a day after their disclosures. We further subdivide the range \(t_{pd} > 0\) into the same three parts as in Section 4.

The \(t\)-test with \(A = t_{pd}, B = t_{ed},\) and \(X = Y = \text{"aggregate data set"}\) yields \(p \approx 0\) which leads us to accepting the alternative hypothesis that, compared to hackers (benign and malicious collectively), vendors take more time on average to patch a vulnerability.

5.1 Evolution of Patching Behavior

In Figure 11 we observe that till 2005, the percentage of vulnerabilities patched on or before disclosure dates consistently decreased. Keeping in view the fact that independent organizations inform the vendors about vulnerabilities well before disclosing them [20], such a poor patching behavior of vendors indicates that security was not a major concern for vendors at that time. However, we see a significant improvement after 2005. Since 2008, vendors have been providing patches for more than 80\% of total vulnerabilities till their disclosure dates. A possible reason for this can be that it has become more common to not report vulnerabilities publicly, rather, the vendors pay for vulnerabilities.

5.2 Patching of Types of Vulnerabilities

From Figure 12, we observe that vendors are generally slower in patching the PHP vulnerabilities. A possible explanation is that for the majority of PHP vulnerabilities, the exploits are released by the benign hackers as implied by the large percentage size of the subgroup corresponding to \(t_{ed} = 0\) in Figure 7. The vendors are quicker in patching the EXE vulnerabilities as these vulnerabilities pose high security risk because they usually have higher CVSS scores.

5.3 Patching Trend for Vendors and Products

Here we study the behavior of the selected vendors in patching the vulnerabilities in their products. Figures 13 and 14 show the patch data for selected vendors and products.

Closed-source vendors are typically profit based organizations and have more resources to secure their products as compared to open-source vendors. Therefore, we expect better patching behavior from closed-source vendors. Figure 13 confirms this intuition as Microsoft, Apple, and
the patched vulnerabilities for open-source vendors. Applying we observe significantly smaller percentages and quantity of vulnerabilities on or before disclosure dates. In comparison, Fig. 13. Patched vulnerabilities w.r.t vendors
to reject the null hypothesis of Safari against Firefox. More quickly compared to Internet Explorer but the test fails
Windows as compared to its remaining products. The following two rules show this: (1) vnd=Microsoft prod=Windows XP typ=EXE → fps=0-day. (2) vnd=Microsoft prod=Internet Explorer typ=EXE → fps=+1 month.

Apple also patches vulnerabilities in its operating systems as soon as they are disclosed. The following rule highlights this trend: vnd=Apple prod=MAC OS typ=EXE → fps=0-day. Following rule shows that Apple generally takes about a week to fix DoS vulnerabilities even if they are exploited on the day they are disclosed: vnd=Apple prod=MAC OS typ=DoS → fps≤1 week. Other rules show that Apple takes about a month after disclosure to patch the EXE vulnerabilities although it is one of the are prevalent types.

Sun is quicker in patching all kinds of vulnerabilities except XSS. Sun fixes DoS vulnerabilities before their disclosure, which is a better performance as compared to Microsoft and Apple. For Mozilla, EXE vulnerabilities are mostly patched on or before the day of disclosure; however, SQL vulnerabilities are not patched for months.

6 Patching vs. Exploitation
In this section, we compare the quickness of vendors with hackers. We study the trends in fps values of vulnerabilities present in the PE-subset.

fps < 0 shows that a vulnerability was patched before its exploitation irrespective of whether or not it was disclosed. The inherent time-lag between the release of patches by vendors and their installation by end-users motivates the hackers to write exploits for vulnerabilities even after corresponding patches have been released. In our PE-subset, 31.7% of all the vulnerabilities fall in this range.

fps = 0 means that a given vulnerability was exploited on the day its patch was released. 21.8% of the vulnerabilities fall in this range.

fps > 0 shows that an exploit for a given vulnerability was released before the vendor patched it. A total of 46.4% of vulnerabilities have fps > 0. The larger percentage of
$t_{pc} > 0$ compared to $t_{pc} < 0$ indicates that hackers have generally been quicker in exploiting the vulnerabilities as compared to vendors in patching. This observation affirms the result of the first $t$-test presented in Section 5.

### 6.1 Patching vs. Exploitation: Over the Years

From Figure 16 we can see the same behavior as observed in Section 5.1: patching response of vendors was poor till around 2005 and a large percentage of vulnerabilities was being exploited before being patched. In 2006, the situation was so bad that the patches for about 38% of the vulnerabilities were released more than a month after their exploitation. However, after 2007 a significant improvement can be observed in the vendor response. It is encouraging to see that since 2008, over 70% of all the vulnerabilities have been patched on or before the release date of their exploits. From the discussion in this section and Sections 4.1 and 5.1, we can conclude that the security state of the software industry has been improving for the last 3 years.

![Fig. 16. Yearly change in patching vs. exploitation trend for $t_{pc}$](image)

### 6.2 Patching vs. Exploitation: Vendors & Products

It can be seen from Figure 17 that for all vendors except Oracle and Sun, the percentage size of the subgroups corresponding to $t_{pc} > 0$ is greater than that for $t_{pc} < 0$. The magnitude of the difference between the percentage sizes of $t_{pc} < 0$ and $t_{pc} > 0$ can serve as a measure to gauge the agility of the vendors in reference to hackers. We can see that among the vendors, only Oracle and Sun are faster than hackers, whereas hackers are, on average, faster than all other vendors. From Figure 18 we can see that, compared to hackers, Microsoft and Sun are quicker for Windows and Solaris respectively.

![Fig. 17. Patched vulnerabilities for vendors relative to exploit dates](image)

### 6.3 Patching vs. Exploitation: CVSS Scores

From Figure 19, it can be seen that for Microsoft and Apple, approximately the same percentage of vulnerabilities belonging to medium and high risk categories are patched before the release of their exploits. However, the percentage

![Fig. 18. Patched vulnerabilities for products relative to exploit dates](image)

![Fig. 19. Patched vulns. relative to exploited vulns.: CVSS score](image)

### 7 Software Diversity

In fault tolerant systems, several redundant components, that can perform the same task independently, are deployed in parallel. The redundancy ensures failure diversity i.e., if some components fail, the system should still be able to perform the required tasks [24]. In case of system security, if all redundant components are instances of same software, a single exploit can compromise them all causing complete system failure. Such situations can be avoided by either using software developed by different vendors or by the same vendor at different points in time. The differences in software code reduce the chances of existence of same vulnerabilities in all redundant components, and thus provide failure diversity.

We study the number of common vulnerabilities in different operating systems as well as in different versions of
the operating systems and determine whether the use of multiple operating systems can provide failure diversity or not.

We focus our attention only on the commonly used operating systems for two reasons. First, nearly all software systems built today rely on these operating systems. At times, supposedly secure systems are compromised not by exploiting vulnerabilities in application software but by compromising a critical component in the operating system. Given the variety of operating systems available, diversity at the operating system level can be a reasonable way of providing good security. Second, our data set contains enough information about vulnerabilities only in the operating systems to make statistically significant observations.

The key challenge in determining the number of common vulnerabilities in multiple operating systems is to automatically identify vulnerabilities that are same. Manually identifying same vulnerabilities is not possible due to the size of the data set. We first describe our automatic technique to identify same vulnerabilities in the data set. Second, we discuss the quantity and types of common vulnerabilities in different combinations of operating systems. Third, we study some trends in common vulnerabilities. Last, we present insights regarding code reuse in multiple operating systems from same vendors.

7.1 Identifying Same Vulnerabilities
Frequently in our aggregate data set, vulnerabilities that are actually the same are reported under multiple CVE-IDs. For example, consider following two vulnerabilities: CVE-2009-0232 and CVE-2010-1883. The text description of CVE-2009-0232 is: integer overflow in the Embedded OpenType (EOT) Font Engine in Microsoft Windows 2000 SP4, XP SP2 and SP3, Server 2003 SP2, Vista Gold, SP1, and SP2, and Server 2008 Gold and SP2 allows remote attackers to execute arbitrary code via a crafted name table, aka “Embedded OpenType Font Integer Overflow Vulnerability”. The text description of CVE-2010-1883 is: integer overflow in the Embedded OpenType (EOT) Font Engine in Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP1 and SP2, Windows Server 2008 Gold, SP2, and R2, and Windows 7 allows remote attackers to execute arbitrary code via a crafted table in an embedded font, aka “Embedded OpenType Font Integer Overflow Vulnerability”. These two text descriptions show that CVE-2009-0232 and CVE-2010-1883 are actually two entries of the same vulnerability. There are numerous other examples of multiple entries of same vulnerability. For example, CVE-2008-2332 and CVE-2010-0043 are two entries for the same vulnerability in Image I/O framework of Apple. Similarly, CVE-2011-0245 and CVE-2011-1374 are two entries for a buffer overflow vulnerability in two different versions of Apple QuickTime.

Occurrence of the same vulnerability in the data set under multiple CVE-IDs happens due to several reasons, the most prominent of which is the discovery of same vulnerability in different software products at different times. For example, for CVE-2009-0232 and CVE-2010-1883, a probable reason behind NVD adding the second entry in the database instead of updating CVE-2009-0232 could be that the disclosure dates of this same vulnerability for Windows 7 and other versions of Windows were different.

To identify and group vulnerabilities that are same, we processed the text descriptions of vulnerabilities in our data set. Specifically, we performed following four steps to identify and group vulnerabilities that are same but reported under multiple CVE-IDs. First, we clustered all vulnerabilities into 7 groups, as explained in Section 3.4. Second, from the vulnerabilities in each cluster, we removed the attributes corresponding to the dominant keywords for that cluster.

We remove the dominant keywords because they exist in the description of majority of vulnerabilities in a given cluster and thus, do not provide useful information to decide whether a pair of vulnerabilities is same or not. For example, in the text descriptions of CVE-2009-0232 and CVE-2010-1883 given above, the words execute, code, and overflow are the dominant keywords and are, respectively, present in the text descriptions of 99%, 96%, and 58% CVE-IDs; so we remove them. It is the common non-dominant keywords in the description of a pair of vulnerabilities that indicate the extent to which the two vulnerabilities are similar because they are common only among the vulnerabilities that are same. In our example, the words EOT, embedded, and opentype are the non-dominant keywords and are, respectively, present in the text descriptions of only 0.1%, 0.8% and 0.15% CVE-IDs.

Third, for each cluster, after removing the attributes corresponding to dominant keywords in that cluster, we agglomerated the vulnerabilities with Ward’s Linkage [15] and Jaccard Coefficient [14] to obtain a dendrogram. The dendrogram obtained from agglomeration facilitates grouping the vulnerabilities that have no less than a desired percentage of common non-dominant keywords. Fourth, we cut each dendrogram at a height of $J = 0.05$ resulting in small sub-clusters, where each non-singleton sub-cluster has at least 95% common non-dominant keywords. Therefore, each sub-cluster contains vulnerabilities that are actually the same but reported under different CVE-IDs. All examples that we have given of vulnerabilities reported under multiple CVE-IDs were found in the database using these four steps.

7.2 Common Vulnerabilities in Operating Systems
After identifying and grouping the vulnerabilities that are same, we counted number of common vulnerabilities among different operating systems. Specifically, we studied the following four categories of operating systems: Windows, Linux, Mac, and Solaris. For Windows, we studied Windows 98, 98SE, 2000, ME, XP, 2003 Server, Vista, 2008 Server, and Windows 7. We don’t consider Windows 2012 Server and Windows 8 because the number of reported vulnerabilities for them are yet too few to make any statistically significant observations. For Linux, we studied Enterprise, Fedora, and Ubuntu. For Mac we studied OS X and OS X Server. We also studied Solaris and OpenSolaris.

Tables 3 and 4 give several stats about common vulnerabilities among different operating systems. The second column in each table shows several combinations of operating systems along with the total number of vulnerabilities disclosed for each operating system. We have placed different combinations in the table such that the operating systems that belong to the same category lie next to each other and in chronological order wherever possible. We found 33 pairs of
operating systems with at least ten common vulnerabilities and 84 pairs with at least one common vulnerability. Due to space limitations, we do not list all of them in Table 3. The third and fourth columns in the tables show the total number of common vulnerabilities and the percentage of common vulnerabilities, respectively. In the next nine columns, each set of three consecutive columns gives the number of common vulnerabilities for the three CVSS metrics: Access Vector, Access Complexity, and Integrity Impact. Recall from Section 2.1 that each of these three CVSS metrics has three values. The last four columns in Table 3 give percentages of different types of common vulnerabilities. We do not show percentages for PHP and SQL vulnerabilities because they are all 0 for all pairs of operating systems.

From these two tables, our general observation is that the use of multiple operating systems from the same vendor will not provide good failure diversity because they contain a lot of common vulnerabilities between different releases. We observe this trend primarily in Mac and Windows. For example, there are 530 vulnerabilities that are common in Mac’s consumer version and server version. Similarly, in almost all consecutive release pairs of Windows, the percentage of vulnerabilities that are common is quite large. For different flavors of Linux, however, the percentage of vulnerabilities that exist in all of them or in each pair is low (less than 20%). A possible reason is that vendor of each flavor of Linux not only customizes its GUI but also the Linux kernel.

Our analysis reveals that to implement failure diversity, ideally one should deploy operating systems from multiple vendors. This is because there are hardly any common vulnerabilities in operating systems from different vendors. For example, there is only one common vulnerability in Windows XP and Solaris. Similarly, there is only one common vulnerability in Mac OS and Windows Vista. Deploying operating systems from multiple vendors may, however, not be feasible in some scenarios such as when the application that has to be run is available only for one category of operating systems. In such a scenario, Tables 3 and 4 can be used to identify an appropriate combination of operating systems that will provide best failure diversity for given requirements. For example, if the requirement is to deploy two operating systems that provide robustness against DoS attacks, a good choice would be Windows XP and Windows 7 because these two operating systems have least percentage of common DoS vulnerabilities. Similarly, if the requirement is to deploy two Windows based operating systems, that are resilient against network based attacks, on a server that is located in a physically secure location, a good choice would be Windows 2008 Server and Windows 7 because although they have high percentage of locally exploitable common vulnerabilities, they have lowest percentage of remotely exploitable common vulnerabilities.

Another interesting observation from Table 3 is that the majority of the common vulnerabilities are either EXE vulnerabilities or DoS vulnerabilities. A possible reason for the non-existence of remaining four types of vulnerabilities can be that they are more related to internet usage on web browsers or applications such as data bases, and do not depend on the operating system that is hosting those applications.

### 7.3 Trends in Vulnerabilities Reported Multiple Times

Next, we study how close are the multiple CVSS scores of the same vulnerability that is reported multiple times under different CVE-IDs. We also study how far apart in time multiple entries of the same vulnerability lie.

We observed that for majority of vulnerabilities that are reported multiple times, the difference between the CVSS scores in their corresponding multiple entries is 0. Figure 20(a) shows the difference between CVSS scores in multiple entries of the same vulnerabilities. The horizontal axis shows the difference in CVSS scores between multiple entries and the vertical axis shows the fraction of vulnerabilities corresponding to the difference values. Recall from Section 2.1 that CVSS score lies in the range $[0, 10]$. We observe from Figure 20(a) that for over 90% of vulnerabilities that are reported multiple times in the database, the difference between CVSS scores in multiple entries is no greater than 2. This shows that CVSS is a well designed system that assigns consistent scores to multiple entries of the same vulnerabilities, even when the entries are separated by several years.

![Figure 20](image)

**Fig. 20.** Difference in values for some vulnerabilities reported under multiple CVE-IDs

Figure 20(b) shows the difference between dates in multiple entries of the same corresponding vulnerabilities. The horizontal axis shows the difference in disclosure dates between multiple entries and the vertical axis shows the number of vulnerabilities corresponding to the difference values. For approximately 50% of the vulnerabilities, subsequent entries are reported within a year of the first entry. For approximately 90% of the vulnerabilities, subsequent entries are reported in less than 5 years from the first entry. Most of the vulnerabilities that exceed 5 years belong to products for which new versions are released frequently such as Apple QuickTime and Adobe Shockwave.

### 7.4 Code Reuse

The existence of multiple common vulnerabilities in consecutive releases of the same category of operating systems shows the reuse of code from the previous release of the operating system into the next release. For example, in case of Windows, there are 36 vulnerabilities that are common in Windows 98, 98SE, 2000, and XP. One such vulnerability is CVE-2004-0123, which is a vulnerability in the abstract syntax notation library, ASN.1, used by these four operating systems. Therefore, although Windows XP was completely redesigned, we see common vulnerabilities between it and Windows 98 due to the reuse of such libraries. Similarly, in Windows XP, 2003 Server, Vista, 2008 Server, and Windows 7, there are 127 common vulnerabilities, which shows that the code from Windows XP was carried forward to
Windows 2003 Server to Windows Vista to Windows 2008 Server and finally to Windows 7. One such vulnerability is CVE-2009-0091, which is a vulnerability in Microsoft .NET framework. As all these operating systems use .NET framework, they all carry this common vulnerability. As before, although Windows 7 was redesigned, we see common vulnerabilities in all releases from Windows XP to Windows 7 due to use of such common code. Similar trends exist in the products of other operating system vendors.

While the code reuse itself is not harmful, the observation that same vulnerabilities exist in multiple releases shows that vendors do not thoroughly reanalyze the legacy code for potential security threats by using up-to-date security verification techniques before using it in next releases.

### 8 IMPLICATIONS

Observations from our study have important implications in software design, development, deployment, and management. We discuss them below.

#### 8.1 Software Design

The analysis of access requirements, risk level, and functionality of vulnerabilities presented in Sections 3.3, 3.2, and 3.4 respectively, can reveal inherent flaws in software design process for specific products and vendors. For instance, if a particular software series has more than the typical number of instances of buffer overflow vulnerabilities, then this may reflect a lack of sanity checks in socket read processes. From our data set, we observed that DoS is the most exploited vulnerability type in Solaris accounting for 38.85% of all its vulnerabilities. At the same time, only 11.7% of vulnerabilities in OS X involve DoS, which shows that Solaris is more susceptible to DoS attacks compared to OS X. The observation mentioned above implies that Solaris developers need to take additional steps to make the design more robust to DoS attacks.

#### 8.2 Code Development Practices

The analysis of vulnerability life cycles during the evolution of a given software can reveal insights about potential flaws in its code development and testing practices. In particular, a correlation analysis of count of vulnerabilities across different software and vendors can highlight important differences in code development practices. For instance, we observe in Figure 13 that the percentage sizes of the subgroups corresponding to \( t_{pub} > 0 \) for open-source vendors (Linux, Redhat) are significantly greater than those of closed-source vendors (Microsoft, Apple). This observation highlights an important insight into the code development practices of open-source vendors which typically rely on contributions from a group of volunteer developers. On the other hand, closed-source vendors have dedicated resources to fix newly disclosed vulnerabilities as soon as possible. Therefore, open-source vendors tend to have a slower patch response compared to closed-source vendors. Our work complements the work done by Shin et al. in using different metrics to predict software vulnerabilities [25].
8.3 Customers’ Assessment of Software
The analysis presented in this paper also has direct implications in product assessment, certification, and security recommendations to consumers. Several commercial products e.g. eEye Digital Security (http://www.eeye.com) and Arellia (http://www.arellia.com/), can leverage the presented analysis for product recommendation and design of future security policies. For example, given that the exploits of vulnerabilities have already been released, our measurement analysis showed that Sun releases patches for 96% of the vulnerabilities within a month; whereas, Microsoft, Apple, and Linux provide patches for only 69%, 74%, and 65% of vulnerabilities in the same time period. Therefore, if the patch response of vendor is of prime importance to a customer, then the products from Sun should be preferred. As another example, if a customer’s infrastructure has less security policies. For example, given that the exploits of vulnerabilities have already been released, our measurement analysis showed that Sun releases patches for 96% of the vulnerabilities within a month; whereas, Microsoft, Apple, and Linux provide patches for only 69%, 74%, and 65% of vulnerabilities in the same time period. Therefore, if the patch response of vendor is of prime importance to a customer, then the products from Sun should be preferred. As another example, if a customer’s infrastructure has less tolerance for DoS attacks, then it is more suitable to deploy Mac OS X, which has the lowest percentage of DoS vulnerabilities compared to other operating systems. Likewise, if a customer requires more robustness to buffer overflow attacks, then it is more suitable to deploy Solaris because buffer overflow vulnerabilities account for about 20% of all the vulnerabilities in Windows and Mac but only 13% in Solaris.

9 RELATED WORK
The major focus of the work on large scale analysis of vulnerabilities has been on the development of vulnerability discovery models (VDMs). Some work has also been done to understand the economic impacts of vulnerability disclosures in software. We briefly describe the work that has been done in these areas in relation to our work.

9.1 Large Scale Vulnerability Analysis
The work most relevant to ours is that of Frei et al. [3], who presented a large scale study of the discovery, disclosure, exploit, and patch dates of vulnerabilities. They analyzed about 14000 vulnerabilities and showed that till 2006, the hackers were quicker than vendors. This observation is in accordance with what we presented in this paper but we also show that in the last three years, the response of vendors has been improving. Frei et al.’s work does not differentiate between vendors and types of vulnerabilities.

In [26], studies the life-cycle of vulnerabilities from the time a software is released till the time the first vulnerability is discovered. They show that the time till the discovery of the first vulnerability is a function of the hackers’ familiarity with the system and the amount of legacy code. In [27], the authors propose to use semantic templates to help the developers understand the vulnerabilities and their artifacts. This work only focuses on understanding the technical details of a disclosed vulnerability and does not study any large scale trends in vulnerabilities.

9.2 Studies on Disclosure and Patching
In [28], authors have studied the economic aspects of the quickness of vendors in releasing patches for Internet based vulnerabilities. In [29], authors show that on average a vendor loses 0.6% of the stock price with the disclosure of a vulnerability. In [8], authors show that a vendor with more competitors patches the vulnerabilities more quickly. In [7], they show that the vulnerability disclosure accelerates the patch release. Although their work is based upon a small data set of just 354 vulnerabilities disclosed till 2003, they observe, like us, that the closed-source vendors are quicker in patching the disclosed vulnerabilities. These studies, however, do not develop any insight into the individual behaviors of vendors and hackers.

In [30], using a small data set, authors make a claim that there is no difference between the patching behavior of open and closed-source vendors.

They make this conflicting observation because they only consider the percentage of patched vulnerabilities as a measure of goodness of a vendor. This is unreasonable because without analyzing the duration between disclosure dates and patch dates, one can not determine how active a vendor is in fixing vulnerabilities in its products.

9.3 Modeling and Classification
The motivation behind the work on VDMs is to enable the prediction of quantity and timing of vulnerability discoveries in new software. Four notable VDMs have been proposed: (1) Anderson Thermodynamic Model [2], (2) Rescorla Linear Model [6], (3) Rescorla Exponential Model [6], and (4) Alhazmi-Malaiya Logistic Model [5]. Another work focused on modeling the time interval between disclosure date of vulnerabilities and their corresponding exploit, patch, and discovery dates [31]. A recent work by Bozorgi et al. extracted various features from NVD and OSVDB and used machine learning techniques to predict whether a recently disclosed vulnerability will be exploited within a given time or not [32]. Our focus, however, is not the prediction rather the study of phases of vulnerability life cycle in reference to different variables along with several aspects associated with the nature of vulnerabilities.

9.4 Software Diversity
In [9], Garcia et al. studied whether operating system diversity is feasible or not. They also observed that for some combinations of operating systems, the number of common vulnerabilities were few. Although comprehensive, their work lacks two aspects that we have studied. First, they did not consider the types of vulnerabilities (such as EXE, DoS etc.) that are common in multiple operating systems and can thus not be used to determine the best pair of operating systems that provides robustness against a certain type of vulnerability. Similarly, they did not study the access complexity and integrity impact of common vulnerabilities. Second, they performed their study on only 1887 vulnerabilities and did not do text processing to identify common vulnerabilities. In contrast, we used all vulnerabilities reported until April 2013 for the four categories of operating systems.

10 CONCLUSION
In this paper, we presented a large scale study of various aspects associated with software vulnerabilities during their life cycle. We aggregated a large software vulnerability data set containing 56077 vulnerabilities disclosed till 2013. Our study showed that the number of vulnerabilities being disclosed every year stopped increasing between 2008
and 2012. We showed that the most primitive and most exploited form of vulnerabilities are DoS and EXE; however, the numbers of SQL, XSS, and PHP vulnerabilities have increased significantly. We also observed that the percentage of remotely exploitable vulnerabilities has gradually increased to over 80% of all the vulnerabilities. Since 2008, vendors have been becoming increasingly agile in patching the vulnerabilities and the access complexity of vulnerabilities has been increasing. Our findings also highlighted that patching of vulnerabilities in closed-source software is faster compared to open-source software and at the same time the exploitation is slower. Our study also shows that the use of different versions of operating system from the same vendor does not provide good failure diversity. The best failure diversity can be achieved by using operating systems from different vendors.

REFERENCES


