Software Security Economics: Theory, in Practice

Stephan Neuhaus Bernhard Plattner
Eidgenössische Technische Hochschule Zürich

April 26, 2012

Abstract

In economic models of cybersecurity, security investment yields positive, but diminishing, returns. If that were true for software vulnerabilities, fix rates should decrease, whereas the time between successive fixes should go up as vulnerabilities become fewer and harder to fix.

In this work, we examine the empirical evidence for this hypothesis for Mozilla, Apache httpd and Apache Tomcat over the last years. By looking at 292 vulnerability reports for Mozilla, 66 for Apache, and 21 for Tomcat, we find that the number of people committing vulnerability fixes changes proportionally to the number of vulnerability fixes for Mozilla and Tomcat, but not for Apache httpd.

Our findings do not support the hypothesis that vulnerability fix rates decline. It seems as if the supply of easily fixable vulnerabilities is not running out and returns are not diminishing (yet).

Additionally, software security has traditionally been viewed as an arms race between an attackers and defenders. Recent work in an unrelated field has produced precise mathematical models for such arms races, but again, the evidence we find is scant and does not support the hypothesis of an arms race (of this kind).

1 Introduction

In standard economic models of cybersecurity, security can be bought incrementally, but there are diminishing returns: some investment later buys less security than the same investment earlier [17]. From comparing several cybersecurity models, Rue et al. find that the security function (the function that says how much return one gets for how much investment) that underlies every economic model of cybersecurity is increasing and concave[1].

In order to view software security in economic terms, we assume this basic workflow:

[1] The authors use 'convex' instead of 'concave', and by 'convex' they mean "any twice continuously differentiable function". But unless that function has a negative second derivative, the diminishing returns don't happen. However, a negative second derivative is a criterion of *concavity*, not convexity.

1. Programmer A commits a *vulnerability*, which is an error that has security consequences.

2. That vulnerability is discovered and *assigned* to programmer B for fixing. We might have $A = B$.

3. Programmer B *fixes* the vulnerability, either by making all the required changes himself (even in code not written by him), or by making still other people contacting the people who own the code that is to be corrected.

4. A fix might not solve the vulnerabilty completely, causing steps 2 and 3 to be *repeated*.

If fixing vulnerabilities could be modeled as a security function, one expends a certain amount of work and reduces the total number of vulnerabilities by one. As time goes by, vulnerabilities become fewer and harder to fix, so returns will be diminishing. Certainly, adding more people to fix vulnerabilities will at some point only bring diminishing returns as administrative overhead eats up the potential for increased productivity. That this occurs is clearly the expectation of at least some economists [17].

It is also of economic interest to know how often vulnerability fixes need to be repeated for a single vulnerabilty. Clearly, the fewer the better.

Another concern is the size of the *security team*, by which we mean the number of developers people who fix vulnerabilities. The size of the security team ought to depend on the code ownership model. In one code ownership model that we call *individual code ownership*, the person having written the code in question retains ownership of that code and is expected to fix any bugs pertaining to it, including vulnerabilities. In this case, we will have $A = B$ in step 2, and we will have A contact other code owners in the workflow aboveIn. In the *communal code ownership* model, in principle anyone can fix anybody else's code, so we will in general have $A \neq B$, and B will fix the vulnerability all by himself.

We can see from this that individual code ownership should lead to a large security team, since many code owners will commit vulnerability fixes, whether they are interested in security or not, whereas in communal code ownership, checkins will be made by people who like fixing vulnerabilities, and the security team will therefore be smaller. This has staffing consequences, since in individual code ownership, *all* developers will have potentially to be schooled in how to fix security issues. This is different from writing secure code, which all developers ought to know anyhow.

From a different perspective, software security is often seen as an arms race between hackers and programmers, sometimes termed "Red Queen" (for the attackers) and "Blue King" (for the programmers). The term comes from Lewis Carroll's *Through the Looking-Glass*, where the Red Queen complains that "it takes all the running you can do, to keep in the same place" [4]. In our case, the Red Queen tries to find and exploit vulnerabilities, which the Blue King then eventually patches. If the Blue King is successful, the Red Queen would then

have to expend time and energy, just to keep her supply of possible exploits constant ("running [...] to keep in the same place"). In this case, the time between successive vulnerability fixes should increase and obey a power law [7].

Our objects of study are three large software systems: first, the Mozilla suite (including Firefox and Thunderbird, but also lesser-known products like Seamonkey); second, the Apache HTTP server httpd; and third, the Apache Tomcat application server. We chose these projects because they are widely used, because they represent a wide variety of vulnerability-prone, internet-facing software, and because development data such as source code repositories and bug and vulnerability databases are freely available.

Our contribution is twofold. First, from the theoretical considerations above, we have the following predictions, which we check on our three software systems:

- vulnerability fix rates should decrease as diminishing returns set in (we find no evidence that this prediction is true);
- the size of the security team should be correlated with the vulnerability fix rate or not, depending on the code ownership model (again we find no evidence that this is true); and
- the time between vulnerability fixes should increase and obey a power law (the time tends very roughly to increase, but the data do not support a power law).

Second, we also find that:

- the quality of vulnerability fixes is good: most vulnerabilities are fixed once, with no need to fix the fix; and
- the distribution of vulnerability fixes per day obeys some, and perhaps the same, heavy-tailed distribution for all three applications. This is a result that was not derived by a research question, but came from the data analysis. We do not yet have a theoretical explanation for this behaviour, but speculate on possible causes; see Section 4.2.

The rest of this paper is organised as follows. First, we describe our data collection (Section 2) and evaluation methodology (Section 3). Then we describe our findings and speculate about possible causes for the results (Section 4). Finally, we describe related work (Section 5), analyse threats to validity (Section 6) and conclude with conclusions and future work (Section 7). Appendix A contains instructions on how to replicate our findings, Appendix B gives the algorithm we used to reconstruct checkins from CVS logs, and Appendix C gives a more formal treatment of the transformations we applied to the data.

2 Data Collection

For our purposes, a *checkin* or *commit* is the uploading of a set of related changes to source files to a source code repository. Checkins are usually perceived to be atomic, but this is not always the case; see below.

Figure 1: A typical MFSA, showing the bug identifier.

When programmers fix a vulnerability, or part of a vulnerability, they commit those changes to the source code for which they have the responsibility or authority to change. The systems where checkins and vulnerability information are stored are usually separate, so in order to find out which checkins fixed vulnerabilities, they have first to be united. This section spells out in more detail how this process works for the three software systems under observation.

2.1 Mozilla: Reconstituting Checkins

The Mozilla Foundation uses Mozilla Foundation Security Advisories (MFSAs) to report security issues in Mozilla products.[2] Our dataset contains 417 MFSAs, starting with MFSA 2005-01, issued January 21, 2005, and ending with MFSA 2011-18, issued April 28, 2011.

MFSAs can contain references to bug reports, and there can be zero or more such references; see Figure 1. These references in turn contain the bug identifier, chosen by the bug tracking software. (The identifier does not carry any semantics.) While not all bugs are vulnerabilities, all vulnerabilities are bugs, so we will identify a vulnerability in Mozilla by its bug identifier. If the programmer committing the fix for a vulnerability mentions its bug identifier in the commit message—as is usual in Mozilla—we can identify the as fixing that vulnerability. Typical commit messages contain "bug n", "bug=n", or "b=n".

Not all bug identifiers that are mentioned in MFSAs appear in checkin messages. We call a bug identifier *unassigned* when there is no checkin carrying that identifier; otherwise we call it *assigned*.

[2] `http://www.mozilla.org/security/announce/`

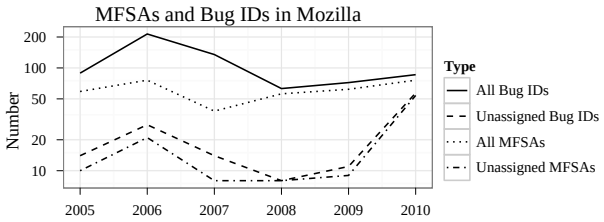

Figure 2: Number of MFSAs and their associated bug identifiers, by year. The 'Year' column contains the year in the MFSA name, which is not necessarily the same as its publication year. Note the logarithmic y axis.

Also, not all MFSAs contain such bug identifier references. For example, MFSA 2010-79 references a Java LiveConnect bug and an entry in the Common Vulnerabilities and Exposures database, but no bug identifier. In all, only 382 out of 417 MFSAs have such references. In analogy to bug identifiers, we call an MFSA *assigned* if it contains a bug identifier, and *unassigned* otherwise.

Even worse, only 292 MFSAs were fixed using assigned bug identifiers. The earliest example of an MFSA that we could not associate with a checkin is MFSA 2005-12 ("javascript: Livefeed bookmarks can steal cookies"). This MFSA is associated with the bug identifier 265668, which appears nowhere in the CVS log messages.

One particular aspect of the Mozilla checkins is that they have been historically stored in a CVS version archive. After March 2007, there has been a Mercurial repository too, but the main work went on in the CVS repository, which was considered to be authoritative. Only since October 2010 has the "relic master copy" officially switched to the Mercurial repository.[3]

In the top two lines in Figure 2, we see that since 2008, there is a trend towards "one MFSA, one bug identifier". This is unlikely to happen by chance, so we suspect either a conscious effort by the Mozilla team to assign only one bug identifier per vulnerability, or a bug identifier now tends to be assigned only after the corresponding vulnerability is discovered.

Generally, the gap between the total number of bug identifiers or MFSAs on the one hand and the number of unassigned bug identifiers or MFSAs on the other is closing since 2009. This seems to indicate that the main work is now going on in the Mercurial repository.

Mercurial commits all changed files as a single changeset. CVS, however, does not have this notion of a checkin transaction; files are individually version-controlled by RCS, so there is no easy way to ask "which files were affected by the last checkin?". This is perfectly fine in itself, but since from a developer perspective, fixes occur in checkins, we therefore need to reconstruct those checkins from the individual version-controlled files. Also, since there are 639,783 individual file checkins, or *CVS log entries*, in our dataset, the reconstruction algorithm

[3]See comment on changeset 56642:882525a98119.

needs to be efficient. We describe our algorithm in Appendix B; when applied to our dataset, we found 186,170 checkins in the Mozilla CVS, ranging from March 28, 1998 to February 22, 2011.

2.2 Apache httpd and Tomcat: Assigning Checkins

The source code and other development artefacts for Apache httpd (from now on simply called httpd) does not offer a way to link vulnerabilities and their fixing checkins. Apache publishes security information about httpd[4] using the Common Vulnerabilities and Exposures (CVE) database. Each report also contains timestamps when the security team was made aware of the vulnerability, then the issue was published, when the update was released, and which versions are affected. What is missing is the link to a bug report (called a 'Problem Report' by the developers, and abbreviated PR). Therefore, we went manually through all reported vulnerabilities and tried to find the fixing commit ourselves.

Such an approach naturally introduces uncertainties. We have therefore labeled our vulnerability-to-commit mapping with the degree of certainty that the commit in question is actually the one that fixes the named vulnerability. There are four certainty levels:

- **Certain.** The commit message contains the CVE name and asserts that it fixes the issue described in it.
- **High.** The commit message mentions issues thematically related to the CVE message and the timeline fits (commits must come after the security team was made aware of the issue, but before a fix is released), but the CVE identifier is not explicitly mentioned.
- **Low.** One or more of the above indicators (commit message, timeline) fits, but not all of them. The CVE identifier is not mentioned.
- **Unassigned.** No commit had any of the above indicators.

Out of the 100 CVE entries for Apache, 34 were categorised as *unassigned*, leaving 66 reports.

The situation for Tomcat is better. From 2008 onwards, the vulnerability reports[5] always contain the revision number of the fixing checkin. Conversely and unfortunately, none of the reports from 2007 or earlier have any such attribution, so that fully 68 out of the 89 CVEs for Tomcat are unassigned, leaving 21 CVEs, whose attribution to a fixing checkin, however, is absolutely certain.

Httpd and Tomcat both reside in SVN repositories. SVN also lacks the ability to retrieve changesets, but the situation is much improved over CVS, since SVN labels commits with their own unique revision number. Therefore, one can simply group all log file entries with the same revision number into one checkin.

[4] `http://httpd.apache.org/security/vulnerabilities_`x`.html`, $x \in \{13, 20, 22, 23\}$.
[5] `http://tomcat.apache.org/security-`x`.html`, where $x \in \{5, 6, 7\}$.

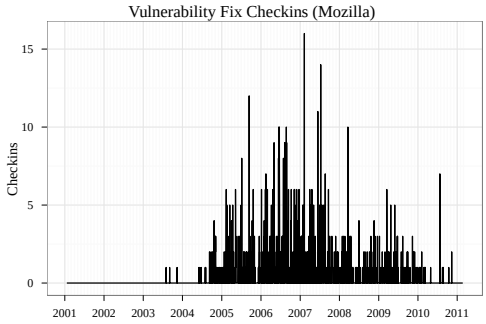

Figure 3: Fixes against time for Mozilla. No trend is directly apparent.

We found 18,803 checkins in the Apache SVN, ranging from November 18, 1996 to March 10, 2011, and 17,688 checkins in the Tomcat SVN, ranging from May 12, 2001 to April 23, 2011.

3 Methodology

3.1 Vulnerability Fixes

Trends in vulnerability fix rates cannot be read directly off the checkins, since it is not a priori clear that every checkin is a fix. However, this is a reasonable assumption because of three observations:

- each such checkin is usually *believed* to be a fix by the developer doing the checkin, even if the fix needs to be modified later;
- checkins that are backports of vulnerability fixes to earlier releases constitute vulnerability fixes in themselves, since the fix will have to be adapted to the peculiarities of the earlier release; and
- the number of checkins for any given vulnerability is small (but see below for a discussion of actual results).

Plotting the number of vulnerability-fixing checkins on any given day against time directly gives only a confused picture. For example, plotting this number for Mozilla gives Figure 3. While it is clear that fix rates have peaked in 2007—during the lifetime of Mozilla 2.0—and have declined since then, some smoothing would give a clearer picture.

Instead of plotting fixes directly against time, we plot the moving average and the moving average fraction. Both move a window of constant size over the data: the moving average computes the sample mean in each window, and the moving average fraction computes the ratio of two moving averages. See Appendix C for a more formal treatment.

We plot moving averages for checkins with a window size of 365 days, therefore this moving average will start one year after the start of the original series and will average values of the previous year. We also plot moving average fractions in order to express vulnerability-fixing checkins as a fraction of all checkins, also with a window size of 365 days.

In order to analyse the distribution of checkins per day, we plot the number of vulnerability-fixing checkins on the x axis and the number of days with this number of checkins on the y axis on a log-log scale. We chose to bin checkins by day because it is a small enough bin size so that important effects would still be present in the data and not lost in larger bins.

3.2 Size of the Security Team

In order to investigate the size of the security team, we look at the number of unique committers in a given time window, in our case with a window size of 365 days. In addition, in order to see how many vulnerabilities a person fixes in a given year, we divide the number of vulnerabilities in the last year by the number of committers.

3.3 Red Queen/Blue King

Johnson et al. have used a "Red Queen" model to explain the power laws they found when they looked at the time between successive insurgent attacks with coalition fatalities in Iraq and Afghanistan: not only did the time between days with alliance fatalities obey a power law, the exponent could also be predicted from the interval between the first two fatal days! [7] If software security were such a Red Queen race, we should see the same power laws, and we should be able to tell who is winning the race by looking at the trend: if the time between successive vulnerability fixes is growing, the defending Blue King wins; otherwise, the attacking Red Queen wins.

They explained this regularity with a dynamic Red Queen model. In Red Queen models in evolutionary biology, an entity needs to adapt continuously in order to survive. In the context of insurgent attacks, Johnson et al. model the race between the Red Queen insurgents and the Blue King alliance with a random walk model, which is then responsible for the observed power laws.

More formally, if vulnerabilities are fixed at times $t_1, \ldots, t_N$, we then look at the sequence $\langle \delta_1, \ldots, \delta_{N-1} \rangle$, where $\delta_k = t_{k+1} - t_k$ for $1 \leq k < N$, and fit this sequence to the model $\log \delta_k = \log a + b \log k$.

4 Findings and Discussion

4.1 Vulnerability Fixes Over Time

Figure 4 (left) shows the moving average of vulnerability-fixing checkins for Mozilla, httpd, and Tomcat. We can see that httpd and Tomcat have had the

same fix rates (within half an order of magnitude, or a factor of about 3.2), but Mozilla's fix rates are about two orders of magnitude higher.

As we suspected already from the raw data in Figure 3, the fix rate for Mozilla peaked in 2007 and then declined. What was not apparent in that figure, however, is how dramatic the decline is. Fix rates decline from about 1.4 per day to under 0.1 per day, which is a stunning 93% reduction. However, it is possible that most vulnerability-fixing checkins now occur in Mercurial repository, and not in the CVS. This is supported by looking at a moving average of all checkins (not shown), where we can see that not just the vulnerability fixes per day have gone down, but also the number of all checkins per day. There is very little activity left in the Mozilla CVS.

Having established that the strong decline in Mozilla is probably an artifact of repository usage, the next interesting feature of that graph is the incline and peak in 2007. The period with the highest vulnerability fix rates in Mozilla corresponds to the lifetime of Firefox 2 (released on 24 October 2006, end of life in December 2008)[6]. We know from other studies [9] that Firefox 2 is unlike other Firefox versions because almost none of its source code was inherited by later versions. Together with the high number of vulnerability fixes during Firefox 2's lifetime, we conclude that something in its architecture must have made it inherently unsafe to use so that it was completely phased out.

The general shape for httpd is similar to that of Mozilla, but the peak is in 2005, not in 2007, and the decline in fix rates is not as strong. In fact, one could argue that the fix rate has been about constant since 2007.

Tomcat is a comparative latecomer, but the fix rate seems to be stable, with an upward trend in 2010. It is apparent that the data is much grainier than the data for either Mozilla or httpd, simply because there are much fewer checkins, so each checkin represents a rather larger jump. As we can see, the fix rates rose until 2009, stayed essentially constant during 2009 and 2010, and took off in 2011.

In Figure 4 (right), we see the moving average fraction of vulnerability-fixing checkins in all checkins. For Mozilla, we can see that vulnerability-fixing came into prominence by 2007, and, after a brief dip in 2009, rose to a new height in 2010, after which it fell again. We attribute the comparatively large proportion of vulnerability fixes in 2010 to a gradual move to the Mercurial repository, because vulnerability patches would have a proportionately greater need of being backported to the CVS than other checkins. For httpd, the long-term trend seems to go down, whereas for Tomcat, it seems to be rising.

There is no evidence in the data that vulnerability fix rates are declining overall, neither over time nor as a fraction of all checkins. The decline in Mozilla is explained by development occurring elsewhere, and the rates for httpd and Tomcat simply show no decline.

[6] See `http://en.wikipedia.org/wiki/Firefox_2`. We would not normally cite a Wikipedia article, but this page seems to be the only publicly available information on this topic.

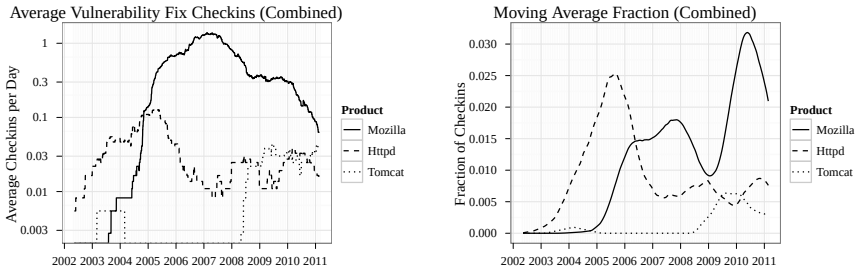

Figure 4: Combined moving average of vulnerability-fixing checkins for all three products, log-linear scale (left), and combined moving average of fraction of vulnerability-fixing checkins in all checkins, for all three products, linear (*not* log-linear!) scale.

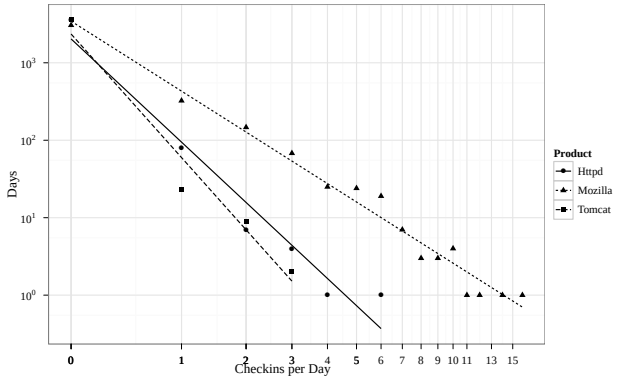

Figure 5: Number of days versus number of checkins, on a log-log scale, together with their respective least-squares regression lines.

4.2 Distribution of Number of Checkins Per Day

Figure 5 shows the distribution of the number of fixes per day on a log-log scale. It is evident that the data points lie very nearly on straight lines. We take this as evidence that all three distributions are heavy-tailed.[7]

If we view the set of vulnerabilities as a reservoir, there are *on periods* of mean length μ_{on}, in which new code is written and new vulnerabilities added

[7]Even though a linear regression on the model $\log \text{days} = \log a + b \log(\text{checkins} + 1)$ gives excellent p- and R^2-values, we cannot infer from this that the distribution obeys a power law. This is because (1) parameter estimates for power law distributions from linear regression is prone to large systematic biases, (2) the data do not span sufficiently many orders of magnitude for a reliable check, and (3) even with much data, power laws are very hard to distinguish from other heavy-tailed distributions such as the log-normal distribution. [5] Fortunately, the precise nature of the distribution is not important for this work, since we are here concerned with an empirical description and not with forecasting. The problems with estimating power laws with linear regression were brought to our attention by one of the anonymous reviewers.

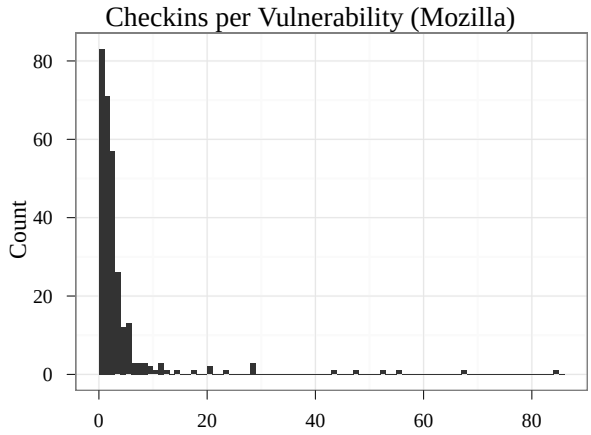

Figure 6: Checkins per Vulnerability for Mozilla.

to the reservoir, and *off periods* of mean length μ_{off}, in which vulnerabilities are fixed and hence removed from the reservoir. If vulnerabilities are added at unit rate during on periods and removed at a rate greater than $\mu_{\text{on}}/(\mu_{\text{on}} + \mu_{\text{off}})$ during off periods, then the number of vulnerabilities V will be heavy tailed with $P(V > x) \sim cx^{-(\alpha-1)}L(x)$, where c and α are constants and L a slowly varying function[8]. If we now also assume that at any day, a constant fraction of these available vulnerabilities will be fixed, it follows that the number of days will also be heavy tailed. (The idea for this model and the notation used is from a survey paper on heavy tail modeling [16, p. 1807]).

We can also explain the different slopes in Figure 5: they are associated with project size. Mozilla is by far the biggest project, containing at the time of writing 95,617 files in a freshly checked out working directory, whereas httpd only has 21,136 (including tests; the pure source is just 9,300 files), and Tomcat 21,726. Larger project size is thus associated with a shallower slope and this is intuitively pleasing, since we would expect that larger projects are also more active, and hence there are bigger chances that there are two or more vulnerability fixes on any given day. But of course, three projects are too few to enable a meaningful statistical analysis of this relationship.

4.3 Number of Checkins Per Vulnerability

Figure 6 shows the number of checkins per vulnerability for Mozilla, whereas we show the much smaller datasets for Httpd and Tomcat in Table 1.

Some Mozilla vulnerabilities took a large number of commits to fix. To take the most extreme example, MFSA 2006-64 ("Crashes with evidence of memory corruption (rv:1.8.0.7)") took 85 (sic!) commits to fix. Looking at that

[8] A real function L is slowly varying if for all real $c > 0$ we have $\lim_{x\to\infty} L(cx)/L(x) = 1$.

	Checkins			
Product	1	2	3	4
Httpd	33	33		
Tomcat	1	14	4	2
Mozilla	83	71	57	26

Table 1: Number of Checkins per Vulnerability (httpd and Tomcat, with the first four Mozilla values for comparison).

MFSA, we find that this vulnerability is host to 29 bug identifiers, and that this MFSA is in fact a blanket vulnerability under which to file any bug report that is concerned with crashing due to memory corruption: there are six bug identifiers associated with "crashes involving tables", one on "heap corruption using XSLTProcessor", four involving the "JavaScript engine", fully 17 because of "crashes involving DHTML", and one more which seems to have been a regression. In other words, *any* memory corruption issue at this time was seen as an instance of this vulnerability.

Other vulnerabilities with large numbers of checkins are a consequence of Mozilla's bug-fixing process. A developer first commits a fix to the trunk of the respective product, and pushes a patch out to a *try server*. On the try servers, a large test suite is run on the patched product. If a test fails and a patch is held responsible for the failure, that patch is undone, and a new fix attempted. Due to the repository organisation, this backout counts as a new commit, and will again carry the bug ID in the commit message.

After having made it through the try server, the patched product is upgraded to mozilla-central (recently renamed to mozilla-incoming), a large repository, where all products are again subjected to tests. Again, if a product fails a test, and if a patch is held responsible, the patch is again undone, and a new solution needs again to be attempted.

It is curious that almost all vulnerabilities in Tomcat need two commits to fix. For example, CVE 2010-1157 is fixed in r936540, which was committed on April 22, 2010 at 00:12:05 +0200, and in r936541, which was committed about 90 seconds later. Analysis reveals however that the both checkins are backports of a fix committed on the trunk (for the then unreleased Tomcat 7). So the large number of vulnerabilities needing two commits is simply because the fixes are committed separately for the different versions.

4.4 Size of Security Team

Figure 7 shows the size of the security team in the last 365-day period (left) and the number of vulnerability fixes per committer in that period (right). It is difficult to tell visually how well the size of the security team (left graph in Figure 7) tracks the number of vulnerability fixes (Figure 4), but the linear correlation coefficients are clear: there is a strong association between the two

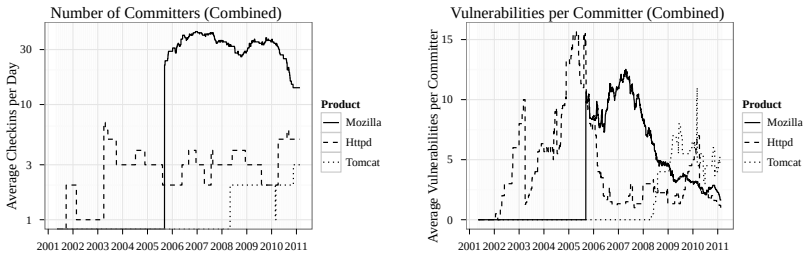

Figure 7: Moving sum of unique vulnerability fix committers on log-linear scale (left), and moving rate of commits per committer on linear scale (right).

quantities for Mozilla and Tomcat ($\rho = 0.77$ and 0.92, respectively), and a very weak one for httpd ($\rho = 0.22$). See also Figure 4.

The findings for Mozilla and Tomcat mean either that there is a staffing process in place that adapts the security team to apparent needs, that there are always enough people there to fix vulnerabilities, or that the original authors of code are responsible for any fixes in that code. The number of vulnerability fixes per committer (Figure 7, right), decreasing for Mozilla, almost constant since 2009 for Tomcat, support the second hypothesis. This is also the reason why the declining rate of commits per committer for Mozilla cannot be used to support increasing marginal cost of fixing vulnerabilities.

The findings for httpd are striking. The lack of correlation between vulnerability fixes and number of people committing such fixes means that there is *no effort to staff the security team proportionally to the needs*, which in turn means that *the amount of work expected of each committer will be proportional to the amount of vulnerabilities discovered*. That is a potentially dangerous situation, since it can lead to overload. On the other hand, the number of checkins per vulnerability is very low, indicating a good quality of vulnerability fix checkins, since they generally do not need to be revised.

The absence of a correlation between vulnerability fixes and fixers could indicate that there are always enough people available to fix vulnerabilities. In that case, it would not make sense to worry about staffing. It could also be that, from a software development perspective, fixing vulnerabilities is no big deal. If fixing a vulnerability is no more difficult than fixing any other bug, then the same people who fix ordinary bugs can also fix vulnerabilities, and it would again make no sense to worry about staffing. Or it could indicate that there is no planning in place to adapt the security-conscious staff to needs. To be fair, httpd is an open-source project, where staffing is likely not to be the result of a cental planning process.

4.5 Red Queen/Blue King

As explained in Section 3.3, in a race between a constantly adapting Red Queen and a defending Blue King, the time between consecutive events (vulnerability-

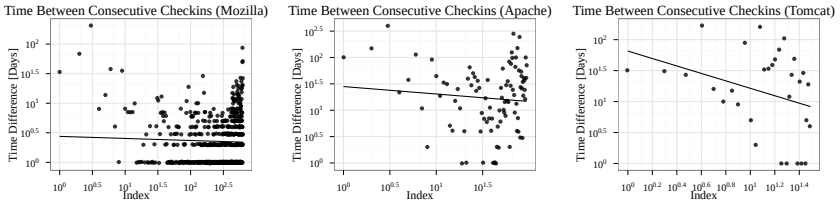

Figure 8: Time between consecutive checkins for Mozilla (left), httpd (middle), and Tomcat (right), on log-log scale with least-squares regression line.

fixing checkins in our case) should obey a power law. When we actually plot the time between consecutive checkins, we see in all cases a downward trend that is, however, far removed from the uncannily precise fits in Johnson et al.'s paper [7]. The p-values (0.37 for Mozilla, 0.38 for httpd, and 0.07 for Tomcat) are as disappointing as the R^2 values ($2.8 \cdot 10^{-4}$ for Mozilla, $2.4 \cdot 10^{-3}$ for httpd, and 0.077 for Tomcat).

With this data and this analysis, we cannot confirm a Red Queen race.

5 Related Work

As far as we know, we are the first to investigate vulnerability fix rates over time; none of the papers below address this problem. The closest work is Ozment and Schechter's study of vulnerabilities in OpenBSD [12]. Their aim, like ours, is to find out whether software security gets better as the software ages (wine) or worse (milk). In order to analyse this, Ozment and Schechter look at the number of *foundational vulnerabilities*, that is, the vulnerabilities that have been in the software from the first release. They conclude that software security in OpenBSD is indeed getting better with age in the sense that fewer and fewer reported vulnerabilities are foundational. Similarly, Massacci et al. presented a study on Firefox evolution, and looked especially at the question of whether vulnerabilities were inherited or foundational [9], but did not compare Firefox with other software.

In our study, we have a different viewpoint: we look at whether a constant effort in fixing vulnerabilities is bringing diminishing returns. In doing this, we not only consider foundational vulnerabilities (which by definition can only become fewer), but also vulnerabilities that are introduced as the software evolves.

In the area of software engineering in general, one of the earliest attempts to show bug fixes—not necessarily vulnerabilities—is in software visualisation work; see, for example, work by Baker [1] or by Ball and Eick [2]. Such visualisations can display the fix-on-fix rate (the rate at which fixes need to be fixed), but not the fix rates themselves.

There is much research on bug introduction and fix rates, but that research does in general not look at how these rates change over time. For example,

Phipps analysed bug and productivity rates for Java and C++ [13], and Kim et al. [8] worked on bug introduction rates per author, based on earlier work by Śliwerski et al. identifying bug-introducing changes [20].

Rescorla modeled the vulnerability lifecycle in order to determine if looking for vulnerabilities was a good idea [15], whereas Massacci et al. looked at Firefox vulnerabilities, but with the intent of determining which source of vulnerability information would enable one to answer which type of research question [10].

Neuhaus et al. studied Mozilla vulnerabilities in order to predict so far unknown vulnerabilities [11] and Schryen studies many OSS projects in order to find out whether open-source security is a myth [19].

Heavy tailed distributions have been extensively studied for network structures, the earliest example being Price's 1965 discovery that networks of scientific citations obey a power law [6]. A more mathematical treatment of heavy tail modeling is given in the survey article by Resnick [16].

6 Threats to Validity

Study Size. We are looking at only three software projects. It is possible that looking at more (and more diverse) projects would give different results. However, all such studies would be restricted to open source software, and in that area, the chosen projects are representative in terms of market share and attack surface.

Biases in Data Selection. Bird et al. have described the bias that is present in bug-fix datasets, and the consequent bias that results when these datasets are used for prediction [3]. It is certainly true that we have not been able to map all vulnerabilities to their fixing checkins. However, since we are here only concerned with an empirical description, not with prediction, we argue that the data we have is for the most part representative.

Unknown Noise in Data. From previous studies we know that vulnerability information can be noisy, and that the noise is often not even quantifiable. This is because we lack information about the exact process by which vulnerability information is acquired and published. In our case, this information comes directly from the vendor, and since we are only concerned with the time when a vulnerability has been *fixed* (as opposed to when it has been *discovered*), and since we get this information directly from the source code repository, we are confident that our results are robust.

Confusing Data. The picture emerging from analysing these three software systems is far from unified; instead, each system has its own idiosyncrasies. This may create a confusing data set where inter-system comparisons may be difficult. Still, we believe that the phenomena we are investigating should be robust with respect to slight differences in the precise meanings of words like 'vulnerability' or 'fix'.

7 Conclusion and Future Work

In this work, we gave evidence that vulnerability fixing is not an activity that can be modeled by an increasing concave function, as we would have in a standard cybersecurity economic model. Instead, the situation is different for each of the three projects: for Mozilla, the vulnerability fix rate decreases as predicted (but there has been a switchover to another repository, so this decrease is suspect, since the decrease is abrupt and is concurrent with the switchover), for httpd it stays constant, and for Tomcat it increases. Also, vulnerability fixing is not a Red Queen race since we did not find any evidence of the resulting power laws. Whatever the reasons for these findings, they are evidence that standard cybersecurity economic models are not easily applied to software security.

We unexpectedly found evidence that the distribution of the number of fixes per day is heavy tailed. While we can at this point only speculate about the causes for this relationship, as far as we know, we are the first to notice this.

Looking at the number of people fixing vulnerabilities, we found that the pool of people available for fixing vulnerabilities changes proportionally with the demand for Mozilla and Tomcat, but not for httpd. Generally, vulnerability fixes have good quality, because they do not need to be revised.

In future work, we will:

- investigate probable models for the heavy tailed distributions exhibited in Figure 5;
- find models that predict fix rates for the three projects; and
- explore the Red Queen races in order to see if perhaps the predicted relationship holds in subsets of the data.

8 Acknowledgments

We thank Sandy Clark, Jonathan M. Smith and Matt Blaze for constructive discussions and for finding reference [7]; Brian Trammell for suggesting the title of this paper; the Tomcat security team for answering our questions; Christian Holler for information about the Mozilla development process; Dominik Schatzmann for excellent suggestions on early drafts of the paper; and the anonymous reviewers for raising many excellent points and making helpful suggestions.

References

[1] Marla J. Baker and Stephen G. Eick. Visualizing software systems. In *Proc. 16th International Conference on Software Engineering*, ICSE '94, pages 59–67, 1994.

[2] T. Ball and S.G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[3] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proc. ESEC/FSE '09*, pages 121–130, 2009.

[4] Lewis Carroll. *Through the Looking-Glass*. Macmillan and Co., 1871.

[5] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51:661–703, 2009. DOI 10.1137/070710111.

[6] Derek J. de Solla Price. Networks of scientific papers. *Science*, 149(3683):510–515, 1965.

[7] Neil Johnson, Spencer Carran, Joel Botner, Kyle Fontaine, Nathan Laxague, Philip Nuetzel, Jessica Turnley, and Brian Tivnan. Pattern in escalations in insurgent and terrorist activity. *Science*, 333(6038):81–84, July 2011.

[8] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr. Automatic identification of bug introducing changes. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, September 2006.

[9] Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. After-life vulnerabilities: A study on firefox evolution, its vulnerabilities, and fixes. In *Proc. ESSoS 2011*, volume 6542 of *Lecture Notes in Computer Science*, pages 195–208, 2011.

[10] Fabio Massacci and Viet Hung Nguyen. Which is the right source for vulnerability studies?: an empirical analysis on Mozilla Firefox. In *Proc. 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, pages 4:1–4:8, 2010.

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

[12] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Usenix Security Symposium*, pages 93–104, August 2006.

[13] Geoffrey Phipps. Comparing observed bug and productivity rates for Java and C++. *Software Practice and Experience*, 29:345–358, 1999.

[14] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[15] Eric Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[16] Sidney I. Resnick. Heavy tail modeling and teletraffic data. *The Annals of Statistics*, 25(8):1805–1869, 1997.

[17] Rachel Rue and Shari Lawrence Pfleeger. Making the best use of cybersecurity economic models. *IEEE Security & Privacy*, 7:52–60, 2009.

[18] Jeffrey A. Ryan and Josh M. Ulrich. *xts: Extensible Time Series*, 2011. R package version 0.8-0.

[19] Guido Schryen. Is open source security a myth? what does vulnerability and patch data say? *CACM*, 54(5):130–140, May 2011.

[20] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. Second International Workshop on Mining Software Repositories*, pages 24–28, May 2005.

[21] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

[22] Achim Zeileis and Gabor Grothendieck. zoo: S3 infrastructure for regular and irregular time series. *J. Statistical Software*, 14(6):1–27, 2005.

[23] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

A Replication Guide

In this Appendix, we describe how to use the scripts we used to create the figures in this paper. This allows easy checking and replication of our results.

Prerequisites are: the R program [14] and the R packages ggplot2 [21], xts [18] and zoo [22], a MySQL database server and client programs, a Bourne shell-compatible shell, a Perl 5 interpreter, a GNU-compatible make program,[9] a sed interpreter, and a "touch" utility. Any Linux or Mac system will do, as will any recent Cygnus system under Windows.

1. Obtain, then unpack *fixrates.tar.gz* from from `ftp://ftp.tik.ee.ethz.ch/pub/publications/WEIS2012/fixrates.tar.gz`. This will cause directories *src/* and *data/* to be created.

2. Create a MySQL user 'root' with password 'root', and privileges to create a database called 'vulnerabilities' and all privileges in that database. If you do not wish to create such a user, you need to modify *src/Makefile* and substitute different user names and passwords.

[9] Will probably run with other makes too.

3. Create a directory *paper/figures/*. The plots will appear in this directory. Also create a directory *out/*.

4. Adapt *src/Makefile* to the correct way of accessing the MySQL database (macros `MYSQL` and `MYSQLD`).

5. Change your working directory to *src* and type 'make'. This will cause the database to be populated, the relevant time series to be extracted, and the figures to be plotted.

B Reconstructing Checkins from CVS Logs

For our purposes, a CVS log entry e contains the file name $e.f$ of the file being checked in; the time $e.t$ of the checkin; the identifier $e.c$ of the person doing the checkin; and a message $e.m$ describing the changes.

A CVS *commit* can be viewed as a collection of CVE log entries that were created because some user typed "`cvs commit`". These entries will have the same c and m values, and will have related t values, since the changed files will be checked in close together. In other words, a CVS commit consists of all the CVS log entries that have the same committer and message, and were committed not too far apart. This gives rise to Algorithm 1[10]

This algorithm first sorts the CVS entries into checkin time order. Then it goes over the list elements one by one. The elements seen in the outer loop (lines 3–15) start new transactions. In the inner loop (lines 5–12), we look for log entries that have matching committers and messages, until we have exhausted our sliding window δ. Matching CVS entries are removed from the list so that they cannot be mistaken for new transactions later.

If there are n CVS log entries, line 6 of Algorithm 1 will be executed at most $n(n-1)/2$ times. To see this, consider the case where each of the n log entries comes from a different `cvs checkin` operation and that all checkins have been done within δ time. Then the first outer loop will consider the first log entry, then the inner loop will look at all the other $n-1$ entries in a vain search for a match. The next outer iteration will look at the second log entry and the inner loop will look at the remaining $n-2$ elements for a match, and so on. In all, the outer loop will be executed n times, and the inner loop will be executed $n-k$ times on outer iteration k, where $1 \leq k \leq n$. In total, there will therefore have been $\sum_{k=1}^{n} n-k = n(n-1)/2$ loop steps, which is a lower bound on the number of times that line 6 is executed.

At the same time, no arrangement of log file entries will cause line 6 to be executed more often. This is because the outer loop can take at most n iterations, since it will at most iterate over the whole list. Also, on iteration k, the inner loop can at most take $n-k$ iterations, since there are at most $n-k$ log entries left to examine. This is therefore an upper bound on the number

[10]This algorithm is described in passing in work by Zimmermann et al. [23], but we are not aware of any formal statement of the algorithm or an analysis of its running time.

Algorithm 1 CVS Commits from CVS Log

This algorithm computes a set of CVS commits from a CVS log file. The log file is given as a linked list of CVS log entries (supporting "head" and "next" operations). The CVS transactions are returned as a list of sets of CVS log entries. Every list member represents one CVS commit.
This algorithm also uses a parameter δ that says how far apart two log entries must be in order to belong to different transactions, even when they have the same message and committer.

Require: A CVS log L, and a threshold $\delta > 0$.
Ensure: A list C of CVS transactions, sorted by checkin time.

```
 1: Sort L by time and assign the result to S
 2: C ← ⟨⟩; a ← S.head
 3: while a ≠ Λ do
 4:    c ← {a}; t ← a.t; n ← a.next
 5:    while n ≠ Λ and n.t − t < δ do
 6:       if n.c = a.c and n.m = a.m then
 7:          c ← c ∪ {n}; t ← n.t; p ← n; n ← n.next
 8:          Remove p from S
 9:       else
10:          n ← n.next
11:       end if
12:    end while
13:    Add c to the end of C
14:    a ← a.next
15: end while
```

of times that line 6 is executed. Since both lower and upper limits are equal, Algorithm 1's worst-case running time is exactly $n(n-1)/2$ steps.

On average, however, this algorithm will perform much better because unrelated checkins will tend not to happen close together, and checkins will usually commit their files very quickly. A time-sorted CVS log will therefore consist mostly of clusters of related entries (belonging to the same checkin), separated by relatively large time spans. In this case, Algorithm 1's running time will be proportional to n, since every log entry is touched exactly once.

C Moving Averages and Fractions

Let $X = \langle x_1, \ldots, x_N \rangle$ be a sequence of numbers. Then the *moving average with a window size of* n (where $n \leq N$) is defined as the sequence $\mu(X, n) = \langle \mu(X, n, n), \ldots \mu(X, n, N) \rangle$, where

$$\mu(X, n, j) = \frac{1}{n} \sum_{k=0}^{n-1} x_{j-k} \quad \text{for } n \leq j \leq N.$$

The moving average will act as a low-pass filter because strong zig-zagging movements will disappear in the average.

Let $Y = \langle y_1, \ldots, y_N \rangle$ be a sequence of numbers such that $0 \leq x_j \leq y_j$ for $1 \leq j \leq N$. In our example, y_j is the total number of checkins on day j. Then we define $\phi(X, Y, n) = \langle \phi_n, \ldots, \phi_N \rangle$, the *moving-average fraction of X and Y with a window size of n* as

$$\phi_j = \begin{cases} \mu(X, n, j)/\mu(Y, n, j) & \text{if } \mu(Y, n, j) \neq 0, \\ 0 & \text{otherwise} \end{cases}$$

for $n \leq j \leq N$.