A Quantitative Analysis into the Economics of Correcting Software Bugs

Conference Paper · January 2011

DOI: 10.1007/978-3-642-21323-6_25 · Source: DBLP

2 authors:

Craig Wright
Association for Computing Machinery
27 PUBLICATIONS **253** CITATIONS

SEE PROFILE

Tanveer Zia
Charles Sturt University
123 PUBLICATIONS **1,253** CITATIONS

SEE PROFILE

A Quantitative Analysis into the Economics of Correcting Software Bugs

Craig S Wright[1] and Tanveer A Zia[2]

School of Computing and Mathematics
Charles Sturt University, NSW 2678
[1]cwrigh20@postoffice.csu.edu.au, [2]tzia@csu.edu.au

Abstract. Using a quantitative study of in-house coding practices, we demonstrate the notion that programming needs to move from "Lines of Code per day" as a productivity measure to a measure that takes debugging and documentation into account. This could be something such as "Lines of clean, simple, correct, well-documented code per day", but with bugs propagating into the 6th iteration of patches, a new paradigm needs to be developed. Finding flaws in software, whether these have a security related cost or not, is an essential component of software development. When these bugs result in security vulnerabilities, the importance of testing becomes even more critical. Many studies have been conducted using the practices of large software vendors as a basis, but few studies have looked at in-house development practices. This paper uses an empirical study of in-house software coding practices in Australian companies to both demonstrate that there is an economic limit to how far testing should proceed as well as noting the deficiencies in the existing approaches.

Keywords: Software Development Life Cycle, Model Checking, Software Verification, Empirical studies

1 Introduction

The deficiency of published quantitative data on software development and systems design has been a major ground for software engineering's failure to ascertain a proper scientific foundation. Past studies into coding practice have focused on software vendors. These developers have many distinctions from in-house projects that are not incorporated into the practices and do not align well with in-house corporate code development. In the past, building software was the only option but as the industry developed, the build vs. buy argument has swung back towards in-house development with the uptake of Internet connected systems. In general, this has been targeted towards specialized web databases and online systems with office systems and mainstream commercial applications becoming a 'buy' decision.

As companies move more and more to using the web and as 'cloud applications' become accepted, in-house development is becoming more common. This paper uses an empirical study of in-house software coding practices in Australian companies to

both demonstrate that there is an economic limit to how far testing should proceed as well as noting the deficiencies in the existing approaches.

1.1 Related Work

Other studies of coding processes and reliability have been conducted over the last few decades. The majority of these have been based either on studies of large systems [3, 8] and mainframe based operations [8] or have analyzed software vendors [7]. In the few cases where coding practices within individual organization have been quantitatively analyzed, the organizations have been nearly always large telecommunications firms [1, 2, 5, 6, 8] or have focused on SCADA and other critical system providers [9] or are non-quantitative approaches [12, 13].

Whilst these results are extremely valuable, they fail to reflect the state of affairs within the vast majority of organizations. With far more small to medium businesses coupled with comparatively few large organizations with highly focused and dedicated large scale development teams (as can be found in any software vendor), an analysis of in-house practice is critical to both security and the economics of in-house coding.

As the Internet comes to become all persuasive, internal coding functions are only likely to become more prevalent and hence more crucial to the security of the organization.

1.2 Our contribution

In section 2 we present an analysis of the empirical study completed to determine the cost of finding, testing and fixing software bugs. We model the discovery of bugs or vulnerabilities in Section 3 using Cobb-Douglas function and calculate the defect rate per SLOC (source line of codes) using Bayesian calculations. Finally paper is summarized and concluded in Section 4.

2 An analysis of coding practice

A series of 277 coding projects in 15 companies with in-house developers was analyzed over multiple years. The costs, both in terms of time and as a function of financial expenditure were recorded. The analysis recorded: format string errors, integer overflows, buffer overruns, SQL injection, cross-site scripting, race conditions, and command injection. The code samples were analyzed by the authors using a combination of static tools and manual verification to the OWASP[1] and SANS[2] secure coding guidelines during both the development and maintenance phases. For the 277 coding projects, the following data fields have been collected:

[1] http://www.owasp.org/index.php/Secure_Coding_Principles
[2] http://www.sans-ssi.org/

- the total number of hours
    - Coding / Debugging (each recorded separately)
- tloc (thousand lines of source code)
- the number of bugs (both initially and over time as patches are released)

The coding projects where developed using a combination of the Java, C# (.Net), PhP and C++ languages. The authors collected data between June 2008 and December 2010 during a series of Audits of both code security and system security code associated with [14]. The code projects came from a combination of financial services companies, media companies and web development firms. The data will be released online by the authors.

It is clear from the results that there is an optimized ideal for software testing. Fig. 1 demonstrates the costs of testing and notes how each subsequent bug costs more to find than the previous one. The costs of finding bugs go up as the cost of software is tested to remove more bugs.

It has been noted that "*there is a clear intuitive basis for believing that complex programs have more faults in them than simple programs*" [9]. As the size and hence complexity of the code samples increased, the amount of time and costs required to write and debug the code increased (Fig. 2). What was unexpected was that the number of bugs/LOC did not significantly change as the size of the program increased (Fig. 3). Rather, there was a slight, but statistically insignificant decline in the number of bugs noted in more complex programs per line of code. So whilst the number of bugs did increase, this occurred in a linear fashion to the cost increase which occurred exponentially.

The calculated number of hours per line of code (Fig. 2) increased exponentially with an exponent of around 1.56. In this study, the largest program sampled was approximately 300,000 SLOC (source lines of code). This relates directly to complexity with longer programs costing more both in time and money.

A financial calculation of internal costs of mitigating the bugs was also conducted based on the bugs found within a year of the code being released. This period was selected as it comes to reason that if the bug has not been found in a 12 month period, it would be expensive to find.

The results of the analysis of the data demonstrated that the costs of testing can be analyzed. In Fig. 1, the cost (calculated as a function of analysis time) of finding each additional bug is exponentially more expensive than the last to find. As a consequence, this also increases the mean cost of the project. The more bugs are sought, the higher the cost. This is offset against the costs of fixing bugs in the field later in the paper.

Of particular interest is the distribution of bugs as a percentage of code. We can see that there is no strong correlation between the levels of bugs in code and the length of the code ($R^2 = -14.48$). There are more bugs in large code, but the number of bugs per line does not increase greatly.

The distribution of bugs was fairly even for all sizes of code in the study and was positively skewed. The numbers of bugs discovered was far too high. The reported numbers of bugs in large commercial software releases average around 4% [9]. The mean (Fig. 4) for the study was 5.529% (or 55 errors per 1,000 lines of source code). Many of these are functional and did not pose security threats, but the economics of repairing the remaining bugs remains.

We need to move from "*Lines of Code per day*" as a productivity measure to a measure that takes debugging and documentation into account. This could be something such as "*Lines of clean, simple, correct, well-documented code per day*" [3]. This also has problems, but it does go a long way towards creating a measure that incorporates the true costs of coding.

The primary issue comes from an argument to parsimony. The coder who can create a small, fast and effective code sample in 200 lines where another programmer would require 2,000 may have created a more productive function. The smaller number of lines requires less upkeep and can be verified far easier than the larger counterpart.

Software maintenance introduces more bugs. Through an analysis of the debugging progresses and the program fixes, it was clear that systems deteriorate over time. What we see is that the first iteration of bug fixes leads to a second and subsequent series of fixes. In each set of fixes, there is a 20-50% (mean of 34% $\pm$ 8%[3]) of the fix creating another round of bugs. This drops on the second round of fixes, but starts to rise on the 3rd and subsequent rounds. In a smaller set of code, the low overall volume of bugs limits the number of iterations, but the larger code samples led to up to 6 iterations. This would be expected to be even larger on extremely large programs (such as an Operating System).

The ratio of software bugs in the patching process was less than that of the initial release, but over the life of a program that has been maintained for several years, the total number of bugs introduced through patches can be larger than that of the initial release.

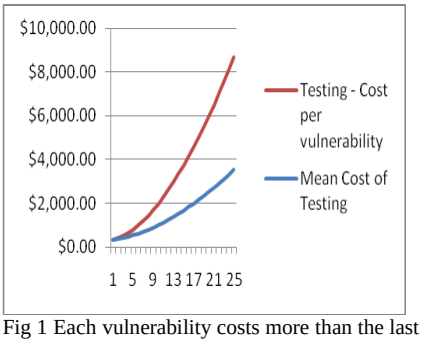

Fig 1 Each vulnerability costs more than the last to mitigate

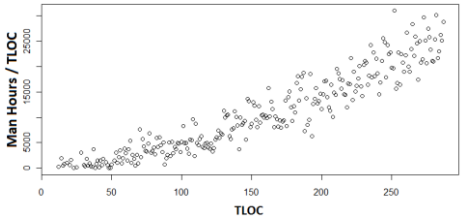

Fig. 2 Program size against Coding time

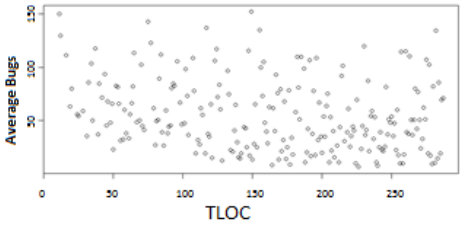

Fig. 3 Program size against Bugs

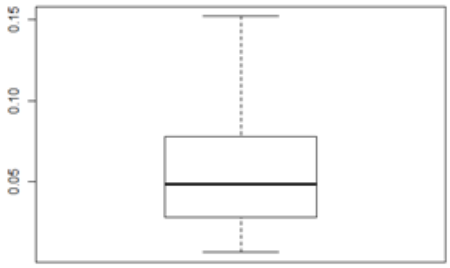

Fig. 4 Box plot of the distribution of Bugs/TLOC

[3] 95% Confidence Interval or $\alpha = 5\%$

3 Vulnerability Modeling

Vulnerability rates can be modeled extremely accurately for major products. Those with an extremely small user base can also be modeled, but the results will fluctuate due to large confidence intervals. What most people miss is that the number of vulnerabilities or bugs in software is fixed at release. Once the software has been created, the number of bugs is a set value. What varies stochastically is the number of bugs discovered at any time.

This is also simple to model, the variance being based on the number of users (both benign and malicious) of the software. As this value tends to infinity (a large user-base), the addition of any further users makes only a marginal variation in the function. Small user-bases of course have large variations as more people pay attention (such as the release of software vulnerability).

This is a Cobb-Douglass function [10] with the number of users and the rate of decay as variables. For largely deployed software (such as Microsoft's Office suite or the Mozilla browser), the function of the number of vulnerabilities for a program given the size of the program can be approximated as a Poisson decay function.

3.1 Modeling the discovery of bugs/vulnerabilities in software

The discovery of software bugs can be mapped to the amount of time that has been used in both actively examining the product as well as the passive search for bugs (using the software).

The study found that a Cobb Douglass function with $\alpha=1.6$ and $F(x)= c \times TLOC + \varepsilon$ where c and C are constant values with the function $G(x)^{\beta}$ is constant for a given number of users or installations and expresses the rate at which users report bugs. This equation increases to a set limit as the number of users increase. In the case of widely deployed software installations (such as Microsoft Word or Adobe Acrobat) and highly frequented Internet sites, this value tends towards $G(x)=1$.

3.2 Equations for Bug Discovery

For a static software system under uniform usage the rate of change in, N, the number of defects discovered is directly proportional to the number of defects in the system,

$$\frac{d}{dt}N(t) = \alpha N(t) \qquad (1).$$

A Static system is defined as one that experiences no new development, only defect repair. Likewise, uniform usage is based on same number of runs/unit time. As the user-base of the product tends to infinity, this becomes a better assumption.

If we set time T to be any reference epoch, then N satisfies

$$N(t) = N(t)e^{-\alpha(t-T)} \tag{2}$$

This means we can observe the accumulated number of defects at time t, A(t), where

$$A(t) = N(t)\left(1-e^{-\alpha(t-T)}\right) \tag{3}$$

With continuous development, an added function to model the ongoing addition of code is also required. Each instantaneous additional code segment (patch fix or feature) can be modeled in a similar manner.

What we do not have is the decay rate and we need to be able to calculate this. For software with a large user-base that has been running for a sufficient epoch of time, this is simple.

This problem is the same as having a jar with an unknown but set number of red and white balls. If we have a selection of balls that have been drawn, we can estimate the ratio of red and white balls in the jar.

Likewise, if we have two jars with approximately the same number of balls in approximately the same ratio, and we add balls from the second jar to the first periodically, we have a most mathematically complex and difficult problem, but one that has a solution.

This reflects the updating of existing software. In addition, with knowledge if the defect rates as bugs are patched (that is the rate of errors for each patch), we can calculate the expected numbers of bugs over the software lifecycle. In each case, the number of bugs from each iteration of patching added *34% ± 8%* more bugs than the last iteration.

$$\begin{aligned} A(t) &= \sum_{i=0}^{k} A_i(t) = A_0(t) + A_1(t) + \ldots + A_k(t) \\ &\approx A_0(t) + \beta A_0(t) + \beta^2 A_0(t) + \ldots + \beta^k A_0(t) \qquad \beta \le 1 \end{aligned} \tag{5}$$

In the study, this would come to

$$A(t) = A_0(t)\sum_{i=0}^{6}(0.34)^i = 1.514A_0(t) \tag{6}.$$

So over the life of the software, there are 1.51 times the original number of bugs that are introduced through patching.

Where we have a new software product, we have prior information. We can calculate the defect rate per SLOC, the rate for other products from the team, the size of the software (in SLOC) etc. This information becomes the posterior distribution. This is where Bayesian calculations [11] are used.

t = time

λ_B = (Mean) Number of Bugs / TLOC (Thousand Lines of Code)
L = SLOC (Source Lines of Code)

So, more generally, if a software release has L lines of code and the expected number of lines of code per defect is λ_B, then the a priori distribution of defects in the release is a Poisson P_β distribution where β is the ratio of new lines of code to average number of lines/bug (L/λ_B)

$$P_\beta(n_{defects}) = \frac{\beta^n e^{-\beta}}{n!} \tag{7}$$

The conditional distribution for the number of defects in a software release given a defect discovery T units of time since the last discovery is

$$P_\beta(n\_defects) = \frac{\beta^n}{n!} e^{-\beta} \tag{8}.$$

Suppose the defect discovery (decay) constant is α and β is the a priori expected number of defects (code size/lines of code per defect). If we observe defects at time intervals of T_1, T_2, ..., T_k, then the conditional distribution of remaining defects is Poisson:

$$P_{(\beta e^{-\alpha(T_1+T_2+\ldots+T_k)})}\left(n_{defects}\right) = \frac{(\beta e^{-\alpha(T_1+T_2+\ldots+T_k)})^n}{n!} e^{-\beta e^{-\alpha(T_1+T_2+\ldots+T_k)}} \tag{9}$$

This is the a priori expected number of defects scaled by the decay factor of the exponential discovery model.

As new releases to the software are made, the distribution of defects remains Poisson with the expected number of defects being the number remaining from the last release, γ plus those introduced, β, by the independent introduction of new functionality.

$$P_{(\beta+\gamma)}[n] = \frac{e^{-(\beta+\gamma)}(\beta+\gamma)^n}{n!} \tag{10}.$$

It is thus possible to observe the time that elapses since the last discovery of a vulnerability. This value is dependent upon the number of vulnerabilities in the system and the number of users of the software. The more vulnerabilities, the faster the discovery rate of flaws. Likewise, the more users of the software, the faster the existing vulnerabilities are found (through both formal and adverse discovery).

4 Conclusion

To represent the effect of security expenditure in minimizing bugs against investment over time and the result as expected returns (or profit) we see that there are expenditure inflection points. What we see is that spending too much on security has a limiting function on profit. Also too little expenditure has a negative effect on profit

as the cost of discovering bugs post release increases. This is where risk analysis comes into its own. The idea is to choose an optimal expenditure on security that limits the losses. Money should be spent on security until that last dollar returns at least a dollar in mitigated expected loss. Once the expenditure of a dollar returns less than a dollar, the incremental investment is wasted. Here, the software coder has to optimize the testing process.

Modeling and understanding program risks is essential if we are to minimize risk and create better code. It was clear from this study that organizational coding expresses a far higher rate of bugs per line of code than is expressed in specialized software companies. Insufficient testing is being conducted in many companies who have in-house coding teams. This is leading to higher costs and lower overall security.

The goal for any coding team should be how many lines of good code are produced, not how many lines of code are written and then sent to be fixed.

5 References

[1] Anderson., R.: Why information security is hard, an economic perspective. In 17th Annual Computer Security Applications Conf. Dec 2001 New Orleans LA. (2001)

[2] Carman, D. W., Dolinsky, A. A., Lyu, M.R. & Yu, J.S: Software Reliability Engineering Study of a Large-Scale Telecommunications System. Proc. Sixth Int'l Symp. Software Reliability Eng., pp. 350-359 (1995)

[3] Connell, C: It's Not About Lines of Code. http://www.developer.com/java/other/article.php/988641 (Viewed 15 Mar 2010)

[4] Daskalantonakis, M. K: A Practical View of Software Measurement and Implementation Experiences within Motorola. IEEE Trans. Software Eng., 18(11), 998-1,010, (1992)

[5] Kaaniche, K. & Kanoun, K: Reliability of a Telecommunications System. Proc. Seventh Int'l Symp. Software Reliability Eng., pp. 207-212, (1996)

[6] Khoshgoftaar, T.M., Allen, E.B., Kalaichelvan, K.S. & Goel, N: Early Quality Prediction: A Case Study in Telecommunications. IEEE Trans. Software Eng., 13(1), 65-71 (1996)

[7] Levendel, Y: Reliability Analysis of Large Software Systems: Defects Data Modeling. IEEE Trans. Software Eng., 16(2), 141-52 (1990)

[8] Mills, H. D: Top-down programming in large systems. Debugging techniques in large systems, R. Rustin Ed., Englewoods Cliffs, N.J. Prentice-Hall (1971)

[9] Munson J.C.,, & Khoshgoftaar, T.M: The Detection of Fault-Prone Programs, IEEE Transactions on Software Engineering, 18(5), 423-433, (1992)

[10] Cobb, C. W. and P. H. Douglas: A theory of production. American Economic Review 18(1):139-165. Supplement, Papers and Proceedings of the Fortieth Annual Meeting of the American Economic Association (1928)

[11] Bayes, Thomas: An essay towards solving a problem in the doctrine of chances." Philosophical Transactions of the Royal Society. 53: 370-418 (1763)

[12] Bacon, D. F., Chen, Y., Parkes, D., & Rao, M. A market-based approach to software evolution. Paper presented at the Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. (2009)

[13] Sestoft, Peter, Systematic software testing IT University of Copenhagen, Denmark1 Version 2, 2008-02-25. (2008)

[14] Wright, Craig S. The not so Mythical IDS Man-Month: Or Brooks and the rule of information security, ISSRE (2010)