Does open-source software development tend to result in products that are of higher quality than their closed-source counterparts in terms of security, functionality, and ease-of-use? 

**INTRODUCTION**

Traditionally, most software on the market has been closed-source, meaning users pay for licenses but do not gain access to its source code. A growing number of programs, however, are open-source, in which the general public has access to the source code. The release of the source code enables users to find and fix bugs, add new features, and change it so that it works with other devices or systems.

Supporters of open-source development claim that the release of the source code results in better software. Comparing closed-source software is often constrained by the number of programmers they can hire. The creators of open-source programs, by contrast, are not limited in numbers by the number of users they have, and the expertise of paid users. In short, Raymond (2001, 19), one of the main advocates for open-source development, states that “given enough eyeballs, all bugs are shallow.” He suggests that there are two categories of open-source software development: the “cathedral” style in which one small group is responsible for updates and releases (much like a typical closed-source project), and the “bazaar” style in which anyone can contribute improvements (Raymond 2001). Those who oppose open-source development often claim that the lack of centralized management is often characteristic of open-source projects (especially those that follow Raymond’s “bazaar” style) and that the fact that most of the programmers are volunteers hinders the development process. Supporters of open-source development contend that such transparency ensures that the code is reviewed by many people around the world, eliminating the need for project management (Raghunathan et al. 2005, 903-918).

**LITERATURE REVIEW**

Paulson et al. (2004, 246-256) tested three different pieces of open-source software (the Apache web server, the Linux kernel, and the GCC compiler) and three pieces of closed-source software (which could not be named due to nondisclosure agreements). They determined that open-source development tends to foster creativity, as evidenced by the fact that many of these projects exhibited 26 times the growth in the number of functions added over time than the proprietary products included in the study. Over time, the rate of modification of functions in the open-source software decreased more than that of the closed-source software. This validates O’Reilly’s claim that open-source software becomes more stable than closed-source software (Paulson et al. 2004, 246-256; O’Reilly 1999, 32-37).

Contrary to Raymond’s findings, Paulson et al. (2004, 246-256) showed that the open-source and closed-source software had comparable growth percentages in the number of functions and lines of code. The overall complexity and average complexity of the open-source software exceeded that of the proprietary products. Unlike the closed-source products studied, the open-source projects exhibited a direct correlation between the number of functions added and the number of functions modified. This disproves O’Reilly’s hypothesis that open-source software is more modular (Paulson et al. 2004, 246-256; O’Reilly 1999, 32-37). In their conclusion, Paulson et al. (2004, 246-256) suggested that open-source developers should take steps to prevent the tight coupling encountered in the study and institute limits on function complexity. It should be noted, however, that the closed-source software studied by Paulson et al. (2004, 246-256) was intended for “real-time embedded systems,” while the open-source software tested fell into the realm of “applications and systems software.” Software developed for real-time embedded systems must be simpler because these devices have less memory, storage space, and processing power than devices such as laptops, desktops, and servers. They noted that complexity is influenced by factors such as “programming style” and “organizational procedures,” which may vary across domains (Paulson et al. 2004, 246-256). In addition, they did not examine the factors that could have led to the comparable growth percentages, such as the programming language, development environment, or project management system used.

Schryen (2011, 130-140) examined 17 different, widely used open-source and closed-source pieces of software. Some of the open-source software used, such as Thunderbird, Apache, OpenOffice, PostgreSQL, and Red Hat Enterprise Linux, were developed using Raymond’s “cathedrals”-style. Debian Linux, MySQL, and Firefox, the remaining open-source programs studied, were developed under Raymond’s “bazaars” style. The closed-source packages in the study included Windows 2000, Windows XP, Microsoft Outlook Express, Internet Explorer, IBM’s DB2, Oracle’s 10g, and Microsoft Internet Information Systems (IIS) (Schryen 2011, 130-140). He used the National Institute of Science and Technology’s National Vulnerability Database (NVD) to compute the average severity of vulnerabilities, the “mean time between vulnerability disclosures,” and the “development of vulnerability disclosures over time” (Schryen 2011, 130-140). After analyzing this data, Schryen concluded that how vulnerabilities are patched depends on the vendor, not on the software development method. However, the MITRE CVE and NVD databases do not necessarily contain each and every bug discovered, or the actual dates of discovery. Both “black hat” (evil) and “white hat” (good) hackers struggle with the question of how and when to disclose newly discovered vulnerabilities. Black hats will often refuse to release their discoveries to increase their reputation or will only tell a few people who will then use the vulnerabilities to attack users. White hats will often hesitate to release their discoveries, because vulnerabilities can help whiten hats and black hats alike. The NVD does not necessarily contain each and every vulnerability with a CVE identifier (Schryen 2011, 130-140). Raghunathan et al. (2005, 903-918) created a mathematical model to study the quality of open-source and closed-source software. Their model developed new market conditions. According to the model, the quality of both types of software increases in the face of competition, but the quality of the open-source software is independent from that of the closed-source software. Closed-source vendors can adjust the price so that their customer bases consist entirely of those who are more willing to pay for higher quality. If the quality or cost of closed-source software decreases under competition, the quality of the open-source software will increase (Raghunathan et al. 2005, 903-918).

The mathematical model presented by Raghunathan et al. (2005, 903-918) assumes, however, that open-source programming is a competition only one coder can win. Therefore, it does not fully take into account all of the collaboration that may occur between programmers. It also assumes that closed-source software will be developed by teams of programmers paid by the software vendors. The size and skill of these teams will be limited by the resources of the vendors. Managers will attempt to decrease the amount of wasted work, and closed-source programmers will not view competition as a use. Users then download free licenses to lower the development costs. The model also assumed that demand only depends on quality and cost (Raghunathan et al. 2005, 903-918).

**ACKNOWLEDGMENTS**

Warren Breggenv, Ph.D., Provost and Vice President for Academic Affairs
Vish Prasad, Ph.D., Vice President for Research and Economic Development
Costin Turculescu, Ph.D., Dean of Engineering
Gloria C. Cox, Ph.D., Dean, Honors College
 Barnett Bryant, Ph.D., Dean, Department of Computer Science and Engineering, College of Engineering

**REFERENCES**


**RESEARCH METHODOLOGY**

Part 1: Security

To evaluate the security of the open-source and closed-source software, I will analyze the NIST National Vulnerability Database, as Schryen (2011) did in his study. This appears to be the most comprehensive dataset available on vulnerabilities.

Part 2: Functionality and Ease-of-Use

To evaluate functionality and ease-of-use, I will randomly select a group of subjects from a variety of backgrounds and with different levels of experience with computers. After the subjects sign the required consent form, I will administer a pre-study survey. The subjects will then perform using nothing but the products’ official documentation and knowledge base. I will configure the web browser(s) installed on the computers used for the study so that the subjects can only access these sites and a survey page for the project. The tasks will vary depending on the type of program.

The participants will complete a set of questions following each task. I will ask the participants to upload some type of proof that they have completed these tasks. The format of the proof will vary depending on the type of program. For the web browser example described above, a subject will upload a copy of the document to the survey page after completing each task. With other programs, such as web browsers, I will likely ask the participants to upload screenshots after completing each step to the survey page.

I will then give the subjects two hours to continue to use the software in question. This is to take into account, that, according to Raghunathan et al. (2005), users will have varying expectations of the software at different times. The participants will then take a post-study survey.

**ACKNOWLEDGMENTS**

Logan Widick, Department of Computer Science and Engineering, College of Engineering
Faculty Mentor: Diane Verrill, Department of Political Science, College of Arts and Sciences

(Barrett Bryant, Ph.D., Chair, Department of Computer Science and Engineering, College of Engineering; Vish Prasad, Ph.D., Vice President for Research and Economic Development; and Costin Turculescu, Ph.D., Dean of Engineering)