ABSTRACT

ANBALAGAN, PRASANTH. A Study of Software Security Problem Disclosure, Correction and Patching Processes. (Under the direction of Dr. Mladen Vouk.)

Quantitative analysis of software security problems plays an important role in understanding software security. Information on how and when software security problems are disclosed, exploited in the field, fixed by developers and patched by users, is often analysed from a calendar time perspective. This provides worst-case assessment and effort-to-fix information, but is not directly related to actual operational impact of the discovered problems. Given that security problems are a subset of the more general category of software problems, employing usage metrics typically found in classical software reliability engineering, such as inservice time, appears to be a reasonable approach for assessing security problems. The main goal of this thesis is to investigate operational software security problem disclosure, correction and patching processes through publicly available information, and through that improve our understanding of of the issues, as well as enable better process and defense planning and related decision making.

One of the issues one runs into almost immediately when studying open software security data is the distributed nature and diversity of such data. The data reside in numerous data bases, in different formats, and it is a challenge to collect that information. The first step in the current work was to develop a set of tools for automated collection of linked information across public repositories. Investigated were products that follow a process of full disclosure of security problems before fixes are available (we call them "full disclosure" products), and those that disclose security problems along with fixes and possibly only limited information about them (we call them "limited disclosure" products). To analyse and understand collected information, a comprehensive security problem response model was developed that describes interactions of events associated with users, developers, attackers, software security problems, and fixes. The model captures the states through which a software may go based on the discovery, disclosure, exploit, failure, and correction of security problems. The model distinguishes itself from published models by emphasizing roles and operational impact perspectives.

As part of the analyses, two sub-models are investigated for estimating the disclosure of unique security problems - the classical Logarithmic Poisson Execution Time (LPET) model, and a Bayesian model. The latter model was included to capture the subjective views of risk and exposure. Both models were found to work well - the LPET in the context of security problem rates across releases, and the Bayesian model in the context of disclosure of security problems per release.

In combination with experimental data, the overall model was also used to investigate secu-

rity problem disclosure, correction and patching policies. Time to discovery, time-to-disclosure, time-to-intrusion, time-to-patch-availability, and time-to-patch-application are some of the metrics in this context. Empirical results tell us that between 30% and 80% of the reported problems will fail in the field only if end-users interact with the attack mechanism (e.g., opening a malicious attachment in an email). We classify such problems as "voluntary" security problems. Early warning/disclosure of such problems may help users in taking precautions. An interesting question is "Under what conditions is the policy of early disclosure of voluntary security problems a good one?". This is discussed from the perspectives where a) users do not intervene in the installation of patches (we call it "automatic updates") and b) where users do intervene (we call it "non-automatic updates"). For a given set of values of the process metrics under consideration, it is shown what percentage of users should heed the warning for the policy of early disclosure to be effective. Several other such policies are examined and discussed.

A Study of Software Security Problem Disclosure, Correction and Patching Processes

by
Prasanth Anbalagan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Laurie Williams	Dr. Xuxian Jiang
Dr. Jason Osborne	Dr. Mladen Vouk Chair of Advisory Committee

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

DEDICATION

To my parents.

BIOGRAPHY

Prasanth Anbalagan received his Bachelor of Engineering in Computer Science and Engineering from Coimbatore Institute of Technology, India, in 2003. From 2003 to 2005, he worked as a Software Engineer at HCL Technologies, Cisco's Offshore Development Center in India. He received his M.S in Computer Science from North Carolina State University in 2008 and continued as a doctoral student with an emphasis on Software Security and Reliabiltiy Engineering working under Dr. Mladen Vouk. Prasanth's research interests include software security, automated software engineering and data analysis.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Mladen Vouk, for his valuable guidance and support during my Ph.D program. I thank Drs. Laurie Williams, Xuxian Jiang, and Jason Osborne for serving on my dissertation committee and providing valuable suggestions in improving this research work. This work was supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

Heartfelt thanks go to Ms. Linda Honeycutt and Ms. Carol Allen in the Department of Computer Science. Special thanks go to my wife and my family members, and friends for their continuous support and encouragement that led to a successful completion of the Ph.D program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation and Problem

Security problems (or fault, or flaws), discovery and disclosure of such problems, failure of security problems in the field, and correction of such problems, are some of the important considerations in evaluating software quality. In analysing the operational security quality of software systems with large number of users, researchers often focus on security problem counts and frequencies independently of the operational profile of the software that harbors them [40, 4, 33, 65]. However, one may be able to see a different behavior when including usage metrics in to security analyses. For example, Figures 1.1 and 1.2 show the number of security faults disclosed for Firefox 3.0 per calendar unit time (e.g., weeks), and per software execution unit time or inservice unit time (e.g., inservice weeks [50], [32]) respectively.

It is interesting to observe that that the number of security problems disclosed per unit time remains approximately constant when viewed from the calendar perspective. An observation might be that the security quality of the software is probably not improving. On the other hand, when viewed from the usage perspective, an observation might be that the usage of the product is intensifying and yet we are not finding more problems over time, so perhaps things are getting better, and perhaps the impact of problems is smaller (e.g., less new problems per inservice hour) [32]. Thus, an operational view can be helpful and should to be taken into account explicitly when analysing the security profile of a software product.

In practice, the operational profile of a software product (i.e., frequency or amount of use of the product, its individual functions, and operations [50] includes not only the users and the legitimate use of that software, but also the intervention of attackers. The latter may intensify (and change the operational profile) after a security problem is disclosed. When a disclosure is accompanied by a fix or a work-around, end-user perceptions of the level of security of the product may improve, i.e., security reliability of the product may appear to increase. When

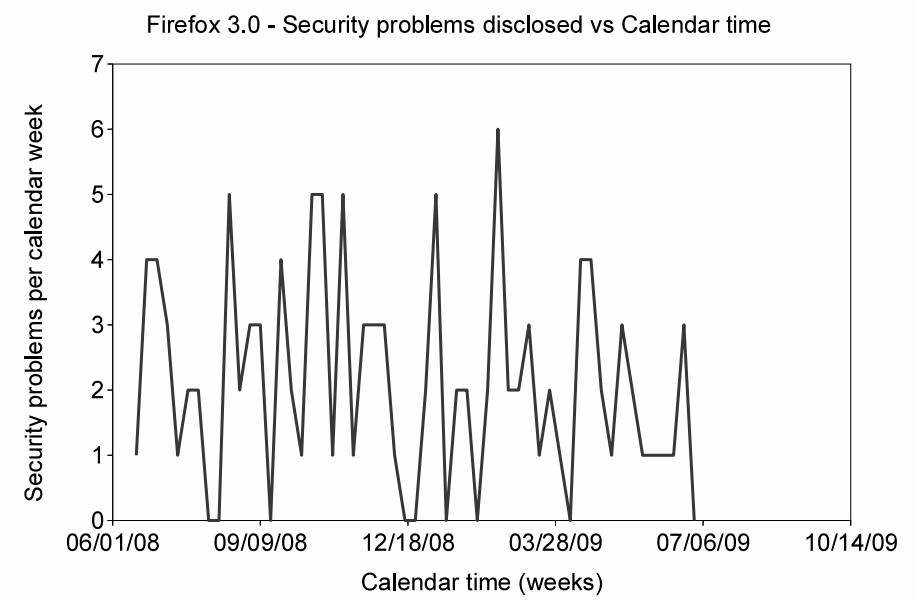

Figure 1.1: Security problems disclosed vs calendar time

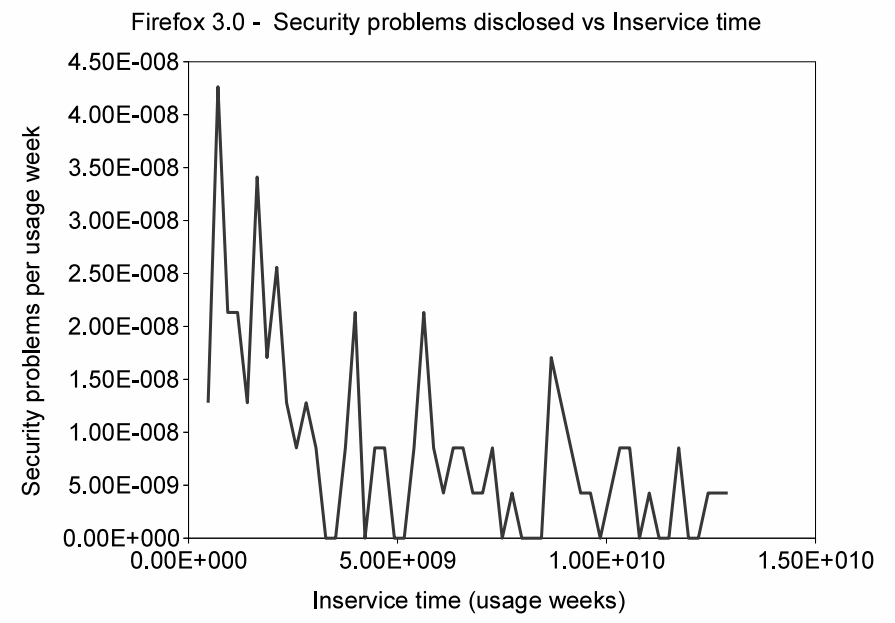

Figure 1.2: Security problems disclosed vs inservice time

disclosures are not directly accompanied by instant repair of the problem in the form of a fix or a workaround by developers, end-user perceptions may be less favorable despite the fact that in actuality the probability that the product will fail due to a security breach may not have changed in a particular environment. When information such as software usage level, developer response process, user patching process, and attacker intrusion processes are (publicly) available, it is necessary to also consider these factors in security analyses. Therefore, in this work, we recognize roles and usage based metrics in analysing publicly disclosed security problems, as well as field usage levels of a product.

1.2 Solution Overview

There were four principal goals for the work described in this dissertation 1) develop an automation framework to facilitate collection and collation of data on software security problems from public repositories, 2) develop and assess a state-based process model of the security problem discovery, disclosure, correction, exploit, failure, and patching, 3) develop security sub-models, as a part of the overall process model, to predict disclosure events of security problems, and 3) use empirical data and simulations with the overall process model to asses several practices for handling disclosure, correction and patching of security problems.

1.3 Summary of Contributions

The principal contributions and findings are:

- **Data Collection** One of the issues is the distributed nature of security data and difficulties in collecting that data [66]. A toolset for automated collection of linked information across public repositories was developed to address the issue [11]. The toolset has been effectively used as a part of the research reported here to collect information on both security and non-security problems for a broad range of software products [Chapter 2]. Investigations included different versions of products such as Firefox, Linux, SeaMonkey, Microsoft Internet Explorer, Windows XP, Adobe products, etc. Analysis of the collected data yielded some interesting results. For example,
  - About 0.05% to 5% of the total number of problems reported are related to security [11]. This is consistent with what other researchers [4, 2, 55, 3] have reported.
  - Security problem reports tend to have a different profile than non-security related problem reporting and correction [8]. Time to security problem report and correction appears to be relatively constant. Time to non-security problem report and

correction tends to decrease as the product progresses in time, possibly indicating that product is getting better [Chapter 3].

- On the average, about 35% of the total security problems receive public exploits. Thirty-four percent (34%) of the total security problems receive exploits even before developers discover the problem and are disclosed as a result of an exploit, and only 1.3% of the problems receive public exploits after the security problem is publicly disclosed [9].
- Only about 0.05% to 2% of the total number of software security problems in a software result in field failures. This is a very small, albeit potentially dangerous number. Ninety-five percent (95%) of the field failures impact less than 50 systems (limited impact) and 3% of the field failures impact between 50 and 1000 systems (medium spread impact), and 2% of the field failures impact more than 1000 systems (broad spread impact). However, popular security threats during the past 10 years are known to have infected as many as 50,000 to 50 million users.
- A significant proportion of real software security problems (ranging from 30% to 80%) (so called "voluntary" problems) will fail in the field only when users interact with the attack mechanism [9]. This suggests that these failures could be avoided if users are alerted of such problems, i.e., a disclosure is made to users in a timely manner, and users take precautions. This is further evident from the study of popular security threats during the past 10 years, where 50% of the security threats involved voluntary security problems [Chapter 4].
- Fedora and Suse Linux distributions show positive results by resolving high and medium severity[1] security problem reports without a backlog. On the other hand, Ubuntu and Redhat Enterprise Linux distributions show backlogs in resolving high and medium severity security problem reports.
- In products that follow a process where full disclosure of security problems happen before fixes are available, the time taken to release patches for failed security problems is about a week. In products that do not follow a process of full disclosure of security problems, the time taken to release patches for failed security problems ranges from a week to a month.

- **Process Modeling.** To analyse and understand collected information, we developed a security problem response model that describes the interactions of events associated with users, developers, attackers and security problems, and fixes. The model captures the

[1]The classification of severity of security problems by the National Vulnerability Database was used in the analysis

states through which a system may go based on the discovery and disclosure of security problems, exploits, field failures, type of problems, correction status (e.g., a developer fixes a security problem or not), etc. The model is quite comprehensive, and distinguishes itself from published models by emphasizing roles and usage perspectives [26, 15, 56].

- **Predictive Modeling** As part of the security problem response modeling, one would be interested in being able to predict things like mean-time-to-problem-correction, mean-time-to-problem-disclosure, etc. If the process does not change, one might be able to use classical software and system reliability models. An assumption is that security problem behave in much the same way as non-security problems. Transfer of such models into security space is still at an early stage (e.g., [66, 4, 60, 55]). In [55, 60], researchers used classical software reliability models on security fault data, from a calendar time perspective, but without much success. An observation in this context is that classical reliability models work much better when the exposure is expressed not in calendar time, but in terms of system usage or inservice time (e.g., [50], [19]). This led to investigation of two security-oriented models for description and prediction of some of the transitions in our overall model. For example,

  - **Classical Reliability Models** This work finds that classical reliability models (such as Logarithmic Poisson Execution Time model [49]) can well predict problem rates per release and across software releases in the context of operational use of a system [8, 58, 9, 10].
  - **Bayesian Disclosure Model** It is important to note that software security includes not only objective measures, but also subjective measures such as implied variability in the operational profile, influence of attacks, and subjective impressions of exposure and severity of the problems, etc. In order to reflect the uncertainty in the security behavior of a system as well as the uncertainty in our knowledge about the security behavior of a system, Bayesian approaches used in classical software reliability models were extended in this work to the security space. [12]. This work finds that a Bayesian disclosure model can predict the disclosure of security problems with respect to the operational use of a software at a performance level (relative error of 26%) at least on par with that of some other classical software reliability models.

- **Analysis of Practices and Policies**

  While the security process model mentioned above (Chapter 3) is comprehensive and includes a reasonably complete security problem response state space, the model is complex and is not easy to solve analytically. In order to understand the interactions of various

events covered by the security model, and guide users and developers in making appropriate decisions, we used simulation in combination with real measurements. This yielded some interesting results. For example,

- The time-to-fix a security problem (measured from the discovery of a problem to the release of a patch) is shorter in "full disclosure" software products than "limited disclosure" software products. Obviously, and perhaps not unexpectedly, "full disclosure" software products' need and have quicker response to security problems than "limited disclosure" software products.
- "Full disclosure" software products that follow "automatic updates" will impact less number of systems with actual failures than "limited disclosure" software products that follow non-automatic updates. Some examples of the former category are Mozilla Firefox and Google Chrome, and the latter category are Internet Explorer and Apple Safari.
- Using publicly available security related information for software products, users can estimate the optimal time to apply security patches when switching between products. For example, can a user of Internet Explorer with an average time-to-patch of 30 days, determine the optimal time to apply security patches after switching to Firefox and continue to be as secure as when using Internet Explorer? The model developed in this work suggests that a user should apply patches within 5 days of a patch release in Firefox in order to achieve this.
- Early disclosure/early waring of voluntary security problems may help users in taking precautions. Can a developer or a vendor decide under what conditions the policy of early disclosure of voluntary security problems is a good one? Perhaps developers may want to take this opportunity to allow additional time to fix a security problem. To do so, they need to know what percentage of the users must heed the warning to compensate for the additional impact due to the delay in fixing the security problem. For example, consider that a Firefox developer makes an early disclosure of 1 week before the "normal fix time". To allow an additional time of 1 week to fix the security problem, there must be a high probability that at least 19% of users (as opposed to no users taking precautions in the 1 week slot) take precautions to achieve protection as the 1 week process.

1.4 Outline

The rest of the thesis is organized as follows. In Chapter 2, we discuss the security data collection and empirical results from the collected data. In Chapter 3, we discuss the security

problem response model. In Chapter 4, we discuss the two predictive modeling approaches. In Chapter 5, we discuss the simulation based analyses. In Chapter 6, we discuss the related work. In Chapter 7, we conclude the thesis with a discussion on the future work.

Chapter 2

Empirical Analysis

2.1 Introduction

In order to analyse software security problems, we need to collect security related information like the number of security problems disclosed, , time-to-discovery, time-to-disclose, time-to-fix the security problems, software usage level in terms of the number of downloads or registrations or active users, time-to-intrusion, etc. We do this by developing a toolset that automatically retrieves such security related information from public repositories. We then analyse the collected data to understand various characteristics of security problems from the perspective of events related to users, developers and attackers.

2.2 Mining Security Data From Public Repositories

Mining software repositories is an important activity when analysing projects. Mining information across multiple data sources is one of the challenges [74, 28]. We observed that relevant information from one repository can complement the mining activity on another repository. For example, the Bugzilla bug tracker for *Mozilla*[1] and *Red Hat* projects[2] contain custom "keywords", textual tags, that help identify specific categories of faults in the database. The keyword *security* relates to a security bug[3]. This could have been used to identify the security faults in projects like *Fedora*[4], *Firefox*[5] etc. But the usage of the keyword was not consistent across bug reports. The Common Vulnerability Exposure[6](CVE) site maintains information about publicly known vulnerabilities. The vulnerabilities tagged by CVE are contained in the

[1]https://bugzilla.mozilla.org/
[2]https://bugzilla.redhat.com/
[3]https://bugzilla.redhat.com/describekeywords.cgi
[4]http://fedoraproject.org/
[5]http://www.mozilla.com/en-US/firefox/
[6]http://cve.mitre.org/

National Vulnerability Database[7](NVD), a U.S. government repository devised to manage vulnerability data. The NVD database lists vulnerabilities specific to different types of products including *Fedora (Red Hat)*, *Firefox (Mozilla)* etc. The *external resource* section of each vulnerability listed in the NVD has a mapping or link to the faults in their respective bug-tracking system. This implies that information from the NVD can be utilised in mining security faults in projects like *Firefox*, *Fedora* etc.

For projects, like Ubuntu, that deploy the Launchpad bug tracker, the search engine allows to search for faults with CVE tags. These CVE tags in turn can be used to collect linked vulnerability characteristics in terms of the nature of exploits, impacts, and their metric values present in the the NVD. Extracting such information would give additional context to information available through individual repositories. Manually extracting such information is tedious. In this Chapter we

- Discuss a framework that helps in automatically mining linked information across repositories.
- Discuss its implementation in the context of identifying security related information in projects that use the Bugzilla bug tracker by mining linked information from the NVD.
- Discuss its implementation in the context of mining security faults from repositories that deploy the Launchpad bug tracker along with related vulnerability information from the NVD.
- Discuss the use of the tool on the *Fedora*, *Ubuntu*, *Suse*, *RedHat*, and *Firefox* projects.

2.2.1 Approach

Figure 2.1 shows an overview of our framework. It consists of: the exploratory study, the HTML downloader, and the HTML parser. we discuss the components in detail in the following sections.

Exploratory study

Exploratory study refers to the preliminary work done to identify the relation between the repositories under consideration and collect details required to extract data from the repositories. The output of this study would be the *repository relation identifier*, *URL patterns*, and the *data formats*.

We define a repository relation as an identifier obtained from one repository that would help in search of information in another repository. For example, the vulnerability summary provided

[7]http://nvd.nist.gov/

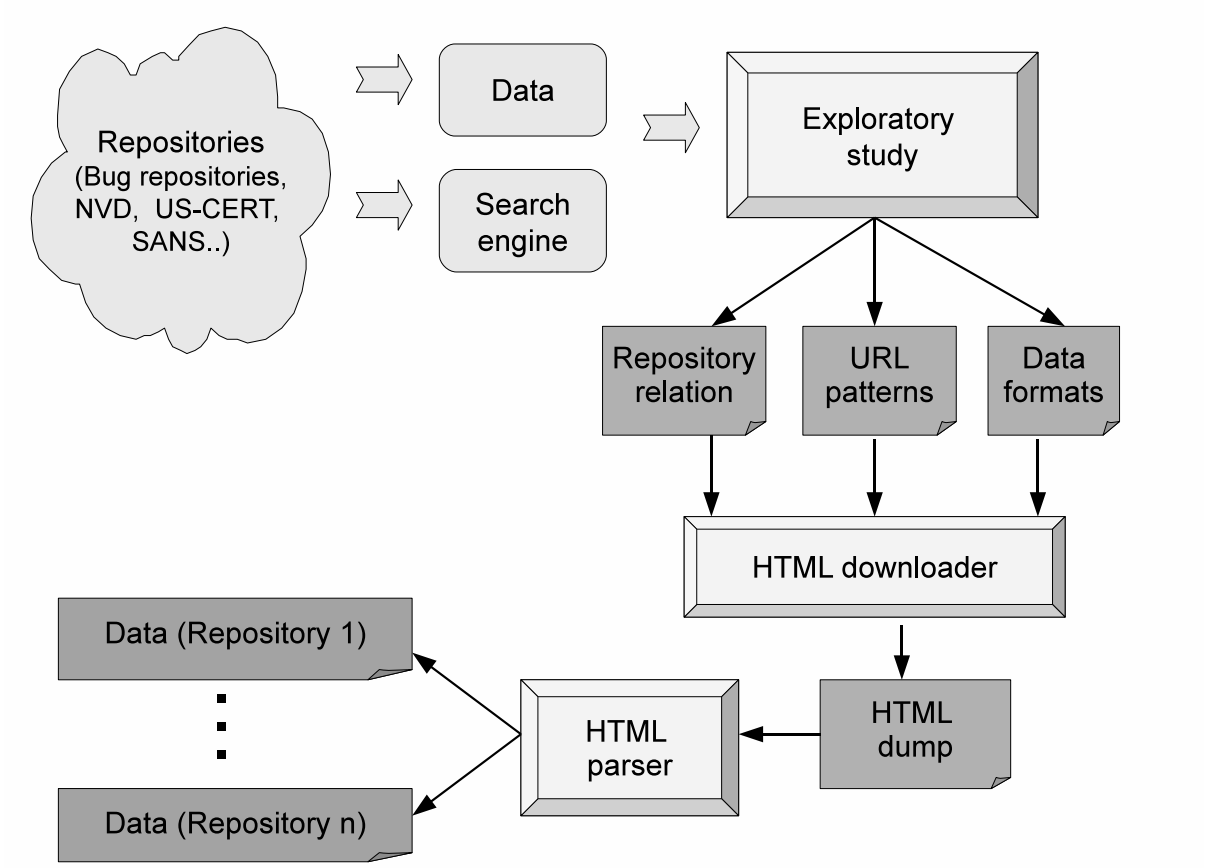

Figure 2.1: Framework

by the NVD contains a mapping or link to individual faults in the project's bug repositories. Consider the vulnerability summary for the *Firefox* vulnerability CVE-2007-3656[8] in the NVD. The external source section of this summary has the link *"https://bugzilla.mozilla.org/show_bug.cgi?id=387333"* which maps to the *Firefox* bug repository. We call this mapping or link a repository relation. Also, the vulnerabilities listed in NVD have a unique CVE tag (eg.CVE-2007-3656) associated with each of them. We observed that bug repositories use this CVE tag either in the bug summary or the bug description section. This unique CVE id could also be used as a repository relation. Projects like *Ubuntu* quote the CVE tag in their bug reports. This may not be true for all projects. It is necessary to carefully examine the repositories, understand their usage of keywords or tags, and make sure the usage is consistent across bug reports.

URL patterns refer to the patterns observed in the HTTP or HTTPS requests used by the

[8]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-3656

search engines of the respective project's bug tracker. In our implementation, the patterns are simple URL links observed in the address bar of the browser when one uses the search engines but without the parameter values. For example, consider the URL used by the search engine in the Redhat's Bugzilla "$https://bugzilla.redhat.com/buglist.cgi?query\_format = advanced\&classification = Fedora\&product = Fedora\&bug\_status = NEW$". This URL is used to list all *NEW* faults for the *Fedora* product. Here the *URL pattern* would include the url but without the parameters *Fedora* and *New*. Parameters will be passed as arguments to the *HTML downloader* based on the URL pattern and type of data or product required. We need to identify the *URL pattern* for generic search engine usage as well as for viewing individual sections in a bug report. The latter is required to extract information for each bug while the former is required to collect the list of all the faults in the bug repository using generic search requests. For example, "$https://bugzilla.redhat.com/show\_activity.cgi?id =$ " refers to the *URL pattern* to view the activity log of each bug report. As the bug ids are collected, this *URL pattern* will be appended with the individual bug ids and executed to view their activity log.

Data formats refer to the formats in which the required data is stored in the repositories. In our implementation, we collected information in terms of calendar time when the bug was reported and fixed. For example, Redhat's Bugzilla maintains the timing using *Year-month-day hour-minute-seconds* format (Example: 2007-01-11 08:20:30). Vulnerability characteristics in NVD are storied in the format (AV: x /AC: x /Au: x /C: x /I: x /A: x), where the attribute AV is the access complexity, Au is the authentication complexity, C is the confidentiality impact, I is the integrity impact, A is the availability impact, and x is the respective attribute's value.

HTML downloader and HTML parser

The *HTMl downloader* uses the *URL patterns* to automatically collect information from the repositories based on the parameter values passed during execution. The output of the *HTML downloader* is the HTML dump of the search result returned for each of the HTML requests automatically invoked. In other words, the output is the HTML version of the webpage that would be displayed when a user searches for information using the search engine. The *HTML parser* uses the *data formats* obtained from the *exploratory study* component to parse this HTML output and extract the required information. We used libraries from HTMLScraper[9], an open source project, to implement the *HTML downloader* and *HTML parser* components of the tool.

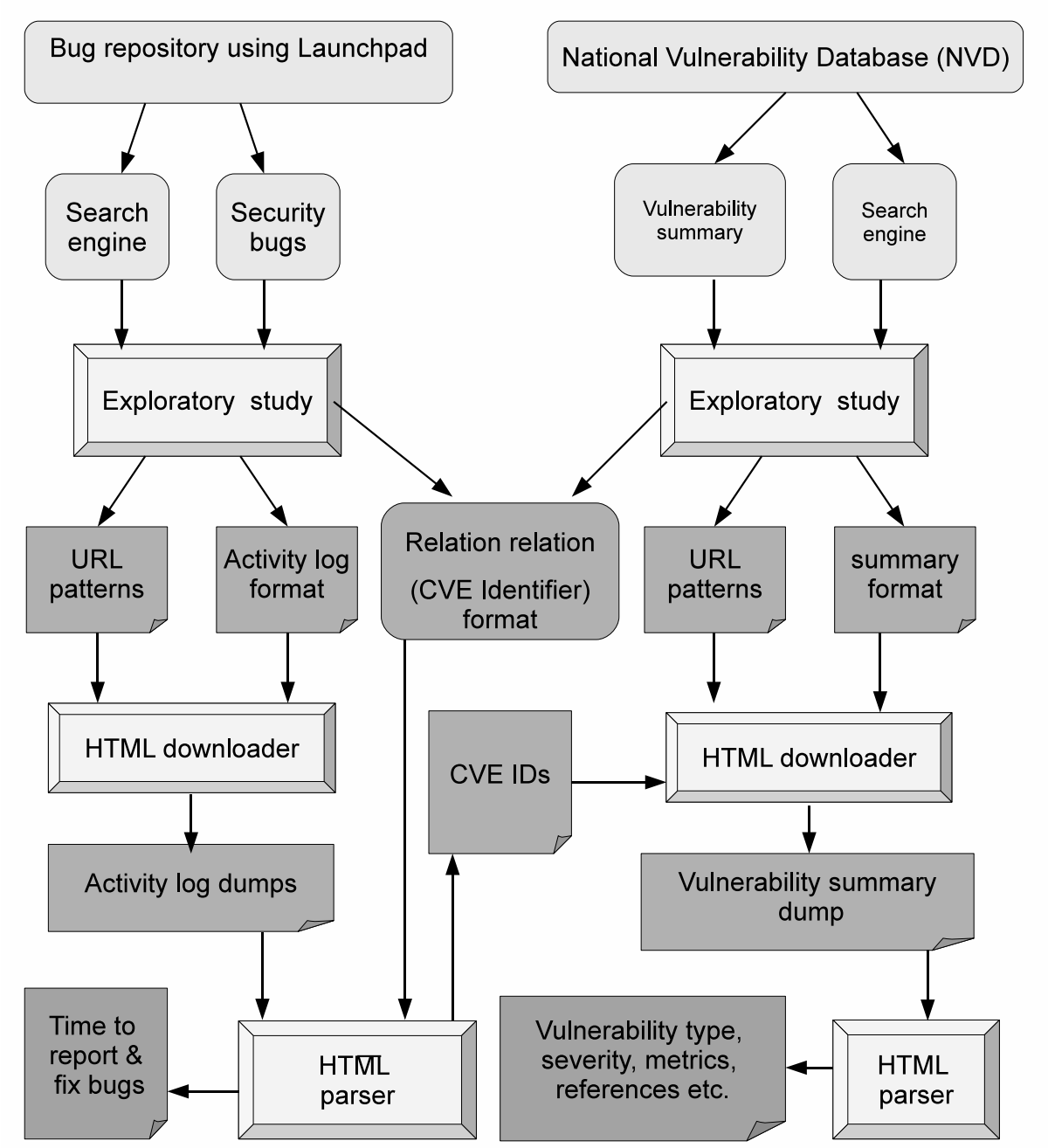

Figure 2.2: NVD and bug repositories using Launchpad

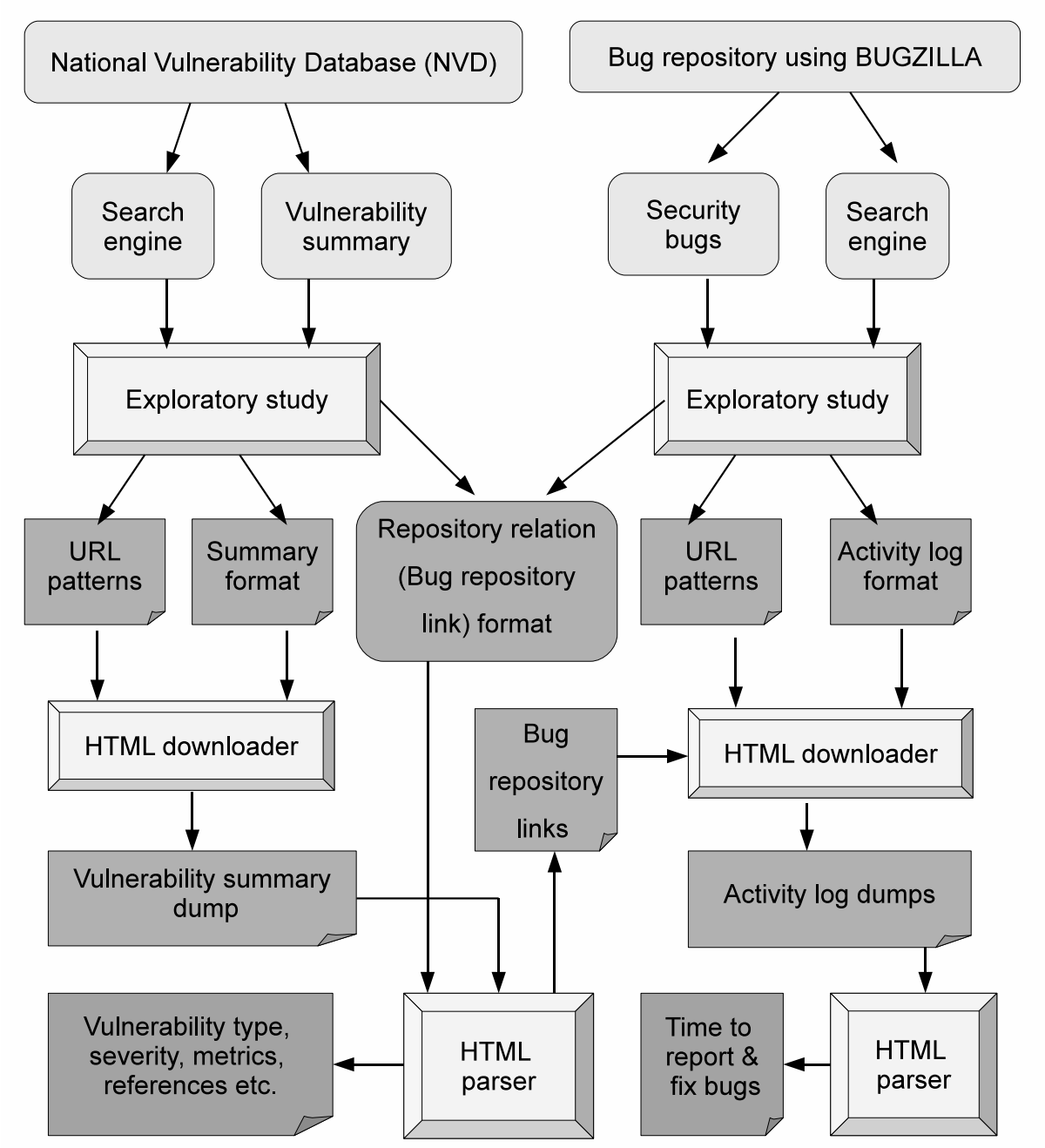

Figure 2.3: NVD and bug repositories using Bugzilla

2.2.2 Implementation

Figures 2.3 and 2.2 show the implementation of our framework between the NVD and projects that deploy BUGZILLA and Launchpad bug trackers respectively. Figure 2 shows the implementation where information from the NVD is used to identify security faults in the respective project's bug repositories. Figure 2.3 shows the implementation where security faults are directly mined from project bug repositories that deploy Launchpad bug tracker and then collect related vulnerability characteristics from the NVD.

In our first implementaion, a list of CVE ids for vulnerabilities specific to a project is given as input to the *HTML downloader*. The *HTML downloader* retrieves the HTML data of the vulnerability summary for each vulnerability in the list using their CVE ids. In this case, each CVE id is appended to the URL pattern "$http://web.nvd.nist.gov/view/vuln/detail?vulnId=$" to retrieve the vulnerability summary. Then the *HTML parser* identifies the mapping or link to the bug repositories from the HTML output. Next, the *HTML downloader* uses the identified links and retrieves the HTML data of the activity log section of each bug report in the respective bug repository. The activity log contains the timestamp when the bug was reported and fixed. Based on the data format in which the timestamps were stored, the *HTML parser* identifies the date and time when each bug was reported and fixed. This implementation was evaluated on projects like *Firefox* and *Fedora*. We collected the timestamps for use in our published works [8, 58, 10, 9, 12].

In our second implementation, the *HTML downloader* automatically retrieves the list of faults from the bug repository using the URL pattern similar to the implementation that works on projects with Launchpad bug tracker. Launchpad bug tracker allows to search for faults tagged with CVE id. The CVE ids are displayed along with the bug ids when one uses the search engine. The *HTML parser* parses this output and extracts the bug ids, and the CVE ids associated with them. Once the bug ids and CVE ids are extracted, the activity log data and the vulnerability characteristics were extracted by using the *HTML downloader* and the *HTML parser* in a similar fashion as in the first implementation. This implementation was evaluated on the *Ubuntu* project. It is possible to find more than one repository relation and retrieve data across repositories. In our implementations, we identified the relation as the mapping or links present in the NVD vulnerability summary. Since these links could be used directly by the HTML downloader, we considered such a repository relation. One could also search for CVE ids in the summary section of the faults, and there by identify security faults. In this case, the tool can be modified to collect the CVE ids from each of the faults and then retrieve the corresponding vulnerability information from the NVD. We have implemented this approach for Suse and RedHat projects.

[9] http://sourceforge.net/projects/HTMLscraper/

Table 2.1: Tool statistics

Project	Non-security	Security	Total	Percentage
Fedora (1 - 8)	48077	908	48985	1.85
Ubuntu (4.10 - 8.10)	71408	1086	72494	1.49
OpenSuse (10.2 - 10.3)	8747	66	8813	0.75
SuseLinux (10.1 - 10.2)	6975	56	7031	0.80
Firefox (1 - 2)	8506	367	8139	4.30
RedHat Enterprise Linux (2.1 - 5)	22496	822	23318	3.52

2.2.3 Discussion

We collected time specific information for security faults from the individual bug repositories along with the data from the NVD. Table 2.1 shows the data collected in terms of the number of security and non-security faults for *Fedora* (releases 1 to 8), SuseLinux (releases 10.1, 10.2) OpenSuse (releases 10.2, 10.3), Ubuntu (releases 4.10 to 8.10), and RedHat Enterprise Linux (releases 2.1 to 5). From the table, we find that security faults account for roughly 0.5% to 5% of the total number of faults. This is consistent with the rates reported by other researchers [3, 2]. The security faults have been identified using the approaches explained in this paper. We have successfully used the tool in this work in collecting data for various analyses. We plan to make the tool open source software and release it on Sourceforge[10].

2.3 Empirical Results

A much discussed issue is the disclosure of vulnerabilities. Information about vulnerabilities is either kept secret until the problem is resolved and is then released when a fix is available, or the information about vulnerabilities is made available before the problem has been resolved and a fix has been released. An argument in favor of non-disclosure is that releasing information about a security problem may increase the risk for end-users since it makes a larger number of potential hackers aware of it, and until a fix is available, this increases the exposure of end-users to danger. A counter-argument may be that disclosure of security problems helps alert end-

[10]http://sourceforge.net/

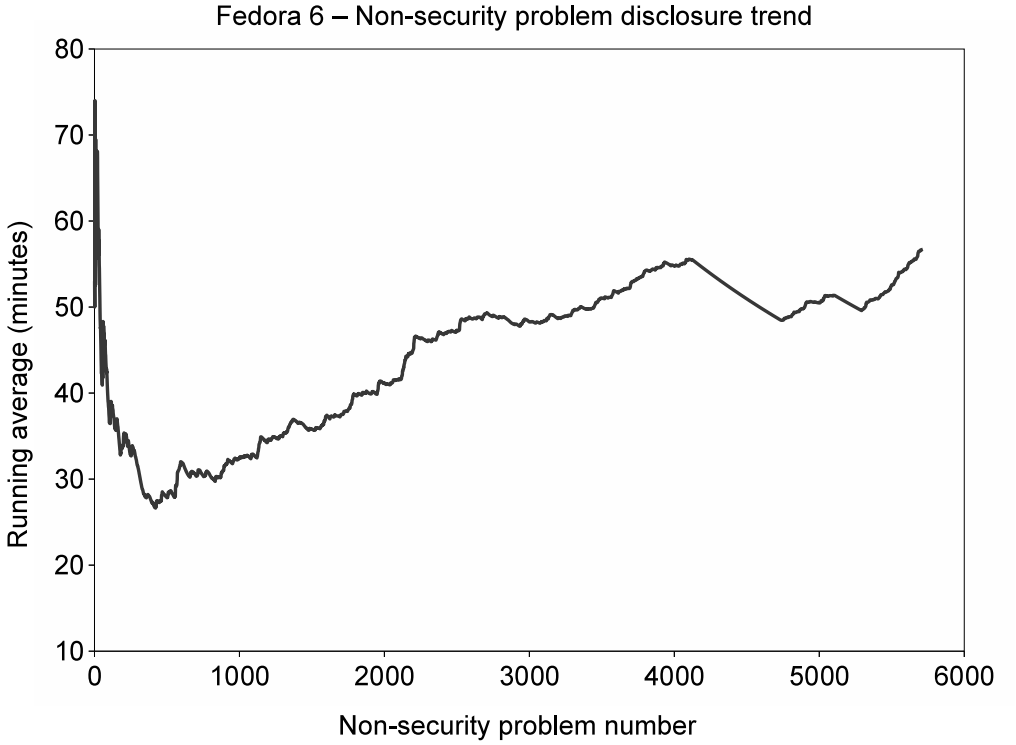

Figure 2.4: Trend test for non-security problems

users in taking precautions against exploits. In this section, we study various characteristics of the collected vulnerabilities.

2.3.1 Security vs Non-security Problems

Software Reliability Models typically model the behavior of a software that each time the software experiences a failure, the fault that caused the failure is fixed, and the reliability changes over time. When a software is released, typically there is an increase in the number of reported failures since the product users start reporting problems. However, one can expect a decrease in the number of failures and associated faults over time as the product is becoming stable [50]. To verify this behavior for non-security and security problems, we use the arithmetic mean test [34] (also knowns as the running average or moving average test), a frequently used trend test to detect reliability growth. Figure 2.4 shows the arithmetic mean trend test for non-security problems disclosed for Fedora 6 from the release of Fedora 6 until the release of the successive version, Fedora 7. The horizontal axis shows the non-security problems reported and the vertical axis shows the running average computed using the inter-arrival problem disclosure

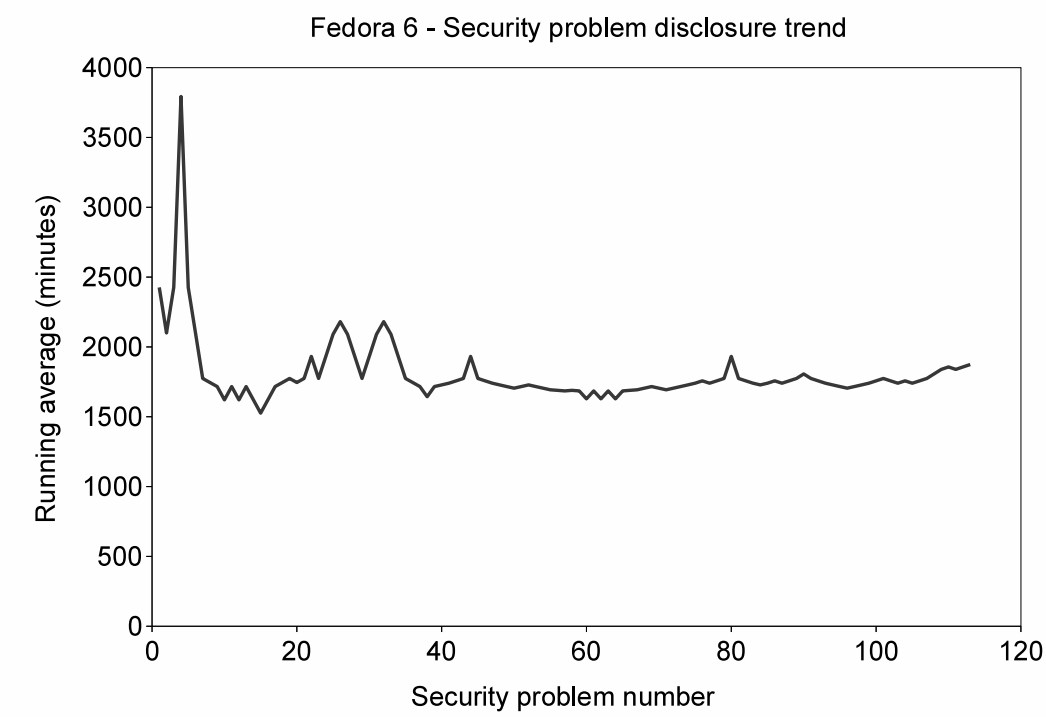

Figure 2.5: Trend test for security problems

times. From the Figure, we observe a decrease in the inter-arrival times initially, i.e., increase in the number of non-security problems, and then the inter-arrival times continue to increase showing that the product is getting stable. But for security problem reports, we see that inter-arrival times appear to be relatively constant (Figure 2.5). Similarly the correction of non-security problems show a decreasing trend and security problems show no trend. The decreasing trend with non-security problems may imply that developers tend to fix problems faster as the next release approaches in order to resolve as many problems possible in the older release.

2.3.2 Voluntary and Involuntary Security Problems

One facet of how a security problem is exploited in the field is whether a user is deceived into interacting with the attack mechanism to get exploited (we call these voluntary failures). An example is a recent security alert from Adobe[11]. Adobe alerted of the presence of a critical security problem in Adobe Reader and Acrobat, on May 1, 2009, even before releasing the fix

[11] http://www.adobe.com

Table 2.2: Security problems from the NVD

Project	Total	Voluntary	Percentage
Fedora	908	192	21.1
Firefox 2.0	225	76	33.8
SeaMonkey	177	78	44.1
Ubuntu	1105	244	22.1
RedHat	822	79	9.61
Thunderbird	165	61	37.0
OpenOffice	28	23	82.1
Suse	56	4	7.14
Apache Server	127	13	10.2
WindowsXP	440	126	28.6
IE 5.0	119	69	58.0
IE 6.0	352	115	32.7
Overall	4524	1070	23.7

on May 12, 2009. The alert says that when a user opens a malicious PDF file from untrusted source, the application would crash and the malicious PDF file can execute arbitrary code and take control of the affected system[12]. Adobe cautioned users from opening files from untrusted sources, and provided tentative solutions in preventing applications from automatically opening PDF files. One thing to note is that a failure of this security problem is possible when users voluntarily interact with the attack mechanism by opening a malicious file. As already mentioned, we call such problems as "voluntary" security problems. On the other hand, we call the problems that do not require any voluntary interaction from users as "involuntary" security problems.

Information on whether a security problem requires user to voluntarily interact with an attack mechanism or not, is available in the National Vulnerability Database, a U.S. government repository devised to manage vulnerability data. Table 2.2 shows the total number of security problems along with the number of "voluntary" type security problems. On the average 24% of the security problems are of "voluntary" type. In projects like Suse Linux (7.14%), voluntary exposure may not be significant. But for projects like Firefox, Internet Explorer, SeaMonkey, etc., it is significant.

[12]http://www.adobe.com/support/security/advisories/apsa09-02.html

2.3.3 Security Exploits

We define "exploit" as the proof-of-concept code or a piece of source code that shows how one can practically turn the vulnerability in to a failure. A security problem exploit is either known or not. Security problems in the OSVDB contain details on whether a security problem has a public exploit (we call them "exploit known" problems), or not. The category of "exploit unknown" problems tells us that an exploit is not known for a security problem. However, it is possible that the security problem may have an exploit that is not publicly known and may or may not have been turned in to failures. Also, for the security problems in the OSVDB, there is no process to distinguish between a proof of concept code and an actual field failure. Security problems tagged with a public exploit known may indicate either a proof of concept code or an actual field failure. Nevertheless, we still are interested in observing the process of correcting security problems knowing that the problems have a public exploit either in the form of an exploit code or an actual failure. Table 2.3 shows the proportion of vulnerabilities with exploits known and unknown. From the table, we find that security problems with exploits known constitute about 35% of the security problems found in the OSVDB. Existing studies capture the process of correcting the overall set of security problems [8]. Since the security problems with exploits known constitute a significant fraction, it is worthwhile to study the process of correcting these problems.

Knowing that a security problem has a public exploit is likely to prompt the developers to fix problems faster. For example, consider the RedHat vulnerability #12727[13] from OSVDB. This problem was disclosed on Jan 06, 2005 and an exploit was disclosed on Feb 12, 2005. A problem report corresponding to this vulnerability is maintained in the RedHat's bug database with the bug id #144099[14]. Soon after an exploit for the security problem was disclosed on Feb 12, 2005, the security response team closed the bug with a fix and issued a security update or patch on Feb 15, 2005. Although the developers may have been working on the problems, it appears that disclosure of a public exploit prompted the security response team in completing the correction activity and releasing the patch. In this paper, we study quantitatively how often security problems are disclosed, how many receive public exploits, and how quickly they are corrected. We distinguish the process of correcting such problems from that of the security problems for which an exploit is unknown.

Table 2.4 shows the data for security problems with public exploits - a) those problems where the exploit was reported first and the problem was disclosed immediately, b) those problems where the exploit was reported first, but the problem was disclosed after a certain period of time, and c) those that received a public exploit after disclosure of the security problem. We can observe that the proportion of security problems that received public exploits is about 35%

[13] http://osvdb.org/show/osvdb/12727

[14] $https://bugzilla.redhat.com/show_activity.cgi?id = 144099$

Table 2.3: Data on exploits known/unknown

Category	Number of security problems	Percentage
Exploit known	15508	35.48%
Exploit unknown	28202	64.52%
Total	43710	100.00%

Table 2.4: Voluntary/Involuntary

Total number of security problems : 43710 (OSVDB)		
Category	Involuntary	Voluntary
Exploit reported first and problem disclosed immediately	13306 (30.4%)	1489 (3.4%)
Exploit disclosed first and problem disclosed after a certain time period	168 (0.4%)	20 (0.1%)
Exploit received after public disclosure of the problem	463 (1.1%)	62 (0.2%)
Exploit known (total)	13937 (31.9%)	1571 (3.6%)
Exploit unknown (total)	24166 (55.3%)	4036 (9.2%)
Total	38103 (87.2%)	5607 (12.8%)

(inclusive of voluntary and involuntary type security problems). About 65% (55.3% and 9.2%) of the security problems remain with exploit unknown. Compared to the proportion of security problems observed for individual projects in Table 2.4, the overall percentage of voluntary type security problems in the OSVDB database is about 12.8%. The data from OSVDB is a pool of security problems for a large number of products. While studying such a collection of security problems may give an overall picture, it is still necessary to study individual products in detail to obtain a self-consistent set of probabilities. Tables 2.5 shows the classification for Bugzilla product. Table 2.5 shows Bugzilla data along with the exploit details. From Table 2.5, we find that the proportion of Bugzilla product security problems exploited is about 27%. This is consistent with the rates reported by other researchers [1].

2.3.4 Security Failures

We collected data from the NVD to identify what percentage of security problems result in actual field failures. Table 2.6 shows the total number of security problems disclosed through

Table 2.5: Bugzilla security problems

Total no of security problems : 107		
Category	Voluntary	Involuntary
Exploit reported first and problem disclosed immediately	0 (0%)	28 (26.17%)
Exploit received after disclosure of the problem	0 (0%)	1 (0.93%)
Exploit known (total)	0 (0%)	29 (27.1%)
Exploit unknown (total)	9 (8.41%)	69 (64.49%)
Total	9 (8.41%)	98 (91.59%)

Table 2.6: Security failures

No	Product	Total number of security problems	Number of security problems failed	Percentage
1	Windows XP	558	12	2.15%
2	Internet Explorer	113	4	3.54%
3	Firefox	418	2	0.48%
4	Acrobat Reader	156	7	4.49%
5	Dokuwiki	26	1	3.85%
6	OpenOffice	32	1	3.13%
7	Redhat Linux	213	2	0.94%
Average number of security problems failed				1.91%

the NVD for a few products and the number of security problems that were exploited in the field for each product. The data shown in the table represent entries in the NVD from January, 2007 to December, 2010. From the table, we observe that, on the average 2% of the total security problems reported for a product result in field failures.

Table 2.7 shows the data collected from Symatec[15] repository on security threats. Information on security threats includes virus, worms, trojans, etc. In Table 2.7 a low impact security threat implies that the number of systems infected is in the range 0 to 49, a medium spread security threat implies that the number of systems infected is in the range 50 to 999 and a broad spread security threat implies that the number of systems infected is more than 1000. The database does not have further information on the broad spread security threats. But a study on some of the popular security threats reported during the last 10 years (Table 2.8)

[15]http://us.norton.com/security_response/threatexplorer/index.jsp

Table 2.7: Symantec information on security threats (2000-2010)

No	Security Threat	Number of threats	Percentage	Number of systems infected per threat
1	Low impact	7882	95%	0-49
2	Medium spread	270	3%	50-999
3	Broad Spread	137	2%	1000+
4	Total	8289	100%	

Table 2.8: Broad spread security threats

No	Threat Name	Systems Infected	Voluntary
1	Morris	6000	No
2	Melissa	1000000	Yes
3	ILOVEYOU	50000000	Yes
4	CodeRed	359000	No
5	SirCAM	2300000	No
6	KLEZ.H	391000	Yes
7	SQLSlammer	74855	No
8	SoBIG	1000000	Yes
9	MSBlaster	8000000-16000000	No
10	MyDoom	1000000	Yes
11	Sasser	200000-1000000	No
12	Storm	1600000	Yes
12	Conficker	8976038	No
13	Chernobyl CIH	700000-1000000	Yes
14	July 09 Cyber Attack	50000	No

revealed that broad spread security threats impacted about 50,000 to 50 million systems.

2.4 Case Study - Security Problems in Linux Distributions

Existing work like [6, 35] focuses primiarily on the analysis of the general category of problem reports, application of reliability models on security problems [8], estimation of security failure rates [58], etc. Although, security problem reports and failures, are a subset of the general category of problems and failures [50], our earlier work [8] showed that security problem reports have characteristics different from the general category of problem reports in terms of reporting and correcting problems across releases. To evaluate the security of a project, it is necessary to know how often one is exposed to security problem reports, how many receive public exploits, how many fail in the field, which of those problem reports and how quickly

they are corrected, etc. We study security problem reports from eight releases of Fedora[16], nine releases of Ubuntu[17], four releases of RedHat Enterprise Linux (RHEL)[18] and two releases of Suse Linux[19] distributions. We compute quantitative measures by extending software reliability metrics like mean time to problem report, mean time to repair, system usage, rates, etc., to the security space. Using the measures, we analyse which of those security problem reports and how promptly they are corrected, and discuss the security of each of the projects.

2.4.1 Metrics

Figure 2.6 shows a sample window of three periods encompassing four problem reports. The problem reports are marked with the time when the problems were reported (R_i) and fixed(F_i). In Figure 2.6, X_i gives the time to problem report (TTPR), the amount of time system is operational without a problem being reported, Z_i gives the time to problem exploit (TTPE), the amount of time system is operational without a problem being exploited, and Y_i gives the time to problem correction (TTPC), the amount of time taken to fix a problem report. Following the empirical study, I discuss

Mean Time to Problem Report

Mean Time To Problem Report(MTTPR) is the mean or average of all times to problem report during a particular interval. This is analogous to mean-time-to-failures which is the average amount of time a system is operational without failures during a particular time intervial. In the period i-1, the Mean Time to Problem Report would be estimated as the average of X1 and X2.

Mean Time to Problem Exploit

Mean Time To Problem Exploit(MTTPE) is the mean or average of all TTPEs during a particular interval. We define MTTPE as the average amount of time a system is operational without exploits during a particular time interval. In the period i, the MTTPE would be estimated as the average of Z2 and Z3.

Mean Time to Problem Correction

Mean Time To Problem Correction(MTTPC) is defined as the average of all times to problem correction (TTPC) during a particular interval. Once the problem is reported, on the average,

[16] http://fedoraproject.org/
[17] http://www.ubuntu.com/
[18] http://www.redhat.com/
[19] http://www.novell.com/linux/

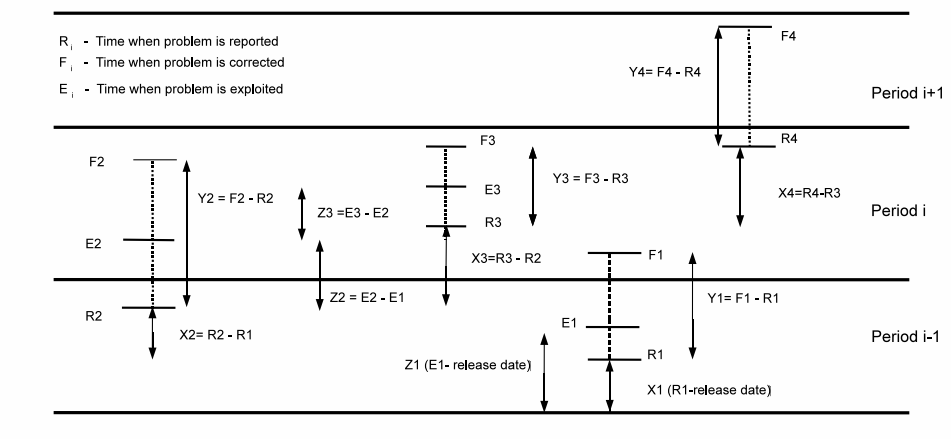

Figure 2.6: Problem report, exploit and correction time intervals

it might take that long to apply a fix and terminate exposure to that particular problem. For period i in Figure 2.6, Y2 and Y3 indicate the time spent to fix problem reports 3 and 4. The total correction time spent in the period i is given by the sum of time periods Y2 and Y3. The Mean Time to Problem Correction is given by the total correction time divided by the number of problems reported in a period. For period i, the Mean Time to Problem Correction is the total correction time divided by two (since only problems 3 and 4 were reported in that period). There is one thing to note, however. Once a problem is reported, its TTPC may be longer than the next window of interest. In that case, exposure to announced or open problems needs to include not only the problems found in that particular period, but also the ones found earlier and not yet resolved.

System Usage

Ideally, we would have liked to have the usage information for every system running a FEDORA release. Usage information like the total number of systems in operation during a given time, the total operation time as well as the downtime for each FEDORA release, etc. Due to the unavailability of such information, we focused on the download statistics maintained by FEDORA project. What is available in the database is the number of downloads and the number of registrations of the FEDORA product on weekly basis. We use this information to estimate the number of installations that might run FEDORA and from that make an estimate of an upper bound on inservice time.

We assume that at any one time, the number of downloads or installations or unique IP addresses represent the total number of operational systems. Although the number of systems operational in the field may be higher, FEDORA Statistics affirm that their numbers provide a

reasonable estimate of the actual FEDORA usage[20]. Based on the download statistics, we can calculate the inservice time i.e., product of the total number of systems and the time they have been operational. The FEDORA download statistics is only available from FEDORA release 6. Therefore our analyses were done on FEDORA releases 6, 7 and 8.

Calendar time or the operating time is defined as the cumulative sum of time after release of the software [50]. Inservice time refers to cumulative sum of the execution time of software [50, 19]. In our case, an upper bound on the inservice time can be estimated as

$$Inservice\ time(t) = \sum_{i=1}^{t} \Delta t_i \times n_i \tag{2.1}$$

$$\Delta t_i = t_i - t_{i-1} \tag{2.2}$$

where Δt_i is time interval at i, and n_i is the total number of installations (registrations) in time interval i. In our analysis, the inservice time has been calculated from the point the software was released until the time for which the usage statistics were available.

Rates

In our analysis, we estimate problem exposure rates, problem correction rates, and problem exploit rates. We define problem exposure rate as the number of unresolved or open problems a system has in the time period of interest. These problems can be hidden, or they may have been publicly disclosed. They can refer to any type of problem, or to security problems only. In our case, we focus on security problems that have been publicly disclosed but not yet fixed during a particular time period. Since an end-user is exposed to not only new problems disclosed during a particular time period, but also unresolved problems from the previous time period which could also potentially be exploited by any malicious user, we need to consider problems reported during the current time period as well as unresolved problems from the previous time periods. This is different from the problem report rate that identifies only the unique number of problems reported during a particular time periods. We call this as "problem disclosure rate". We define problem correction rate as the number of security problems resolved in the time period of interest. We define problem exploit rate as the number of security problems that receive exploits (including exploits and actual failures in the field) in the time period of interest.

[20] http://fedoraproject.org/wiki/Statistics

Table 2.9: Fedora Statistics

Statistics	F 1	F 2	F 3	F 4	F 5	F 6	F 7	F 8
Security(PR_s)	31	128	82	154	201	194	92	26
Non-security(PR_{ns})	3141	4124	7528	7644	7805	7877	5468	4490
Total(PR_t)	3172	4206	7656	7798	8006	8071	5560	4516
% of (PR_s)	0.98	3.01	1.07	1.97	2.51	2.4	1.65	0.58
High	13	54	46	63	74	51	23	8
High (%)	42%	42%	56%	41%	37%	26%	25%	31%
Medium	15	62	26	73	93	113	60	16
Medium(%)	48%	48%	32%	47%	46%	58%	65%	62%
Low	3	12	10	18	34	30	9	2
Low(%)	10%	9%	12%	12%	17%	16%	10%	8%
Downloads	-	-	-	-	-	2864875	1920667	2152583
From	-	-	-	-	-	10/24/06	04/23/07	11/08/07
To	-	-	-	-	-	05/31/07	11/07/07	02/22/08

Problem exposure rates

Let the problem exposure rate [8] for publicly disclosed security problems (λ_d) in the time interval i be estimated as,

$$\lambda_d \;=\; \frac{((n_o)_i + (n_f)_i)}{(n_i \times \Delta t_i)} \tag{2.3}$$

where $(n_o)_i$ represents the number of open problem reports from the previous week, $(n_f)_i$ represents number of problems reported in time interval Δt_i (e.g., given in minutes). Typically, the number of users (n_i) in time interval Δt_i is very large compared to the total number of problem reports considered $((n_o)_i + (n_f)_i)$ [8]. It is worth noting again that λ_d is different from the inverse of the $MTTPR_i$. The latter reflects arrival time of unique problem reports in the calendar time frame, while the former reflects the number of open reports with respect to the inservice time during time period i. We may wish to use λ when discussing the quality of the system as a whole, and MTTPR in the context of calendar-based problem correction rate. In the event one is interested in computing the problem disclosure rate, then only the number of problems reported in time interval Δt_i, i.e., $(n_f)_i$ needs to be considered.

Problem correction rates

The problem correction rate for publicly disclosed problems (inclusive of voluntary and involuntary problems) can be estimated through the inverse of the Mean Time To Problem Correction (MTTPC), where this time refers only to disclosed security problem reports in time frame, i.e.,

$$\mu_d = \frac{1}{MTTPC_i} \tag{2.4}$$

Problem exploit rates

In practice, data such as the exploits, actual field failures, operational profile of the system, etc., are required to estimate system exploit rate accurately. Information about the actual field failures was not available in the case of Fedora and the number of problem reports does not tell how many failures are experienced by an end-user. However empirical results from [1] show on the average the proportion of publicly disclosed as well as undisclosed problems received exploits. We use this to estimate the exploit rates. We estimate exploit rates for publicly disclosed security problems using [58]. If we assume that P_e is the probability that a publicly disclosed problem report received an exploit,

$$\lambda_d = \frac{1}{n_i} \times \frac{(n_f)_i + (n_o)_i}{\Delta t_i} \times P_e \tag{2.5}$$

$$\lambda_{de} = \lambda_d \times P_e \tag{2.6}$$

Where λ_{de} is an estimate of Fedora exploit rate for publicly disclosed problems. In our illustration, we use $P_e = 0.22$ for the proportion publicly disclosed problems that received exploits from [1]. Next, we present an empirical study of security problem reports from the Fedora, the Ubuntu, the RedHat Enterprise Linux (RHEL) and the Suse Linux distributions, analyse and discuss which type of security problem reports and how frequently they are reported, and how promptly they are corrected.

2.4.2 Data

Table 2.9 shows the number of problem reports and installations running more recent releases of Fedora. Information like the severity (high, medium or low) for each security problem were collected from the National Vulnerability Database[21]. Rows *High*, *Medium*,*Low* show the number of security problems under each category and their percentage in the total number of problems PR_s. Tables 2.10, 2.11 and 2.12 show similar statistics for Ubuntu, RHEL and Suse. We consider system usage to analyse security from the perspective of the system as a whole, i.e., usage of the total number of systems in operation during a particular time period, the total operation time as well as downtime for each linux installation, etc. We use inservice time [58] to approximate the system usage as a whole. We calculate the inservice time as the product of the total number of downloads or installations (n_i) during a particular time period and the time they have been operational [58]. We use week as the time period of interest. The Fedora

[21] http://nvd.nist.gov/

Table 2.10: Ubuntu Statistics

Statistics	4.10	5.04	5.10	6.04	6.10	7.04	7.10	8.04	8.10
Security(PR_s)	107	51	54	34	58	186	336	192	87
Non-security(PR_{ns})	4177	5145	11248	6947	9029	9482	12085	9876	3388
Total(PR_t)	4284	5196	11302	6981	9087	9668	12421	10068	3475
% of (PR_s)	2.50	0.98	0.48	0.49	0.64	1.92	2.71	1.91	2.57
High	47	23	24	16	22	52	134	70	19
High(%)	44%	45%	44%	47%	38%	28%	40%	36%	22%
Medium	38	23	23	16	32	108	178	107	56
Medium(%)	36%	45%	43%	47%	55%	58%	53%	56%	64%
Low	22	5	7	2	4	26	24	15	12
Low(%)	20%	10%	13%	6%	7%	14%	7%	8%	14%

download statistics is only available from Fedora release 6 onwards. Therefore our analyses on Fedora as a whole have been done on releases 6, 7 and 8.

2.4.3 Numerical Results and Discussion

We illustrate the rates computed for Fedora 6, Ubuntu 8.04, RHEL 5, and Suse Linux 10.2 and discuss the results.

Fedora Security

We consider problem exposure rates from two perspectives. That of Fedora as a whole with n_i systems running during a particular week, and that of a particular system that may be under attack. In the first case, there are n_i systems running Fedora during a particular week. Over these systems, a random Fedora user is exposed to $(n_o)_i + (n_f)_i$ unique problems. This includes the unique problems reported weekly and unique problems unresolved from previous week. Since n_i is very large (Table 2.9), the exposure rate of a random Fedora user to unique problems during any particular week is usually very small. From the perspective of Fedora as a whole, exposure rate to unique problems is relatively low, and a relatively small number of patches should be able to take care of those problems. The exposure rate for a random anonymous system is of the order of 10E-09 security problems per minute (4 problems per week, Table 2.13). From the perspective of Fedora as a whole, these 4 unique problems are reported only from running a very large number of Fedora systems (about 1 to 2 million Fedora systems). The Fedora security correction rate (Table 2.13) is thus adequate for the product as a whole. Further, not all of these problems actually receive exploits. For example, [1] show that about only 22% of disclosed problems receive exploits.

Table 2.11: RedHat Enterprise Linux Statistics

Statistics	RHEL 2.1	RHEL 3	RHEL 4	RHEL 5
Security (PR_s)	127	125	458	112
Non-security(PR_{ns})	1528	6400	7682	7708
Total(PR_t)	1655	6525	8140	7820
% of (PR_s)	7.6	1.9	5.6	1.4
High	20 (16%)	23 (18%)	155 (34%)	27 (24%)
Medium	43 (34%)	51 (41%)	215 (47%)	66 (59%)
low	64 (50%)	51 (41%)	88 (19%)	19 (17%)

From the perspective of an individual system identified and targeted, the exposure rate is about 10E-04 (4 problems per week, Table 2.13). The correction rate (Table 2.13) is smaller than the exposure rate of a system that may be under attack. For example, let us consider the problem exposure (λ_d) and correction rates (μ_d) for disclosed security problems. On an average during a particular week, out of 4 problems, only half of them are corrected. This leaves the system exposed to approximately 2 additional problems per week. This implies a backlog in the overall problem resolution. Also, from Table 2.13, we can observe that the correction rate for high and medium severity problems is enough to keep up with the disclosed, but that of low severity problem reports is not. Resolution of such high and medium severity problems with no backlogs is a very good news for Fedora project. If one assumes that only about 22% of the disclosed problems receive exploits [1], the correction rate for high and medium severity problems is good enough to keep up with the exploitable (Table 2.13). Certainly, as an individual user who may not be typically singled out for an attack, one might feel comfortable knowing that there are nearly about 2 million systems operating per week and that chances of a random attack are relatively low. On the other hand, those that may be running high visbility sites, have a higher probability of being targetted. Even in such cases of being singled out, one may feel safe considering that correction rates for high and medium severity problems keep up with exposure rates.

Ubuntu Security

On an average during a particular week, out of 7 disclosed problems, only less than one problem is corrected. This leaves the system exposed to approximately 6 problems per week. This implies a significant backlog in the problem resolution. From Table 2.13, we can observe that the correction rate for low severity problems is enough to keep up with the disclosed, but that of high and medium severity problems are not. Even under the assumption that only about 22% of the disclosed problems receive exploits [1], the resolution rates do not keep up with

Table 2.12: SuseLinux Statistics

Statistics	Suse 10.1	Suse 10.2
Security(PR_s)	19	37
Non-security(PR_{ns})	3104	3871
Total(PR_t)	3123	3908
% of (PR_s)	0.61	0.95
High	10(53%)	13(35%)
Medium	5(26%)	19(51%)
Low	4(21%)	5(14%)

the exploitable. On the average 33% (resulting from λ_d and μ_d in Table 5) of the exploitable high severity problems and 40% of the exploitable medium severity problems reported per week remain unresolved.

Suse Security

The overall resolution rate for the disclosed security problems does not show presence of any backlog. High and medium severity problems keep up well with the disclosed problems. We can observe a backlog with the low severity problems which may not be significant.

RedHat Enterprise Linux Security

The overall resolution rate for the disclosed security problems as well as that of high and medium severity problems show presence of backlogs. On the average 20% (resulting from λ_d and μ_d in Table 2.13) of the high severity problems reported per week remain unresolved. But if one considers that only about 22% of the disclosed problems receive exploits [1], then the problem correction rates are good enough to keep up with the exploitable.

Table 2.13: Rates and Average values

Rates	Fedora (whole)	Fedora (Individual)	Ubuntu (Individual)	Suse (Individual)	RedHat (Individual)
Problem exposure rates in security problem reports/minute (per week)					
λ_d	1.6E-09	3.6E-04 (4)	7.1E-04 (7)	8.2E-05 (1)	2.6E-04 (3)
λ_h	2.7E-10	1.1E-04 (1)	2.6E-04 (3)	3.7E-05 (0.4)	5.5E-05 (0.5)
λ_m	9.7E-10	1.8E-04 (2)	3.9E-04 (4)	3.7E-05 (0.4)	1.6E-04 (1.6)
λ_l	3.4E-10	6.1E-05 (0.6)	5.5E-05 (0.6)	1.2E-05 (0.1)	4.2E-05 (0.4)
Exploit rates in security problem reports that received exploits/minute (per week)					
λ_{de}	3.5E-10	7.9E-05 (0.8)	1.6E-04 (2)	1.8E-05 (0.2)	5.7E-05 (0.6)
λ_{he}	6.0E-11	2.5E-05 (0.3)	5.7E-05 (0.6)	8.2E-06 (0.1)	1.2E-05 (0.1)
λ_{me}	2.1E-10	4.0E-05 (0.4)	8.7E-05 (1)	8.2E-06 (0.1)	3.5E-05 (0.4)
λ_{le}	7.5E-11	1.3E-05 (0.1)	1.2E-05 (0.1)	2.7E-06 (0.03)	9.3E-06 (0.1)
Correction rates in security problem reports/minute (per week)					
μ_d	1.9E-04 (2)		4.3E-05 (0.4)	1.2E-03 (12)	3.8E-05 (0.5)
μ_h	3.2E-04 (3)		3.7E-05 (0.4)	8.5E-004 (9)	3.5E-05 (0.4)
μ_m	2.0E-03 (20)		5.7E-005 (0.6)	6.5E-04 (7)	5.2E-05 (0.5)
μ_l	2.3E-05 (0.2)		5.7E-04 (6)	1.7E-05 (0.2)	4.5E-05 (0.5)

Chapter 3

Security Problem Response Model

3.1 Introduction

To analyse and understand collected information, we developed a model that describes the interactions of events associated with users, developers, attackers and system security. The model is quite comprehensive, and distinguishes itself from published models by emphasizing role perspectives [26, 15, 56].

3.2 Model

In this section, we present a model based on our analysis of data on security problems in Chapter 2 (Section 2.3). The model captures the states through which a system may go based on the discovery of security problems by users, developers or attackers, type of security problems (characterized based on user involvement in the attack mechanism - voluntary or involuntary security problems), exploit status (an exploit for a security problem is known or unknown), failure status (an attacker exploits a security problem in the field failure or not), and their correction status (a developer fixes a security problem or not). The model describes the states generic for both voluntary and involuntary security problems. Figure 3.1 shows the security model. Self transition for each state is not shown for the sake of clarity. The security states are described as follows.

- **good** represents the start state where the system may contain undisclosed and unknown security problems. A transition to the same state indicates that such problems, if present, remain with the exploit, failure and fix status unknown.
- **exploit$_{(h)}$** represents the state where a security problem is first known publicly through an exploit, i.e., an exploit for the security problem is disclosed before developers discover

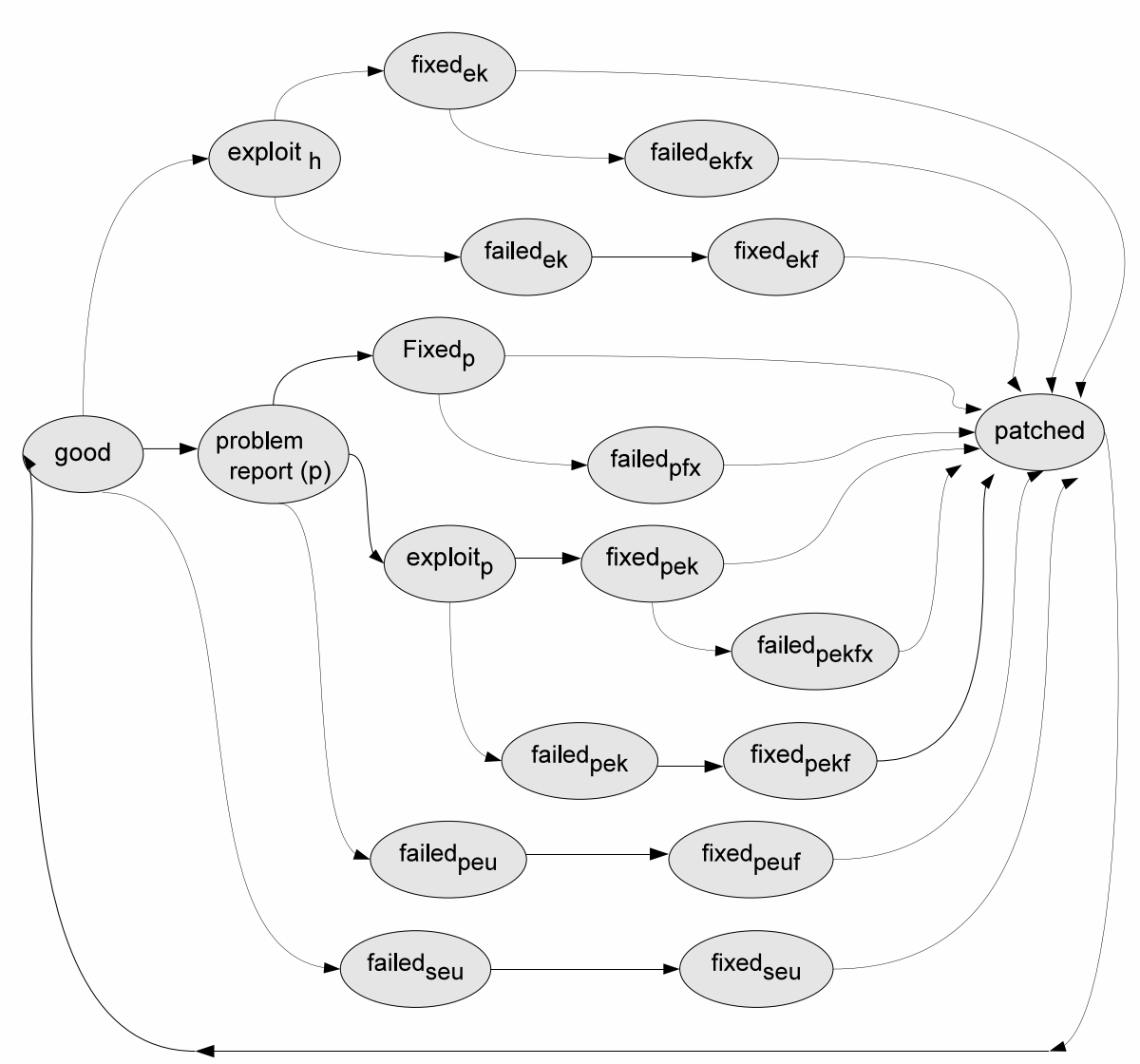

Figure 3.1: Security problem response model

the security problem. A transition to the same state indicates that such problems are neither fixed by the developers nor exploited by the attackers. This occurs when the exploit published is not of enough severity to attract the attention of developers and attackers.

- **fixed$_{(ek)}$** represents the state where a security problem disclosed in the form of an exploit is fixed by the developers. We use "fixed" to indicate not only the fix of a security problem by the developers, but also the release of the security patch to the users.

- **failed$_{(ek)}$** represents the state where a security problem disclosed in the form of an exploit is exploited by attackers before developers fix the problem. Attackers take advantage of the public exploit code and causes field failures.

- **failed$_{(ekfx)}$** represents the state where a security problem disclosed in the form of an exploit and fixed by developers is exploited by attackers before users apply the security patch. Attackers take advantage of the public exploit code and cause field failures.

- **fixed$_{(ekf)}$** represents the state where a security problem disclosed in the form of an exploit and exploited by attackers is fixed by the developers.

- **problem report (p)** represents the state where developers discover a security problem first and disclose it in the form of a problem report.

- **fixed$_{(p)}$** represents the state where a security problem disclosed in the form of a problem report is fixed by developers before attackers exploit the security problem in the field.

- **failed$_{(pfx)}$** represents the state where a security problem disclosed in the form of a problem report and fixed by developers is exploited by attackers in the field before users apply the security patch.

- **exploit$_{(p)}$** represents the state where an exploit for a security problem disclosed in the form of a problem report is publicly disclosed.

- **fixed$_{(pek)}$** represents the state where a security problem disclosed in the form of a problem report and a public exploit for the problem is known, is fixed by the developers.

- **failed$_{(pekfx)}$** represents the state where a security problem disclosed in the form of a problem report and a public exploit is known for the problem, and fixed by the developers, is exploited in the field before users apply the security patch.

- **failed$_{(pek)}$** represents the state where a security problem disclosed in the form of a problem report and a public exploit is known for the problem, is exploited by the attackers before the security problem is fixed by developers.

- **fixed$_{(pekf)}$** represents the state where a security problem disclosed in the form of a problem report and a public exploit is known for the problem, and exploited by attackers in the field before the security problem is fixed by developers, is now fixed by the developers.

- **failed$_{(seu)}$** represents the state where a secret or hidden security problem is exploited by attackers before developers discover the problem.

- **fixed$_{(seu)}$** represents the state where a secret or hidden security problem exploited by attackers is fixed by developers.

- **patched** represents the state where users apply the security patch released.

Next, we present an application of the security problem response model in describing the security problems from from Open Source Vulnerability Data Base.

3.3 Case Study - Security Problems from Open Source Vulnerability Data Base

In this section, we discuss the security model based on estimated rates from the OSVDB data. To be consistent with the data from the OSVDB, we present a reduced state space of the security model where an exploit would include both exploit code as well as failure of security problems in the field. In addition, we generate states for both voluntary and involuntary type of security problems. The states are described as follows.

- **good** or **hidden** represents the start state where the system may contain undisclosed and unknown problems. A transition to the same state indicates that such problems, if present, remain with the exploit and fix status unknown.

- **disclosed$_{(v)}$** represents the disclosure of "voluntary" security problems with exploits unknown. The disclosure is primarily through means other than public exploits. For example, a vendor could identify the problems during testing. A transition to the same state indicates that such problems remain with the exploit and fix status unknown. Similarly **disclosed$_{(nv)}$** represents the disclosure of "involuntary" security problems with exploit unknown. **disclosed$_{(sv)}$** and **disclosed$_{(snv)}$** represent the disclosure of 'voluntary", and "involuntary" security problems due to a public exploit.

- **exploit$_{(v)}$** represents the exploit of disclosed "voluntary" security problems. Similarly **exploited$_{(nv)}$** represents the exploit of disclosed "involuntary" security problems. The states **exploit$_{(sv)}$** and **exploit$_{(snv)}$** represent the exploits of hidden "voluntary" and "involuntary" security problems.

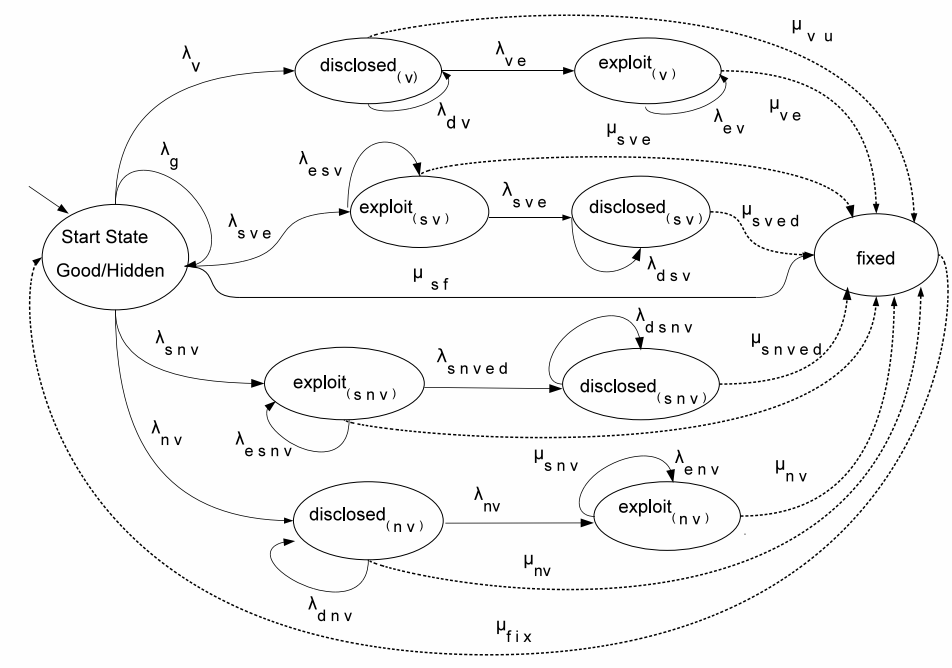

Figure 3.2: Security model

- **fixed$_{(s)}$** represents the system state where the security problems are fixed. From the data, we observed that for those problem reports that were fixed, the patch release occurred before the problem reports were updated in the bug tracking system as fixed. Hence the state **fixed** implies that the security problem is fixed and the patch has been released. For the transition to occur, the user must apply the patch.

Figure 3.3 shows the reduced state space model with transition rates. The basic assumption is that in the very small period dt not more than one transition takes place. The transition rates from one state to the other are as follows:

- λ_v (**good/hidden** to **disclosed$_{(v)}$**) - rate of disclosure of "voluntary" security problems. Similarly λ_{nv} represents the disclosure of "involuntary" problems.

- λ_{ve} (**disclosed$_{(v)}$** to **exploit$_{(v)}$**) - rate of exploit of disclosed "voluntary" security problems. Similarly λ_{nve} , λ_{sve}, and λ_{snve} represent the exploits of disclosed "involuntary" security problems, hidden "voluntary" and "involuntary" security problems.

- λ_{sved} (**exploit$_{(sv)}$** to **disclosed$_{(sv)}$**) - rate of disclosure of "voluntary" security problems along with exploits. Similarly λ_{snved} represents the disclosure of "involuntary" security problems along with exploits. Note that we are interested in only the first exploit and

not subsequent exploits. Hence no transition to exploits from the states **disclosed$_{(sv)}$**) and **disclosed$_{(snv)}$**). This is because we are interested in observing the influence of the information that an exploit for a problem exists, on the process of correcting the problem. Not the failures.

- μ_{vu} (**disclosed$_{(v)}$** to **fixed**) - rate of fix of disclosed "voluntary" security problems with exploit unknown. Similarly μ_{nvu} represent fixing of disclosed "involuntary" security problems with exploit unknown.

- μ_{ve} (**exploit$_{(v)}$** to **fixed**) - rate of fix of disclosed "voluntary" security problems with exploit known. Similarly μ_{nve} represents fixing of disclosed "involuntary" security problems with exploit known, and μ_{sve}, and μ_{snve} represent fixing of "voluntary" and "involuntary" security problems disclosed immediately along with exploits.

- μ_{sved} (**disclosed$_{(sv)}$** to **fixed**) - rate of fix of disclosed "voluntary" security problems with exploit known. Similarly μ_{snved} represent fixing of disclosed 'involuntary" problems with exploit known.

- μ_{sf} (**good** to **fixed**) - rate of fix of security problems before being disclosed or exploit is publicly known. Similar to μ_{sve}, and μ_{snve}, we include μ_{sf} for completeness of the model.

- μ_{fix} (**fixed** to **good**) - rate of release of fixes for the security problems.

The rates λ_{dv}, λ_{ev}, λ_{g}, λ_{esv}, λ_{dsv}, λ_{esnv}, λ_{dsnv}, λ_{dnv} ,λ_{env} for the states {$disclosed_{(v)}$, $exploit_{(v)}$, Good, $exploit_{(sv)}$, $disclosed_{(sv)}$, $exploit_{(snv)}$, $disclosed_{(snv)}$, $disclosed_{(nv)}$, and $exploit_{(nv)}$ } represent transitions to the same state itself. We base our analysis on information such as the discovery date, disclosure date, exploit date, and fix date of security problems, project download statistics, etc. Using the data, we compute rates of disclosure, exploit, and correction of security problems with respect to inservice time as well as calendar time, and discuss how promptly problems with exploits known are resolved compared to those for which an exploit is still unknown.

3.3.1 Numerical Results and Discussion

Table 3.1 shows the rates computed for FEDORA as a whole using inservice time and from the perspective of a single system using calendar time. Considering inservice time, the proportion of unique voluntary and involuntary problems that receive exploits during the usage of about 2 million systems in a particular week, is very low. Further, the correction rate is very high compared to the exploit rate with respect to inservice time that even the very low percentage of security problems with exploit known gets patched quickly. Considering calendar time, we find

Table 3.1: Fedora rates

Rates	Fedora (whole)	Fedora (single)
Security problem disclosure rates in security problems/min (per week)		
Voluntary	2.5E-10	8.0E-05 (1)
Involuntary	3.8E-10	2.8E-04 (3)
Security problem exploit rates in security problems exploit/min (per week)		
Voluntary	5.6E-11	4.3E-05 (0.6)
Involuntary	8.4E-11	1.2E-04 (1.4)
Security problem correction rates in security problems/min (per week)		
Voluntary	4.1E-05 (0.4)	
Involuntary	2.0E-04 (2.0)	

that the correction rate is enough to patch the involuntary type security problems with exploit known but not the voluntary type problems. This may be because developers may focus more on problems where user does not have control over in failures.

Table 3.3 shows the rates computed using the vulnerabilities collected from the OSVDB database. The table shows the average rates of transition from one state to the other. The rates in this table present an overall view of the general pool of vulnerabilities and cannot be attributed to the general behavior of a single product. On the average, "voluntary" security problems with exploit unknown are disclosed at the rate of 5.5 problems per week. The disclosed "voluntary" security problems are fixed, before receiving exploits, at the rate of 1.59 problems per week and receive exploits before being fixed at the rate of 0.15 problems per week. This shows a 28.9% chance that "voluntary" security problems disclosed per week are fixed before receiving exploits and only a 2.72% chance of receiving exploits before they are fixed. The disclosed "voluntary" security problems remain in the disclosed state, i.e., without the exploit status or fix status known at the rate of 3.86 problems per week. This leaves about 70.18% chance that "voluntary" security problems disclosed per week remain without the exploit or the fix status known. For "voluntary" security problems that receive exploits after disclosure, the correction rate of is high enough that no backlogs are observed.

On the average, "involuntary" security problems with exploit unknown are disclosed at the rate of 17.7 problems per week. The disclosed "involuntary" security problems are fixed, before receiving exploits, at the rate of 2.19 problems per week and receive exploits, before being fixed, at the rate of 0.48 problems per week. This shows a 12.37% chance that "involuntary" security

Table 3.2: Bugzilla rates (number of security problems per minute (per week))

From	To	Mean per minute (per week)
good/hidden to	$disclosed_{(v)}$	3.1E-006 (0.03)
$disclosed_{(v)}$	to fixed	4.6E-004 (4.63)
good/hidden to	$exploited_{(snv)}$	1.1E-005 (0.11)
$exploited_{(snv)}$	$disclosed_{(snv)}$	disclosed immediately
$disclosed_{(snv)}$	fixed	2.4E-003 (23.7)
good/hidden to	$disclosed_{(nv)}$	1.5E-005 (0.15)
$disclosed_{(nv)}$	fixed	1.5E-003 (14.6)

problems disclosed per week are fixed before receiving exploits and only a 2.71% chance of receiving exploits before they are fixed. The disclosed "involuntary" security problems remain in the disclosed state, i.e., without the exploit status or fix status known at the rate of 15.03 problems per week. This leaves about 84.75% chance that "involuntary" security problems disclosed per week remain without the exploit or the fix status known. For the "involuntary" security problems that receive exploits after disclosure, the correction rate of is high enough that no backlogs are observed.

On the average, hidden "voluntary" security problems are disclosed as a result of an exploit at the rate of 3 problems per week. Disclosed "voluntary" security problems with exploit known are fixed at the rate of 2.65 problems per week and remain in the exploit state, i.e., without the fix status known at the rate of 0.35 problems per week. This shows a 88.3% chance that "voluntary" security problems disclosed per week as a result of exploit are fixed and 11.35% chance of remaining without the fix status known.

On the average, hidden "involuntary" security problems are disclosed as a result of an exploit at the rate of 9.9 problems per week. Disclosed "involuntary" security problems with exploit known are fixed at the rate of 5.4 problems per week and remain in the exploit state, i.e., without the fix status known at the rate of 4.5 problems per week. This shows a 55.6% chance that "involuntary" security problems disclosed per week as a result of exploit are fixed and 44.4% chance of remaining without the fix status known.

During a particular week, there is a high chance of security problems remain with exploit and fix status unknown after being disclosed. This only implies that the solution status is unknown but not necessarily that the problems remain unfixed. Since OSVDB updates the problem reports with the workarounds available with references back to the vendor or third

party references, the problem reports may not be updated when the final patch is available from the vendor. In order to identify the availability of the actual patch one may need to refer back to the actual vendor or product. While collecting this data for all vulnerabilities may be tedious, we collect the details for Bugzilla product with the help of associated problem report information in the Bugzilla bug tracking system and discuss the model below.

Table 3.2 shows the rates for a single product - Bugzilla, computed based on the data presented in Table 2.5. Since there no voluntary type problems with exploit known, transitions rates involving voluntary type problems with exploit known are not present in the table. Also, the involuntary type security problems are disclosed immediately upon exploit. From Table 3.2, we find that in general the correction rate for Bugzilla is high enough to keep with the disclosure and exploit rates without any backlogs. We can also observe that voluntary type security problems with exploit known are corrected at a much faster rate compared to the involuntary type problems with exploit unknown. Based on this and presence of no backlogs, one can assess how promptly a project reacts to a security problem knowing whether an exploit for the problem is known or unknown.

Table 3.3: OSVDB rates (number of security problems per minute (per week))

From	To	Mean per minute (per week)
good/hidden	**disclosed$_{(v)}$**	5.43E-004 (5.5)
good/hidden	**disclosed$_{(nv)}$**	1.76E-003 (17.7)
good/hidden	**exploited$_{(sv)}$**	2.98E-004 (3.0)
good/hidden	**exploited$_{(snv)}$**	9.90E-004 (9.9)
disclosed$_{(v)}$	**exploit$_{(v)}$**	1.46E-005 (0.15)
disclosed$_{(v)}$	**disclosed$_{(v)}$**	3.83E-004 (3.86)
disclosed$_{(nv)}$	**exploit$_{(nv)}$**	4.76E-005 (0.48)
exploit$_{(sv)}$	**fixed**	0 (0)
exploit$_{(sv)}$	**disclosed$_{(sv)}$**	2.98E-004 (3.0)
exploit$_{(sv)}$	**exploit$_{(sv)}$**	0 (0)
exploit$_{(snv)}$	**fixed**	0 (0)
exploit$_{(snv)}$	**disclosed$_{(snv)}$**	9.90E-004 (9.9)
exploit$_{(snv)}$	**exploit$_{(snv)}$**	0 (0)
disclosed$_{(v)}$	**fixed**	1.58E-004 (1.59)
disclosed$_{(nv)}$	**fixed**	2.17E-004 (2.19)
disclosed$_{(nv)}$	**disclosed$_{(nv)}$**	1.49E-003 (15.03)
exploit$_{(v)}$	**fixed**	1.83E-004 (1.84)
exploit$_{(v)}$	**exploit$_{(v)}$**	0 (0)
disclosed$_{(snv)}$	**fixed**	5.43E-004 (5.5)
disclosed$_{(snv)}$	**disclosed$_{(snv)}$**	4.36E-004 (4.4)
disclosed$_{(sv)}$	**fixed**	2.63E-004 (2.65)
disclosed$_{(sv)}$	**disclosed$_{(sv)}$**	3.47E-005 (0.35)
exploit$_{(nv)}$	**fixed**	2.80E-004 (2.82)
exploit$_{(nv)}$	**exploit$_{(nv)}$**	0 (0)
good/hidden	**fixed**	N/A
good/hidden	**good/hidden**	N/A

Chapter 4

Predictive Modeling

4.1 Classical Software Reliability Model on Security Data

In this section, we investigate if classical reliability models (such as Logarithmic Poisson Execution Time model [49]) can predict problem rates in the context of operational use of a system. Figure 4.1 shows the problem disclosure rate for Fedora 6. Since the problem exposure rate appears to be a decreasing function of inservice time, we observed that Musa's Logarithmic Poisson Execution Time model [49] can be applied to Fedora 6 and other release data to model the problem exposure rate. Figures 4.1 also shows the LPET fit ($\beta 0 = 82.83, \beta 1 = 4.4E\text{-}06$) for Fedora 6 problem disclosure rate. We see that classical reliability analysis models can be used to describe time dependent behavior of security problem disclosure rates.

Prediction across releases

We observed that different Fedora releases exhibit similar characteristics with respect to security problem report and correction [8]. For example, Figures 4.2 and 4.3 show that the mean time to problem report for Fedora releases 6 and 7 are similar in characteristic.

Based on this, we observed that the parameters estimated for a particular release of the software can be applied to the next release as well. This may be useful in assessing the next version's possible security problem disclosure rate using current version, as well as in planning when to make a transition to a new version. For example, Fedora 7 disclosed problem disclosure rate (Figure 4.4) can be modeled using parameters from Musa's LPET model [49] fit for Fedora 6 (Figure 4.1).

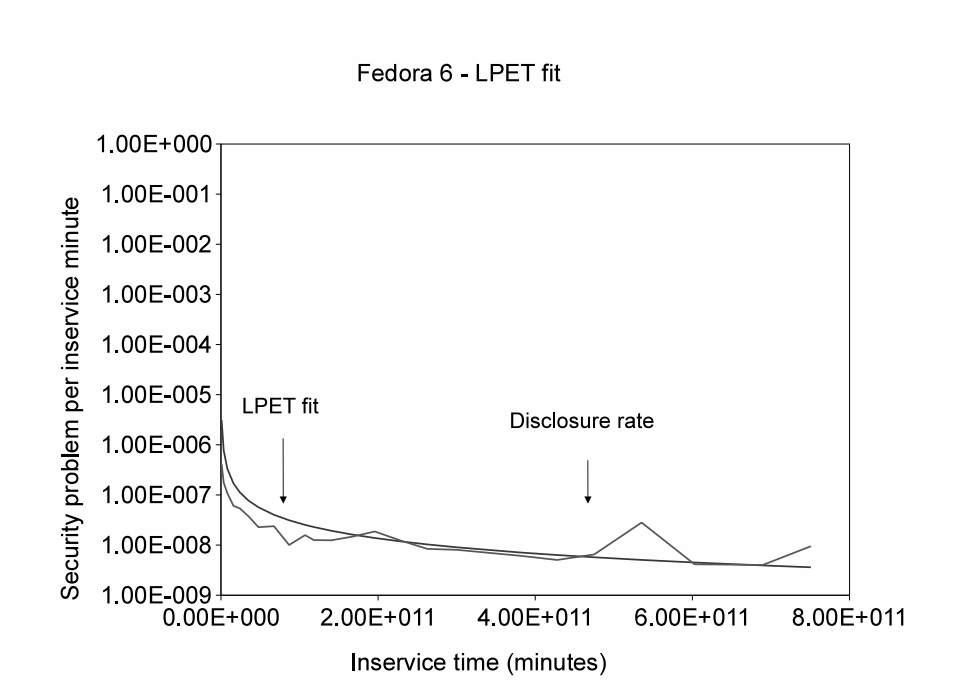

Figure 4.1: Fedora 6 - LPET fit (security problems per minute)

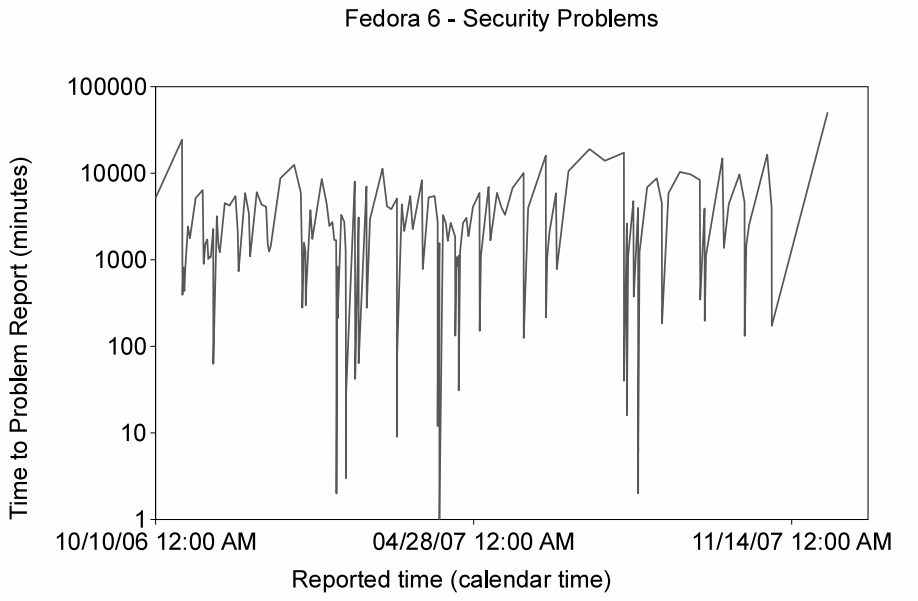

Figure 4.2: Fedora 6 - Security problems

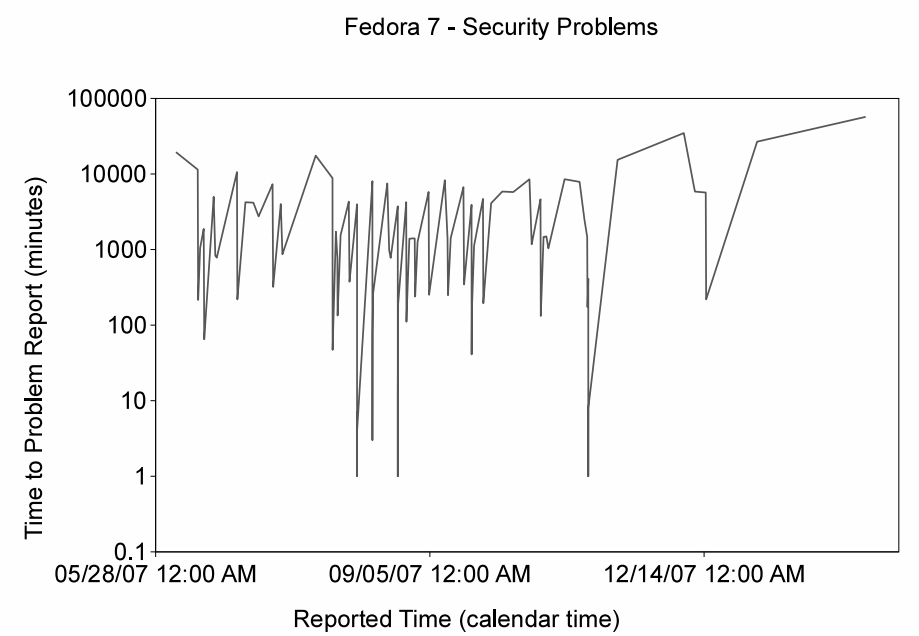

Figure 4.3: Fedora 7 - Security problems

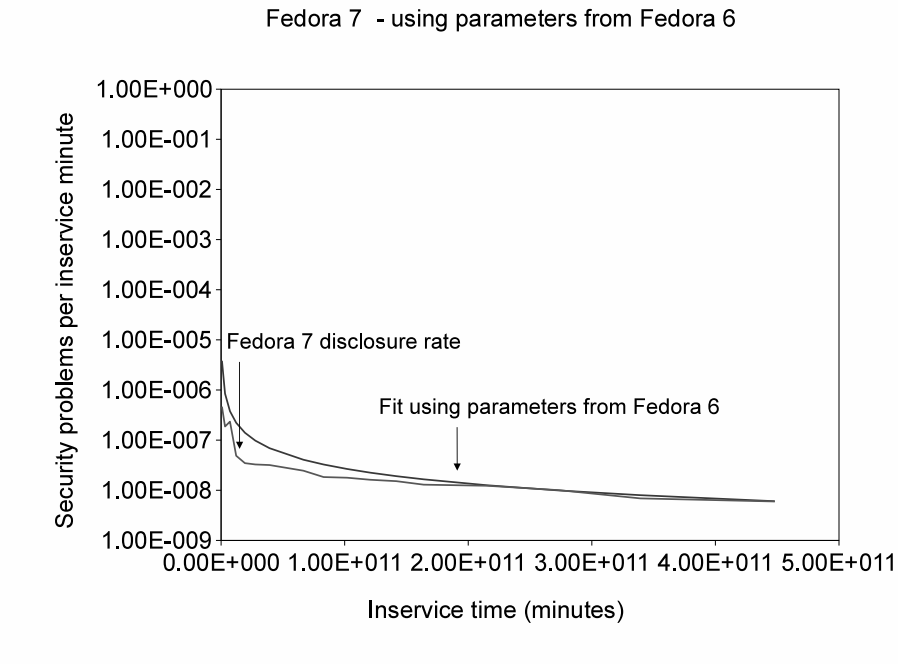

Figure 4.4: Fedora-7 fit using parameters from Fedora-6

4.2 Bayesian Model

Software security has both an objective and a subjective component. Therefore a comprehensive model of software security qualities of a product needs to incorporate both objective measures, such as security problem disclosure, repair and, failure rates, as well as less objective metrics such as implied variability in the operational profile, influence of attacks, and subjective impressions of exposure and severity of the problems, etc. In this section, we show how Bayesian approaches used in classical software reliability models can be adapted for use in the security context. First, we observe the behavior of the software system in constant execution time intervals, an approach similar to classical software reliability models. We assumed that the number of downloads is proportional to usage of the product in the field and that all systems were operational during the entire calendar period. This allowed us to estimate an upper bound on the execution time as the product of the total number of systems and the calendar time. In assuming that all systems are operational during the entire calendar period, we do not account for the variability in the operational use of each system. To address this, we also extend the model by incorporating randomness in the usage of each system to account for the variability in the operational use brought by the intervention of attackers.

4.2.1 Modeling Disclosure related Beliefs

In real life operational profile of a software product can include not only legitimate use of that software, but also attacks by hackers. The latter may intensify (and change the operational profile) after a vulnerability is disclosed. When a disclosure is accompanied by a fix or a work-around, end-user perceptions of the level of security of the product may improve, i.e., security reliability of the product may appear to increase. When disclosures are not directly accompanied by instant repair of the problem, end-user perceptions may be less favorable despite the fact that in actuality the probability that the product will fail due to a security breach may not have changed in a particular environment. Therefore we distinguish two groups of security disclosure

1. those that are accompanied by fixes, distribution of weekly fixes by a vendor, or those where the end-user, due to the nature of the vulnerability, can implement an instant fix, problem avoidance or work-around solution.

2. those that require an external fix, and have the fix follow a measurable time (μ) later, or possibly never.

Next, we discuss the subjective impacts of the security disclosures, and adapt a classical software reliability engineering Bayesian model to this context.

Sources of uncertainty

Software tends to come with faults. As it ages some of those faults are detected and removed, but other faults could be introduces during that process as well. We consider security vulnerabilities as a subset of the general pool of faults. The difference, however, is that security faults imply malicious intent and exploitation that may deliberately amplify a relatively innocuous deficiency in software which, on its own, may not cause a software failure. If we accept that a certain fraction of the operational profile of network-based software are attacks or attempts to exploit its vulnerabilities, then a software product (running on many installations and with many users) will experience disclosure of its vulnerabilities in a relatively random fashion [1]. Of course, attacks may intensify after a problem disclosure, and that may change both the actual probability that a machine will suffer a security failure and the perceived probability. On the other hand, in the case of closed source projects, disclosures of security problems may occur at regular intervals - often after fixes for the problem are available [2].

While software may originally come with faults, once they are disclosed perceptions of the user can change. How that perception, belief in the security quality of the software, changes may depend on several factors. First of all, disclosure of unique faults does not really say anything about the failures (or exploits) due to those vulnerabilities, nor how often those occur. Therefore, just disclosing a problem without some additional action may only worsen the perception an end-user may have of that software. If disclosure is accompanied by a fix, then in many situations perceptions of the users will become more positive (i.e., software security reliability perception may grow).

There is also the issue of trust. Open source projects, particularly when the community is large, come with a certain belief that disclosures are made in earnest and as soon as issues are discovered. If, in addition a disclosed vulnerability is of the voluntary class, or somehow else can be mitigated (e.g., blocking of certain ports to external access), or an actual fix is available, then this is equivalent to failure reporting with instant fault repair in traditional software reliability engineering. In general the trust, or belief, in the software from the end-user security perspective can increase, remain the same, or decrease with every vulnerability disclosure.

One can discuss the probability from a frequentist or from a Bayesian perspectives [38, 36]. For example, given some assumptions, the rate of disclosing a unique security fault in a time period can be estimated as the ratio between the number of disclosures and the time over which those disclosures have been made This can then be translated into probability that a disclosure (or a security failure) will be observed over a certain time period in the future, etc.

However, typically software projects of interest have a large number of users with perhaps

[1] http://magazine.redhat.com/category/red-hat-enterprise-linux/
[2] http://www.microsoft.com/security/msrc/report/disclosure.aspx

widely varying operational profiles. In that case it may not be justifiable to just use frequencies and point estimates to assess system behaviors, especially if the nature of the faults and failures is such that there is a considerable variability in the interpretation of their impact, or even whether an impact can occur or not. Security failures and faults fall into that domain. We believe that in this case analysis of the disclosures of the problems, through conditional probability distributions, i.e., a Bayesian view [39]., may provide a better envelope for estimating their possible impact. Bayesian models have been used with success in the past to describe and predict behavior of general class of software failures ([36, 39, 38]), but their application to software security has been very limited.

One thing to remember is that while security faults are in many ways similar to non-security faults (i.e., they are due to an omission in the specifications, or are a physical defect in the code, a user error, etc.) they also exhibit many behavioral differences. For example, they may not manifest in physical failure of the system but only in more ephemeral (but still unintended) behaviors such as system slow down, unauthorized access to the system and its content, they may not manifest unless there is a correlated series of inputs, including perhaps end-user cooperation (voluntary failures), etc.. Thus focusing on disclosures may be more telling than focusing on the actual security breaches (or actual security failure intensity) so long as one remembers that not every disclosed (potential) vulnerability is actually exploitable. Furthermore, security failures often require operational conditions and inputs which are considerably different from the operational profile envisioned for the software under considerations. This, and the unknown factor of attacker behaviors (timing, interest, location, etc.) can drastically alter operational profile of a system or a group of systems.

Data

In this paper, we used data for Firefox 3.0, Firefox 3.5 and SeaMonkey 1.1. The data, including information on the number of downloads, number of security faults, report time, and repair time, were collected using [11] from Mozilla[3] pubic repositories. We considered only security faults disclosed after the release of each product. We assumed that the number of downloads is proportional to usage of the product in the field. This allowed us to estimate an upper bound on the inservice time as the product of the number of systems using the product and the calendar time since product release.

Model

We use a variant of Littlewoods approach to Bayesian modeling [36, 39, 38], except we apply the reasoning to analysis of software security vulnerability disclosures.

[3] http://www.mozilla.org/security/known-vulnerabilities/

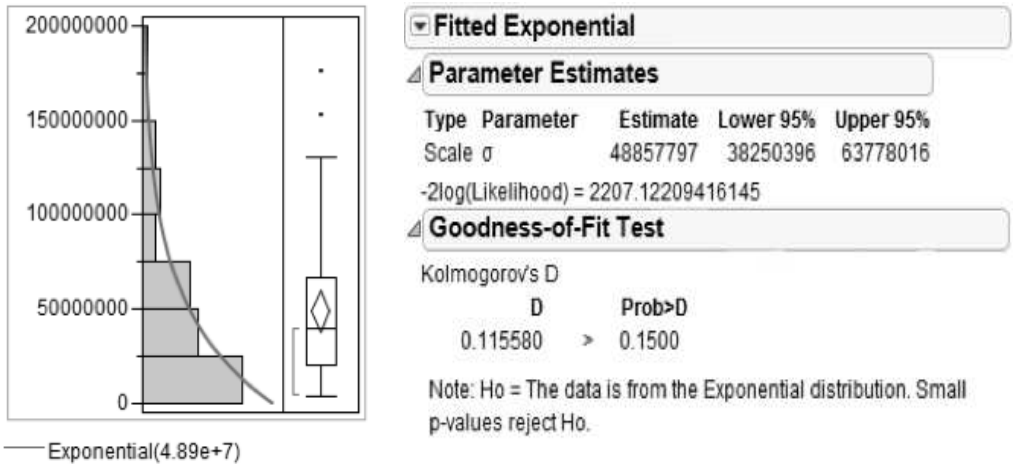

Figure 4.5: Test for exponential distribution of time between disclosures

A obvious source of uncertainty is random disclosure of unique security faults during the operational use of the software. Typically such random events are assumed to follow a Poisson process [50]. But security researchers [66, 55] have questioned the validity of making such assumptions. This has prompted us to check on this assumption.

To test for a Poisson process, it is necessary to establish whether inter arrival times of unique security fault disclosures follow an exponential distribution, and whether the number of unique security faults disclosed in a fixed time periods follow a Poisson distribution [63].

We tested for the exponential distribution of inter arrival times and poisson distribution of number of unique security faults disclosed using JMP[4]. Figures 5.14 and 4.6 show the fit of exponential and poisson distributions to the data on the inter arrival times of unique security faults and data on number of unique security faults disclosed during a fixed execution time interval. From the figure, we observer that there is not enough evidence to conclude that the inter-arrival times of unique security faults do not follow an exponential distribution and the number of unique security faults disclosed during a particular execution time interval do not follow a poisson distribution.

Similar to Littlewood's model [36], we now focus on the time-to-unique-security-fault disclosure distribution, and time-to-repair of such faults. Hence the data for this model is a sequence of execution time intervals, i.e., the time interval between successive disclosures of unique security faults measured in inservice times, and time to repair those faults. Since we have established that the inter arrival time of unique security fault reports is exponentially

[4]http://www.jmp.com/

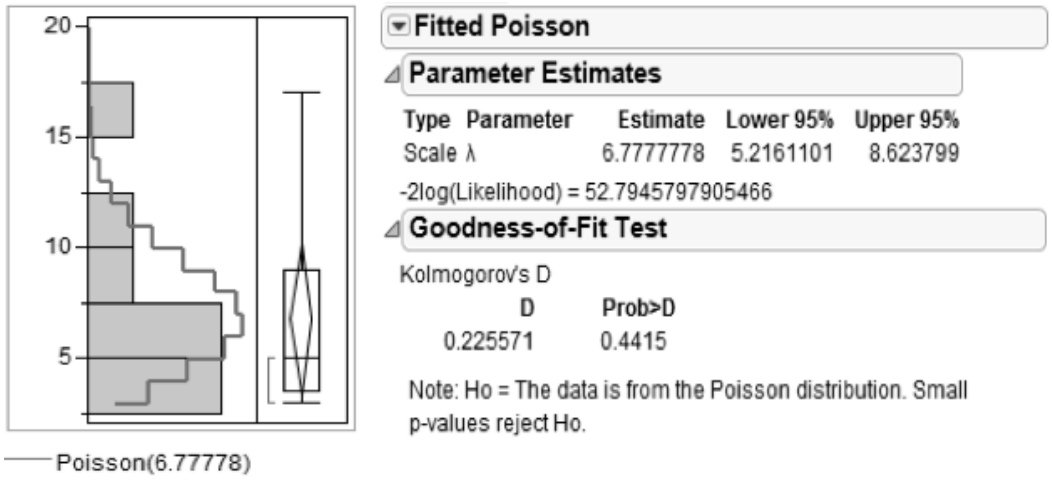

Figure 4.6: Test for poisson distribution of number of unique disclosures

distributed, then assuming we know the disclosure rate of unique security faults, its probability distribution function is given by

$$pdf(t_i|\lambda_i) = \lambda_i e^{-\lambda_i t_i} \tag{4.1}$$

where t_i is the execution time between the disclosure of unique security faults and λ_i is the disclosure rate of unique security faults at disclosure event i.

Unfortunately, disclosure rate is not a known quantity to start with. Furthermore there is a considerable amount of uncertainty attached to this value due, among other things, to possible attacker induced changes in the operational profile, and other perhaps perception-based decisions. Hence we treat the disclosure rate as a random variable.

Figure 4.7 shows a very simplified disclosure model adapted from the more comprehensive model discussed in [9]. λ denotes problem disclosure rate (e.g., problems per unit time), and μ problem repair rate (problems repaired per unit time. Software is in the Good/Hidden state during normal operation (hidden refers to latent or hidden faults that have not yet been disclosed). When a security fault is disclosed it transitions into Disclosed state and stays there until the fault is repaired (fixed), after which it transitions back to the Good/Hidden state.

Faults that are disclosed but are not repaired ($\mu = 0$, or $\mu >> \lambda$) are likely to increase concerns about the system. A relatively constant fault discovery rate (λ) over longer periods of time may do the same since it may imply either a software that had new faults injected, or a software that has a very large pool of problems, or perhaps a software where some of the faults are kept secret. Usually in open source projects security faults are disclosed immediately

upon discovery and then they are fixed. If repair time is short (e.g., $\mu > \lambda$) faults will not accumulate, and that is good. Some disclosures, such as those in the voluntary category allow the end-user to mitigate their possible impact immediately making μ effectively infinite or at least much larger than λ. But security faults may also be kept secret upon discovery, then fixed and disclosed along with the fix. In that case public μ is technically infinite or at least much larger than λ. This is typically the case in closed source projects. This may increase the confidence of a user if one believes that the fix is a good one. On the other hand, there real value of μ usually is not know in those cases, and so it is not clear how long end-users have been exposed to the danger. If at the same time λ has been relatively constant over extended time periods the confidence may decrease because one may become apprehensive about what other security problems may have been concealed from a user and currently leave the user exposed to attacks.

Open source proponents favor disclosure of security faults since they believe in transparency, and that public disclosures allow issues to be addressed quickly, allow users to take precautions against any exploits [9], and in general guard against security through obscurity approach. If the rate of public disclosures of security faults is a decreasing function of time, one also needs to make certain that the associated repair time is sufficiently short that the actual pool of security faults in a software is a decreasing function of time.

Lets consider the case where on the average $\lambda_i < \lambda_{i-1}$ for event i. We would also need to insure that the average repair rate is large enough that there is no accumulation of faults. If there is uncertainty about the values of λ, we at least would like to ensure that it is probable that the public disclosure of a security faults, and its repair action rate, increases the confidence of a users. Then in both cases, from the model perspective μ=∞, or at least $\mu >> \lambda$. If the mean-time-to-repair is zero or a user is not certain in the fix that has been provided, then the governing uncertainty is that in λ. If a user is certain that the fault has indeed been fixed, then for λ, this can be stated as

$$P(\lambda_i < l) \geq P(\lambda_{i-1} < l), for\ all\ l, i$$

Let the distribution function of λ_i be denoted by G(l,i,α), where α is a parameter or vector of parameters. Then,

$$G(l, i-1, \alpha) \leq G(l, i, \alpha)$$

This requires us to choose a parametric family that satisfies the ordering of the distribution functions. Similar to Littlewood et al. [37, 36], we use Gamma distribution to represent the disclosure rate. Note that the scaling function (β) in the Gamma distribution need to be a function appropriate to satisfy the above ordering of the distribution functions. For example, it

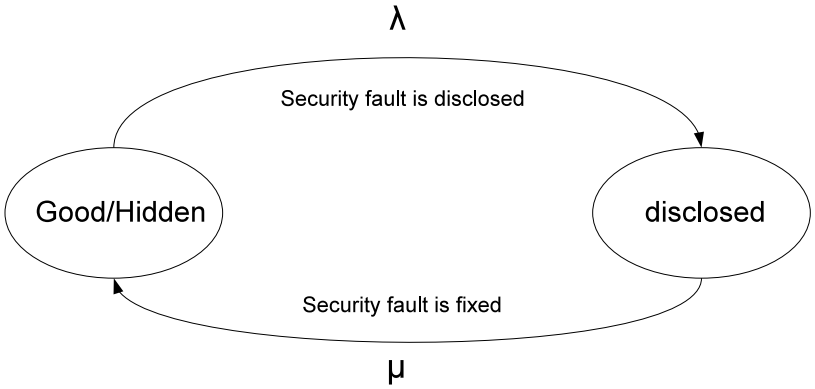

Figure 4.7: Disclosure model

could be monotonically increasing to satisfy the increasing order of the distribution functions. Ideally, the scaling function should be a function that takes in to account the repair action of developers (for example, the repair rate), the number of users operational, attackers' behavior on the security problem disclosed including the variability in the operational use by the users, etc. This can be reasoned as follows. Assume the developer does not fix the fault immediately but starts working on the fix at his own perusal. While the fault is still being fixed the user has a workaround to avoid any exploit. One may argue that the fault is not yet fixed and may decrease the confidence of the user. But from the perspective of a user whose belief in the open source community is sufficiently greater to overcome any pessimism at the disclosure, the confidence increases considering that the user has a workaround to avoid exploit and that the developer will eventually fix the fault. This is also evident from our data since all faults were eventually fixed.

We make the following assumptions

1. The operational use of the software and the intervention of attackers together form the operational environment.

2. The disclosure of unique security faults is random and follows a Poisson process.

3. The disclosure rate of unique security faults at any time is a random variable and is assumed to follow a Gamma distribution.

4. The security faults are fixed immediately upon disclosure and no new security faults are introduced by the repair action.

In the following, for simplicity, neither λ, nor the Gamma shape (α) or scaling (β) parameters are shown as functions of time and event counts. This may not actually reflect the reality.

Security faults often are not fixed immediately. Therefore we would need to account for the actual repair time in the fault correction process. Furthermore, operational profile may change and λ, α and β and their distributions are very likely functions of time, and certainly of perceptions. This is addressed in Section 4.3

Consider that we observe the software when a total of τ execution minutes have passed and n unique security faults have been disclosed. We are interested in the time to next disclosure of a unique security fault. The conditional distribution is given by,

$$
\begin{aligned}
&pdf(\lambda|\ n\ security\ faults\ disclosed\ in\ (0,\tau)) \\
&= \frac{(Pr(n\ faults\ disclosed\ in\ (0,\tau))|\lambda).pdf(\lambda)}{\int (Pr(n\ faults\ disclosed\ in\ (0,\tau))|\lambda).pdf(\lambda).d\lambda} \\
&= \frac{\frac{\lambda\tau}{n!}.e^{-\lambda\tau}.pdf(\lambda)}{\int \frac{\lambda\tau}{n!}.e^{-\lambda\tau}.pdf(\lambda).d\lambda}
\end{aligned}
\tag{4.2}
$$

Since we have assumed that the disclosure rate follows a Gamma distribution, we have

$$
pdf(\lambda) = \frac{\beta^{\alpha}.\lambda^{\alpha-1}.e^{-\beta\lambda}}{\Gamma(\alpha)} \tag{4.3}
$$

Using (3) in (2) we get,

$$
\begin{aligned}
&pdf(\lambda|\ n\ security\ faults\ disclosed\ in\ (0,\tau)) \\
&= \frac{\frac{\lambda\tau}{n!}.e^{-\lambda\tau}.\frac{\beta^{\alpha}.\lambda^{\alpha-1}.e^{-\beta\lambda}}{\Gamma(\alpha)}}{\int \frac{\lambda\tau}{n!}.e^{-\lambda\tau}.\frac{\beta^{\alpha}.\lambda^{\alpha-1}.e^{-\beta\lambda}}{\Gamma(\alpha)}.d\lambda} \\
&= \frac{(\beta+\tau)^{(\alpha+n)}.\lambda^{\alpha+n-1}.e^{-\lambda(\beta+\tau)}}{\Gamma(\alpha+n)}
\end{aligned}
\tag{4.4}
$$

which is gamma distributed with parameters ($\alpha+n$,$\beta+\tau$). We had already mentioned that,

$$
pdf(t|\Lambda=\lambda) = \lambda e^{-\lambda.t} \tag{4.5}
$$

From (5), the unconditional time-to-next-security-fault disclosure distribution is,

$$\begin{aligned} pdf(t) &= \int_0^\infty pdf(t|\Lambda=\lambda).pdf(\lambda|\ n\ security\ faults \\ &\qquad disclosed\ in\ (0,\tau)).d\lambda \\ &= \frac{(\beta+\tau)^{(\alpha+n)}.(\alpha+n)}{(\beta+\tau+t)^{(\alpha+n+1)}} \end{aligned} \tag{4.6}$$

which is a Pareto distribution. Next, we are interested in computing the security fault disclosure rate. The failure rate of a program is computed from the reliability of the program, i.e.,

$$\begin{aligned} R(t) &= P(T>t) \\ &= 1-P(T<=t) \\ &= 1-cdf(t) \end{aligned} \tag{4.7}$$

$$\lambda(t) = \frac{-R'(t)}{R(t)} \tag{4.8}$$

We use a similar approach in computing the security fault disclosure rate. In our case, we adapt the reliability of the program to represent the probability that there is no disclosure of unique security faults up to time t. Note that t is measured from the τ point on, so these equations actually reflect piecewise nature of the problem. Although, this may not be the actual reliability of a program, we use it to compute the disclosure rate of unique security faults. Since the time to next disclosure follows a Pareto distribution, we get,

$$R(t) = \frac{(\beta+\tau)^{(\alpha+n)}}{(\beta+\tau+t)^{(\alpha+n)}} \tag{4.9}$$

$$\lambda(t) = \frac{\alpha+n}{(\beta+\tau+t)} \tag{4.10}$$

Similar to Littlewood's approach [37, 36], we have obtained the unconditional unique security fault disclosure rate for an open source software for which a total of "n" unique security faults have been disclosed in a total execution time of τ.

Table 4.1: Security faults statistics

Project	Firefox 3.0	Firefox 3.5	SeaMonkey 1.1
Number of users at the end of the period	450,459,683	311,992,340	2,309,549
Period of interest (weeks)	54	50	50
Number of security faults reported	125	57	69
Average number of disclosures per inservice week	5.53E-09	1.23E-08	8.69E-07
Standard deviation	1.24E-08	2.62E-08	1.39E-06
Average number of fixes per inservice week	4.63E-09	6.91E-07	2.91E-06
Standard deviation	1.41E-08	1.80E-08	8.05E-06

4.2.2 Evaluation

We use security data from Firefox releases 3.0 and 3.5, and from SeaMonkey release 1.1 to discuss the Bayesian model in the context of experimental information.

We continue to use inservice time as the primary exposure metric. Figures 4.8, 4.9, and 4.10 show the number of security problems disclosed per unit inservice time for Firefox releases 3.0, 3.5 and SeaMonkey release 1.1 respectively. The horizontal axis shows the inservice time in inservice weeks - an estimate of the number of operational systems multiplied by the number of weeks in operation. For each week we have taken the data on how many software downloads were recorded. We made the assumption that all downloaded systems were installed and activated, and then used that information and the information from previous weeks to compute an upper bound on the cumulative number of systems in service that week. The vertical axis shows the number of security problems disclosed per unit inservice week. From the figures, we observe that the number of security problems disclosed over the inservice time show a decreasing trend.

Table 4.1 summarizes some of the more interesting statistics about the three releases. These statistics are over the whole observed period noted in the table. The total number of security problems is relatively small. This is particularly notable since there were millions of downloads of these systems during the observed period. We see that for most part repair processes are keeping up with problem reporting (e.g., Figures 4.14, 4.15, 4.16), but that in some cases issues took a very long time to fix (from the data on average disclosures and repair and their standard deviation).

So how does this get captured by a model. The current Bayesian model assumes instantaneous fault repair. In about 30% of the cases (voluntary problems) that is feasible [9], but that is not the case in general. However, problem repair rates are at least keeping up with issues, although with some delay (Table 4.1). Figures 4.11 , 4.12 , and 4.13 show how model fits

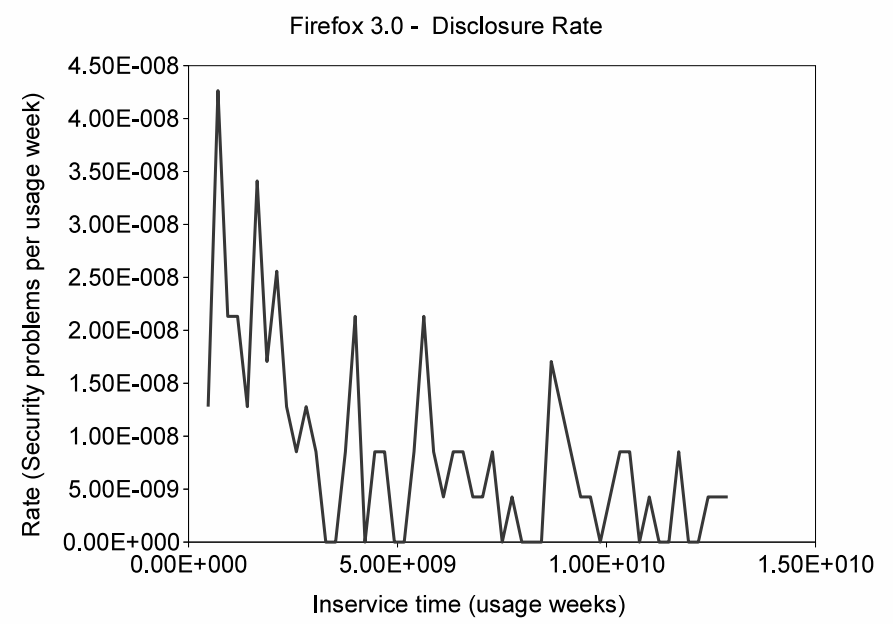

Figure 4.8: Firefox-3.0 disclosure rate

observed data and how it might predict what would happen in the future. Table 4.2 shows the parameter values. From Figures 4.8, 4.9 and 4.10 and the parameter values, we can observe the parameter estimates correspond to the actual values. The fits shown in the figures are for a specific variant of the model where scaling factor is constant. Variants where it is a linear function of disclosure events or a quadratic functions have been considered in the past for non-security software faults ([21, 50]). Similar to classical reliability models, we measured the predictive accuracy using the measure of median relative error. On the average, we observed a median relative error of about 34% when the predictions have been tried out after observing the data for about 20% of the total execution time. Next, we extend the model to include more subjective views on the variability in the operational profile.

So all this shows that the model (in its present form) is at least consistent with the observed data. However, the reality is that a classical reliability model such as Musas could also fit the disclosure data quite well [8]. The power of the Bayesian model will start showing when we add to the mix the problem repair time delays, and the subjective views that modify the Gamma distribution shape and scaling parameters. As already noted, the current model probably applies to voluntary class of security faults It could also be applied to closed source software for which commercial vendors distribute a fix along with the disclosure. In the cases considered here, we know that all the reported problems were eventually fixed. All this indicates that there

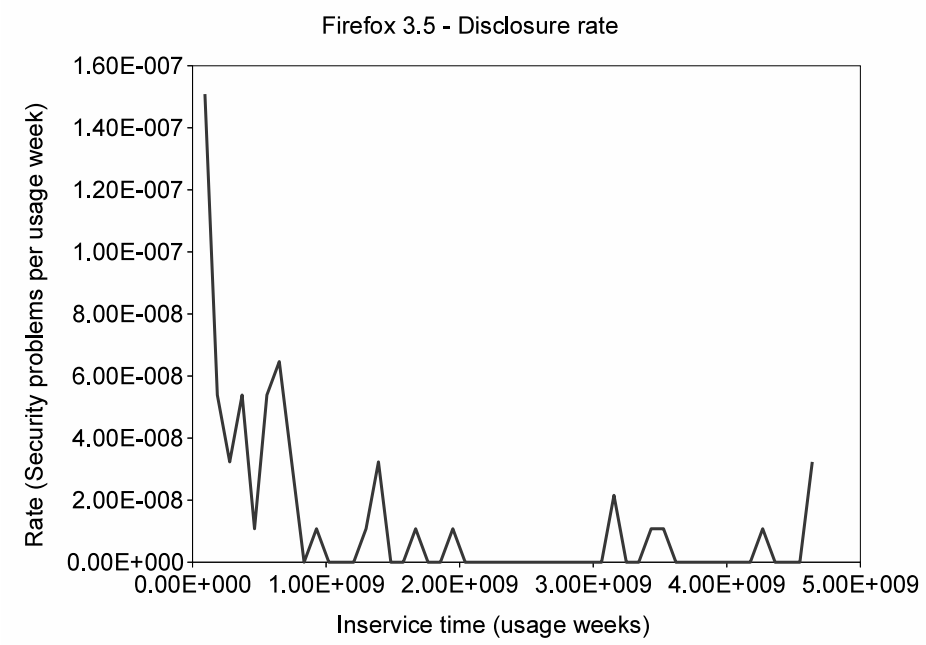

Figure 4.9: Firefox-3.5 disclosure rate

Table 4.2: Disclosure rate prediction results

No	Product	Training set	Test set	alpha (α)	beta (β)	Median Relative Error
1	Firefox 3.0	20%	80%	2	3.9E08	27.94%
2	Firefox 3.5	20%	80%	2	5.7E07	39.16%
3	SeaMonkey 1.1	20%	80%	2	2.8E05	34.29%

is a genuine reduction in the security issues. This should increase the trust in this particular open source software.

We plan to extend the model to incorporate more subjective views. For example, delay in problem repair should result in a partial loss of trust, an increase in the perceived probability that the system might fail due to a security breach. That probability also will depend on the probability that the system would be attacked (or have its operational profile changed externally). Similarly, a consistently level λ might, after a certain period of time erode confidence in the product despite disclosures that come with fixes. One would really want to see genuinely reducing λ. Of course, as was shown in [9] λ could reduce simply due to advent of a newer version of the product and shifting of end-user interests to that version.

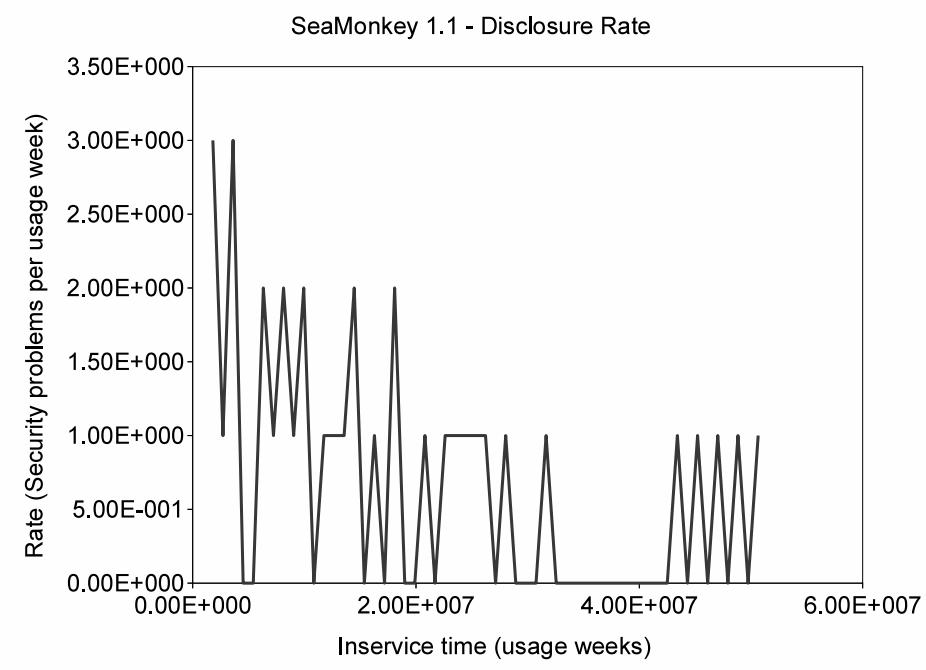

Figure 4.10: SeaMonkey-1.1 disclosure rate

4.3 Variability in the Operational Use

In this section, we extend the disclosure model to incorporate the variability in the operational use of a software product. We observe the behavior of a product from the view point of growth in the number of operational systems. The usage time for each of the system is treated as a random variable with a prior distribution associated to it. The model captures the number of unique security problems disclosed in the form of unique security faults, exploits, and security failures during the operational use of a software product. We choose a poisson process to describe the number of security problems disclosed with respect to the operational use of a software system. Figure 4.17 shows the model developed using openBUGS[5], a software package widely used for Bayesian analysis.

In Figure 4.17, "m" represents the number of intervals in which we observe the number of unique security problems disclosed. Each interval "m" consists of a constant number of the systems operational. This is similar in approach to classical software reliability models where the behavior of the system is observed in constant execution time intervals. In each of the "m" intervals, the usage of the number of systems operational is considered a random variable associated with a prior distribution. Ideally, the prior distribution should contain usage

[5]http://www.openbugs.info/w/

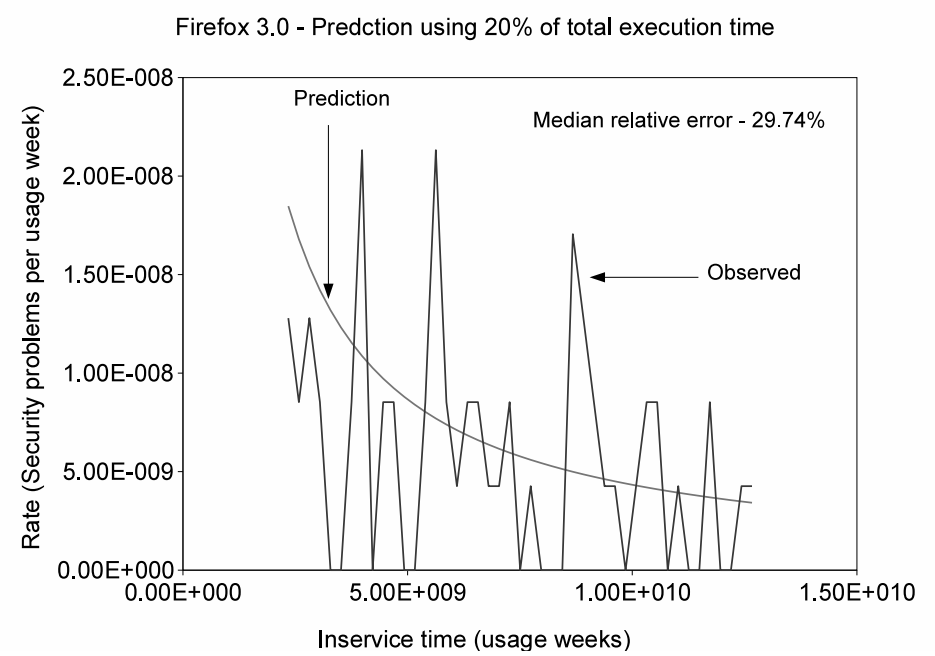

Figure 4.11: Firefox-3.0 prediction using 20% of the total execution time

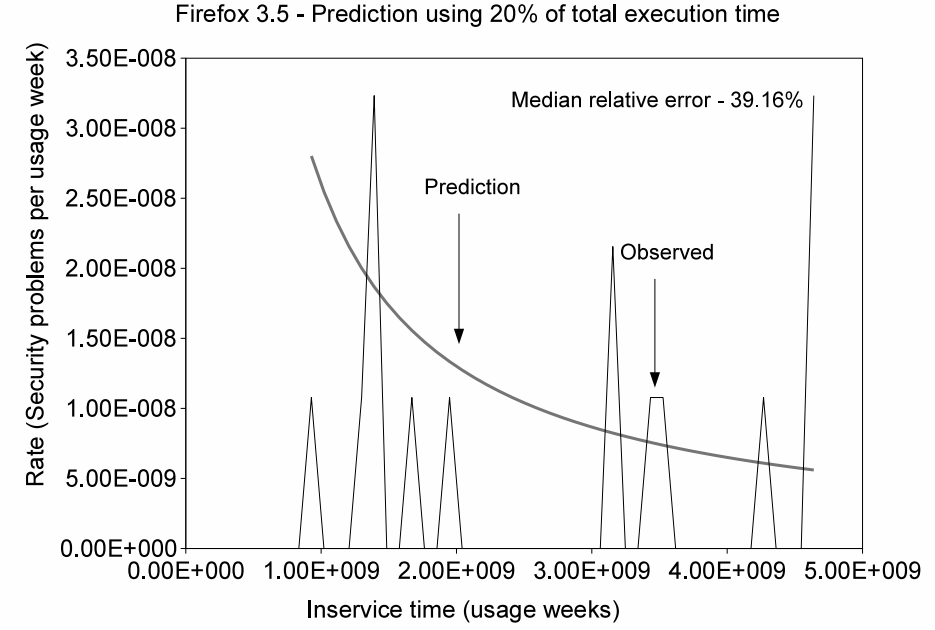

Figure 4.12: Firefox-3.5 prediction using 20% of the total execution time

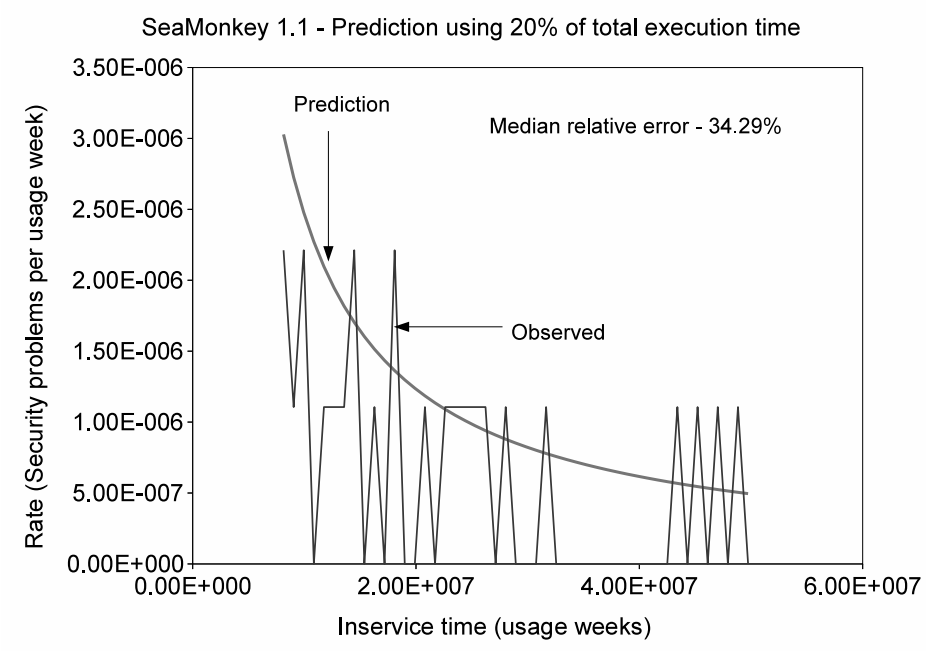

Figure 4.13: SeaMonkey-1.1 prediction using 20% of the total execution time

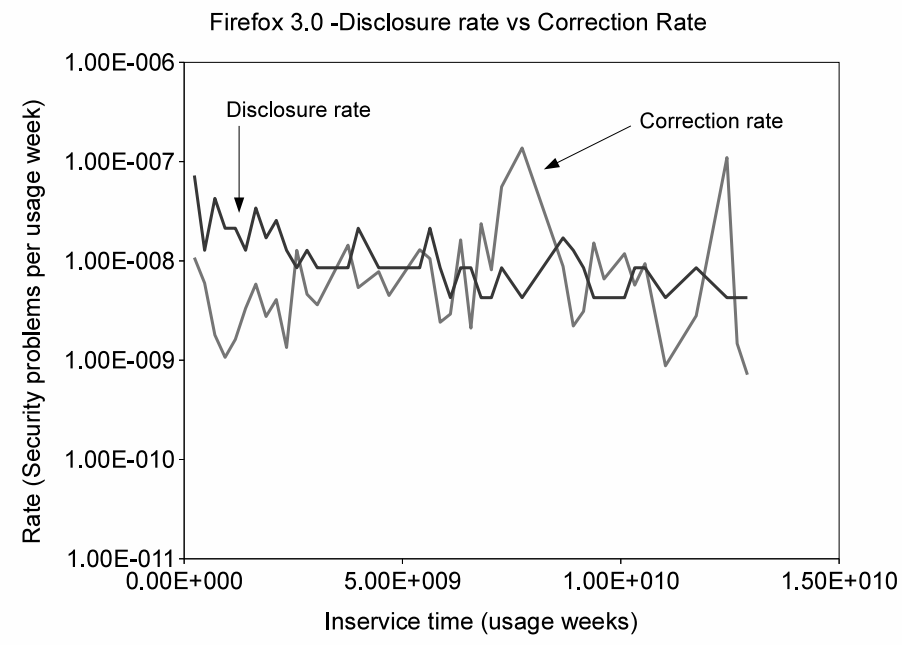

Figure 4.14: Firefox-3.0 disclosure Rate vs correction rate

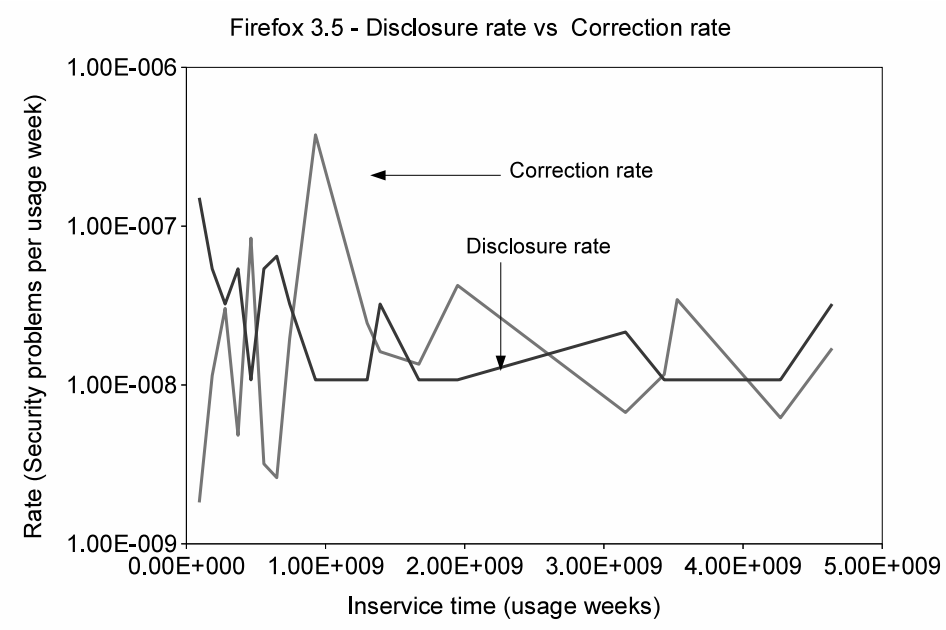

Figure 4.15: Firefox-3.5 disclosure rate vs correction rate

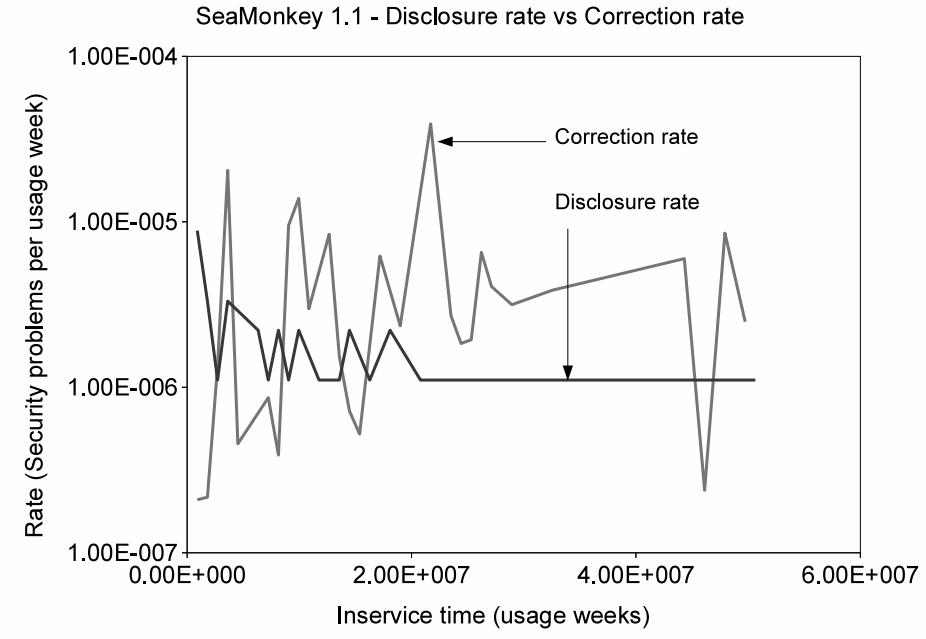

Figure 4.16: SeaMonkey-1.1 disclosure rate vs correction rate

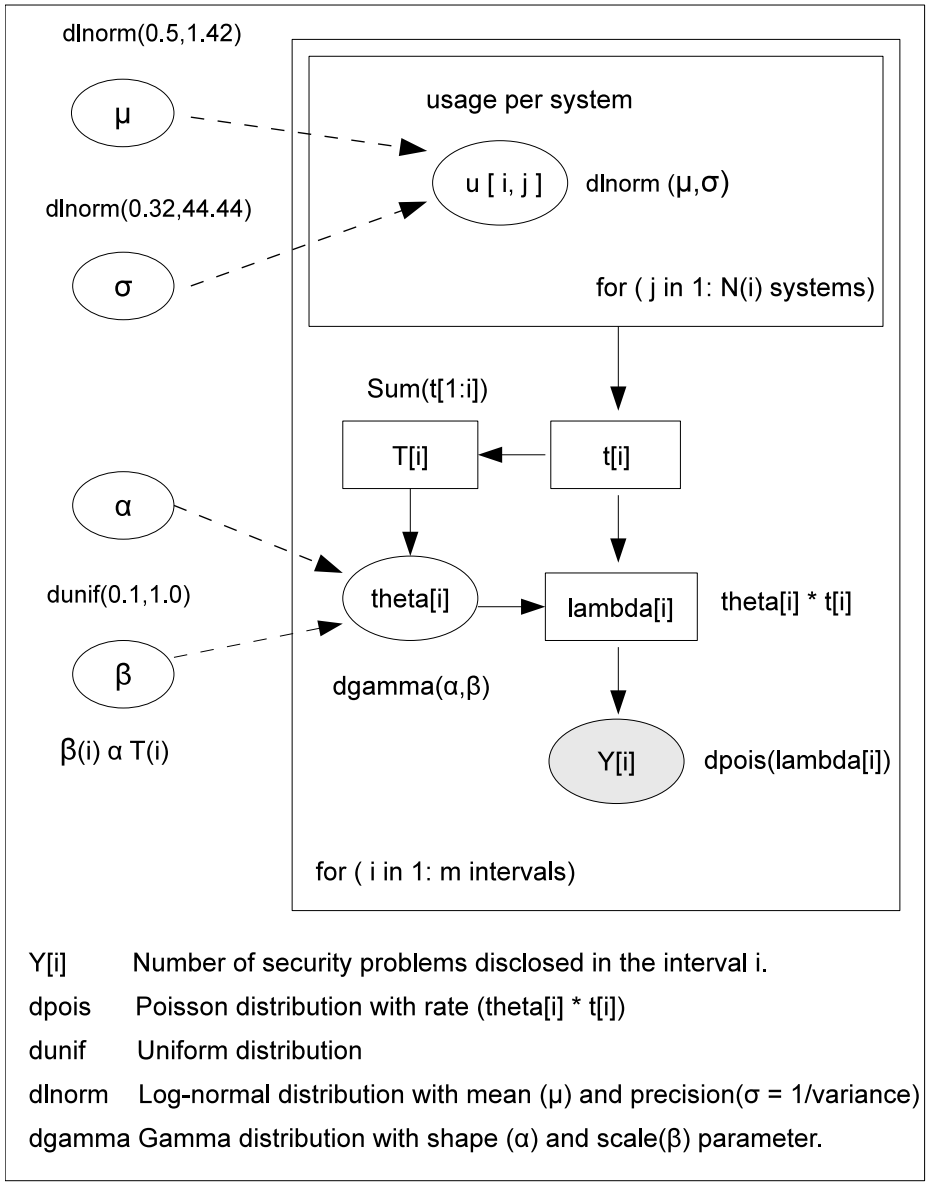

Figure 4.17: Bayesian model with variability in operational use

information not only about the legitimate use of a software product, but also the interruption in the normal usage due of security problems. We observed various studies[6] [7] [7, 57] on the usage behavior of software systems for various categories of users. Although it unlikely that these studies may include security related information, we assume that information from these studies would be appropriate to represent the variability in the operational use of a software. Since we observed several studies on the same topic of usage behavior, we allow the parameters of the prior distribution to vary with their own prior distributions. The parameters of the hyperpriors have been determined based on the above studies. In Figure 4.17 , μ (mean) and σ (precision, which is the inverse of variance) represent the prior parameters. Each of parameters are associated with hyperpriors, i.e., μ is given a prior distribution that is log-normally distributed with mean 0.5 and precision 1.42, and σ is given a prior distribution that is log-normally distributed with mean 0.32 and precision 44.44. As discussed earlier, the number of security problems disclosed is assumed to follow a poisson distribution.

$$Y[i] = Poisson(theta[i] * t[i]) \tag{4.11}$$

where theta[i] represents the disclosure rate and t[i] represents the total usage of N[i] users in the interval [i]. A conjugate gamma prior distribution is assumed for the disclosure rates. It is necessary to reflect the subjective view in the disclosure of security problems using the shape or the scale parameter in gamma prior for the disclosure rate. Similar to the model in 4.2.1, we assume that the scale parameter is function that it is directly proportional to the growth in the operational usage of the software system. We assumed an informative prior for α parameter, a uniform distribution between 1 and 2, based on the Firefox 3.0, 3.5 and SeaMonkey 1.1 results from Table 4.2. We used OpenBUGS[8], a widely used Bayesian analysis tool, to simulate the posterior densities, perform sensitivity analysis on the choice of the prior distributions as well as predict future observations.

4.3.1 Sensitivity Analysis

To determine the sensitivity of the choice of prior distributions, we examined the behavior of the estimate of theta and Y by varying the values on their prior distributions. Since we chose informative priors in our analysis, we use the same distributions for the priors, but with different values. Tables 4.3 and 4.4 show the different estimates of theta and Y for different prior parameter values. The results show that the estimates of Y and theta, including the mean and the 95% Bayesian confidence interval, vary only slightly. Based on the results, we conclude that Y and theta are insensitive to the choice of the prior distribution.

[6] http://blog.mozilla.com/metrics/2010/08/26/who-are-our-firefox-4-beta-users/
[7] http://www.webuse.umd.edu/handouts/tutorial/SDA-GSS-Tutorial.pdf
[8] http://www.openbugs.info/w/

Table 4.3: Sensitivity analysis for parameter alpha on Y

No	Prior	Y(Mean)	95% Prediction Interval
1	dunif(1.1, 1.9)	2.14	{0, 9}
2	dunif(1.2, 1.8)	2.34	{0, 10}
3	dunif(1.3, 1.7)	2.63	{0, 11}
4	dunif(1.4, 1.6)	2.47	{0, 10}
5	dunif(1.5, 2.0)	2.31	{0, 9}

Table 4.4: Sensitivity analysis for parameter mu and sigma on theta

No	Prior (mu)	Prior (sigma)	theta(Mean)	95% Prediction Interval
1	dlnorm(0.5,1.42)	dlnorm(0.32,44.44)	4.175	{2.21, 6.76}
2	dlnorm(0.1,1.21)	dlnorm(0.12,42.44)	4.161	{2.21, 6.74}
3	dlnorm(0.2,1.31)	dlnorm(0.22,43.44)	4.166	{2.22, 6.78}
4	dlnorm(0.3,1.51)	dlnorm(0.42,45.44)	4.179	{2.17, 6.75}
5	dlnorm(0.4,1.61)	dlnorm(0.52,46.44)	4.174	{2.19, 6.79}

4.3.2 Prediction

We evaluated the model on security problems from Firefox 3.6. Figure 4.18 shows the number of security problems disclosed with respect to the operational use of the software. Figure 4.18 shows how model predicts future observations. We measured the predictive accuracy using the measure of median relative error. We observed a median relative error of 24% in predicting the number of security problems disclosed. Based on these results, we conclude that Bayesian security problem disclosure model can be used with success in predicting the disclosure events of security problems.

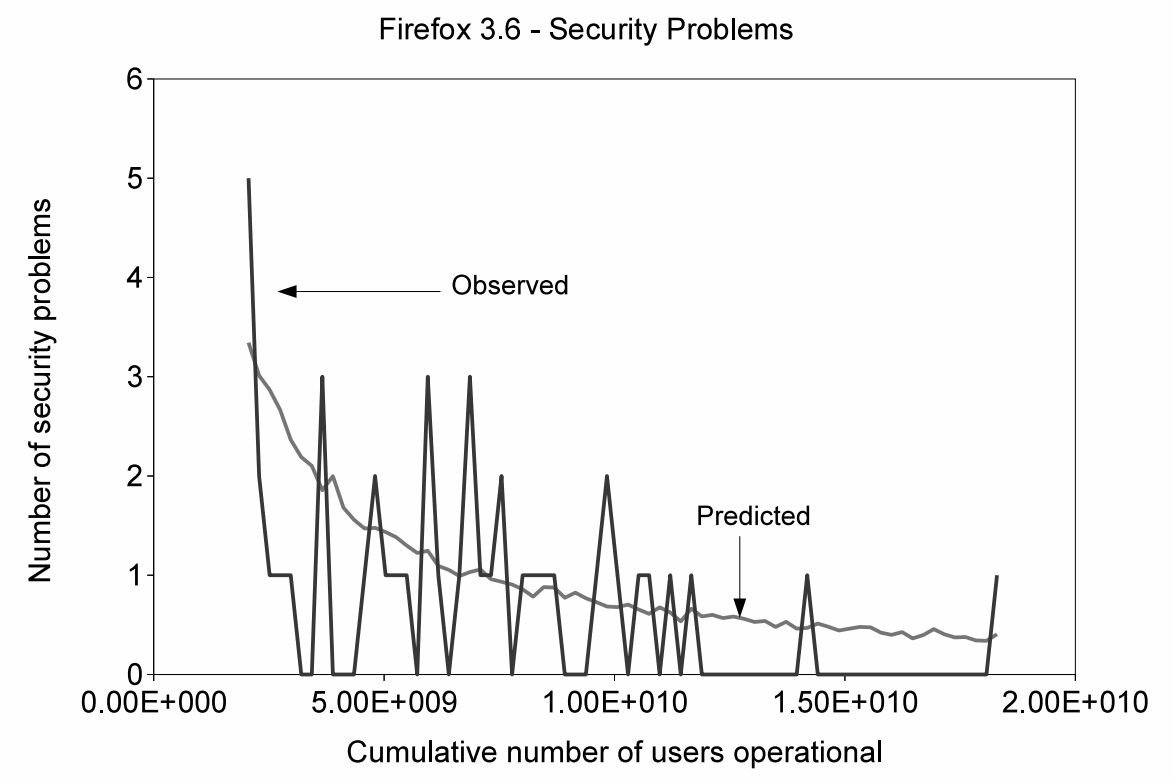

Figure 4.18: Firefox-3.6 security problems

Chapter 5

Analysis of Disclosure and Patching Policies

5.1 Introduction

While the security problem response model is comprehensive, it is difficult to solve analytically. We use simulation in combination with measurements for a variety of software products, to understand the interaction of events in the overall process model and guide users, and developers in making process related decisions.

5.2 Time-line View of the Security Model

Figure 5.1 shows the timeline view of the security model along with the overall process model. In Figure 5.1, time-to-discover is the time between the main release of a software in which the problem is present and when the security problem is discovered. Typically "full disclosure" products tend to have shorter releases compared to "limited disclosure" products. So we chose to consider the discovery time period from the release of the main version to prevent any bias in the measurements. For example, if a security problem is discovered in Firefox 3.6.11 ("full disclosure") and is also present in the earlier releases 3.5 and 3.6, then we consider the time-to-discovery as the time from release of 3.6 until the time it was discovered in 3.6.11. Time-to-disclose is the time between the discovery of a security problem and when it is publicly disclosed. "Full disclosure" products tend to disclose problems first and then fix it, and "limited disclosure" products disclose problems along with the fix. Typically open source products fall under the latter category. Data on the time taken to discover and disclose software security problems were collected from a public list of security problems maintained by IDefense[1], a security intelligence

[1] http://labs.idefense.com/about/

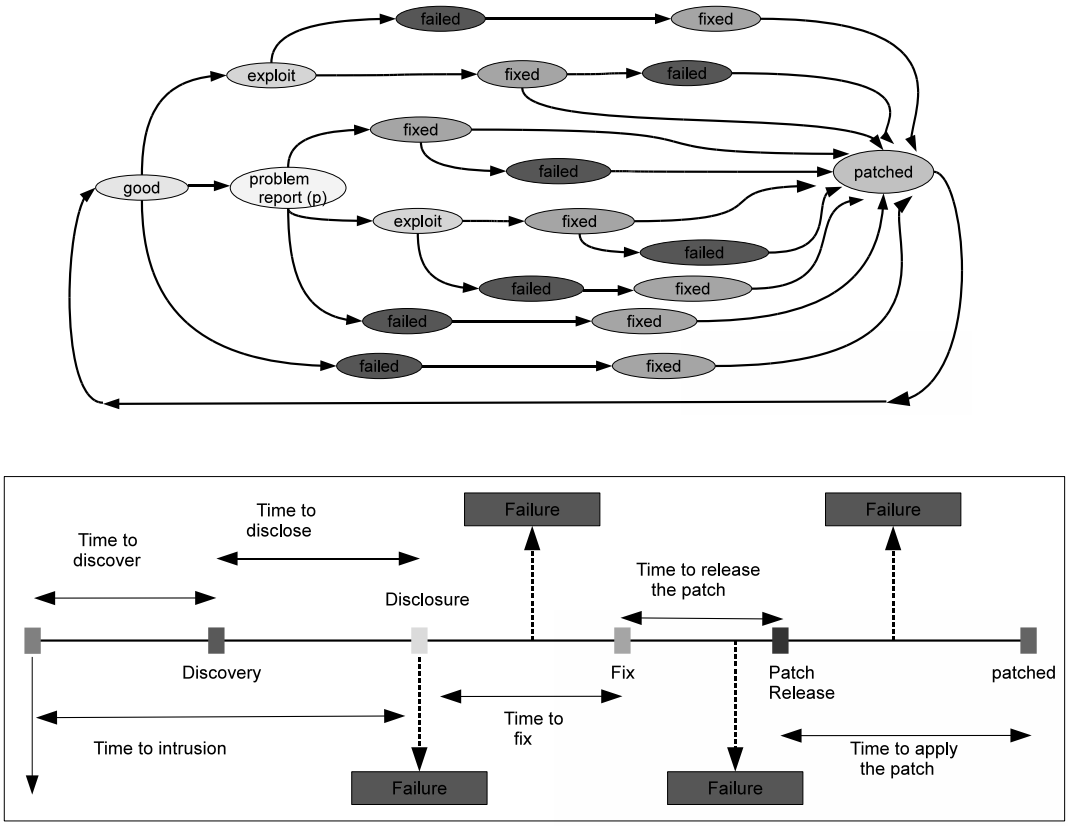

Figure 5.1: Timeline view of the security model

firm aimed at providing software security services to governments and private organizations. The data collected from IDefense's repository included security problems discovered during the years 2007 to 2011. Figures 5.2 and 5.3 show the time between discovery of security problems for "full disclosure" and "limited disclosure" products.

Time-to-fix security problems is the time by developers to fix the problem. In "limited disclosure" products, where security problems are disclosed along with the fix (including release of a patch), the time-to-fix security problems is included within the time-to-disclose the problems. When security problems are disclosed as a result of field failures, software products tend to have a different correction profile than for unfailed security problems. In "full disclosure" products, the time taken to fix a failed security problem (including release of the patch) is within a week. while in "limited disclosure" products, the time taken to fix failed security problems range from a week to a month. In full-disclosure products, the time taken to fix failed

security problems ranges from 1 to 7 days. In limited-disclosure products, the time taken to fix failed security problems ranged from 7 to 40 days. Figure 5.10 shows the time-to-fix security problems for failed and unfailed security problems. Time-to-release patches is the time taken in "full disclosure" products to release a patch for an unfailed problem. Figure 5.11 shows the time-to-release security patches in "full disclosure" products.

Time-to-intrusion is the time taken for a malicious user to effect a security breach in the field. From the information available in the NVD and individual project repositories, we measured time-to-intrusion as the time from the point of the main release of the software version until when a security breach happened. Since we are interested in the time-line from when discovery of a problem happens, we assumed that attackers have a similar time-to-discovery as developers or experts in security firms. This assumption is reasonable as attackers are not different from experts in security firms, since attackers are also security experts, but only with a malicious intent. Under this assumption, we computed the time-to-intrusion as the time between the discovery of a security problem and when a security failure begins to happen in the field. Figures 5.6, 5.7, 5.8 and 5.9 show time-to-intrusion of security problems.

Information on failure of a security problem in the field and the time taken to fix such failed security problems were collected from the information available in the NVD and individual project repositories. Time-to-fix for unfailed security problems, i.e., those problems that did not experience a failure in the field, were measured based on data presented in Chapter 2. To collect data on the user patching behavior, we used the results from [24] which discusses the user patching process for the browsers Firefox, Chrome, Opera, Safari and Internet Explorer. The patching process described for Firefox and Chrome in [24] is "automatic updates" where there is no user intervention in installation of security patches. This captures the trend for patch processes where no user intervention is present. Similar is the case for "non-automatic updates", i.e., there is user intervention in installation of security patches, in Safari, Internet Explorer and Opera.

Figure 5.12 shows the active number of users for Firefox and Internet Explorer. The active number of users for Firefox 3.6 was obtained from Mozilla developers [2]. Using the values obtained for Firefox and the public report on browser statistics[3], we computed the active number of users for Internet Explorer. While the user profile may not be representative of all "full disclosure" and "limited disclosure" products, we believe it is reasonable to assume such a growth in the field usage for any software product. The Figures referenced above also show the distributions and functions fitted to the data. The fitted parameter values are shown in Table 5.1. The distributions fitted to the data and the parameter values are consistent with what other researchers have reported [26]. In our current analyses, we do not explicitly analyse

[2] http://www.mozilla.org/

[3] $http://www.w3schools.com/browsers/browsers_stats.asp$

the impact of a public exploit code on field failures. We plan to address this limitation in our future work.

In our simulations, we captured the events of discovery and disclosure of security problems by developers and attackers, correction of security problems and release of patches by developers, and installation of patches by users. We also capture the propagation of security failures in the event of failure of a security problem in the field. Next, we discuss the simulations and results in detail. In the simulations, we generate samples from the probability distributions of various parameters like the time-between-discovery, time-to-fix, time-to-release-patch, etc. The overall process begins with the discovery of security problems. Based on the data from Chapter 2, we make the following assumptions. One percent of the discovered security problems are disclosed as a result of field failures and 99% of the discovered security problems are disclosed in the form of problem reports or public exploits. One percent of the security problems disclosed through problem reports or public exploits result in field failures after disclosure. For failures that happen after disclosure, the point of failure could be during the time a problem is fixed by the developer or after the release of patches, depending the sample generated for time-to-fix and time-to-intrusion. In the event a failure happens during the fix operation, then expedited fixing is assumed by generating samples from the probability distributions of time-to-fix-failed-problems. Until the release of patches, we generate samples from the probability distributions and then use the power function to simulate the patching process.

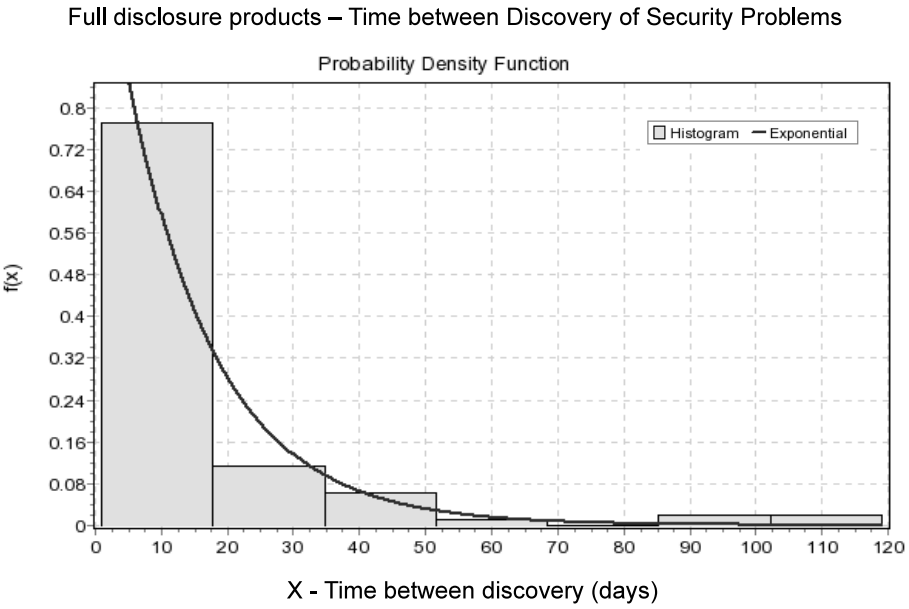

Figure 5.2: Full disclosure products - Time between discovery of security problems

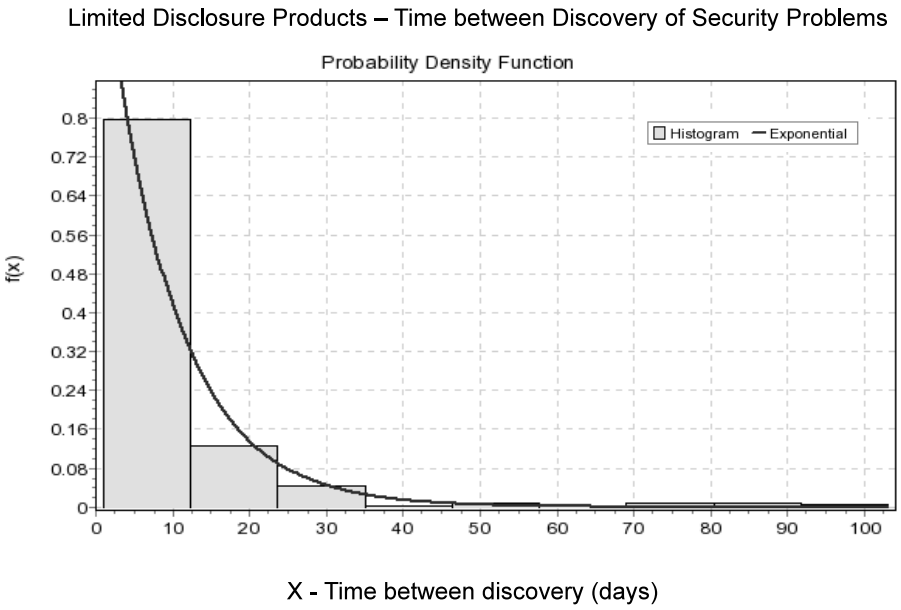

Figure 5.3: Limited disclosure products - Time between discovery of security problems

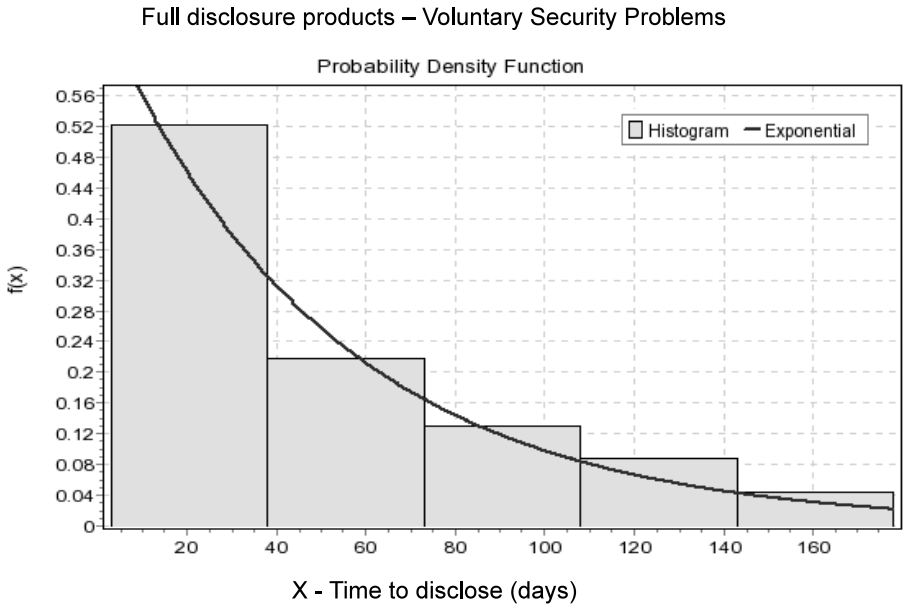

Figure 5.4: Full disclosure products - Disclosure of voluntary security problems

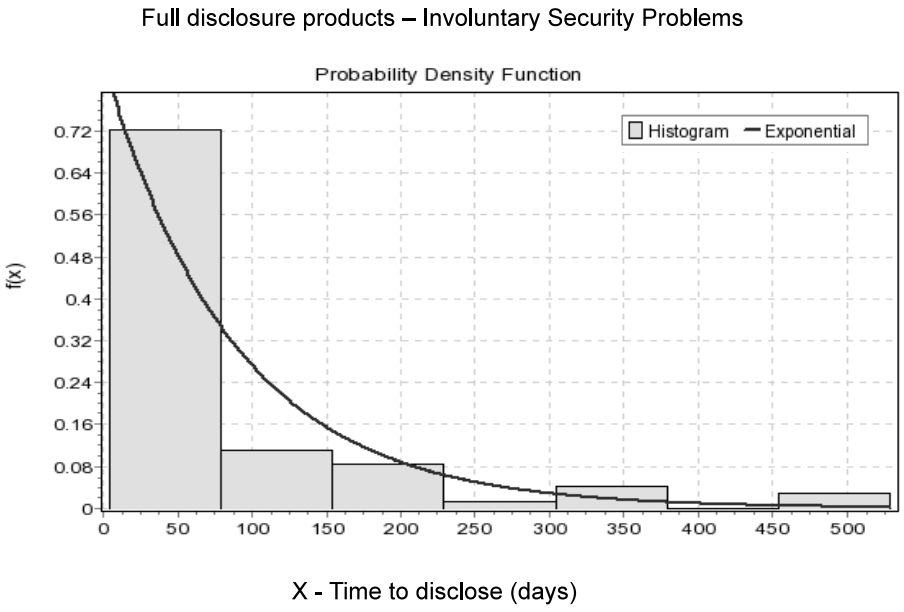

Figure 5.5: Full disclosure products - Disclosure of involuntary security problems

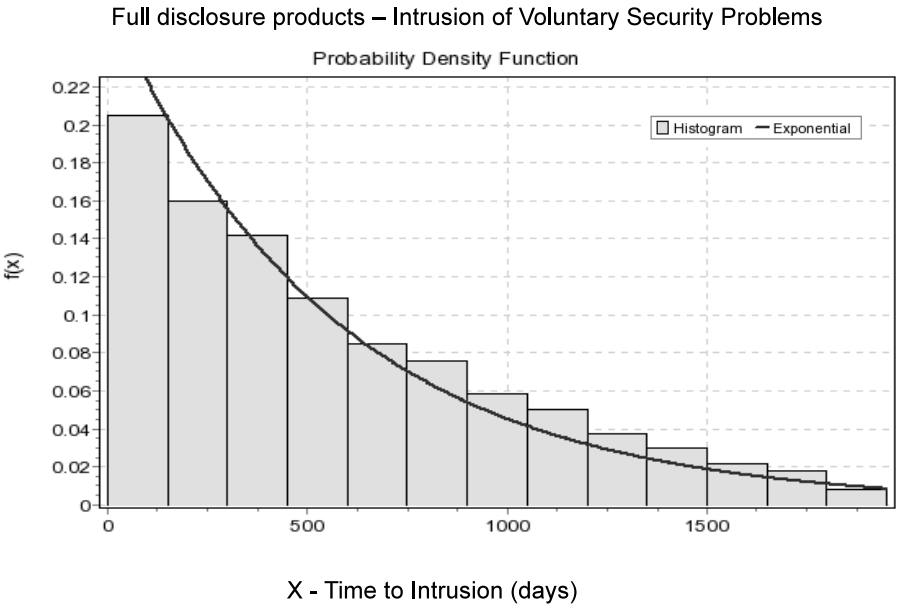

Figure 5.6: Full disclosure products - Intrusion of voluntary security problems

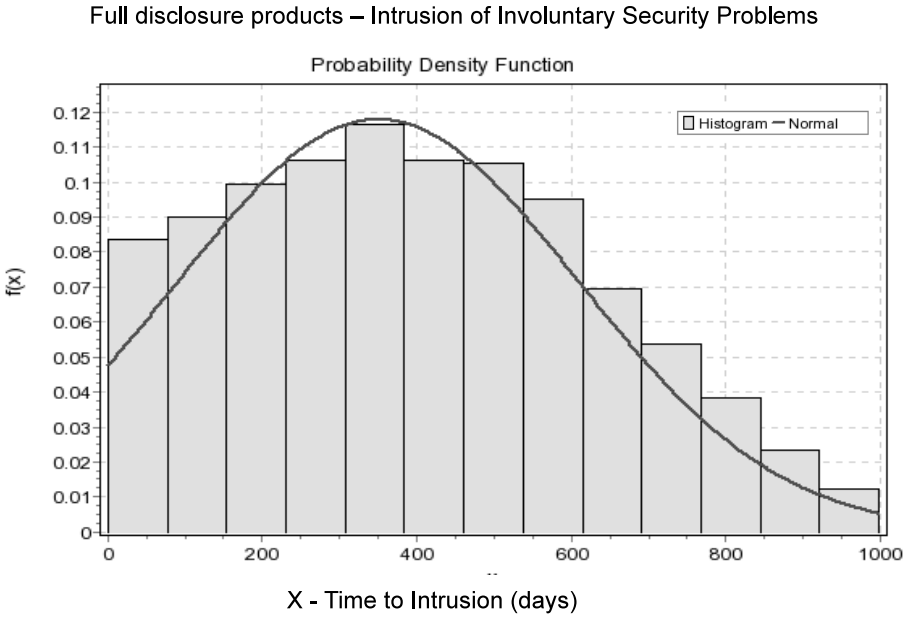

Figure 5.7: Full disclosure products - Intrusion of involuntary security problems

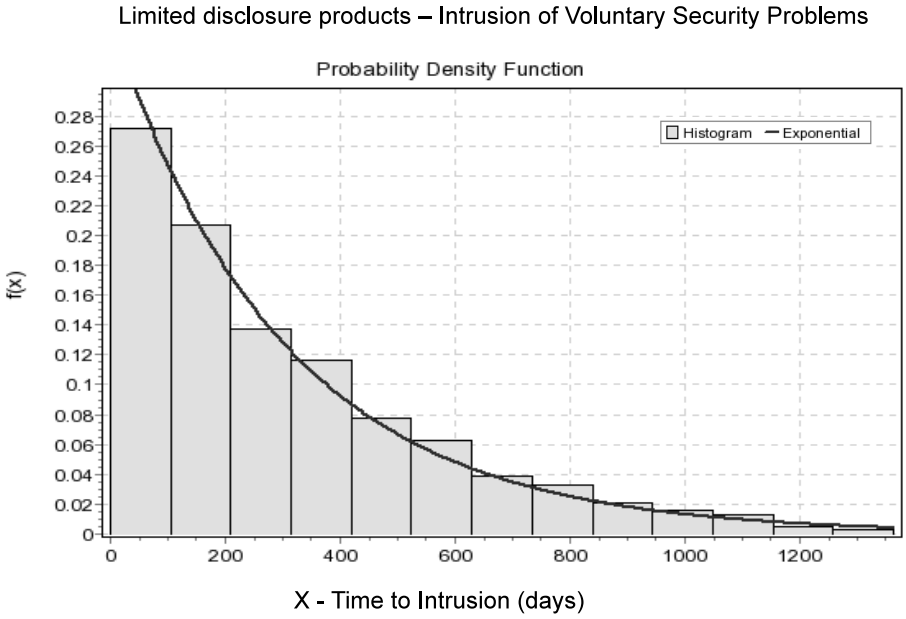

Figure 5.8: Limited disclosure products - Intrusion of voluntary security problems

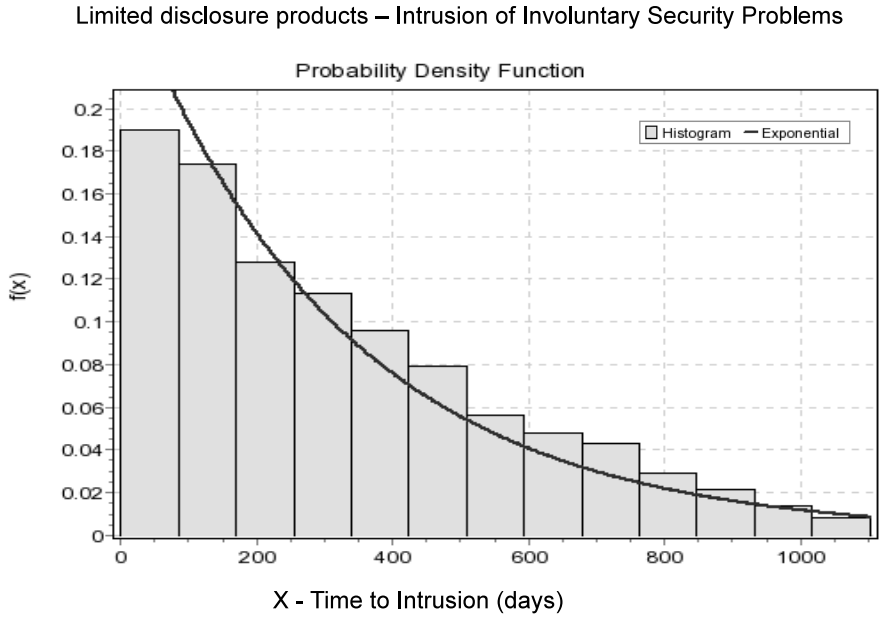

Figure 5.9: Limited disclosure products - Intrusion of involuntary security problems

Table 5.1: Timeline events parameters

No	Security Problem Events	Distribution	Parameters
1	Time between discovery (full disclosure products)	exponential	$\lambda = 0.0739$
2	Time between discovery (limited disclosure products)	exponential	$\lambda = 0.1117$
3	Time-to-disclosure (full disclosure products - voluntary)	exponential	$\lambda = 0.0238$
4	Time-to-disclosure (full disclosure products - involuntary)	exponential	$\lambda = 0.0227$
5	Time-to-disclosure (limited disclosure products - voluntary)	exponential	$\lambda = 0.0086$
6	Time-to-disclosure (limited disclosure products - involuntary)	exponential	$\lambda = 0.0050$
7	Time-to-fix (full disclosure products - unfailed)	exponential	$\lambda = 0.0510$
8	Time-to-release patches (full disclosure products)	normal	$\mu = 25, \sigma = 25$
9	Time-to-fix (full disclosure products-failed)	uniform	$a = 1, b = 7$
10	Time-to-fix (limited disclosure products-failed)	uniform	$a = 7, b = 40$
11	Time-to-intrusion (full disclosure products - voluntary)	exponential	$\lambda = 0.0028$
12	Time-to-intrusion (full disclosure products - involuntary)	normal	$\mu = 350, \sigma = 260$
13	Time-to-intrusion (limited disclosure products - voluntary)	exponential	$\lambda = 0.00329$
14	Time-to-intrusion (limited disclosure products - involuntary)	exponential	$\lambda = 0.00313$

Table 5.2: User patch trend

No	Type of project	Fit	Parameters
1	full disclosure products	Power function	$a = 60.17, b = 0.30$
2	limited disclosure products	Power function	$a = 39.34, b = 0.45$

Table 5.3: Active number of users daily

No	Type of project	Fit	Parameters
1	full disclosure products	Power function	$a = 6.209, b = 1.683$
2	limited disclosure products	Power function	$a = 11.44, b = 1.683$

5.3 Simulation Results and Discussion

We performed simulations to study the impact of disclosure, correction and patching policies on user exposure to security problems and failures. We assume that a developer would be interested in the impact on the software product as a whole and computed the total number of systems exposed to security problems as well as the total number of systems experiencing security failures. We assumed that a user would be interested in the impact in terms of the being attacked and computed the probability that a user is a victim during security failures. We discuss how various disclosure, correction and user patching policies impact the measures computed.

5.3.1 Developer Response to Security Problems

Guido et al. [67] hypothesize that open source and closed source security problem response process do not differ in their impact on the correction of security problem reports. The author compares the proportion of unresolved security problem reports for open source and closed source products, and conclude that there is no significant difference in developers' response to security problems in open source and closed source products. Analysis on the number of unresolved problems (12% to 26% [67]) is only a part of the security problem response process. It is still necessary to understand the impact on the significant proportion of security problems that eventually get fixed in a software, which is about 74% to 88% [67]. We assume that the total number of users exposed between the discovery, disclosure, and correction of security problems (that eventually get fixed in a software) would capture the impact on developers' response to security problems. We address the question "Does "full-disclosure' and " limited disclosure" software products differ in their impact in responding to software security problems?"

To answer the question, we assumed that both products have similar user population and measured the total number of users exposed between the discovery of a security problem and the

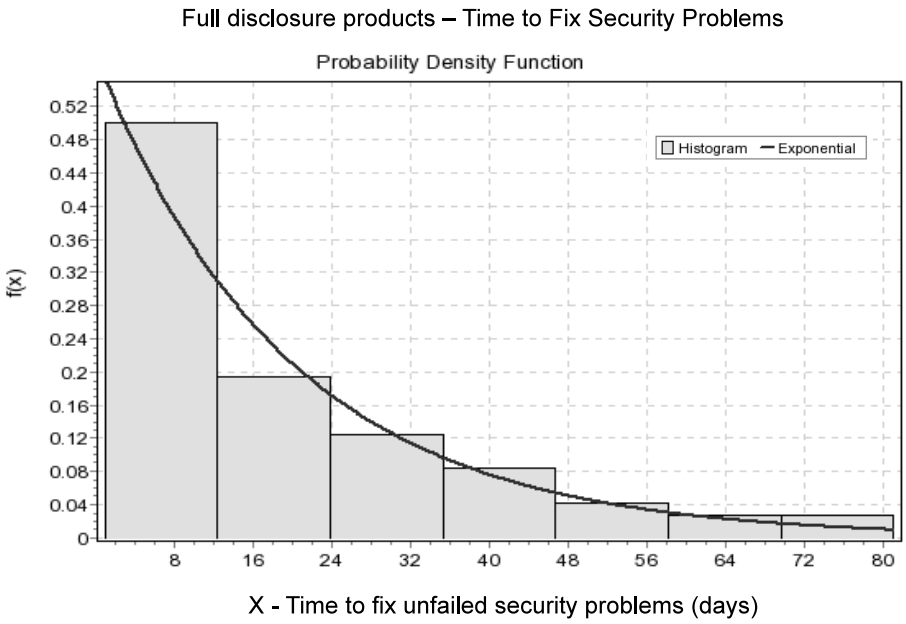

Figure 5.10: Full disclosure products - Time to fix unfailed security problems

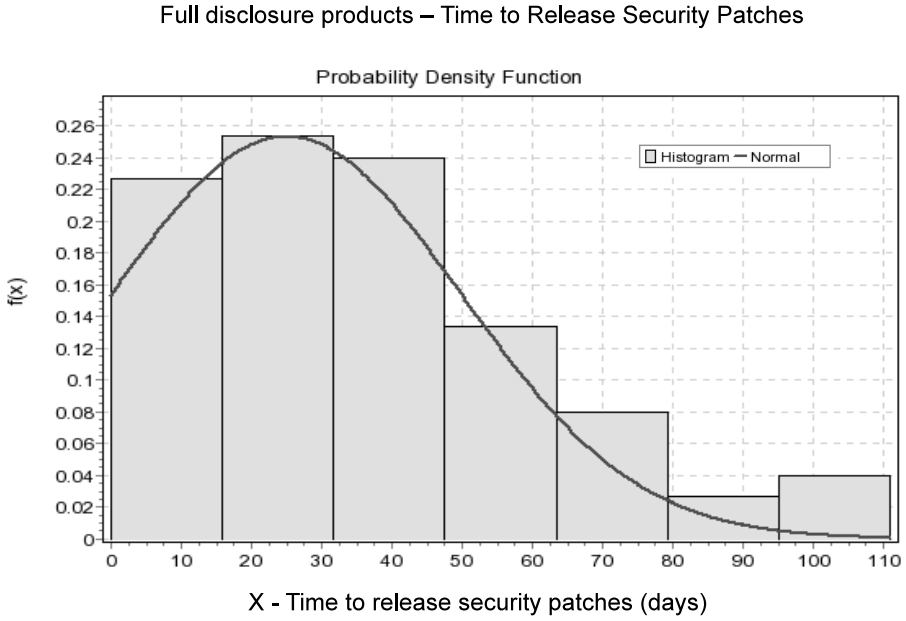

Figure 5.11: Full disclosure products - Time to release security patches

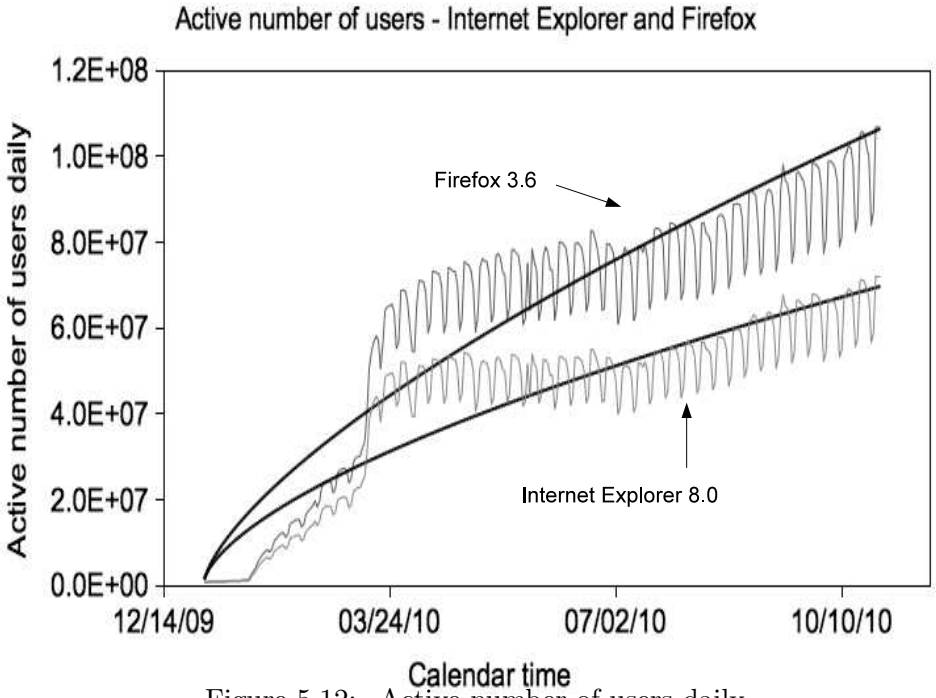

Figure 5.12: Active number of users daily

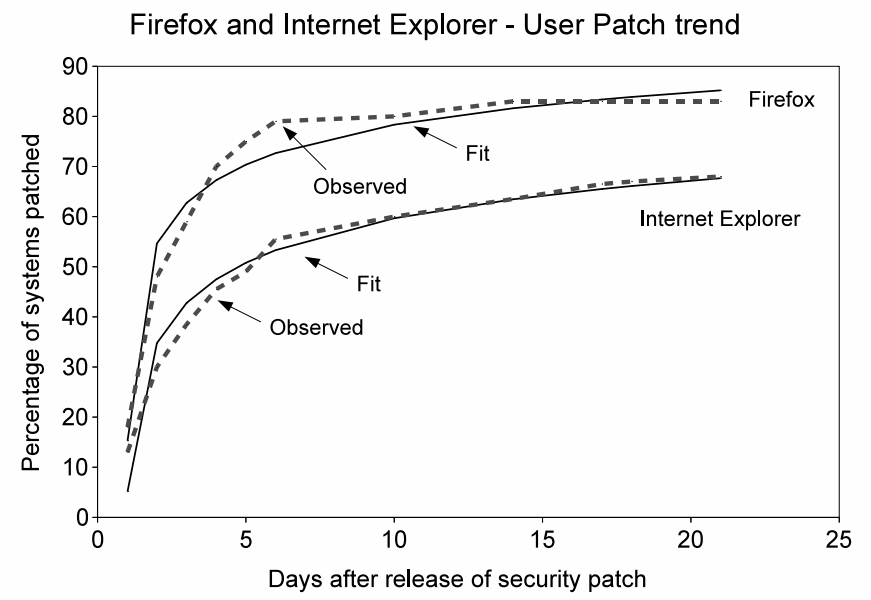

Figure 5.13: Patching behavior

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable Exposed
Classified by Variable Type

Type	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
limited disclosure	4160	14965624.5	14470560.0	82117.3881	3597.50589
full-disclosure	2796	9230821.5	9725886.0	82117.3881	3301.43830

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	9230821.5000
Normal Approximation	
Z	-6.0287
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001
t Approximation	
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001

Z includes a continuity correction of 0.5.

Figure 5.14: Mann-Whitney U test for developer response

patch release. We used Mann-Whitney U [41], a non-parameteric test to determine if there is a significant difference in the number of users exposed to a security problem for "full disclosure" software systems and "limited disclosure" software systems. Figure 5.14 shows the result from Mann-Whitney U test. From the result ($p < 0.0001$), we conclude that there is a significant difference in the number of users exposed to a security problem for "full disclosure" software systems and "limited disclosure" software products. The mean scores show that the number of users exposed for "full disclosure" software systems is less than that of "limited disclosure" software products. From this result, we conclude that "full disclosure" software products tend to respond faster to security problems compared to "limited disclosure" products.

5.3.2 Impact of Security Failures

Ransbotham [59] studied data on exploitation attempts from intrusion detection system logs and hypothesized that security problems in open source software are more risk prone than security problems in closed source software. Considering only exploitation attempts may not provide the complete picture. For example, they do not tell how many of the attempts resulted in failures, how long did the failures last, how many systems failed, what was the project's response to such failures, etc. It is necessary to include these factors in analysing the overall impact of a security problem. In this section, we focus on analysing the impact of correction

and patching policies in "full-disclosure" and "limited disclosure" web browsers on field failures. We address the question "Does "full-disclosure" and "limited disclosure" web browsers differ in their impact on the number of systems affected due to security failures?"

To answer the question, we measure the total number of systems affected during the security failures. From Table 5.1, we can observe that the time taken to fix failed security problems is greater for "full-disclosure" software products than "limited disclosure" software products. Based on this, one may conclude that the total number of users affected during the failure of a security problem would be greater for "limited disclosure" products. But, it is possible for "full-disclosure" browsers to have more systems operational during a particular time than "limited disclosure" browsers (e.g., Figure 5.12). Similar trends were observed for other "full disclosure" and "limited disclosure" browsers [24]. An interesting question is, "What is the overall impact when considering usage profile along with the correction and patching policies?. Next, we discuss four cases on how long attackers tend to exploit a security problem in the field and when does the peak number of infections occur, and for each of the cases discuss the impact of various correction and patching policies on field failures.

In order to simulate security failures, we use the random constant spread model [70] designed after the outbreak of the code-red virus [48, 75]. In this model, the number of users infected at time "t+dt" is given by,

$$n(t+dt) = (N * a(t)) * K(1 - a(t))dt \tag{5.1}$$

where K is the virus infection rate, i.e., the number of systems an infected system can compromise per unit time, N is the number of users vulnerable at time t, a(t) is the proportion of users infected at time t given by,

$$a(t) = \frac{e^{K(t-T)}}{1 + e^{K(t-T)}} \tag{5.2}$$

where T represents the time when the peak number of users are affected. Figure 5.15 shows sample number of users affected for different rates of infection and different times when the peak infections occur.

From existing work, we observed that the random constant spread model model has been used to model the propagation of various computer viruses and the values of K were found to be in the range 0.05 to 2 [68, 48, 75, 16]. In the simulations, we assume a similar range of values for K. For the values of T, it is necessary to make assumptions on how long would the security failures last in the field. One may expect security failures to last until all the users have applied the security patches or taken protective measures. But in practice, security failures may not last longer and may cease to infect users or reduce to an intensity where it no longer affects a significant proportion of users, because of logic errors in the virus source code, bandwidth limitations, etc [47, 48, 75]. In the simulations, we consider different cases on how long the

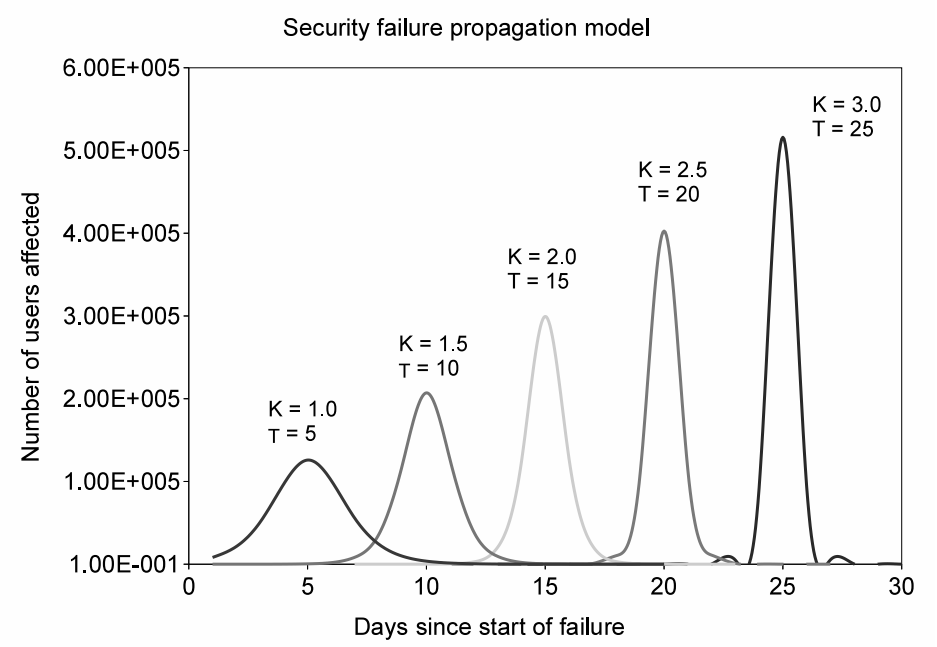

Figure 5.15: Security failure propagation

spread of failures may occur. We discuss the cases where security failures occur for short time periods, until when 80% of the users have applied security patches, until when all the users have applied security patches, etc. Next, we process related questions and decision making.

- **CASE1** We assume that security threats in the form of worms or viruses may be designed to spread mostly during the time when a significant proportion of the systems operational are vulnerable, i.e., last until 80% of the total number of systems are patched. We used Mann-Whitney U [41] test to determine if there is a significant difference in the number of systems affected due to security failures in "full-disclosure" and "limited disclosure" browsers. Based on [24], we use "automatic patching" for "full disclosure" browsers and "non-automatic" patching for "limited disclosure" browsers. Figure 5.16 shows the result from Mann-Whitney U test. From the result ($p < 0.0001$), we find that for case 1, there is a significant difference in the number of systems affected due to security failures. The mean scores show that the number of systems impacted for "limited disclosure" browsers is greater than that of "full-disclosure" browsers.

- **CASE2** Attackers may tend to exploit users even after 80% of the users have applied the patches, i.e., target even when only 20% of the total number of systems are vulnerable. In this case, we assume that security failures last until all the systems are patched. From Figure 5.13, we can see that the patching process can be modeled by a power function. In

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable Infected
Classified by Variable Type

Type	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
limited disclosure	8115	58280822.0	54954780.0	222970.301	7181.86346
full disclosure	5428	33432374.0	36758416.0	222970.301	6159.24355

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	33432374.0000
Normal Approximation	
Z	-14.9170
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001
t Approximation	
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001

Z includes a continuity correction of 0.5.

Figure 5.16: Mann-Whitney U test for security failures (case 1)

order to determine when all of the systems would have been patched, we used the power function to estimate when 100% of the systems are patched. We observed that after 180 days all systems with "full disclosure" browsers would have been patched and after 806 days all systems with "limited disclosure" browsers would have been patched. This may appear to be a long time period for the spread of security failures, but in reality it may not be so. For example, consider the "Conficker" virus that exploited a vulnerability in Microsoft Windows that was patched in October 2008. The virus continued to infect systems until March 2010[4]. So failures had occurred for more than 500 days. Hence we assume that attackers tend to exploit "full disclosure" and "limited disclosure" browsers for about 180 and 806 days. Figure 5.17 shows the result from Mann-Whitney U test for a significant difference in the number of systems affected due to security failures. From the result ($p < 0.0001$), there is a significant difference in the number of systems affected due to security failures. The mean scores show that the number of systems affected for "limited disclosure" browsers is greater than that of "full disclosure" browsers.

- **CASE3** In the above cases, how long failures of a security problem lasts in the field is dependent on the patching behavior of the users. This would translate to different failure periods for the "full disclosure" and "limited disclosure" browsers. So we also consider the case where the failure period and patching process is similar for "full disclosure" and "limited disclosure" browsers. We assumed that security failures last for about a month for "full disclosure" and "limited disclosure" browsers. Figure 5.18 shows the result from Mann-Whitney U test for a significant difference in the number of systems affected

[4]http://www.zdnetasia.com/conficker-fizzled-a-year-ago-but-headache-remains-62062336.htm

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable Infected
Classified by Variable Type

Type	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
limited disclosure	8155	71905372.0	54940235.0	220670.255	8817.33562
full disclosure	5318	18862229.0	35827366.0	220670.255	3546.86517

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	18862229.0000
Normal Approximation	
Z	-76.8800
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001
t Approximation	
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001

Z includes a continuity correction of 0.5.

Figure 5.17: Mann-Whitney U test for security failures (case 2)

due to security failures. From the result ($p < 0.0001$), there is a significant difference in the number of systems affected due to security failures. The mean scores show that the number of systems affected for "limited disclosure" browsers is greater than that of "full disclosure" browsers.

- **CASE4** From Table 2.8, we observe that during the last five years, the failure period for popular security threats were approximately a week. Next, we consider the case of short failure periods. Figure 5.19 shows the result from Mann-Whitney U test. From the result ($p < 0.0001$), we find that there is a significant difference in the number of users affected in the event of security failures. The mean scores show that the number of systems affected for "limited disclosure" browsers is greater than that of "full disclosure" browsers.

5.3.3 Impact of Patching Policies

Early disclosure/early warning of voluntary security problems may alert users in taking precautions against potential attacks. Some interesting questions are "Under what conditions is the policy of early disclosure of voluntary security problems a good one?", "Can developers take this opportunity to allow additional time to fix such problems?", "How can developers determine the percentage of the users that must take precautions to compensate for the delay in fixing the problems?", etc. Time-to-disclose, time-to-fix, time-to-apply-patch (automatic and non-automatic updates), time-to-intrusion and the number of users are the metrics of interest here. For various values of the metrics under consideration, we discuss the impact of change in disclosure, correction and user patching processes, on the number of users exposed at the time

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable Infected
Classified by Variable Type

Type	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
limited disclosure	8163	64455956.5	55753290.0	225985.892	7896.11129
full disclosure	5496	28835013.5	37537680.0	225985.892	5246.54540

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	28835013.5000
Normal Approximation	
Z	-38.5098
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001
t Approximation	
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001

Z includes a continuity correction of 0.5.

Figure 5.18: Mann-Whitney U test for security failures (case 3)

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable Infected
Classified by Variable Type

Type	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
limited disclosure	8151	59889573.5	55606122.0	225605.701	7347.51239
full disclosure	5492	33182972.5	37466424.0	225605.701	6042.05617

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	33182972.5000
Normal Approximation	
Z	-18.9864
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001
t Approximation	
One-Sided Pr < Z	<.0001
Two-Sided Pr > |Z|	<.0001

Z includes a continuity correction of 0.5.

Figure 5.19: Mann-Whitney U test for security failures (case 4)

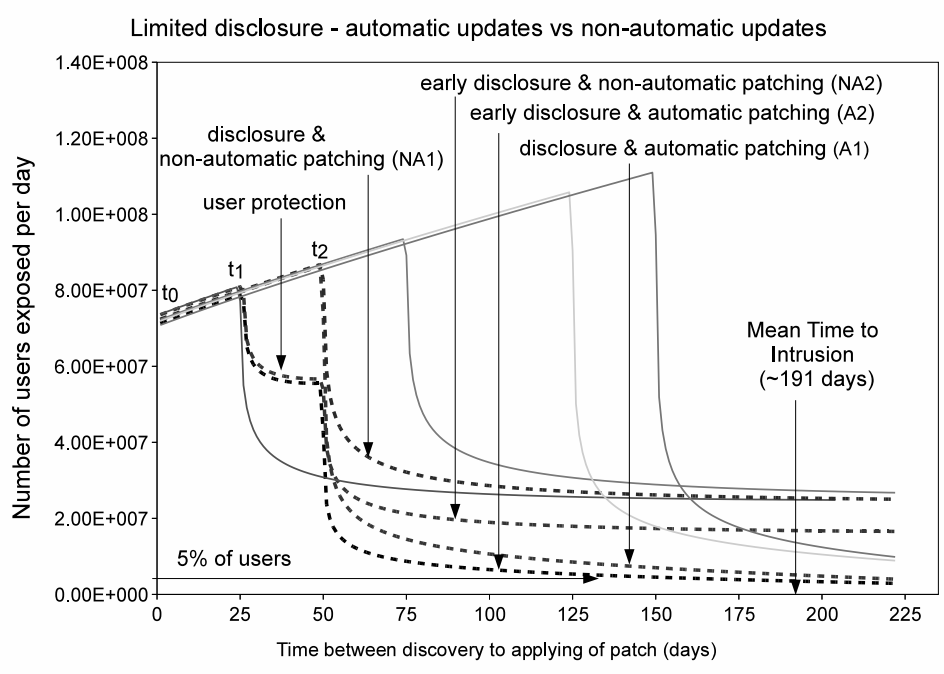

Figure 5.20: Impact of patching policies

of security failures.

Figure 5.20 shows some sample results from this investigation. In Figure 5.20, the horitzontal axis (X-axis) shows the time between between the discovery of a security problem in the field and installation of patches by users measured in calendar days.The vertical axis (Y-axis) shows the number of users exposed to the security problem between discovery and patch installation. For discussion, let us consider the curves shown in dotted lines. t_0 is the time when a security problem is discovered. t_2 is the time when the security problem is publicly disclosed along with the fix. t_1 illustrates a sample case of the early disclosure of the security problem to alert users. NA1 and A1 are the trends when the software product follows the process of disclosure with fix at time t_2 with non-automatic and automatic patch updates. Similarly, NA2 and A2 are the trends when the software product follows early disclosure of the problem at time t_1 and release of fix at time t_2. NA2 is illustrated for the case where a security problem is disclosed 25 days earlier and when 30% of users heed the warning. The mean time-to-intrusion is about 191 days. We assume that a developer would be interested in measuring the proportion of vulnerable systems at the time of a security failure. For example, let us consider that the developer is interested in making sure that at least 95% of the users are protected before a security failure happens. From the Figure, we can observe that for the given time of

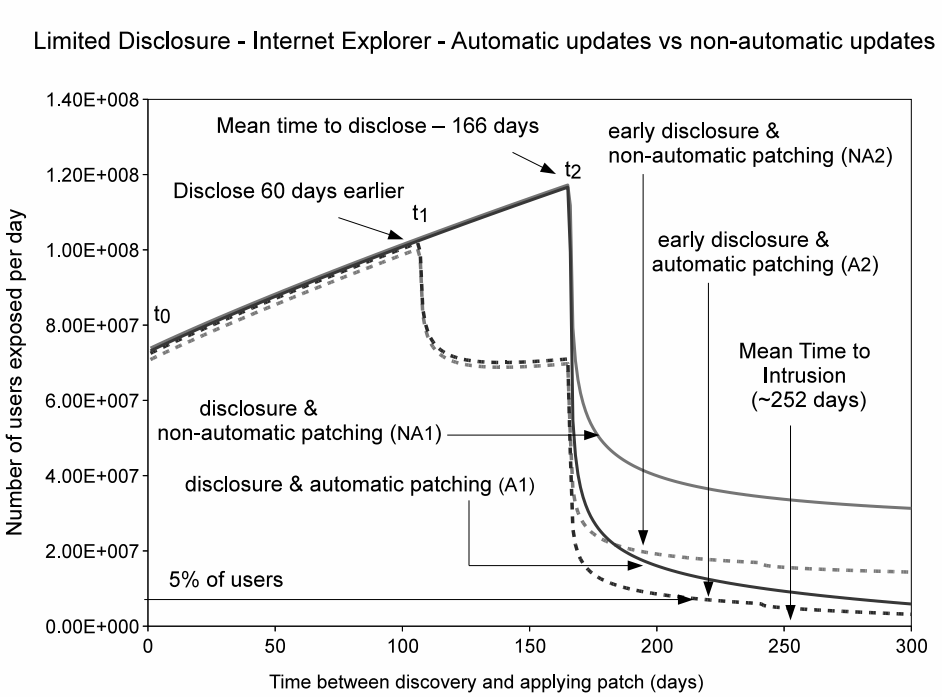

Figure 5.21: Impact of patching policies in Internet Explorer

early disclosure, the developers must ensure that there is a high probability of at least 30% of the users heed the warning to achieve the protection level of 95%. Another way to look at the problems is, given that users heed to warning at a specific rate, developers can determine what is the optimal time to alert users of such problems. Similar techniques can be used with project specific parameter values to guide decisions on real time policies. For example, Figure 5.21 shows, for Internet Explorer, the impact of the patching policies on early disclosure of 60 days and when 30% of the users heed the warning. To protect 95% of Internet Explorer users, developers must ensure that there is a high probability that at least 30% of the users heed the warning in the event of an alert 60 days prior to the normal disclosure process.

Early Disclosure of Security Problems

With early disclosure/warning, developers may take the opportunity to allow additional time to fix such problems. To do so, developers need to determine what percentage of the users that must take precautions to compensate for the delay in fixing the problems. We measure the total number of users exposed with the current process of disclosure and fix of voluntary security problems. Then for a specified early disclosure and additional time taken to fix a security problem, we analyse what percentage of users must take precautions in order to compensate

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable USERS
Classified by Variable PROTECTED

PROTECTED	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
NONE	5276	27438971.5	27923230.0	157169.954	5200.71484
EIGHTEEN	5308	28576848.5	28092590.0	157169.954	5383.73182

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	27438971.5000
Normal Approximation	
Z	-3.0811
One-Sided Pr < Z	0.0010
Two-Sided Pr > |Z|	0.0021
t Approximation	
One-Sided Pr < Z	0.0010
Two-Sided Pr > |Z|	0.0021

Z includes a continuity correction of 0.5.

Figure 5.22: Mann-Whitney U test for early disclosure (18% users)

for the additional delay in time taken to fix the security problem. We conduct the test for an example of early disclosure of 1 week and delay in the correction time by 1 week, in Firefox. Figures 5.22 and 5.23 show that there is a significant difference in the number of users exposed when only 18% and 19% of users take precautions. Figure 5.24 shows that there is no significant difference in the number of users exposed when 20% of the users take precautions as opposed to no users taking precautions with the current process. This indicates that if developers can make sure that at least 20% of users will take precautions because of the early disclosure, then the developers can delay the fix of the security problem by a week and still achieve the same impact as the current process of disclosure and fix of voluntary security problems. Note that, the technique described in Section 5.3.3 can also be used here.

Optimal Patch Time

In this section, we discuss how users can determine the optimal patching time for a security problem when switching between softwares?. We illustrate how a user of "limited disclosure" browser, who typically takes a month to apply security patches, can decide on the optimal patch time when switching to "full disclosure" browsers. We determine the optimal patching behavior of a user when switching between products, using break-even analysis (Figure 5.25). In Figure 5.25, the horizontal axis (X-axis) shows the patching time in days for "full disclosure" browser. The vertical axis (Y-axis) shows the chances of a user not being exploited in the event of a broad spread failure in the field given that the user typically waits until developers to release patches and may not be singled out in an attack. From Figure 5.25, we observe that if the user, after switching to "full disclosure" browsers, cannot apply security patches within 5 days,

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable USERS
Classified by Variable PROTECTED

PROTECTED	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
NONE	5276	28213362.0	28593282.0	162804.907	5347.49090
NINETEEN	5562	30523179.0	30143259.0	162804.907	5487.80636

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	28213362.0000
Normal Approximation	
Z	-2.3336
One-Sided Pr < Z	0.0098
Two-Sided Pr > |Z|	0.0196
t Approximation	
One-Sided Pr < Z	0.0098
Two-Sided Pr > |Z|	0.0196

Z includes a continuity correction of 0.5.

Figure 5.23: Mann-Whitney U test for early disclosure (19% users)

The NPAR1WAY Procedure

Wilcoxon Scores (Rank Sums) for Variable USERS
Classified by Variable PROTECTED

PROTECTED	N	Sum of Scores	Expected Under H0	Std Dev Under H0	Mean Score
NONE	5276	28255261.0	28445554.0	161564.521	5355.43234
TWENTY	5506	29875892.0	29685599.0	161564.521	5426.06102

Average scores were used for ties.

Wilcoxon Two-Sample Test

Statistic	28255261.0000
Normal Approximation	
Z	-1.1778
One-Sided Pr < Z	0.1194
Two-Sided Pr > |Z|	0.2389
t Approximation	
One-Sided Pr < Z	0.1194
Two-Sided Pr > |Z|	0.2389

Z includes a continuity correction of 0.5.

Figure 5.24: Mann-Whitney U test for early disclosure (20% users)

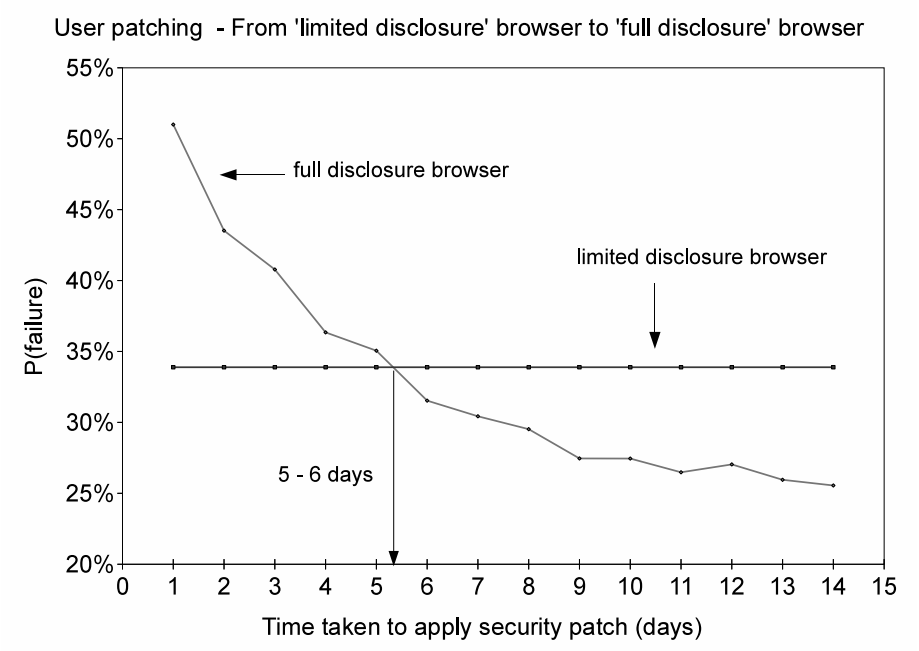

Figure 5.25: Break even analysis for optimal patch time

then such a user is better off using "limited disclosure" browser than "full disclosure" browsers.

Chapter 6

Conclusion

Security is important when evaluating software quality. In analysing the operational security of software systems with large number of users, it is desirable to analyse security problems by taking in to account the operational profile of the software that harbors such problems. In this dissertation, we have showed how software usage levels, along with publicly available information such as developer response process, user patching process, and attacker intrusion processes, can be collected and analysed in a unified context and then used to improve our understanding of security issues, processes and related decision making. The principal contributions and findings of this work are:

- Development of a toolset for automated collection of security related information from public repositories [11]. The toolset has been effectively used as a part of the research reported here to collect information on both security and non-security problems for a broad range of software products [Chapter 2].

- Empirical evidence of several interesting characteristics of security problems. For example, findings on how security problems differ from non-security problems in problem reporting and correction, classification of security problems based on user interaction with the attack mechanism, information on field exploits and failures, etc., are reported [Chapter 2].

- A comprehensive security problem response model that emphasizes roles and software usage perspectives. The model captures the states through which a system may go based on the discovery and disclosure of security problems, exploits, field failures, type of problems, correction status (e.g., a developer fixes a security problem or not), etc. [Chapter 3]

- Two models for some of the security transitions in the security problem response model. First, it is shown that classical reliability models work much better, when extended to software security space, when the exposure is expressed not in calendar time, but in terms

of system usage or inservice time. Second, it is shown that Bayesian disclosure models can predict the disclosure of security problems with respect to the operational use of a software at a performance level on par with that of classical software reliability models [Chapter 4].

- Analysis on the interactions of various events covered by the overall model to provide an understanding on security processes in software products that follow full disclosure of security problems and those that follow limited disclosure of security problems, in combination with automatic and non-automatic types of patching processes. Questions that can help users and developers in making process related decisions are also addressed [Chapter 5].

6.1 Future Work

- In the current framework, the data collection toolset requires human intervention to decide on the repository relations. It will be interesting to explore possible techniques to automatically analyse repositories and identify relation between repositories. We will explore approaches towards addressing the issue and extend the applicability of the toolset beyond the security space.

- Our case studies and analyses include a mix of security related information from a pool of software products. Further, information such as software usage levels are not immediately available. We will continue to extend our analyses as more and more information are available for specific software products. We will continue to refine the model for any project specific characteristics.

- As a part of the Bayesian modeling, we will conduct user surveys to understand the impact of security problems, exploits, failures, etc., on the operational profile of a software as well as subjective impressions of users on the corresponding software products.

- In simulation based analyses, we will extend the analyses to include information on failures due to public exploits and hidden exploits. We used a simple failure propagation model used in our current analyses. We will explore complex models to reflect the spread of failures based on specific usage profiles and software products.

Chapter 7

Related Work

We compare the contributions discussed in this thesis with existing work. In Chapter 2, we introduced a toolset for automated collection of security related information from public repositories. Section 7.1 discusses related work on automated data collection from various repositories. In Chapter 3, we discussed role based based modeling of software security problems. Sections 7.2, 7.3 and 7.4 discuss existing security analyses from the perspective of users, developers, and attackers. In Chapter 4, we discussed predictive modeling approaches. Section 7.5 discusses existing work on predictive modeling of security problems. In Chapter 5, we discussed simulation based analyses in answering security related questions. Section 7.6 discusses existing work on security simulations and studies focusing on the security process.

7.1 Security Data

Robles et al. [62] propose an architecture for automated retrieval and analysis of data from open source repositories like CVS and archives of source packages. The authors discuss on including support for bug tracking systems and mailing lists as future work. Similarly, Neuhaus et al. [52] retrieve vulnerability information from bug databases and CVS repositories. Hassan [28] discusses the challenges in mining software respositories. The author identifies linking data across repositories as one of the open challenges in this area. In this paper, we address this limitation in mining data from individual repositories as well as linked information across repositories.

7.2 Attacker Perspective

Arora et al. [1] empirically study atackers' behavior in exploiting vulnerabilities that are publicly disclosed as well as those that are not yet publicly disclosed. They analyse the trend in the frequency of attacks due to public disclosure of the vulnerabilities and their patches. The

authors conclude that vulnerabilities that are publicly disclosed and patched are attacked more than those that are disclosed and remain unpatched. This study is oriented towards understanding attacker behavior. In a similar direction, Madan et al. [42] study attackers' behavior and system response specific to an intrusion tolerant system, and quantify the security attributes availability, confidentiality and integrity. This study does not account for the frequency with which such vulnerabilities are reported, exploited, or corrected. Lawrence et al. [17] provide a cost benefit analysis in attackers trying to exploit vulnerabilities. Jonsson et al. [33] discuss that attackers follow a learning, attack and innovative phase in carrying out attacks.

Gonzalo et al. [5] discuss the types of web attacks observed in the field. Ristenbatt [61] discusses the types of vulnerabilities observed in different network layers. Newsome et al. [54] focus on attacks targeted at Wireless Sensor Networks. Mirkovic et al. [46] provides a classification of denial of service attacks based on the types of victims exploited. Cuckier et al. [20] analyse attack data and identify characteristics for distinguishing various attacks. Scott et al. [30] discuss what developers and users can do to improve trust in open source.

7.3 Developer Perspective

Schniedwind [65] provides a study on the conditional probability of security failures given that the reliability failures have occured. Littlewood et al. [40] provide an overview of similarities between reliability and security from the perspective of operational behaviour of a system rather than just the safeguards used in the design and implementation phases of the system. Stefan [23] provides a comparison on the total number of exploits experienced and the number of patches released after disclosure of vulnerabilities. This study provides an overall view of exploits and patches available, but does not study the effect of exploits on correcting the vulnerabilities. Huseyin et al. [18] discuss when vendors should release patches and when customers should apply the patches.

William et al. [43] classify security flaws in operating system based on features like error handling, parallelism, etc. Bellowin et al. [14] classify the different types of security faults observed in TCP-IP protocol suite. Mathur et al. [73] classify vulnerabilities based on the cause of the attack, its impact, and fix required for recovery. Tsipenyuk et al. [72] discuss the classification of vulnerabilities based on code quality and the environment in which such vulnerabilities are exploited. Miles et al. [44] discuss a specific type of vulnerability - those that are hidden and then exploited. Fengmin et al. [27] propose a model to analyse the behaviour of an intrusion detection system and its response. Bharat et al. [42] present a semi Markov model to quantify the security attributes - confidentiality, integrity and availability, based on the model proposed by Fengmin et al.

There are many studies that qualitatively discuss the security of open source projects

e.g., [13, 6, 45]. These studies concentrate on attributes like open source policies with respect to source code and problem reports, best practices and process level comparison among various open source projects and similar.

Anderson [13] focuses on whether open access to the design, source code, etc., makes the system vulnerable or not. The author concludes that being open source or closed source does not make a system more vulnerable, but rather the importance lies in specifics of the deployments of the system and the scenarios that may lead to an attack.

In another study, Hoepman et al. [29] have a similar focus as Anderson [13], but restrict their argument only to the source code being disclosed to the public and do not distinguish between any design principles followed. The authors note that making source code open attracts more and more developers, leading to an increased participation in the activities like repairing bugs, testing, etc., The authors conclude that although open source systems may seem to have high exposure to security risk initially, they get better due to such increased involvement from the public.

Smith et al. [69] discuss the best practices and discuss the advantages of applying them in building secure open source systems. The authors study many open source operating systems, note that OpenBSD has better security and discuss use of OpenBSD practices to future projects. The authors suggest following audit process similar to OpenBSD, collaborating with open source projects with simiar security interests, face to face meetings with developers working on open source security, discussion about the management overheads such practices involve, etc.

Michlmayr et al. [45] present a statistical analysis on the defect reports of Debian Linux system. Based on the analysis, they make observations about the defect submission and removal process. The authors study the accumulation of open bug reports and correlate the trend with defect removal rate. Further they study the impacts on defect removal process due to bursts of activity (to sutdy if defects are removed in bursts rather being a gradual removal process), influence of popularity or importance of software packages, importance of modifications and uploads of new versions of the software, maintenance of the software package by one person or a team.

Mockus et al. [6] studied the development and maintenance of the open source projects Apache and Mozilla. The authors discuss processes used to develop the two projects, defect density, number of people developing new features, reporting problems and fixing problems, and functionalities performed by different groups of people on the source code. Authors of [31] discuss user participation in Mozilla open source project in terms of the resources that users can utilise to contribute towards the project. The authors discuss the bug tracking system (Bugzilla) as a tool for project management and a forum for discussion on software development.

Fielding et al. [22] provide a process level discussion on the collaborative software development for Apache server. Scacchi et al. [64] examine existing analyses on free and open

source software development and discuss the incentives behind user participation in open source projects, resources and capabilities required for open source projects, characteristics of a participant, trust and accountability in open source environment etc.

7.4 User perspective

Frei et al. [24] study the user patching behavior with respect to security problems for the popular browsers Firefox, Opera, Chrome, and Safari. In our dissertation, we categorize this user patching behavior based on open source and closed source browsers and perform simulations on the impact of such user patching process on the failure of security problems in the field. Frei et al [25] study the number of security patches an average home user requires to apply during a particular time period to be secure. In this dissertation, we focus on how users can make use of publicly available information and make decisions on secure software usage like the optimal time to apply security patches when moving from open source software to closed source software.

7.5 Predictive Modeling

Alhazmi et al [2, 3, 4] present a vulnerability discovery model and predict the long term and short term number of vulnerabilities for several major operating systems including RedHat Linux. The authors base their assumptions on the rate of discovering vulnerabilities by testers and malicious users, but test their model on the vulnerability report data [56]. The vulnerability report data only tells the time vulnerabilities were publicly disclosed and not when they were actually discovered. In their model, they assume that the number of vulnerabilities (say N) that will eventually be detected for a particular release is a constant (finite). Schryen et al [66] showed that their estimated value of N for Windows XP, Windows NT, Redhat 6.2 and Redhat 7.1 did not match with the actual number of problems eventually discovered for the projects. Further, Alhazmi et al report a significant fit of a linear model for 6 out of the 11 datasets they used. They conclude that linear model would fit only a limited number of datasets. A linear trend in the plot of cumulative number of vulnerabilities and time factor is actually a visual test that shows presence of no trend [71] in the number of unique problems reported. A "no trend" here implies no increasing or decreasing trend. Further this is evident on a majority of the datasets they have used and makes their conclusion weak. Anderson [13] proposed a theoretical model based on thermodynamics for discovery and resolution of vulnerabilities. Anderson did not evaluate his model on any real data. Alhazmi et al [4] tested Anderson's model on Win95, WinXP, or Redhat security data and observed that the model does not fit any of the data.

Thomas et al. [51] study RedHat packages and correlate vulnerabilities with dependencies between packages. Thomas et al. [53] map past vulnerabilities to components, and build a

model to predict vulnerabilities based on past history and function calls. Researchers have tried applying classical software reliability models on security data and while doing so, they often forget to justify the assumptions of the reliability models [56]. Rescorla [60] assumed that security problems are similar in nature to non-security problems and follow a reliability growth pattern as that of non-security problems. Rescorla used the Goel-Okumoto reliability model on security data for Redhat 6.2, WinNT4, Solaris 2.5.1 and FreeBSD 4.0 and was able to fit the model only for Redhat 6.2, and not the others.

Ozment [55] applied seven reliability models to OpenBSD security data set. Ozment classified the dataset under two perspective: failure and fault perspective. He observed that sometimes a set of vulnerabilities would be detected around the same time and a single patch would fix all of them. He termed these as related security faults. Failure perspective data would include the related security faults grouped as one under the notion that one failure could have revealed all those related faults. Fault perspective is where all faults are considered as individual data points. He found that none of the seven reliability models worked on the fault data, while three models with Musa's Logarithmic model being a best fit worked for the failure data. there is no evidence to support that a single failure revealed all the related faults. Further, it is possible to find faults reported around the same time by different sources and different fixes may be bundled together as a single patch release. Thus grouping bugs as failure perspective seems to be weak since this lacks evidence and excludes data points. Not excluding these data points turns the result otherwise which is the analysis from fault perspective. Nevertheless the author concludes that although he could observe a fit on the data, there is no additional information available to justify the assumptions of the model for security data. Thus the results of this analysis is inconclusive.

7.6 Simulation Analyses

Beres et al. [15] use simulation to study security policies and emergency escalations internal to an organization in the event of disclosure of security problems. In this thesis, we focus on a more comprehensive model that analyses the interaction of events in an external environment including the process of discovery and disclosure of security problems, and influence of attackers in the form of security failures. Further, we distinguish the security process of software systems that follow full-disclosure of security problems and those that do not.

REFERENCES

[1] A.Arora, A.Nandkumar, and R.Telang. Does information security attack frequency increase with vulnerability disclosure? an empirical analysis. In *Information Systems Frontiers*, pages 350–362, 2006.

[2] O. H. Alhazmi and Y. K. Malaiya. Modeling the vulnerability discovery process. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 129–138, Washington, DC, USA, 2005. IEEE Computer Society.

[3] O. H. Alhazmi and Y. K. Malaiya. Application of vulnerability discovery models to major operating systems. *Reliability, IEEE Transactions on*, 57(1):14–22, March 2008.

[4] Omar H. Alhazmi, Yashwant K. Malaiya, and Indrajit Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[5] Gonzalo Alvarez and Slobodan Petrovic. A new taxonomy of web attacks suitable for efficient encoding. *Computers and Security*, 22(5):435–449, 2003.

[6] A.Mockus, R.Fielding, and J.Herbsleb. Two case studies of open source software development: Apache and mozilla. In *ACM Transactions on Software Engineering and Methodology*, pages 309–346. ACM Press, 2002.

[7] Murugan Anandarajan and Claire A. Simmers, editors. *Managing web usage in the workplace: a social, ethical and legal perspective*. IGI Publishing, Hershey, PA, USA, 2002.

[8] P. Anbalagan and M. Vouk. Student paper:on reliability analysis of open source software-fedora. In *ISSRE '08: Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering*, Seattle, WA, USA, 2008. IEEE Computer Society.

[9] P. Anbalagan and M. Vouk. Towards a unifying approach in understanding security problems. In *ISSRE '09: Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering*, Bengaluru, India, 2009. IEEE Computer Society.

[10] Prasanth Anbalagan and Mladen Vouk. An empirical study of security problem reports in linux distributions. In *In the Third International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*, October 2009.

[11] Prasanth Anbalagan and Mladen Vouk. On mining data across software repositories. In *Proceedings of the Sixth IEEE Working Conference on Mining Software Repositories*, May 2009.

[12] Prasanth Anbalagan and Mladen Vouk. Towards a bayesian approach in modeling the disclosure of unique security faults in open source projects. *Software Reliability Engineering, International Symposium on*, 0:101–110, 2010.

[13] Ross Anderson. Security in open versus closed systems the dance of boltzmann, coase and moore. In *In Conference on Open Source Software Economics*, pages 1–15. MIT Press, 2002.

[14] S. M. Bellovin and The Tcp/ip Protocol Suite. Security problems in the tcp/ip protocol suite. *Computer Communications Review*, 19:32–48, 1989.

[15] Yolanta Beres, Jonathan Griffin, Simon Shiu, Max Heitman, David Markle, and Peter Ventura. Analysing the performance of security solutions to reduce vulnerability exposure window. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 33–42, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Yuriy Bulygin. Epidemics of mobile worms. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:475–478, 2007.

[17] Lawrence Carin, George Cybenko, and Jeff Hughes. Cybersecurity strategies: The queries methodology. *Computer*, 41(8):20–26, 2008.

[18] Huseyin Cavusoglu, Hasan Cavusoglu, and Jun Zhang. Economics of security patch management. In *Proceedings of the The Fifth Workshop on the Economics of Information Security (WEIS 2006)*, June 2006.

[19] R. Cramp, M. Vouk, and W. Jones. On operational availability of a large software-based telecommunications system. In *Proc. Third Intl. Symposium on Software Reliability Engineering*, pages 358–366, 1992.

[20] Michel Cukier, Robin Berthier, Susmit Panjwani, and Stephanie Tan. A statistical analysis of attack data to separate attacks. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 383–392, Washington, DC, USA, 2006. IEEE Computer Society.

[21] William Farr. Software reliability modeling survey. In Michael R. Lyu, editor, *Handbook of software reliability and system reliability*, chapter 3, pages 71–117. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[22] Roy T. Fielding. Shared leadership in the apache project. *Commun. ACM*, 42(4):42–43, 1999.

[23] Stefan Frei. The dynamics of (in)security. In *ZISC Information Security Colloquium*, January 2007.

[24] Stefan Frei, Thomas Duebendorfer, and Bernhard Plattner. Firefox (in) security update dynamics exposed. *SIGCOMM Comput. Commun. Rev.*, 39:16–22, December 2008.

[25] Stefan Frei and Thomas Kristensen. The security exposure of software portfolios. 2010.

[26] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, LSAD '06, pages 131–138, New York, NY, USA, 2006. ACM.

[27] K. Goeva-Popstojanova, F. Wang, R. Wang, F. Gong, K. Vaidyanathan, K. Trivedi, and B. Muthusamy. Characterizing intrusion tolerant systems using a state transition model. In *DISCEX II: DARPA Information Survivability Conference and Exposition*, volume 2, pages 211–221, 2001.

[28] A.E. Hassan. The road ahead for mining software repositories. In *Proceedings of Frontiers of Software Maintenance, 2008. FoSM 2008*, pages 48–47, 2008.

[29] Jaap henk Hoepman and Bart Jacobs. Increased security through open source. *Communications of the ACM*, 50:79–83, 2007.

[30] Scott A. Hissam, Daniel Plakosh, and Charles B. Weinstock. Trust and vulnerability in open source software. *IEE Proceedings - Software*, 149(1):47–51, 2002.

[31] I.Oeschger and D.Boswell. *Getting your work in to Mozilla*. Oreilly.com, 2000.

[32] William Jones and Mladen A.Vouk. Software reliability field data analysis. In Michael R. Lyu, editor, *Handbook of software reliability and system reliability*, chapter 11, pages 439–489. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[33] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Trans. Softw. Eng.*, 23(4):235–245, 1997.

[34] Karama Kanoun and Jean-Calude Laprie. Software reliability trend analyses from theoretical to practical considerations. *IEEE Transactions on Software Engineering*, 20:740–747, September 1994.

[35] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, October 2006.

[36] B. Littlewood. A bayesian differential debugging model for software reliability. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 129–130, New York, NY, USA, 1981. ACM.

[37] B. Littlewood and J. L. Verrall. A bayesian reliability growth model for computer software. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 22, No.3:332–346, 1973.

[38] Bev Littlewood. How to measure software reliability, and how not to. In *ICSE*, pages 37–45, 1978.

[39] Bev Littlewood. Theories of software reliability: How good are they and how can they be improved? *IEEE Trans. Software Eng.*, 6(5):489–500, 1980.

[40] Bev Littlewood, Sarah Brocklehurst, Norman Fenton, Peter Mellor, David Wright, John Dobson, John Mcdermid, Dieter Gollmann, and Egham Tw Ex. Towards operational measures of computer security. *Journal of Computer Security*, 2:211–229, 1993.

[41] Brian P. Macfie and Philip M. Nufrio. *Applied Statistics for Public Policy*. M.E. Sharpe, Inc., New York, USA, 2006.

[42] Bharat B. Madan, Katerina Goseva-Popstojanova, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. Modeling and quantification of security attributes of software systems. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 505–514, Washington, DC, USA, 2002. IEEE Computer Society.

[43] William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3):230–352, 1974.

[44] Miles A. McQueen, Trevor A. McQueen, Wayne F. Boyer, and May R. Chaffin. Empirical estimates and observations of 0day vulnerabilities. *Hawaii International Conference on System Sciences*, 0:1–12, 2009.

[45] Martin Michlmayr and Anthony Senyard. *A Statistical Analysis of Defects in Debian and Strategies for Improving Quality in Free Software Projects*. Elsevier B.V., 2006.

[46] J. Mirkovic and P. Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communications*, 43, no.2:39–53, 2004.

[47] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1:33–39, July 2003.

[48] David Moore, Colleen Shannon, and k claffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 273–284, New York, NY, USA, 2002. ACM.

[49] J. D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In *Proceedings of the 7th International Conference on Software Engineering*, pages 230–238, Piscataway, NJ, USA, 1984. IEEE Press.

[50] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software reliability: measurement, prediction, application*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[51] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. Technical Report DISI-09-007, Ingegneria e Scienza dell'Informazione, University of Trento, January 2009.

[52] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, October 2007.

[53] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, October 2007.

[54] J. Newsome. The sybil attack in sensor networks: Analysis and defenses. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, pages 259–268, October 2004.

[55] Andy Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In *In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, eds., Quality Of Protection: Security Measurements and Metrics*, May 2006.

[56] Andy Ozment. Improving vulnerability discovery models. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 6–11, 2007.

[57] Robinson. J. P, Kestnbaum. M, Neustadtl. A., and Alvarez. A. S., editors. *The Internet and Other Uses of Time, in The Internet in Everyday Life (eds B. Wellman and C. Haythornthwaite)*. Blackwell Publishers Ltd, Oxford, UK, 2008.

[58] P.Anbalagan and M.A.Vouk. Security failure estimation for open source software: An empirical approach. In *Proceedings of the 1st Workshop on Dependable Software Engineering 2008 (WDSE 2008)*, Seattle/Redmond, Washington, USA, 2008.

[59] Sam Ransbotham. An empirical analysis of exploitation attempts based on vulnerabilities in open source software. In *Proc. Workshop on the Economics of Information Security*, 2010.

[60] Eric Rescorla. Is finding security holes a good idea? In *IEEE Security and Privacy*, Jan-Feb 2005.

[61] M. P. Ristenbatt. Methodology for network communication vulnerability analysis. *MILCOM 1988*, 2:493–499, 1988.

[62] Gregorio Robles and Juan Carlos. Gluetheos: Automating the retrieval and analysis of data from publicly available software repositories. In *In Proceedings of the International Workshop on Mining Software Repositories*, pages 28–31, 2004.

[63] Sheldon M. Ross. *Introduction to Probability Models, Ninth Edition*. Academic Press, Inc., Orlando, FL, USA, 2006.

[64] Walt Scacchi. Free/open source software development: recent research results and emerging opportunities. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 459–468, New York, NY, USA, 2007. ACM.

[65] Norman F. Schneidewind. Reliability - security model. In *ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 279–288, Washington, DC, USA, 2006. IEEE Computer Society.

[66] Guido Schryen and Rouven Kadura. Open source vs. closed source software: towards measuring security. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2016–2023, New York, NY, USA, 2009. ACM.

[67] Guido Schryen and Eliot Rich. Increasing software security through open source or closed source development? empirics suggest that we have asked the wrong question. *Hawaii International Conference on System Sciences*, 0:1–10, 2010.

[68] Giuseppe Serazzi and Stefano Zanero. Computer virus propagation models. In *In Tutorials of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS03)*. Springer-Verlag, 2003.

[69] Jonathan M. Smith, Michael B. Greenwald, Sotiris Ioannidis, Angelos D. Keromytis, Ben Laurie, Douglas Maughan, Dale Rahn, and Jason Wright. Experiences enhancing open source security in the posse project. In Stefan Koch, editor, *Free/Open Source Software Development*, pages 242–257, Hershey, PA, 2004. Idea Group Publishing.

[70] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[71] Paul Tobias. *Assessing Product Reliability, Handbook of Statistical Methods*. National Institute of Standards and Technology, and SEMATECH, Washington, DC, USA, 1991.

[72] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security and Privacy*, 3(6):81–84, 2005.

[73] Security Breaches Wenliang, Wenliang Du, and Aditya P. Mathur. Categorization of software errors that led to security breaches. In *In 21st National Information Systems Security Conference*, pages 392–407, 1998.

[74] Thomas Zimmermann. Knowledge collaboration by mining software repositories. In *Proceedings of the 2nd International Workshop on Supporting Knowledge Collaboration in Software Development*, pages 64–65, September 2006.

[75] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 138–147, New York, NY, USA, 2002. ACM.