The Life and Death of Statically Detected Vulnerabilities: an Empirical Study

Massimiliano Di Penta, Luigi Cerulo, Lerina Aversano
Dept. of Engineering, University of Sannio, Benevento, Italy

Abstract

Vulnerable statements constitute a major problem for developers and maintainers of networking systems. Their presence can ease the success of security attacks, aimed at gaining unauthorized access to data and functionality, or at causing system crashes and data loss. Examples of attacks caused by source code vulnerabilities are buffer overflows, command injections, and cross-site scripting.

This paper reports on an empirical study, conducted across four networking systems, aimed at observing the evolution and decay of vulnerabilities detected by three freely available static analysis tools. In particular, the study compares the decay of different kinds of vulnerabilities, characterizes the decay likelihood through probability density functions, and reports a quantitative and qualitative analysis of the reasons for vulnerability removals. The study is performed by using a framework that traces the evolution of source code fragments across subsequent commits.

Key words: Software Vulnerabilities; Mining Software Repositories; Empirical Study

1. Introduction

Vulnerable instructions are, very often, the cause of serious problems such as security attacks, system failures or crashes. In his PhD thesis [1] Krsul defined a software vulnerability as “an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy”. For business-critical systems, the presence of vulnerable instructions in the source code is often the cause of security attacks or, in other cases, of system failures or crashes. The problem is particularly relevant for any system that can be accessed over the Internet: e-banking or e-commerce systems, but also networking utilities such as Web proxies or file sharing systems, and of course Web servers. All these systems can be attacked from hackers with the objective of getting unauthorized access to system or data, or simply to cause denial of services or data loss. The number of attacks caused by some kinds of vulnerabilities is scaring: it has been reported by CERT that statements vulnerable to buffer overflows are the cause of 50% of software attacks. Recent studies report an increasing trend in terms of other kinds of vulnerabilities, specifically cross-site scripting and SQL injection. In other cases, even when no attack is performed vulnerability can cause system failures/crashes, which can be a considerable risk for safety-critical systems.

Detecting the presence of such instructions is therefore crucial to ensure high security and reliability. Indeed, security advisories are regularly published—see for example those of Linux distributions (www.debian.org/security and www.redhat.com/security) and those published by CERT, or by securityfocus. These advisories, however, are posted when a problem already occurred in the application, a problem that was very often caused by the introduction in the source code of vulnerable statements. This highlights the needs to identify potential problems when they are introduced, and to keep track of them during the software system lifetime, as it is done, for example for source code clones [2].

A number of automatic tools have been developed for the identification of potentially vulnerable source code statements. Most of these tools rely on static source code analysis performed in different ways: some tools merely use pattern matching e.g., with the aim of identifying programming language functions that are known to be vulnerable, while others perform a more accurate analysis, including data-flow analysis. Although

---

1 www.cert.org
2 http://cwe.mitre.org/documents/vuln-trends/index.html
3 wwwdebian.org/security and www.redhat.com/security
4 www.microsoft.com/technet/security/advisory/default.aspx
5 www.securityfocus.com

Preprint submitted to Elsevier

March 14, 2009
several vulnerability detection tools exist and their effectiveness has been assessed by tool developers, up to now the literature lacks of studies aimed at analyzing how the presence of vulnerabilities varies during a software system lifetime, i.e., to what extent new vulnerabilities tend to be introduced when new code is added, and to what extent during the time developers modify the system to protect it against vulnerability attacks. Nowadays, the availability of code repositories for many open source systems, of techniques for integrating data from versioning systems—Concurrent Versions Systems (CVS) or SubVersion (SVN)—of bug/tracking systems, and of techniques to perform software historical analyses [3, 4], forms the basis for this kind of study.

This paper describes an approach, relying on the use of an enhanced differencing tool [5, 6], to track the evolution of vulnerable statements (from here simply referred as “vulnerabilities”) as detected by static vulnerability detection tools. Then, it reports an empirical study on the evolution of vulnerabilities detected in three network applications, namely a Web caching proxy (Squid), a file/printer service (Samba), and a Web application framework (Horde), the first two developed in C and the third one in PHP. Specifically, the study aims at investigating:

- how the presence of vulnerabilities varies over the time, in particular whether their density exhibits a particular trend or whether a massive vulnerability introduction and removal can be observed in particular contexts, e.g., in proximity of a new release;
- how fast vulnerabilities belonging to different categories decay, and whether such a decay follows any particular statistical distribution;
- how vulnerabilities tend to be removed during the time, i.e., whether they disappear because the vulnerable source code line is removed or changed, or whether they disappear even if the vulnerable line remains in the system.

Also, we analyze to what extent the vulnerability was documented by developers in CVS/SVN commit notes, and repeat the above analyses to documented vulnerabilities only.

The paper is organized as follows. After a discussion of related work, Section 3 introduces the static vulnerability detection tools we used in this study, and overviews a taxonomy of vulnerabilities we investigate. Section 4 describes the process followed to detect vulnerabilities and track their evolution. Section 5 describes the empirical study definition, the context, and outlines the study research questions. Results are reported and discussed in Section 6, while threats to validity are discussed in Section 7. Finally, Section 8 concludes the paper and outlines directions for future work.

2. Related Work

Previous work related to ours falls in the following categories: (i) identification of vulnerabilities in source code, (ii) defects prediction and prevention, and (iii) maintenance of vulnerable code.

2.1. Identification of vulnerabilities in source code

Prediction of software failures and vulnerabilities has been faced with a wide number of approaches, based mainly on static and dynamic analyses. Static tools detect the usage of potentially vulnerable functions, perform integer range analysis and track the string manipulation operations. Among the existing tools, it is worth mentioning: FlawFinder7, Rats8, ITS48, and Splint9. Such tools attempt to locate potential vulnerabilities, in particular buffer overflows, based on a lexical analysis of the program. The main weakness of static tools is that they are rather imprecise, since the problem of statically detecting buffer overflows is, in general, undecidable. Recently, Cifuentes et al. [7, 8] proposed a precise vulnerability identification tool (Parfait), able to outperform freely available tools, such as Splint, in terms of both precision and recall.

DaCosta et al. [9] proposed an approach to evaluate the security vulnerability likelihood of a function based on the assumption that a function near a source of input may have a high probability to being vulnerable.

A wide number of research works is related to buffer overflow detection and prevention. Many other tools are available, for example Purify (Hastings and Joyce, 1998), a memory debugger program used by software developers to detect memory access errors in programs, especially those written in C or C++. Ruwase and Lam [10] proposed an approach and a tool to prevent and detect buffer overflows.

Traditional testing strategies provide a marginal contribution to generate test cases capable to actually discover buffer overflows. To this aim, specific approaches have been developed. Korel and Al-Yami [11] generated test cases that violate some assertion conditions,

---

6http://www.dwheeler.com/flawfinder
7http://www.securesoft.com/rats.php
8http://www.cigital.com/its4
9http://www.splint.org
while Tracey et al. [12] used genetic algorithms and simulated annealing to generate test data with the purpose of exercising the exception handling. The evaluation of these approaches, however, was limited to 200 LOC programs. Del Grosso et al. [13] showed how program slicing and data dependency analysis can be effectively used to reduce the genetic algorithm search space to determine whether exists a cause-effect relationship between program inputs and variables used in likely dangerous statements. Dahn and Mancoridis [14] presented a tool which uses TXL, a program transformations language, to secure C programs against run-time stack buffer overflows by transforming stack allocated arrays into heap allocated arrays automatically at compile time. Wang et al. [15] detected buffer vulnerabilities in both stack and heap memory as well as potential buffer overflows in library functions by runtime checking through using automatically inferred assertions and program transformation.

SQL-injection is another practice of attack which exploits weak validation of textual input used to build database queries. Merlo et al. [16] combined static analysis, dynamic analysis, and code reengineering to automatically protect applications written in PHP from SQL-injection attacks. Huang et al. [17] detected Web application vulnerabilities by using dynamic analysis approaches, while Scott and Sharp [18] proposed a proxy-based approach to prevent cross site scripting attacks.

2.2. Defects prediction and prevention

Studies related to defect prediction and prevention usually focus on software defects which include also software vulnerabilities. We share with them the analysis method and the measurement approaches.

Mockus et. al. [19] used a fine-grained analysis to predict defect correction effort and the time-interval for which such an effort is needed. Similarly to us, they use a fine-grained analysis approach, although we focus on vulnerability decay rather than on defect removal effort. Previous works [20, 21, 22] found that Weibull and exponential distributions capture defect-occurrence behavior across a wide range of systems. We find that such distributions are suited to model vulnerability decay. Calzolari et al. [23] used the predator-prey model borrowed from ecological dynamic system to model maintenance and testing effort. They found that, when programmers start to correct code defects, the effort spent to find new defects has an initial increase, followed by a decrease when almost all defects are removed.

Kim and Ernst [24] proposed an algorithm to prioritize the fixing of warnings detected by tools such as FindBugs, Jlint and PMD. We share with them the idea of using historical information to analyze how warnings—and vulnerabilities in our specific case—are fixed. However, Kim and Ernst focused on warnings removed by bug-fixing directly affecting source code lines containing the warning itself. They found that bug fixings represent a very small percentage of warning/vulnerability removals. In our paper, we focus, instead, on vulnerabilities detected by specific analysis tools, and show how they disappear since other source code lines are changed, e.g., to “protect” the vulnerability.

2.3. Maintenance of vulnerable code

Software repositories allow researchers to perform empirical observations on the evolution of worth noting software entities, such as clones [25, 26] and design patterns [27]. Beside many approaches aimed at identifying vulnerabilities in source code with both static and dynamic analyses, very few studies have been performed about how such vulnerabilities evolve within the software system.

Neuhaus et al. [28] found a correlation between vulnerabilities and #include directives for functions called in C/C++ source files of Mozilla, by performing a historical analysis of its CVS repositories. There are similarities in their analysis process and ours, although our study is exploratory rather than predictive.

Alhazmi et al. [29] modeled the vulnerability discovery process in both commercial and open source operating systems to predict future trends. The main difference with our work is twofold: i) we consider vulnerabilities detected by tools and not those discovered by testing and/or reporting tasks; ii) we do not consider when a vulnerability is discovered but when it is introduced and when it disappears from the system.

Ozment et al. [30] examined the code base of the OpenBSD operating system to determine whether its security is increasing over time, finding that what they classify as foundational vulnerabilities have a median lifetime of at least 2.6 years. We perform a similar study with the following differences: i) we consider three systems from different domains ii) we consider vulnerabilities detected by tools and not those reported in security bulletins; iii) we perform the analysis at snapshot level rather than at release level which allows us to model the decay as a probability distribution function, finding, for example, that the median lifetime of Buffer Overflows is, for the systems we analyzed, shorter than what they found, i.e., in all cases less than one year; iv) we compare the decay time of different kinds of vulnerabilities and model the vulnerability decay probability;
v) we analyze the context (line removal, direct change or co-change) where a vulnerability was removed, and whether developers provided evidence of such a removal in the commit notes.

Li et al. [31] studied how the number of defects related to security issues evolve over time in Mozilla and Apache systems. They consider security issues appearing in different software components, and found that some classes of security issues, such as memory problems, are today less dominant than some decades ago. Our study considers a larger number of vulnerability classes, and as mentioned above performs a thorough fine grained analysis, at source code line level instead of at component level.

In a companion paper [32] we presented a preliminary version of this work. The present paper extends the previous one by analyzing how and in what context vulnerabilities are removed, and to what extent developers document such removals.

3. Detection of Vulnerable Code

For our analyses, we selected three freely available static analysis tools, Splint, Rats, and Pixy\footnote{http://pixybox.seclab.tuwien.ac.at/pixy}, with the aim to increase the range of vulnerability categories that we can detect, and the set of programming languages we can analyze. In the following we report a short description of these three tools and the categories of vulnerability each tool can detect.

Splint, previously known as LCLint, is an open source static analysis tool for ANSI C programs. It uses static analyses to identify code vulnerabilities, including data flow analysis and type checking/inference. Splint employs heuristics that are effective but not complete as it trades off precision for scalability. It implements flow sensitive control flow, merging possible paths at branch points, recognizes loop idioms and determines loop bounds. A study performed on the wu-fptd FTP server [33] revealed that over 101 vulnerabilities detected by Splint, 26 turned out to be real ones.

Rats (Rough Auditing Tool for Security) is an open source tool able to analyze source code written in C, C++, Perl, PHP and Python. It is derived from ITS4 [34], and performs a rough source code analysis based on pattern matching, like the Unix grep tool. More precisely, Rats scans for well-known potentially vulnerable functions, such as scanf, gets, sprintf, strcpy and strcat; however, it does not perform flow analysis to check whether the vulnerable statement is protected in some way. Thus, on the one side the tool is very scalable and able to analyze several languages; on the other side, it can just tell that some particular instructions are used, without analyzing their context: for example Rats detect instructions vulnerable by potential Buffer Overflows, without analyzing whether buffer boundaries are checked somewhere. Also, Rats identifies potential determinism in random number generation functions, without checking how the seed is initialized. Clearly, this would reduce the precision of results [35]. On the other hand, as explained in [34], although to some extent ITS4 and Rats behave like grep, they are able to reduce the number of false positives of grep using a series of heuristics, for example checking whether buffer handling functions (e.g., strcpy) are handling constant strings, or using heuristics to detect race conditions.

Pixy is able to detect cross site scripting (XSS) and SQL injection vulnerabilities in PHP code. It performs a data flow analysis to detect whether it is possible that tainted data reaches sensitive sinks without being properly sanitized. Studies performed on systems with known vulnerabilities reported a false positive rate is about 50% [36].

It is important to mention that percentages of false warnings\footnote{In the following we will use the term “false warning” as this is the term used in literature, e.g., in [33], and more appropriate than “false positive”} reported by the existing literature are mostly based on documented vulnerabilities only. On the one side, in this paper we share this approach and perform a separate analysis on documented vulnerabilities. On the other side, documented vulnerabilities are far to cover all potential vulnerabilities that can occur in a software system; for this reason we also focus on all those instructions that, according to clichés or heuristics defined in a tool, are classified as potentially vulnerable. They represent, in fact, a set of potential problems developers have to deal with.

Table 1 shows the different vulnerability categories that can be detected by the three tools adopted for our study. In the Table we also provide a brief description of each category, organizing them into four groups: Input validation, Memory safety, Race/Control flow condition and Other. The classification is largely inspired to what is reported in tool user manuals and on the Krusl’s PhD thesis [1], which classifies vulnerabilities as related to the adopted exploitation technique.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Category} & \textbf{Sub-category} & \textbf{Description} & \textbf{Tool(s)} \\
\hline
Input validation & Input validation & \\
\hline
Memory safety & Memory safety & \\
\hline
Race/Control flow condition & Race/Control flow condition & \\
\hline
Other & Other & \\
\hline
\end{tabular}
\caption{Vulnerability Categories}
\end{table}
4. Data Extraction

This section describes the process we use to extract data necessary for the analysis of source code vulnerability evolution. The data extraction process consists of the sequence of five steps reported below. The collected information constitutes the raw data\(^\text{1}\) for the empirical analysis.

4.1. Step I: snapshots extraction

Since we are interested to perform a fine-grained analysis of vulnerability evolution, we look at changes committed in the Concurrent Versions Systems (CVS)/SubVersion (SVN) repository rather than looking at releases of the software system. In particular, we rely on a technique [3] that considers the evolution of a software system as a sequence of Snapshots \((S_1, S_2, \ldots, S_m)\) generated by a sequence of Change Sets, representing the changes performed by a developer in terms of added, deleted, and changed source code lines. Change sets can be extracted from a CVS/SVN history log using various approaches. We adopt a time-windowing approach, that considers a change set as sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less than or equal to 200 seconds [4].

4.2. Step II: differences identification and line tracing

To analyze the evolution of vulnerable source code lines over snapshots, we need to identify whether a change committed in the CVS/SVN consists in the addition of new lines, in the removal of existing lines, and/or in the change of existing lines. To this aim, we use an approach that identifies the differences between two versions of a source file detailed in [5, 6]. Specifically, such an approach identifies the set of changed, added and deleted source code lines between two subsequent snapshots by using \texttt{ldiff}\(^\text{2}\), an improved \texttt{diff} algorithm. The tool combines the Levenshtein string edit distance with the vector space models cosine similarity to determine whether a line has been changed or whether, instead, a change consists in the removal of an old line and in the addition of a new one. Also, with respect to the Unix \texttt{diff}, \texttt{ldiff} is able to better identify cases where a code fragment or a line was moved upward or downward in a file. With such an information we can trace the line evolution among subsequent versions of a source code file with a high (over 90%) precision.

4.3. Step III: identification of vulnerable source code lines

Once we have identified snapshots (Step I) and traced the evolution of source code lines (Step II), we need to

\(^1\)http://www.rcost.unisannio.it/mdipenta/vuln-rawdata.tgz

\(^2\)http://rcost.unisannio.it/cerulo/tools.html
identify, for each snapshot, the set of source code lines that, according to the tools described in Section 3, contain vulnerabilities. To this aim, we run the vulnerability detection tools on the set of files that, on each snapshot, have been changed. The output of this step is, for each snapshot and for each source code file modified in the snapshot, the list of vulnerable source code lines with a vulnerability description as extracted by the tool, and a vulnerability classification according to the taxonomy of Section 3. In most cases (e.g., for Splint) the output of the tool consists in a detailed vulnerability description: in such cases the classification is performed by means of a Perl script that analyzes the tool output and classifies each vulnerability by means of a pattern-matching on the vulnerability description.

4.4. Step IV: determining vulnerability changes among snapshots
The last step of this process is to trace a vulnerability across snapshots. In particular, by combining the information extracted at Step II (source code line tracing across snapshots) with the information extracted at Step III (vulnerable source code lines for each snapshot), we are able to determine:

- when (in which snapshot) a vulnerability appears in a source code line for the first time;
- whether a vulnerability detected on line $l_i$ of file $f_j$ in snapshot $s_k$ belongs to the same category of the vulnerability identified on line $l'_j$ in snapshot $s_k$, where $l_j$ in $s_k$ corresponds to $l_i$ in $s_{k+1}$ according to the analysis of Step II;
- when a vulnerability is removed from the system.

Vulnerability removals happen for two reasons. First, a source code line containing a vulnerability can be removed. Second, although the line has not been removed, the vulnerability is not detected anymore, either because of a change occurred on the line itself, or because of a change occurred somewhere else. In the following, we will refer to:

- (R)emoved vulnerability: when a vulnerability is not detected anymore in the system from a given snapshot because the vulnerable line has been removed;
- (D)isappeared vulnerability: when a vulnerability is not detected anymore in the system from a given snapshot although the vulnerable line was not removed (according to $ldiff$), however either that line was changed, or other lines were changed to “protect” the vulnerability;
- $R \cup D$: to consider both.

Without loss of generality and to avoid an overkilling notation, in the following we simply refer to $R \cup D$ as “removed” vulnerability, except for research questions $RQ4$ and $RQ5$, where we will distinguish (R)emoved from (D)isappeared.

4.5. Step V: analyzing documentation of vulnerability removal/disappear
To better understand in what context a vulnerability was removed or disappeared, we inspected all the commit notes of change sets where at least one vulnerability was removed. The inspection was performed by two independent reviewers, who classified whether the change was to a vulnerability removal or not. Every time a disagreement occurred in the classification, a third person was involved in the analysis and a discussion was held to resolve the conflict. Vulnerabilities for which evidence was found in change notes are marked as “documented” and will be subject to separate analyses, as for them we have higher confidence that they are not false warnings.

5. Empirical Study Definition
The goal of this study is to perform a fine-grained analysis on the evolution of vulnerable source code lines as detected by static analysis tools. The purpose is to determine whether different vulnerabilities exhibit a particular evolution trend over the time, and how long vulnerabilities tend to remain in the system. The quality focus is the software system reliability and security, which can be affected by these statements. The perspective is of researchers, aimed at understanding how potentially vulnerable instructions in a system are maintained.

5.1. Context description
The context of this study deals with analyzing the evolution of vulnerable source code lines in three open source systems, namely Squid, Samba, and Horde. Squid\textsuperscript{14} is a Web caching proxy, written in ANSI C, supporting HTTP, HTTPS, and FTP. Samba\textsuperscript{15} is a software suite, written in ANSI C, that provides file and print services and allows for interoperability between Linux/Unix servers and Windows-based clients. Horde\textsuperscript{16} is a general-purpose Web application framework written in PHP, providing classes for dealing with

\footnotesize{\textsuperscript{14}http://www.squid-cache.org
\textsuperscript{15}http://www.samba.org
\textsuperscript{16}http://www.horde.org}
preferences, compression, browser detection, connection tracking, MIME handling, and more. Table 2 reports, for the three systems, the number of snapshots, the range of analyzed releases, the minimum-maximum non-commented KLOC (NKLOC), and number of source code files.

### 5.2. Research Questions

The research questions this study aims at answering are the following:

- **RQ1**: How are detected/removed vulnerabilities distributed across categories? This research question discusses the number of different vulnerabilities detected in the three systems, to understand how systems are prone to different kinds of vulnerabilities. Also, the research question discusses, for each category, the number of vulnerabilities that are removed from the system during the period of observation. In addition, we classify cases for which, when the vulnerability was removed, developers documented it in the commit note. Whenever necessary, we repeat the analysis of the following research questions on the documented vulnerabilities subset.

- **RQ2**: How does the number of vulnerabilities vary during the time? This research question analyzes the evolution of the vulnerability density defined as the number of vulnerabilities per NKLOC, investigating whether the density of a particular vulnerability category exhibits any positive or negative trend. This is useful to identify trends that can be potentially dangerous, i.e., increasing density of vulnerabilities without any subsequent decrease, as well as to highlight vulnerability removal activities or changes introducing a high number of vulnerabilities.

- **RQ3**: How long vulnerabilities tend to remain in the system? This research question analyzes the time interval between the introduction and removal of a vulnerability, i.e., the vulnerability decay time. Specifically, it investigates whether vulnerabilities belonging to different categories have a significantly different decay. Although also in this case the intention of the developers might not be always known, such information can highlight whether particular kinds of vulnerabilities tend to be removed quicker than others, either because they are easier to spot or because they are deemed to be more dangerous. Moreover, this research question investigates whether the decays of different vulnerability categories follow a specific probability distribution. This represents the likelihood a vulnerability has to be removed from the system in our interval of observation.

- **RQ4**: How are vulnerabilities removed? This research question performs a deeper investigation of vulnerabilities that, according to **RQ1**, are removed from the system. In particular, it distinguishes (i) cases where the vulnerable line is deleted; (ii) cases where the vulnerability disappears because its source code line is modified, and (iii) cases where the vulnerable line is left unchanged because other lines have been changed, “protecting” the vulnerability. It is important to note that the second and third case are not disjoint, i.e., a vulnerability disappears because of a change on its source code line or of changes occurring elsewhere.

### 5.3. Analysis Method

In this section we describe how we analyze the extracted data to answer the research questions described above. For all the statistical tests we perform, we assume a significance level of 95%. Statistical analyses are performed using the R tool\(^1\).

For **RQ1**, we mainly report data of the overall number of different vulnerabilities detected and those that were removed. We use a proportion test to check whether the proportion of removed vulnerabilities vary across categories (\(H_0\): there is no difference among proportions). For each kind of vulnerability, we also compute the odds ratio (OR), a measure of effect size for dichotomous, categorical data. An odds [37] indicates how much likely is that an event will occur as opposed to it not occurring. Odds ratio is defined as the ratio of the odds of an event occurring in one group (e.g., experimental group, in our case the set of removed vulnerabilities) to the odds of it occurring in another group (e.g., control group, in our case the set of vulnerabilities that remained alive). If the probabilities of the event in each

---

1. [http://www.r-project.org](http://www.r-project.org)
of the groups are indicated as \( p \) (experimental group) and \( q \) (control group), then the odds ratio is defined as:

\[
OR = \frac{p/(1-p)}{q/(1-q)}
\]  

(1)

An odds ratio of 1 indicates that the condition or event under study is equally likely in both groups. An odds ratio greater than 1 indicates that the condition or event is more likely in the first group. Vice versa, an odds ratio less than 1 indicates that the condition or event is less likely in the first group.

For \( RQ2 \), we mainly perform a qualitative analysis, by using plots where the x-axis indicate the snapshot timestamp (relative to the first snapshot analyzed) and the y-axis indicate the vulnerability density, i.e., the ratio between the number of vulnerabilities and the system size in NKLOC. We also consider when a new system release was made available. In addition to the qualitative analysis, we test whether the vulnerability density time series for different categories are stationary or not, i.e., whether they exhibit a trend. To this aim, we use the Augmented Dickey-Fuller (ADF) test [38] (\( H_{02}: \text{the time series is not stationary} \)).

For \( RQ2 \), first we represent decays using boxplots, where the box represents the interquartile range (IQR), and the vertical line comprises data in the range \( 1.5 \cdot IQR \). Data points outside such a range represent outliers. To test whether different vulnerability categories exhibit significant differences in terms of decay time, we use the Kruskal-Wallis test, which is a non-parametric test for testing the difference among multiple medians. The null hypothesis tested is \( H_{03}: \text{all medians of vulnerability decay times are equal} \). In addition, we use the non-parametric, two-tailed Mann-Whitney test to perform pair-wise comparisons among vulnerability categories. Since we are performing multiple tests on data extracted from the same data set, it is appropriate to correct the significance level using Bonferroni correction [39], i.e., dividing the significance by the number of tests performed (the number of combinations across categories). Truly, in this case the application of the Bonferroni correction is an over-protection against the internal validity threat of fishing rate, since the overall hypothesis is tested using a multiple-median test and the pairwise comparison is only performed for the purpose of comparing categories. For this reason, we report significance of results with and without applying the Bonferroni correction. Other than computing p-values, it is also important to evaluate the difference magnitude. To this aim we use the Cohen \( d \) effect size [40] which is defined, for independent samples, as the difference between the means \( (M_1 \text{ and } M_2) \), divided by the pooled standard deviation \( \sigma \) of both groups \( \sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2} \), i.e.:

\[
d = \frac{M_1 - M_2}{\sigma}
\]  

(2)

The effect size is considered small for \( d \geq 0.2 \), medium for \( d \geq 0.5 \) and large for \( d \geq 0.8 \). Moreover, we attempt to fit distributions of decays for different vulnerability categories to various statistical distributions, namely normal, exponential, Weibull, Gamma, and lognormal. In the following we only focus on exponential and Weibull, since these were the only distributions that fitted to our data, and that were used in literature to model defect decay [21]. The probability density function of the Weibull distribution is defined as:

\[
f(x; k, \beta) = \frac{k}{\beta} \left( \frac{x}{\beta} \right)^{k-1} e^{-(x/\beta)^k}
\]  

(3)

where \( k > 0 \) is the shape parameter and \( \beta > 0 \) is the scale parameter. The exponential distribution is a particular case of Weibull distribution with \( k=1 \). In the exponential distribution the rate parameter \( \lambda \) is often used instead of the scale \( \beta \), where \( \lambda = 1/\beta \). To check whether a distribution could fit our data, we first estimate the distribution parameters using the method of Maximum Likelihood, which maximizes the likelihood that the set of data used for the estimation can be obtained from the statistical distribution modeled with the estimated parameters. Once estimated the distribution parameters, we use a non-parametric test, the Kolmogorov-Smirnov (KS) test to check whether the distribution was able to actually fit the data (\( H_{04}: \text{there is no significant difference between the theoretical distribution and the actual data distribution} \)). Thus, the data set fits the distribution for p-values> 0.05. We use a maximum likelihood estimator available in the \texttt{fitdistr} function included in the MASS package of the \texttt{R} statistical environment and the \texttt{ks.test} again available in \texttt{R}.

For \( RQ4 \), as done in \( RQ1 \), we use a proportion test to check whether there is a significant difference across vulnerability categories for what concerns: (i) (R)emoved or (D)isappeared vulnerabilities; and (ii) vulnerabilities disappeared because of a direct change of the vulnerable code (IN) or by means of a change occurring elsewhere (EW). As for \( RQ1 \), the effect size of differences between R and D, and between IN and EW is measured by means of odds ratio. When analyzing the subset of documented vulnerabilities, we use the Fisher’s exact test [37] instead, as is more accurate than proportion test for small samples.
6. Empirical Study Results

This section reports results of the empirical study defined in Section 5.

6.1. RQ1: How are detected and removed vulnerabilities distributed across categories?

Table 3 reports:

- the overall number of different vulnerabilities detected for each system across snapshots (All);
- the number of removed vulnerabilities (R ∪ D);
- the number of (R ∪ D) vulnerabilities for which developers provided a documentation in the CVS/SVN commit notes;
- the number of alive vulnerabilities (L);
- the Odds Ratio (OR) between the number of (R ∪ D) vulnerabilities, and (L) vulnerabilities.

When looking at vulnerabilities detected by Splint, it can be immediately noticed that Memory Access and Type Mismatch are the most frequent vulnerabilities for both Squid and Samba. This is mainly due to characteristics of the programming language (C), e.g., potential Type Mismatches can be due to the use of implicit type conversions that, however, do not necessarily cause security problems or system crashes. Then, there is a number of potential Buffer Overflow problems: 49 in Samba and 106 in Squid, a system smaller than Samba in terms of size and number of analyzed snapshots. This tends to confirm what claimed by CERT: besides Type Mismatch and Memory Access, Buffer Overflows constitute a major exploitable vulnerability.

Regarding vulnerabilities detected by Rats, it can be noticed that Buffer Overflows are the largest portion of detected vulnerabilities for the two C systems (Squid and Samba). This is, by far, different from what is detected by Splint, and can be due to the different kind of analysis the two tools perform (see Section 3).

The kind of vulnerabilities detected in the PHP system Horde is different. Rats did not detect Buffer Overflows and Memory Access problems, since, during a program execution, the PHP interpreter performs a check of data structure boundaries and memory accesses. The most frequent kind of vulnerability is the generic Input problem, which affects many PHP instructions. Also, there is a number of potential Command Injection problems, which is a relevant vulnerability for Web applications. Although Horde interacts with a database and its code contains SQL statements, Pixy did not detect any SQL injection problem. As reported by securityfocus, SQL injection problems are not present in Horde, which, on the other hand, exhibits many HTML injection vulnerabilities not detected by the tool. Instead, Pixy mainly detected Cross-Site scripting problems, with a higher number of Unconditioned (157 vs 49 Conditioned).

Proportions of removed vulnerabilities should be considered with a particular caution. In fact, vulnerabilities were introduced in the system at different times, therefore a vulnerability could have remained alive just because of its recent introduction. Therefore, we com-

<table>
<thead>
<tr>
<th>System</th>
<th>BO</th>
<th>Ci</th>
<th>FS</th>
<th>T</th>
<th>M</th>
<th>NE</th>
<th>RC</th>
<th>RND</th>
<th>BO</th>
<th>CF</th>
<th>DC</th>
<th>M</th>
<th>TM</th>
<th>ASSC</th>
<th>XSSU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samba</td>
<td>1638</td>
<td>1</td>
<td>462</td>
<td>46</td>
<td>9</td>
<td>55</td>
<td>84</td>
<td>42</td>
<td>106</td>
<td>15</td>
<td>38</td>
<td>408</td>
<td>700</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Rats</td>
<td>805</td>
<td>0</td>
<td>365</td>
<td>23</td>
<td>5</td>
<td>38</td>
<td>45</td>
<td>23</td>
<td>85</td>
<td>10</td>
<td>35</td>
<td>314</td>
<td>316</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Rats</td>
<td>23</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Splint</td>
<td>833</td>
<td>1</td>
<td>97</td>
<td>21</td>
<td>4</td>
<td>17</td>
<td>39</td>
<td>19</td>
<td>21</td>
<td>5</td>
<td>3</td>
<td>94</td>
<td>384</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Pixy</td>
<td>0.9</td>
<td>–</td>
<td>14.1</td>
<td>1.6</td>
<td>5.0</td>
<td>1.3</td>
<td>1.3</td>
<td>16.4</td>
<td>4.0</td>
<td>136.1</td>
<td>11.2</td>
<td>0.7</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>Squid</td>
<td>3305</td>
<td>0</td>
<td>463</td>
<td>287</td>
<td>63</td>
<td>18</td>
<td>146</td>
<td>133</td>
<td>49</td>
<td>9</td>
<td>48</td>
<td>1075</td>
<td>1175</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Rats</td>
<td>2026</td>
<td>0</td>
<td>408</td>
<td>197</td>
<td>36</td>
<td>9</td>
<td>132</td>
<td>72</td>
<td>18</td>
<td>9</td>
<td>37</td>
<td>850</td>
<td>540</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Rats</td>
<td>222</td>
<td>0</td>
<td>35</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>33</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>90</td>
<td>61</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Splint</td>
<td>1279</td>
<td>0</td>
<td>55</td>
<td>90</td>
<td>29</td>
<td>9</td>
<td>14</td>
<td>61</td>
<td>31</td>
<td>0</td>
<td>11</td>
<td>225</td>
<td>635</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Pixy</td>
<td>2.5</td>
<td>–</td>
<td>55.0</td>
<td>4.5</td>
<td>1.5</td>
<td>1.0</td>
<td>88.9</td>
<td>13</td>
<td>0.3</td>
<td>–</td>
<td>11.3</td>
<td>14.3</td>
<td>0.7</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Horde</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>49</td>
<td>157</td>
<td>–</td>
</tr>
<tr>
<td>Rats</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>109</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>49</td>
<td>121</td>
<td>–</td>
</tr>
<tr>
<td>Rats</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>8</td>
<td>–</td>
</tr>
<tr>
<td>Splint</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>36</td>
<td>–</td>
</tr>
<tr>
<td>Pixy</td>
<td>–</td>
<td>–</td>
<td>198.3</td>
<td>–</td>
<td>–</td>
<td>132.2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>11.3</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 3: Counts of detected (All), (R)emoved ∪ (D)isappeared, (R) ∪ (D) documented, (L)ive vulnerabilities, and Odds Ratio of (R ∪ D) with respect to L.
pared the introduction timestamps (in days from the first snapshot analyzed) of removed vulnerabilities with those of alive vulnerabilities (boxplots of timestamps are in a longer technical report [41]). We found that, in all cases the introduction date of live vulnerabilities is significantly higher than for removed vulnerabilities (the difference is significant according to a Mann-Whitney two-tailed test, which always gave a p-value < 0.0001).

Provided the above considerations concerning the different age of vulnerabilities, let us now analyze whether proportions of removed vulnerabilities vary among categories. As said, this result must be treated with caution, however it still provides insights into the measure in which different vulnerabilities were removed, e.g., whether there was any particular category for which more attention was paid, or others that were almost ignored because considered not particularly harmful.

For vulnerabilities detected in Squid with Rats, the proportion test indicated that proportions of R ∪ D vulnerabilities significantly varied across categories (p-value < 2.2 · 10^{-16}), with a relatively higher OR (14.0) for File System problems, and Net problems (OR=5.0). For vulnerabilities detected with Splint, proportions of R ∪ D vulnerabilities varied across categories (p-value < 2.2 · 10^{-16}), with a very high OR for Dead Code (OR=136.1) and very low for Type Mismatches (OR=0.7). In other words, on the one side developers tend to remove unused code, on the other side, they do not care about Type Mismatches since in most cases these are just potential problems (i.e., developers properly use implicit casting).

For Samba–Rats, proportions of R ∪ D vulnerabilities are significantly different among categories (p-value < 2.2·10^{-16}), with a higher OR for Race Check (OR=88.9) and File System problems (OR=55.0), both considered relevant issues for a file and print services like Samba. Also for Samba–Splint there are significant differences (p-value < 2.2·10^{-16}) with, once again, a high OR (11.3) for Dead Code removal.

When analyzing Horde with Pixy, we found that XSS Conditioned vulnerabilities underwent a higher proportion of R ∪ D than XSS Unconditioned vulnerabilities. In the first case, all vulnerabilities are removed, and also in the second case the OR is clearly in favor of R ∪ D (11.3). Results for both vulnerability categories indicate how potential XSS problems are properly handled by developers since, as mentioned in the introduction, XSS attacks are becoming more and more popular. Specifically, XSS Conditioned vulnerabilities (all removed) are related to the PHP register_globals variable, which has been deprecated from PHP 6.0.0. Commit notes from version history indicate changes performed to made the system “work with register_globals = Off.”

Results for Horde–Rats indicated no significant difference across categories (p-value=0.6). All Command Injection problems were removed, and OR are very high for Input Problems (OR=198.3) and Race Check (OR=132.2). This result, together with those obtained with Pixy, indicated how in Horde developers took particular care of eliminating Command Injections, Input Problems and Cross Site Scripting problems, which constitute the major causes of attacks for Web applications. In particular, code inspection, we performed, revealed, for example, that many PHP files contained system command invocations through backticks, as in the following example, taken from file translation.php:

\[
\text{$files = explode("\b", trim($sh));\}}\]

It assigns $files the result obtained from the execution of a command contained in the variable $sh. The invocation of system commands/executables from PHP pages—including the invocation through backticks—has been reported to be a security issue; for this reason it has been suggested to disable backticks (see PHP bug 40030\(^{18}\)), since this would allow the user to inject arbitrary commands. If the string is not correctly pruned, then the command is executed. We found that this kind of vulnerability was removed from Horde: in particular, the above vulnerability disappeared in release 1.42 of the file, because the backtick command was replaced by an alternative function able to safely retrieve the same information.

We did not show OR for documented vulnerabilities, as in most cases are extremely low (below 0.001). The only case in which we found an OR greater than one is for Race Check vulnerabilities in Samba–Rats, where the OR was 5 in favor of removed/disappeared (and documented) vulnerabilities.

Overall, results indicate how over 50% (precisely, 56% Squid-Rats, 60% Squid-Splint, 65% Samba-Rats, 62% Squid-Splint, 94% Horde-Rats, 83% Horde-Pixy) of vulnerabilities are either removed or they disappear, indicating a substantial attention of developers to these issues.

As different tools—Splint and Rats in particular—detect vulnerabilities belonging to the same category, Buffer Overflow and Memory problems, it would be interesting to see if results of the two tools overlap. Surprisingly, we found almost no overlap between the

\(^{18}http://bugs.php.net/bug.php?id=40030\)
two tools, except 31 cases in Squid and 15 in Samba, mainly detected to usages of functions potentially causing buffer overflows, e.g., `sprintf`, `strncpy` and `strcat`. Although, to some extent, one could have expected quite different results for tools doing different kinds of analysis (data-flow vs. pattern matching) the almost total lack of overlap suggests that, to have a comprehensive coverage of potential vulnerabilities in a system, different tools need to be used. We also checked whether the same tool detected, in the same system snapshot, multiple vulnerabilities on the same source code line. We found that tools tend to give a unique vulnerability classification to source code lines. Future work will investigate whether the tool detects different kinds of vulnerabilities on the same line when the line or other parts of the system change over the time, or other parts of the system change.

6.2. RQ2: How does the number of vulnerabilities vary during the time?

To answer RQ2, we analyzed the evolution of vulnerability density over time. Due to space limitations, we report only the most interesting results.

Figure 1 shows, for each system and for each tool, the evolution of vulnerability density (all categories). The Figure also reports, on separate graphs, the evolution of documented vulnerabilities. It can be noted that vulnerabilities detected with different tools exhibit almost the same consistent behavior. Vulnerabilities detected with Rats started with a high density, that tended to decrease over the time; vulnerabilities detected with Splint had a lower density. We also noticed, in all systems we analyzed, patterns on vulnerability density increments followed by sudden decrements. This happened because, when a new pre-release is made available, it usually exhibits a high density of vulnerabilities, that tend to be removed with the release of security patches and updates. Despite the fact that we observed the vulnerability density and not the absolute number of vulnerabilities, we also observed the system evolution in terms of NKLOC (graphs can be found in a longer report [41]) to see if changes of vulnerability density can be due to some sudden changes in the source code size, e.g., due to the introduction of intrinsically non-vulnerable code such as header files. For Samba and Squid, we found that changes in vulnerability density did not occur in correspondence (nor close to) increase of the system size (in both case the size monotonically increased). Different is the case of Horde, where there was huge size increase from about 10 NKLOC to about 40 NKLOC in between release 2.1 and 2.2, and then a sudden decrease again to 10 NKLOC before release 3.0. At the same time (Figure 1-e), we also observed an increase of vulnerability density after release 2.2, followed by a rapid decrease.

Last, but not least, we analyzed the evolution of documented vulnerabilities. For Samba–Rats, we noticed...
(see Figure 1-b) a density increase from release 1.9.16 to release 1.9.18 (where, instead, the density of the whole set of vulnerabilities decreases). This can mainly be due to the large number of potential vulnerabilities Rats considers, of which only some removals are documented. No relevant differences in terms of trend were found for Squid, while for Horde (see Figure 1-f) the only documented vulnerabilities (both Pixy and Rats vulnerabilities) occurred between release 2.1 and 2.2, and were removed right after 2.2.

Interesting behaviors can be noticed by analyzing vulnerability categories separately. Figure 2 shows the density evolution of vulnerability categories detected in Horde with Pixy. There is an increasing trend of XSS Unconditioned vulnerabilities, and a stationary behavior of XSS Conditioned vulnerabilities as confirmed by the ADF test reported in Table 4. XSS Conditioned vulnerabilities, specifically related to the PHP register_globals variable, are partially reduced in correspondence of each major release, and completely removed in the last release candidate 3.2rc2, when the register_globals variable was deprecated.

Figure 3 shows some trends of Buffer Overflows in Squid and Memory problems in Samba, respectively, restricted to a short time window. Both exhibit—over the whole time frame analyzed—a stationary behavior as indicated in Table 4. It is interesting to note, in both cases, a periodic increment of the vulnerability density, due to system evolution and addition of new code, and a subsequent decrement due to security patches. In Squid, the partial increment of Buffer Overflows introduced in release 2.3STABLE3 has been removed with the subsequent security patches released in 2.4STABLE7 and 2.5STABLE7, bringing the vulnerability density back to the level before 2.3STABLE3. After a while, a new increment occurred when new features were added. In Samba, a similar periodic behavior is exhibited between releases 2.0.9 and 3.0.0. The alpha release 2.2.0a contains a local maximum of Memory problems, which is partially reduced by the subsequent (not security-
related) patches from 2.2.1 to 2.2.4, and highly reduced with the security patch 2.2.5. This behavior occurred again with other patches, from 2.2.6 to 2.2.8a, until in 3.0.0 a stable release was produced. Overall, we can see that the vulnerability density tend to increase in correspondence of unstable releases—see for instance the case of XSS Unconditioned vulnerabilities detected in Horde (Figure 2)—and then decrease again in correspondence of stable releases.

The right-side of Table 4 shows results of ADF stationarity test for documented vulnerabilities. While in some cases (Buffer Overflows, Input Problems, Net Problems, Race Check for Samba-Rats, and Memory Problems for Split-Squid) the stationarity is confirmed when looking at documented vulnerabilities, in other cases it is not. This depends on the limited number of documented vulnerabilities, but also to the fact that, since on these vulnerabilities we have evidence of removal, their trend is, in many cases, clearly decreasing.

6.3. **RQ3: How long vulnerabilities tend to remain in the system?**

To answer RQ3 we analyzed, for each vulnerability, its decay. Figure 4 shows boxplots of decays (expressed in days) for each vulnerability category detected in the three systems with the different tools.

The Kruskal-Wallis test indicates a significant difference among different vulnerabilities detected with Splint. In particular, for Squid there exists a significant difference among decays of different vulnerability categories (p-value = 0.00029), and the same happens for Samba (p-value < 1.96 · 10^{-12}). For vulnerabilities detected with Rats, the Kruskal-Wallis test indicates a significant difference for Samba (p-value < 2.2 · 10^{-16}) and Squid (p-value = 7.65 · 10^{-11}), while not for Horde (p-value = 0.19). Finally, for vulnerabilities detected with Pixy on Horde, no significant difference was found between decays of XSS Conditioned and XSS Unconditioned (p-value = 0.069).

To perform a deeper comparison, we compared all pairs of vulnerability categories using a Mann-Whitney, two-tailed test. Results are shown in Table 5 and Table 6 for Rats and Splint respectively. The significant values (without Bonferroni correction) are shown in boldface; where the value is significant even after the correction, the value is followed by a star (*) symbol. The tables also report the Cohen d effect size (negative values indicate that the row-vulnerability mean decay is smaller than the column-vulnerability mean decay).

For Squid–Splint, results obtained for Samba are partially confirmed, i.e., Buffer Overflows decayed significantly faster than Control Flow, Dead Code and Type Mismatch vulnerabilities. For Squid–Rats vulnerabilities, Buffer Overflows decayed significantly faster than File System, Memory, Net and Random Generation problems. File System problems decayed faster than Input, Net, and Race Check problems. Buffer Overflow represents the kind of vulnerability developers tend to remove faster: for a Web proxy like Squid Buffer Overflows can be a major cause of attacks.

For Samba–Splint, we found that Buffer Overflows decayed significantly faster than Control Flow, Memory and Type Mismatch vulnerabilities, suggesting the particular attention paid by developers to this kind of vulnerability, while others such as Memory and Type Mismatch are only potential problems detected by the tool—e.g., occurring when developers use implicit type casting—that however do not require any further action. Also, Dead Code vulnerabilities decayed faster than Memory and Type Mismatch vulnerabilities, indicating the execution of activities such as refactoring aimed at improving the code quality (confirming what found in RQ1 related to periodic patches that decrease vulnerabilities). For Samba–Rats, it emerges that File System problems decayed significantly faster than all other vulnerabilities; this can be due to the specific characteristics of the application (distributed file sharing). Results for Buffer Overflows obtained with Splint were not confirmed by Rats, since the latter detects a larger number of Buffer Overflow vulnerabilities, often due to the use of some specific C functions (e.g., strcpy), which in many cases do not constitute a problem and developers do not change them.

We repeated the same statistics for documented vul-
Figure 4: Boxplot of decays for different vulnerabilities
nerabilities however, mainly because of the limited number of these vulnerabilities, the only significant result confirmed was between Race Check and Input Problems for Samba–Rats, where Race Checks also in the case of documented vulnerabilities were removed/disappeared significantly faster than Input problems (p-value=0.03, d=0.68).

Finally, we investigated, according to the method described in Section 5.3, whether the vulnerability decay can be modeled using any particular statistical distribution. Table 7 reports distribution parameters and KS test p-values for vulnerability categories where a fitting distribution was found. As it can be noticed, in most cases the vulnerability decay fits an exponential distribution, but in two cases—Control Flow Problems and Dead Code for Samba–Splint—where data fitted a Weibull distribution. Comparing this result with existing literature modeling defects decay with statistical distributions [20, 21, 22], it can be noted that vulnerabilities decay follow similar laws to defects decay.

Examples of Cumulative Distribution Functions (CDFs) are shown in Figure 5, where the actual CDF (thick line) is compared with the theoretical CDF (thin line). The fitting with both exponential and Weibull distribution can be explained as follows: vulnerabilities have a high likelihood to be removed—i.e., developers remove or protect them—shortly after their introduction. As time increases, the likelihood a vulnerability has to be removed decreases following an exponential or Weibull probability density function. This might suggest that, whenever, after a while, developers did not care about removing a vulnerability, either the vulnerability was too hard to be spotted, or it was just a false warning raised by the vulnerability detection tool.

Table 8 reports the same statistics of Table 7, limited for vulnerabilities with a documented removal/disappearance. It is worthwhile to note that, when limiting the analysis to documented vulnerabilities only, we found distribution fittings in cases where no fitting was found for the whole data set, while we were not able to confirm results of Table 7. Also, it is interesting to
note the larger number of cases with a Weibull distribution fitting, which exhibits a faster decrease than the exponential distribution: documented vulnerabilities are likely to be representative of known problems, and tend to be removed as soon as possible.

6.4. RQ4: How are vulnerabilities removed?

This Section reports an in-depth analysis of the vulnerabilities that, for different reasons, were removed from the system. First, we distinguish the vulnerabilities that disappear from the system because the vulnerable source code lines is removed (R) from vulnerabilities that disappear because of a change (D). Then, for vulnerabilities that disappeared without removal, we compare the proportion of D due to a change inside the vulnerable line (IN), with cases where the line did not change, thus the vulnerability disappeared because of a change elsewhere (EW). Results are summarized in Table 9.

For Squid–Rats, proportions or R and D significantly vary among categories (p-value=1.3 \cdot 10^{-9}), and in all cases the number of R is greater than the number of D. The higher proportion (OR=0.84) for D can be found for Random Generator problems, where function calls, such as srand(time(NULL)) changed into safer ones squid_srandom(time(NULL)). Proportions varied across categories (p-value<2.2 \cdot 10^{-15}); OR were very high for IN, mainly because of the specific type of vulnerabilities Rats detects, i.e., usage of unsafe functions, replaced locally with alternative, safer, ones. For example, in Horde $curdir=$ `$pwd` was replaced by `Scurdir=getcwd()`, while in Samba, strcat(response,"\n",1) was replaced by `safe_strcat(response,"\n",1)`.

For Squid–Splint, once again proportions of R and D significantly varied among categories (p-value<2.2 \cdot 10^{-16}). Regarding the difference between IN and EW, proportions significantly varied among categories (p-value=7.099 \cdot 10^{-15}), with a higher proportion of IN for Dead Code (OR=1.6). This can be explained because—in the few cases where Dead Code was not removed—the vulnerability was removed by means of changes introducing the usage of (previously) unused variables or functions. In all other cases, vulnerabilities mainly disappeared because of changes elsewhere (OR<1). It is interesting to notice how Splint—because of the kind of analysis it performs—detects a large number of vulnerabilities disappeared because of changes elsewhere while, as said above, Rats is mainly limited to vulnerability removed by means of changes in the same line.

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th>TOOL</th>
<th>VULN. CATEG.</th>
<th>FITTED DIST</th>
<th>PARAM</th>
<th>KS TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squid</td>
<td>Splint</td>
<td>TM</td>
<td>Exp</td>
<td>0.0000</td>
<td>0.70</td>
</tr>
<tr>
<td>Squid</td>
<td>Splint</td>
<td>BO</td>
<td>Weibull</td>
<td>0.0025</td>
<td>0.44</td>
</tr>
<tr>
<td>Squid</td>
<td>Splint</td>
<td>I</td>
<td>Weibull</td>
<td>0.0055</td>
<td>0.82</td>
</tr>
<tr>
<td>Squid</td>
<td>Splint</td>
<td>RC</td>
<td>Weibull</td>
<td>0.0056</td>
<td>0.097</td>
</tr>
</tbody>
</table>

Table 8: Distribution fitting for documented vulnerabilities
Table 9: Counts of vulnerabilities that were (R)emoved and (D)isappeared, the latter because of a change (IN) the line or elsewhere (EW)

<table>
<thead>
<tr>
<th></th>
<th>BO</th>
<th>CI</th>
<th>FS</th>
<th>I</th>
<th>M</th>
<th>NET</th>
<th>RC</th>
<th>RND</th>
<th>BO</th>
<th>CF</th>
<th>DC</th>
<th>M</th>
<th>TM</th>
<th>ASSC</th>
<th>XSSU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RATS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>652</td>
<td>232</td>
<td>223</td>
<td>33</td>
<td>30</td>
<td>35</td>
<td>12</td>
<td>22</td>
<td>5</td>
<td>26</td>
<td>29</td>
<td>35</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>153</td>
<td>133</td>
<td>223</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>63</td>
<td>5</td>
<td>9</td>
<td>285</td>
<td>231</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D_IN</td>
<td>134</td>
<td>131</td>
<td>223</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>22</td>
<td>2</td>
<td>5</td>
<td>17</td>
<td>17</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D_EW</td>
<td>19</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>–</td>
<td>–</td>
<td>41</td>
<td>3</td>
<td>4</td>
<td>268</td>
<td>214</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>SAMBA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>800</td>
<td>224</td>
<td>133</td>
<td>23</td>
<td>6</td>
<td>45</td>
<td>61</td>
<td>8</td>
<td>9</td>
<td>22</td>
<td>143</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>733</td>
<td>215</td>
<td>15</td>
<td>3</td>
<td>87</td>
<td>11</td>
<td>10</td>
<td>29</td>
<td>608</td>
<td>397</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D_IN</td>
<td>67</td>
<td>9</td>
<td>49</td>
<td>–</td>
<td>34</td>
<td>–</td>
<td>8</td>
<td>3</td>
<td>19</td>
<td>577</td>
<td>348</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D_EW</td>
<td>71</td>
<td>2</td>
<td>–</td>
<td>18</td>
<td>–</td>
<td>5</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>13</td>
<td>45</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>HORDE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>–</td>
<td>10</td>
<td>–</td>
<td>165</td>
<td>–</td>
<td>18</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>36</td>
<td>76</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>–</td>
<td>2</td>
<td>–</td>
<td>4</td>
<td>–</td>
<td>5</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>13</td>
<td>45</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D_IN</td>
<td>–</td>
<td>2</td>
<td>–</td>
<td>3</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>2</td>
<td>10</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D_EW</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>4</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>11</td>
<td>35</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

varied among categories (p-value < 2.2 · 10^{-16}), and, as it happened for Squid, odds were in favor of IN.

For Samba–Splint, we did not find any significant difference in proportions of R and D among categories (p-value=0.22).

For Horde–Rats, proportions of R and D significantly varied among categories (p-value=0.0001) although odds are always clearly in favor of (R)emovals. For the low number of vulnerabilities that (D)isappeared without a removal, no significant difference in proportions of IN and EW was found among categories; however, in this case, all Command Injections, and the largest proportion of Input problems (OR=9) disappeared because of an (IN)ternal change of the vulnerable line. Instead, Race Checks (OR=0.06) mainly disappeared because of a change elsewhere. Command Injections and Input problems are mainly solved by replacing potentially dangerous instructions or by inserting string preprocessing functions within the same statement. In Horde, Race Checks problems are mainly related to the use of is_readable() and is_file() PHP functions, which could cause deadlocks when concurrent calls are performed on the same resource. Such vulnerabilities disappeared when some functionalities related to file searching and/or creating were reengineered with safer libraries, such as Horde::CLI and PEAR::File_Finder.

Finally, for Horde–Pixy, no significant difference was found in proportions of R and D among categories (p-value=0.25). A significant difference was neither found between IN and EW, although odds were in favor of EW: all XSS Unconditioned and, a large proportion of XSS Conditioned (OR=0.003), disappeared because of a change elsewhere. The main reason, also reported in 6.3, is related to the fact that recent versions of PHP are more protected from XSS exploits.

Table 10 shows percentages of R, D, D_IN and D_EW for which developers provided a documentation in CVS/SVN commit notes not necessarily related to such vulnerabilities. As explained in Section 5.3, we checked the presence of a significant difference between R and D and between D_IN and D_EW using the Fisher’s exact test. Pairs for which the difference is significant are highlighted in boldface.

We can immediately notice that, as expected, the percentage of documented changes concerning vulnerabilities is in most cases higher (and often significantly higher) for D than for R. This has to be expected: R considers cases where the vulnerable source code line is removed, and this can often happen in the context of other changes, that are reflected in the commit notes, while nothing is said about the vulnerability, that is probably “accidentally” removed. When the vulnerability (D)isappears without removing the source code line, it is more likely that the changes was actually performed with the specific aim of fixing the vulnerability, and therefore it is more likely that the change is documented in the commit notes. In some cases the percentage increases to values between 80% and 100%. Of course, there are some exceptions where the percentage is significantly higher for removals, e.g., Type Mismatches for Squid–Splint, or Memory problems for Samba–Splint. In both systems such documented removals are mainly related to source code cleanup, such as removal of unused data structures and compiler warnings as reported by Squid commit notes, “A bit of cleanups to make GCC happy and removed some unused code.”.

Much in the same way, we compared the percent-
Table 10: Percentages of $R$, $D$, $D_{IN}$ and $D_{EW}$ with evidence of vulnerability removal in commit notes (pairs with significant differences highlighted in boldface)

<table>
<thead>
<tr>
<th>RATS</th>
<th>BO</th>
<th>CI</th>
<th>PS</th>
<th>I</th>
<th>M</th>
<th>NET</th>
<th>RC</th>
<th>RND</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>19%</td>
<td>–</td>
<td>9%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
<td>35%</td>
</tr>
<tr>
<td>$D$</td>
<td>26%</td>
<td>–</td>
<td>81%</td>
<td>0%</td>
<td>50%</td>
<td>0%</td>
<td>0%</td>
<td>44%</td>
</tr>
<tr>
<td>$D_{IN}$</td>
<td>28%</td>
<td>–</td>
<td>81%</td>
<td>0%</td>
<td>50%</td>
<td>0%</td>
<td>0%</td>
<td>44%</td>
</tr>
<tr>
<td>$D_{EW}$</td>
<td>5%</td>
<td>–</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>55%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPLINT</th>
<th>BO</th>
<th>CF</th>
<th>DC</th>
<th>M</th>
<th>TM</th>
<th>XSSC</th>
<th>XSSU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>4%</td>
<td>20%</td>
<td>3%</td>
<td>3%</td>
<td>38%</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$D$</td>
<td>17%</td>
<td>40%</td>
<td>33%</td>
<td>33%</td>
<td>26%</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$D_{IN}$</td>
<td>41%</td>
<td>30%</td>
<td>40%</td>
<td>47%</td>
<td>18%</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$D_{EW}$</td>
<td>5%</td>
<td>33%</td>
<td>25%</td>
<td>88%</td>
<td>26%</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIXY</th>
<th>BO</th>
<th>CF</th>
<th>DC</th>
<th>M</th>
<th>TM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$D$</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>35%</td>
<td>17%</td>
</tr>
<tr>
<td>$D_{IN}$</td>
<td>–</td>
<td>–</td>
<td>100%</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$D_{EW}$</td>
<td>–</td>
<td>–</td>
<td>100%</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

ages of documented $D_{IN}$ and $D_{EW}$: in this case we expect to have significantly higher percentages for $D_{IN}$: it is unlikely that it happened accidentally in the context of other changes as the vulnerability disappeared because of a change directly involving the vulnerable source code line. Results of Table 10 indicate that such an expectation is met: in most cases the percentage for $D_{IN}$ is higher than or at least equal to the percentage for $D_{EW}$, and it is often significantly higher. Notable exceptions are Memory Problems for Squid–Splint, and Race Check for Samba–Rats. In both cases, it is likely that this happened because developers fixed the problem (and documented it) by modifying other source code lines. For example, in Squid the deallocation instruction `free()` has been replaced in February, 1998 by the safer `xfree()` (commit notes: “use `xfree`”) causing many vulnerabilities related to Memory Problem instructions disappearing without changing such instructions. In Samba, Race Check problems have been indirectly removed by a fix, as documented in the commit note: “security problem with multi-user Windows NT servers where the contents of the open-file cache can end up being served out to users who shouldn’t have access”.

7. Threats to Validity

This section discusses threats to validity that can affect the results reported in Section 6, following guidelines provided for case study research [42].

Construct validity threats concern the relationship between theory and observation; in this context they are mainly due to errors introduced in measurements. In particular, our results can be affected by performances of the adopted static vulnerability detection tools. Although tools can detect false warnings—i.e., vulnerabilities that do not really cause problems—most of our study (RQ3 and RQ4) focuses on vulnerabilities that disappear, thus vulnerabilities requiring maintenance activities aimed at removing them. In addition, we have manually identified those change sets that are explicitly documented as being related to vulnerability removal and performed most of the analyses (at least those that could have been done) on these vulnerabilities separately. We limited the degree of subjectiveness in this classification by letting two independent experts performing the classification, and discussing conflicting cases. Of course, while the manual classification detects cases where there is evidence of a documented vulnerability removal, the converse is not true: it might have happened that developers removed the vulnerability and performed other changes at the same time, and did not mention the vulnerability removal in the commit note. Also, as discussed in Section 6.4, there might have been cases of “accidental” vulnerability removal, when a source code fragment was deleted for other reasons.

To provide an idea to what extent the used tools (Splint, Rats, and Pixy) actually detected likely vulnerabilities or whether, instead, they provided false warnings, we manually inspected 24 Splint vulnerabilities, 36 Rats vulnerabilities and 10 Pixy vulnerabilities. Other than covering all vulnerability categories, we looked at the full output produced by tools, which indicates the specific problem or the function related to the vulnerability, with the purpose of covering different kinds of specific problems/unsafe function usages. It must be clear that this has not to be intended a statistical sample used to empirically evaluate the precision of the
adopted tools. Determining whether a potential vulnerability is a true one or a false warning would require a thorough knowledge of the system being analyzed, plus the need to perform, in many cases, data flow analysis or even stress testing, where static analysis would not suffice. Instead, the intent of this inspection was to select vulnerabilities belonging to different categories and identifying cases where the tool tend to fail and case where the tool, very often, actually identifies potential vulnerabilities.

For Splint, the Buffer Overflow problems were actual problems (or at least potential ones), Control Flow Problems were mainly related to the usage of double as i f flags (where rounding could have caused problems) however in half of the cases we inspected the variable was explicitly rounded by using a round function. All Dead Code problems were related to true warnings for variables or functions defined but never used. Type Mismatch problems, in most cases, were harmless problems. Most of them were access with “=−>” to fields of variables that appeared not to be structs or unions. However, we found that variables actually contained pointer to structs or unions; such a false warning might be due to the limited points-to analysis capabilities of Splint. Other problems were also less harmful e.g., related to implicit casting, that however appeared to be correctly performed. The only cases we found as potential problems were related to the lack of assignment to a variable of the result produced by non-void functions. Although in most cases Type Mismatches would not constitute vulnerability problems, it can still be argued that they could make the program more difficult to be understood, that can become error prone: investigating this is, however, out of scope of this paper. As explained in Section 6.4, sometimes developers just remove Type Mismatching problems “to make the compiler happier”.

For Rats, potential Buffer Overflow problems were related to the use of functions like strcpy, strcat, snprintf etc. However, besides heuristics described in [34], Rats does not perform precondition checking on used variables. For this reason, 3 over 10 potential problems analyzed turned out to be false warnings. Input problems were related to the use of functions such as fopen, for which half of the times we found the file name to be a string literal, in other cases a variable whose content was not checked before invoking the function. The same happened for Net Problems in functions such as getH- ostByName. Command injections were related to the use of system or of backticks (in PHP, for Horde). As explained in Section 6.1, backticks are a documented security issue in PHP; the use of system, sometimes was harmless (e.g., with a literal parameter), while in other cases developers realized that the function could have been dangerous, e.g., in the file tools.cc of squid where we found the comment “/* XXX should avoid system(3) */” above the system source code line. Memory problems were due to the usage of functions such as realloc. In at least two third of the cases we analyzed the code properly tested whether the variable returned by realloc was null. Last, but not least, for half of the random generator problems we analyzed, the seed was initialized using the system time (making the random generator safe), while on other case read from a parameter, which could have lead to deterministic sequences. As mentioned in Section 3, Rats does not check how the seed is initialized.

For Pixy, all the identified vulnerabilities were related to potential XSS problems due to accesses to global variables, conditioned or not by the register globals flag. They were actually all potential problems, but in a few cases where the global variable was re-defined, making the XSS attack not possible.

Results of static analysis tools are also affected by the presence of false negatives—i.e., vulnerabilities not detected. At minimum, we performed the analysis using multiple tools—working in different ways and being complementary in detecting different kinds of vulnerabilities, as shown in Section 6.1. Also, although some false negatives can still be present, the number of detected ones is enough to make some considerations about their evolution. Future studies will consider alternative tools. Finally, it is important to highlight that—besides the manual classification we performed—this study analyzes the evolution of vulnerabilities as detected by static analysis tools, despite of the possible incorrectness or incompleteness of such a classification. This, to some extent, is similar to what Kim and Ernst did for compiler warnings [24], which may or may not cause problems.

Threats to internal validity did not affect this particular kind of study, being it an exploratory study [42]. Nevertheless, to better investigate the context in which a vulnerability was removed, we manually analyzed commit notes to understand, where possible, the reasons for a vulnerability removal.

Conclusion validity threats concern the relationship between the treatment and the outcome. We paid attention to not violate assumptions of statistical tests we used (although we mainly used non-parametric tests). The only conclusion validity issue can be related to Type-II error when performing tests on documented vulnerabilities. Due to their limited numbers, in some cases it could happen that we could not reject a null hypothesis due to the limited sample size.
**Reliability validity** threats concern the possibility of replicating this study. Analysis tools are available for downloading, as well as code and bug repositories of the analyzed systems. The data extraction process is detailed in Section 4. Finally, we made raw data available for replication purposes.

Threats to **external validity** concern the possibility to generalize our findings. The study was performed on three different systems representative of different kinds of network applications. Nevertheless, analyses on further systems are desirable, as well as the use of vulnerability detection tools different from those we adopted.

### 8. Conclusions

This paper reported an empirical study aimed at analyzing the evolution of source code vulnerabilities, as detected by three freely available static analysis tools—**Rats**, **Splint**, and **Pixy**, in three widely used open source network systems, namely the **Squid** proxy server, the **Samba** file and printer service, and the **Horde** web mail system. As in a previous study by Kim and Ernst [24], who analyzed the removal of compiler warnings, we analyzed the evolution of vulnerabilities as detected by tools, although we are aware that some vulnerabilities might not be dangerous at all. Also, we realized that there is no “silver bullet” vulnerability detection tool: in fact there is almost no overlap among results of different tools; to have a thorough coverage of potential vulnerabilities, developers should rely on the output of different tools.

When analyzing the evolution of vulnerabilities over the time, we found that, overall, vulnerabilities tend to be removed from the system: although we considered vulnerabilities with a different lifetime, the study indicated that a percentage between 56% (**Squid–Rats**) and 93% (**Horde–Rats**) of the detected vulnerability was removed. For example, developers tried to reduce the presence of data structure accesses vulnerable by buffer overflow attacks. Also, functions deprecated because of security issues tend to be consistently replaced. In many cases, also vulnerabilities that are often considered as a licit programming practice in some programming languages—e.g., memory access or type mismatch problems in C programs—are handled by developers, either for increasing the application security or for reducing its defect-proneness. Even dead code tends to be removed.

We found no particular trend, across the analyzed snapshots, in the vulnerability density: as the system evolves, such a density tends to remain stable. However, what we found is that new vulnerabilities are introduced when new features are added, often in correspondence of pre-releases or experimental releases. Then, before major releases, vulnerabilities are removed by means of security patches. In some cases these security patches are reflected in security bulletins of the system under study (e.g., **Horde**), or even are triggered by security issues raised for the programming language being used.

The decay time generally varies among vulnerability categories: in other words, there are categories that tend to be fixed quicker than others, regardless of the nature of the system analyzed. It is likely that vulnerabilities deemed to be more harmful than others (e.g., buffer overflows) tend to be generally fixed quicker. It is also interesting to see how such a decay can be modeled by means of a probability distribution: in most cases the decay distribution significantly fits theoretical distributions belonging to the family of exponential distributions, namely the Weibull or the exponential distribution. Such distributions have been used in the past to model the defect decay in a software system. This result can be interpreted as follows: vulnerabilities that are really considered as likely dangerous tend to be removed quickly from the system; if this does not happen, it means that, probably, the detected vulnerability does not really constitute a big problem, or even is a false warning detected by the tool.

Vulnerabilities can disappear from the system in three different ways: (i) the vulnerable line is removed, (ii) the vulnerable line is modified, or (iii) the vulnerability disappears because of a change occurring elsewhere, for example aimed at changing a variable type, the size of a data structure, or at adding a control flow construct that “protects” the vulnerability. There are of course vulnerabilities, such as dead code, that are often intrinsically removed by means of code deletion. This happens, for example, when functionalities related to vulnerable statements are completely recoded, e.g., by means of reengineering, causing the removal of such statements. Vulnerabilities that are due to the use of potentially dangerous or deprecated instructions/functions (e.g., **strcpy**), detected by pattern-matching tools such as **Rats**, are often removed by directly modifying the vulnerable source code line, and by replacing the vulnerable instruction with an alternative one. For some vulnerabilities, mainly related to memory problems and buffer overflows, and often detected by tools such as **Splint** able to perform a flow analysis of the subject programs, we also found a high proportion of vulnerabilities removed by means of a change occurring elsewhere. As said above, we believe developers should combine multiple tools to thoroughly analyze the presence of potential vulnerabilities in their source code and, depending
on the kind of analysis performed by the tool and of vulnerability detected, different actions might be needed to remove/protection the vulnerability.  

In many cases, the vulnerability is removed in the context of other changes, and we do not have any information that can tell whether developers intentionally removed the vulnerability—without documenting the removal because it was a minor change in the context of another activity—or whether it happened “accidentally” because for other reasons the vulnerable code was removed. When focusing on the attention on vulnerabilities that disappear without removing the vulnerable source code line, and especially vulnerabilities disappeared because of a direct change on that line, the percentage of cases where this was documented by developers is significantly higher than for cases where the line was removed. In other words, there are many cases in which a potential vulnerability as detected by a static analysis tool is removed in the context of a specific change, documented by developers in commit notes. The following are some excerpts extracted from the commit notes of the analyzed system.

- Squid: “Fixed buffer overflow bug in whois.cc...”; “No longer use scanf() in some places...”; “replaced some strcpy() calls with memset() and strcat()...”.

- Samba: “Don’t use static memory malloc it... Jeremy.”; “move to SAFE_FREE()”; “don’t use gets()!”; “Dead code removal. Not used anywhere. Jeremy.”; “don’t use strcpy()”; “This is a security audit change of the main source. It removed all occurrences of the following functions: sprintf strcpy strcat The replacements are sprintf safe_strcpy and safe_strcat. ... Jeremy.” (It can be noticed here how many vulnerabilities were fixed by the same developer).

- Horde: “... close security hole - stream data directly to browser instead of reading it into memory first.”.

Future work aims at extending the study to other systems, at relating the vulnerability decay with the occurrence of bugs, and at further combining information obtained from static vulnerability detection tools with information obtained by mining other sources such as security bulletins and bug reports related to security issues.

References


[17] Y.-W. Huang, S.-K. Huang, T.-P. Lin, C.-H. Tsai, Web applica-


A. Additional graphs

Figure 7: Evolution of the three systems in terms of NLOC
Figure 6: Introduction timestamps of live and removed/disappeared vulnerabilities

Figure 8: Samba – counts of detected vulnerabilities over time
Figure 9: Squid – counts of detected vulnerabilities over time

(a) Rats

(b) Splint

Figure 10: Horde – counts of detected vulnerabilities over time

(a) Rats

(b) Pixy

25
Figure 11: Samba – vulnerability density over time

Figure 12: Squid – vulnerability density over time
Figure 13: Horde – vulnerability density over time