A Market-Based Approach to Software Evolution

David F. Bacon
IBM Research and Harvard University
dfb@watson.ibm.com

Yiling Chen
Harvard University
yiling@eecs.harvard.edu

David Parkes
Harvard University
parkes@eecs.harvard.edu

Malvika Rao
Harvard University
malvika@eecs.harvard.edu

ABSTRACT

Software correctness has bedeviled the field of computer science since its inception. Software complexity has increased far more quickly than our ability to control it, reaching sizes that are many orders of magnitude beyond the reach of formal or automated verification techniques.

We propose a new paradigm for evaluating "correctness" based on a rich market ecosystem in which coalitions of users bid for features and fixes. Developers, testers, bug reporters, and analysts share in the rewards for responding to those bids. In fact, we suggest that the entire software development process can be driven by a disintermediated market-based mechanism driven by the desires of users and the capabilities of developers.

The abstract, unspecifiable, and unknowable notion of *absolute correctness* is then replaced by quantifiable notions of *correctness demand* (the sum of bids for bugs) and *correctness potential* (the sum of the available profit for fixing those bugs). We then sketch the components of a market design intended to identify bugs, elicit demand for fixing bugs, and source workers for fixing bugs. The ultimate goal is to achieve a more appropriate notion of correctness, in which market forces drive software towards a *correctness equilibrium* in which all bugs for which there is enough value, and with low enough cost to fix, are fixed.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability; D.2.10 [**Design**]: Methodologies; J.4 [**Social and Behavioral Sciences**]: Economics; K.6.3 [**Software Management**]: Software development

General Terms

Design, Economics, Reliability, Verification

1. INTRODUCTION

Specification. Implementation. Verification. Testing. Correctness. These are the concepts upon which rests the human enterprise of software creation, and the discipline of software engineering. And at their intellectual core, even if rarely obtained in practice: *Proof* — the notion that we can, in principle, create a specification for a software artifact, implement it, and prove it correct.

This intellectual foundation for software engineering, the Platonic ideal of the bug-free program, has dominated both practical methodology and theoretical study in computer science.

One of the progenitors of this idealized approach, Edsger Dijkstra, suggested in his 1972 Turing Award lecture that

> *well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability, at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs.*

Today this suggestion seems remarkably naïve, and if anything the situation has gotten worse rather than better.

Various approaches to correctness have been used: formal proofs (with significant progress in recent years using mechanical systems like the Coq proof assistant) and model checking are both severely limited in the scale of software artifact to which they can be applied. Some researchers have acknowledged that absolute correctness is inachievable and suggested alternate approaches, notably Rinard's work in "failure oblivious" computing [13].

Our belief is that we need a fundamental change in our approach to large-scale software systems which relies on organic, self-regulating mechanisms rather than attempting to achieve some absolute, centralized notion of correctness.

In this paper, we explore the possibility of using a market mechanism to drive the evolution of software. The goal is not to be bug free, but rather to be free of bugs that people care about, and that can be fixed economically. We begin by summarizing some approaches to using market systems in various parts of the software development process, and then describe our proposed approach.

2. CROWD-SOURCING APPROACHES

Unlike traditional approaches that seek technical advancements to ensure the correctness of software, many recent systems are "crowd-sourcing" various software development and improvement tasks to the participants of the software ecosystem. Crowd-sourcing is a process that involves the use of a competitive process (though payments, prizes or other forms of reward) for the sourcing of work or information, usually involving problem decomposition into small, modular chunks. In this section we review existing bug tracking systems, vulnerability markets, and online marketplaces that take a crowd-sourcing approach. As we will see, these systems, some of which are market-based, typically only focus on one particular aspect of the software ecosystem.

In their paper "The principles of distributed innovation," Lakhani and Panetta describe three different industries where distributed innovation systems have been implemented successfully [7]. The authors discuss the motivation for people to participate in such systems and the organizing principles of production. The paper considers the Linux operating system as an example that highlights the benefits of work that is organized such that many individuals can self-select and lead elements of development without much ex-ante guidance and control. In fact Linus Torvalds, the founder of Linux, is quoted as follows: "*I would much rather have Brownian motion where a lot of microscopic directed improvements end up pushing the system slowly in a direction that none of the individual developers really had the vision to see on their own*." The paper also points out that there is a relatively high failure rate in these systems. Moreover distributed innovation systems do not seem to be efficient in milestone-based innovation development which requires strict planning and delivery on demand.

2.1 Bug and Vulnerability Reporting

It is natural to engage and incentivize users and free-market testers to report bugs and identify vulnerability of software. Voting-based approaches have been used to allow users to express their preferences over the importance of the bugs. Market-based approaches have been proposed to provide incentives for reporting critical security or vulnerability issues.

2.1.1 Voting-Based Bug Reporting Systems

Bug reporting systems often try to get users' preferences over different reported bugs. One way for users to express such preferences is to allow them to vote on bugs or related issues that they care about. For example, Sun uses *Bug Parade* to track bugs in the Java Virtual Machine. In Bug Parade, every unique email address is given some tokens that could be used to vote on the importance of particular bugs.

Similarly, the Adobe Flex bug and issue management system uses a system called *JIRA* to report and track bugs for the Flex Builder/SDK and ActionScript Compiler projects. JIRA allows a user to cast votes for various issues that may be of relevance to the user. It also allows a user to track a particular issue and be notified of any updates regarding that issue [6]. The aggregated votes allocated to bugs are thought to reflect the level of interest from the user community. As each user is allocated a limited number of votes, users must consider how to cast their votes carefully. Users could also unvote on issues that no longer interest them– this ensures that more important issues are dealt with first [1]. While voting-based systems offer expressiveness in order to understand user preferences they do not provide direct incentives for reporting, other than indirectly through the potential to infuence which bugs receive attention.

2.1.2 Market-Based Vulnerability Reporting Systems

For security-related issues, bug reports are essential to ensure the integrity of the software. Hence, several market-based vulnerability reporting systems have been introduced with the goal of incentivizing users to report bugs and vulnerabilities. We review two such systems below. The first one is currently used in practice, while the second one is a proposed theoretical framework.

Mozilla Security Bug Bounty. The Mozilla Foundation offers a cash award of $500 and a Mozilla T-shirt to anyone who reports a valid, critical, security bug [8]. The bug must meet certain criteria including the following: the bug must be new, it must exist in the latest supported version of Firefox or Thunderbird released by Mozilla, and the bug finder must not have written the buggy code or reviewed that code or contributed in any way towards that code.

Bug reporters are encouraged to work with Mozilla engineers in resolving the bug. Security vulnerabilities are treated in a special way because the consequences of a vulnerability being exploited can be extremely serious. Security bug reports may be kept private for a limited amount of time to enable Mozilla engineers to fix the bug before it is made public. However the bug reporter is allowed to decide when to disclose the bug to the public. The bug reporter may choose to disclose earlier if the bug is being ignored, for example. By not publicizing information about the bug immediately upon discovery and instead reporting it to Mozilla, the bug reporter acts in the interests of the Mozilla project and is compensated accordingly.

Vulnerability Markets. Schechter [14] proposes to use a vulnerability market to incentivize testers to identify vulnerabilities in software. In a vulnerability market, software producers offer time-variant rewards to the first testers who identify vulnerabilities in the software. A minimum reward value R_0 is initially offered, which then grows over time at a rate decided by the producer. When a new vulnerability is reported it is first verified to be genuine. On passing verification, the tester responsible for the report is awarded some portion of the reward. The reward amount is then reset to the minimum amount R_0. Only the first reporter receives the reward. Hence a tester that waits to report a vulnerability increases the amount of the reward but also increases the likelihood that another tester may report the vulnerability first thereby cutting him out of the reward. Clearly in such a scheme the most frequently occurring vulnerabilities will be found first. If the reward remains unclaimed the product is considered to be safe enough to protect information whose total value is at most the amount of the reward. Firms have an incentive to cooperate with the vulnerability market as it provides a measure of the quality of their products and hence their reputation.

Ozment [10] explains that the vulnerability market is essentially an auction– in the style of an open first-price ascending auction, or reverse Dutch auction. There is one buyer of vulnerability in the auction, the producer, and potentially unlimited number of sellers, the testers. The initial price is set to be relatively low and the price rises continuously until accepted by a seller. The valuations of the sellers or testers are private because they depend on the amount of work or cost incurred in locating a vulnerability. The minimum reward value R_0 is the auction's reserve price.

Interestingly it appears that the quality of software and the software producer's investment in patching technology are strategic substitutes. If it is possible to release a patch then software producers enter the market sooner and with software containing a greater number of bugs [9]. It has also been shown that software producers tend to issue patches later than is socially optimal [9]. We note that while both the *Mozilla Bug Bounty* and the vulnerability market provide incentives for reporting vulnerabilities, the valuation of users for a patch for a particular vulnerability is not captured.

2.2 Online Freelance Marketplaces

Whereas bug and vulnerability reporting systems are used to identify issues in software, they often do not couple this identification of issues with a sourcing process to perform work in addressing these issues. This gap is filled with online freelance marketplaces, which are platforms that connect individuals, small-business owners, and even Fortune 500 companies with freelance technology specialists to satisfy their technological needs. The sites provide vivid details about workers' histories and qualifications, and some even feature tools that let the businesses monitor the work they are paying for [4]. Web developers, software programmers and other IT specialists from different countries are readily available and often charge a fraction of the price of local workers. We will examine two such online companies, *RentACoder* and *TopCoder*, below.

While tasks typically are independent projects in RentACoder, individual tasks often are components of a larger project in TopCoder. We will also briefly review the *iTune App Store*, which is a market place for software applications for the iPhone.

2.2.1 TopCoder

TopCoder Inc. uses programming competitions to build professional grade software outsourced by clients ranging from individual entrepreneurs to global Fortune 1000 companies. For every task, participants compete against each other for cash award. The top one or two contestants win the award. TopCoder members can work for a variety of outsourced software projects including the "Bug Races," which is a competition for fixing bugs. Members can fix posted bugs and the first submission that is verified to be fully functional (according to tests and design criteria) is awarded a prize. Other types of software developement projects (software design, software architecture, etc.) use a similar competition format. Boudreau *et al.* [3] argue that the competition platform of TopCoder can increase the quality of the best solution by broadening the search for innovation. We note that TopCoder seems to be used mainly for *de novo* design rather than for improvements or fixes to existing software systems.

Central to TopCoder's methodology appears to be the modularization of software development work [5]. Each project is broken down to the most granular level possible. As a result most pieces of work can be completed in a few hours [2]. In addition the modularization of projects allows for simple evaluation criteria when determining if a submitted software solution or bug fix is indeed complete. It also reduces the likelihood that newly submitted code may exert unintended effects on the existing code base. Hence the winning programmer may be rewarded for his work as soon as it is validated. This has the benefit of reducing uncertainty for TopCoder as well as for the programmers. Interestingly, this modularization (and architecting, more broadly) is itself performed within the same competition platform.

While TopCoder provides some monetary incentives for its participants, its fixed pricing for tasks may not be efficient in identifying the best participants to perform a task. Moreover, TopCoder is also more than a software crowd-sourcing platform, in that it also provides value in establishing a community of talented coders and helping companies to identify talent. TopCoder is also an easily accessible practicing ground for programmers looking to hone their skills. All submissions to a task receive feedback from a peer review process involving multiple metrics on a scorecard, which provides valuable information on how a submission could have been improved [2]. Firms like Google and Microsoft often sponsor screening contests [3], and seem to hire talented participants.

2.2.2 Rent-a-Coder

Rent-a-Coder is another online marketplace that connects buyers to coders. All projects are protected by escrow and through arbitration. Buyers post new projects on the site. Sellers (coders) post questions and submit bids on the projects. The buyer then selects the seller that he or she wants to award the project to and puts the funds into escrow as a payment guarantee. When the work is completed, the buyer releases funds from the escrow account to the seller. If the seller completes the work but the buyer withholds the funds, an arbitrator will step in, test the software if necessary, and release the funds to the seller as appropriate [2].

Different auction types are alllowed within Rent-a-Coder. In an open auction all members of Rent-a-Coder can bid for a project. In a private auction only those coders that the buyer invites may participate. Other arrangements are also possible such as "Pay for Time" where the buyer pays a coder for the time spent on a project rather than for an end result. Rent-a-Coder charges coders a fee ranging from 7.5% to 15% on the profit from work done [12]. Once the work is completed the buyers and sellers may rate each other. This is different from TopCoder, where most of the reputation information is aggregated from directly measurable performance metrics such as the fraction of tests passed by developed code.

2.2.3 iTunes App Store

Another system that provides a market-based approach to a different part of the application development ecosystem is the iTunes App Store for the iPhone. The App Store has a built-in micropayment system where developers can make significant profits even with applications priced at \$0.99. There is also a rating system. Popular applications have thousands of ratings, and it is interesting that a large proportion of which include comments from users about outstanding bugs and desired features. Developers often address these issues explicitly in their comments when releasing updates. In this way, the App Store is effective in better connecting end users and developers.

The App Store shows that a micropayment-based model for software delivery is eminently practical and well-received by consumers, but lacks a way for multiple programmers to contribute to a project or for users to vote or bid for features and fixes in an organized fashion.

3. MARKET-DRIVEN SOFTWARE

We now describe our proposal for using a market-based mechanism to drive the evolution of software to increase its correctness and its functionality. We unify many of the previous partial market-based mechanisms, incorporating bug reporters (as in vulnerability markets), bug voters (as in Bug Parade) and developers and validators (as in TopCoder and Rent-a-Coder).

This proposal is not yet completely fleshed out (or at all implemented). To get some intuitive sense of what we have in mind, imagine a combination of the iTunes App Store and Bugzilla where users can bid micropayments for the fixing of bugs or implementation of new features, and any qualified developer can obtain access to the code and perform the requested work. Users could offer as little as a penny for a fix or feature, and aggregate demand could still be sufficient to make it worthwhile for a developer to satisfy them. Furthermore, the demand can be used by the developer to decide how to concentrate effort to increase quality and attract more users.

One missing element of existing voting-based systems such as Bug Parade is that they do not directly elicit additional work in fixing bugs in released software. Rather, they are used to focus existing programmer resources on problems identified as important by a user community. Our approach is designed to allow users to elicit additional work (and encourage new developers to work with the software) by offering rewards for the work.

More specifically, we consider a market ecosystem around a particular piece of software. The evolution of software is a dynamic process. However we will first present a static snapshot of the software market ecosystem. The ecosystem comprises the following basic entities:

U **Users** of the software. Users might be individuals, corporations, or other entities.

J **Jobs** which the users would like to have performed on the software.

W **Workers** who may perform jobs. Note that workers may also be users and vice versa.

K **Kinds** which are (optionally) used to categorize jobs. Examples are `correctness`, `feature`, `security`, `mac`, etc.

Label L_{kj} is 1 if job j has kind k, 0 otherwise.

Reward R^t_{uj} offered by user u for job j at time t. If user u does not offer a reward for job j at time t, then $R^t_{uj} = 0$.

Cost C^t_{wj} to worker w for performing job j at time t. If worker w is incapable of or uninterested in performing job j at time t, then we consider $C^t_{wj} = \infty$.[1]

The labels exist for categorization and to allow the calculation of aggregate statistics about different categories of work items. The market is agnostic as to these categories. However, users may express preferences by offering rewards for particular kinds of jobs; we will expand on this below.

We assume that rewards are in an actual currency (denoted "\$") and consider for the most part an open market in which the only barrier to entry for a particular worker is their ability to complete a particular job (as opposed to the availability of the source code). A key feature of our market is that it incorporates both the aggregation of user bids (like bug voting systems) and multiple competing workers (like TopCoder). This exchange structure, with information and preferences (e.g., costs for different kinds of work, values for different kinds of fixes) on both sides, is designed to provide for a more efficient market place.

For the time being, we make a number of simplifying assumptions: that users' bids are correctly aggregated (that is, we can determine when two users offer a reward for fixing the "same" bug) and that a single worker can perform a job in its entirety (which requires jobs to be modularized appropriately, as is the case in TopCoder for example). In Section 4.4 we discuss how to use market mechanisms to handle a more realistic scenario.

3.1 User Demand

Given the ecosystem described above, the total reward for a job j at time t is

$$R^t_j = \sum_{u \in U} R^t_{uj}, \tag{1}$$

and we define the *user demand* on a piece of software as the aggregate of the available rewards:

$$\mathcal{R}^t = \sum_{j \in J} R^t_j \tag{2}$$

When $\mathcal{R} = 0$, there is no demand for changes or additions to the software. Note that this does *not* imply that users are satisfied: the demand might be 0 because users don't like the software and no one is using it. We denote the demand for a particular kind k of job at time t as

$$\mathcal{R}^t[k] = \sum_{j \in J} R^t_j \cdot L_{kj} \tag{3}$$

As discussed in the introduction, software has traditionally been considered to have some ideal "correct" state in which there are "no more bugs." Henceforth we will refer to this notion as *absolute correctness*. While it is a laudable goal, for anything but small modules absolute correctness is unachievable, and in many large systems, can not even be completely specified.

[1]Note that we can allow for workers that are capacity constrained by allowing the cost for a worker to adjust to ∞ when already working on another job. Moreover, cost can be considered to already factor in a worker's profit margin so that cost C^t_{wj} indicates the reward that a worker requires for job j at time t to be willing to perform the work.

However, in our market, we can express the *correctness demand* in period t as simply $\mathcal{R}^t[\texttt{correctness}]$. If the correctness demand is 0, it does not mean that there are no bugs in the software – but it does mean that there are no bugs to which any users attach value for fixing. Analogously, we can quantify the demand for security, new features, support for a particular platform, and so on.

3.2 Market Potential and Equilibrium

Intuitively, the jobs that are "worth doing" for workers are those where the cost of performing the work is less than the expected reward. More specifically, we can define the *potential value* of a job j at time t as

$$P^t_j = \max_{w \in W}(R^t_j - C^t_{wj}, 0) \tag{4}$$

That is, the potential value of a job is the reward net cost that can be obtained by the worker who can perform it for the lowest cost (provided that the reward is at least their cost). We define the potential value of the entire job market as:

$$\mathcal{P}^t = \sum_{j \in J} P^t_j \tag{5}$$

The *correctness potential* of a system in period t is then $\mathcal{P}^t[\texttt{correctness}]$. Note that $\mathcal{P}^t[\texttt{correctness}] \leq \mathcal{R}^t[\texttt{correctness}]$, so that if there is no correctness demand then there is necessarily no correctness potential.

The user demand R^t_j can be thought as the price at which users demand for the work on job j (or the bid price). C^t_{wj} then represents the price at which worker w is willing to supply the work for job j (or the ask price). Ideally, whenever the bid price is greater than or equal to the ask price, the market should drive the work to happen. Hence, we define that a system is in *correctness equilibrium* when $\mathcal{P}^t[\texttt{correctness}] = 0$. A system in correctness equilibrium is one where all of the bugs that are "worth fixing" have been fixed. There may still be plenty of latent bugs, or even significant correctness demand, but there are no longer any bugs which a worker can fix without losing money.

3.3 Dynamics in the Market

The notions of correctness demand, potentia and equilibrium have so far been discussed in the context of an (implicitly) static system, where a group of users bid rewards for a job to be performed, and workers decide whether to perform the jobs based on their costs and the economic rewards.

However, a real system is dynamic: the reward for a job may fluctuate (up when users post additional rewards for the job or down if the user has specified an "expiration date" for the reward). Furthermore, workers may decide to attack a problem at different times depending on *how* profitable a job is and whether the reward for a job might increase further, as well as considerations about the level of competition with other works and other opportunities. Thus a piece of software with a large and vibrant user community may have a user demand and a market potential that remain high: as some jobs are resolved, others are submitted.

A perfect equilibrium, for example with $\mathcal{P}^t[\texttt{correctness}] = 0$ for correctness, may never be reached in reality. Instead the system will tend to constantly move towards equilibrium, pushed forwards by market pressure to perform jobs that comes from the potential value associated with a job. We should expect that jobs with high correctness potential to be prioritized ahead of jobs with lower correctness potential; e.g, because there exists a worker who can perform the job at low cost and extract a large fraction of the reward available for completing the job.

A low correctness potential in steady state could indicate either that the system is well-written with little to fix, or easy-to-make fixes, or that it has a large or wealthy user base that values improvements, even if they are costly to implement.

3.4 Inability to Measure Cost

We have defined market potential (e.g., correctness potential) in terms of the costs to each worker of performing each job and the demand for a job. However, it should be noted that these costs may not be known a priori, even by the worker considering the job (and we all know how inaccurate programmers' development time estimates can be). Furthermore, while an individual worker can probably evaluate their cost effectively once they have completed a job, such a worker may not wish to share this information with the marketplace. Nevertheless, the definitions of market potential are useful to provide a framework within which to discuss the behavior of the system. And in fact, the inability of users in the marketplace to know the cost functions is important for stimulating rewards (otherwise users would stop offering rewards as soon as the value potential of the job exceeded 0). A rich market economy could include *market analysts* who estimate the cost function, and sell the information to participants in the system.

3.5 Efficiency

Another important factor not captured so far is market efficiency. Given a set of jobs, generally speaking we would like each job to be performed by the the worker that provides the *best* tradeoff between solution quality and cost of performing the work, rather than simpler the cheapest work. But the model as proposed does not differentiate between different qualities of work; rather, whether or not the reward is available is a binary determination. Another criteria to consider is the *number* of jobs completed. This could be important, for example, if solving a larger number of jobs provides a positive externality on the overall user experience, or helps to have ripple effects in making other jobs easier to complete.

Thus we would like to be able to quantify the efficiency of a particular work allocation, really a work allocation policy in the context of a dynamic system, and to design a market mechanism in such a way that it maximizes efficiency. Quantifying efficiency within a formal model of this dynamic system will be an important direction for future work.

3.6 Bundled Rewards

So far we have discussed individual rewards posted by users. However, some users may wish to influence the broader direction of the software rather than fund particular jobs. Other users may simply not have the time or expertise to offer rewards for particular jobs. *Bundled rewards* provide an approach for handling both of these issues. A user could offer a reward to be split across a set of jobs, and in particular over all of the jobs of a specific kind. A user who was primarily interested in the stability of the system could offer a reward to be split across all of the `correctness` jobs; another user might favor `security`; another might allocate funds to encourage development on the `mac` platform. This is analogous to investors who invest in the Dow Jones Industrial Average instead of in particular stocks.

Funds could be divided evenly across all the jobs in the bundle, or proportionately to the existing rewards ("matching funds") in order to leverage the information extant in the current reward offers. Another question is whether the allocation should be made across the jobs extant when the bundled reward is offered, or whether it should adaptively allocate funds as new jobs are posted. In the latter case, it seems that some amount of hysteresis will be desirable so that workers are not subject to excessive price fluctuations.

4. MARKET DESIGN: COMPONENTS AND CHALLENGES

In the previous section we made two major simplifying assumptions: that an individual worker can complete a job, and that workers always perform quality work that adequately addresses an issue. In particular, there is no quality control in the system. Quality control implies multiple agents with checks and balances. We now present the elements of a market design for a realistic market system that removes these assumptions.

There are four fundamental principles that underly our design:

- **Autonomy**. All of the actions necessary to bring jobs to completion should be driven by market forces; the process is never gated by an entity outside of the market.
- **Inclusiveness**. Everyone who provides information or performs work that leads to improvements should share in the rewards.
- **Transparency**. The system should be transparent with respect to both the flow of money in the market and the tasks performed by workers in the market.
- **Reliability**. The system should be immune to manipulation, robust against attack, and prevent "shallow" work which would have to be re-done later.

There are two fundamental problems that the system must support. First, *information aggregation* from the user base, through an expressive, non-manipulable and easy-to-use process. Specifically the system must capture user valuations for various jobs. It must also elicit certain types of information (for example, the steps to reproduce a bug) that would be useful in performing the job. Finally the system must aggregate all users' preferences and information regarding particular jobs.

Second, *work allocation*: the system must support the efficient allocation of sub-tasks within the software producer organization and across "outside" programmers in the case of open software development. In providing this functionality, it will be important to identify the right people to perform a task and to provide incentives so that an appropriate effort level and technical solution is provided. Jobs also need to be prioritized, factoring both the value to the user base and the cost to perform them.

The components of our market design seek to integrate these two fundamental subproblems into a comprehensive solution.

4.1 Funding

The first question we must answer is how the market is funded. This depends on what kind of business model is being used for the software. However, the market is designed in such a manner that whether the system is open- or closed-source, there are no "downstream dependencies" by other parts of the market that must differentiate where the funding originated.

A general feature of the proposed market-based system is that bids are placed in *escrow* and have an *expiration date*; if the bid expires (no one has performed the work by the "deadline"), then the money is returned to the bidder (or they can choose to "let it ride"). A number of methods are possible, which can all co-exist if desired:

Direct Bidding. A user can directly offer a cash reward.

Escrow from Sale. When a user purchases a piece of software, a fraction of the sale price is placed in escrow for the market. The user can then use this capital to bid. If the user does not bid after some period of time, the money can be returned to the seller or placed into a "general fund" that tracks the market as a whole.

Escrow from Contribution. For a shareware model, since the contribution is voluntary the user can choose the fraction of the shareware contribution to place in the market, and can also choose to withdraw the money when it expires (rather than having it returned to the seller, as above).

Escrow from Registration. Even if the software is free, there may still be capital available. Many large open-source projects have significant sources of corporate support (for instance, Mozilla's largest source of income is from sales of placements in the Firefox search bar). Some of this money can then be placed in the "general fund" and have portions allocated to users upon registration. If the users do not bid, the capital in escrow expires and reverts to the general fund.

Note that in all of these cases, regardless of business model, the proposed market is funded with real money, which can be earned by those contributing to the software. The only differences are the degree to which the money in the market is fungible to the bidders. The money in escrow is handled by a trusted third party (*not* by the seller of the software), which we designate as the "bank."

4.2 Reputation System

Autonomy is the key property in making the market work. To the greatest extent possible we wish to use decentralized market mechanisms rather than centralized control to achieve our goals. For example, rather than having a centralized authority that validates a bug fix, we propose a market mechanism (similar to that adopted in TopCoder) in which other workers are rewarded for providing test cases, and competing developers can challenge the correctness of a solution and possibly "steal" the win from their competitor.

We do acknowledge that there always needs to be some form of centralized control for most software systems, in particular when making large-scale architectural decisions, or choosing among competing sets of features. Open source projects (for example, Linux, Firefox, and Jikes RVM) all have a "core" group which exerts this centralized control, even as they accept work from a wide range of contributors. Thus, simply having a bazaar [11] is untenable– we at least need a small church next to it.

However, centralized control suffers from lack of scaling, is often unncessarily rigid, and can lead to a lack of responsiveness. In order to address this issue, we propose to design a hybrid reputation-based system: every contributor has a *rating*, and every kind of task has a minimal *rating prerequisite*. Ratings can be seeded (or adjusted) by the central authority, so it can directly give its members privileges to perform various operations, or deauthorize external contributors who it decides are acting against the interest of the user base.

As contributors perform work, when they are paid their reputation increases. The idea is that ratings will naturally flow from the initial "core" group designated by the central authority out to an organically growing base of contributors. If the contributor community reaches critical mass the rating system can become self-regulating, without requiring direct intervention by the central authority. Nevertheless, the central authority can retain the right to intervene when needed.

4.3 Payments

Each job will have a set of workers who contributed to various portions of the workflow; e.g., the reporter of a problem, the developer of a test, and the developer of a fix. Generally speaking, we propose that all contributors share in the reward for the job when it is delivered, although some (partial) payments may be made earlier in order to encourage certain kinds of activities. Payments can also be made over time to ensure that contributions are robust; if a contribution is discovered to be flawed after it is delivered, the remaining payments can be halted, and the funds diverted to workers who fix the problem.

The exact way in which the reward is to be divided among the contributors is still open to design, but we see this being modulated by a number of factors: the type of task performed, the rating of the contributor, and the "shape" of the demand trajectory (see Section 4.5). It also seems desirable for the system to be self-adapting. If a job "stalls" at a particular point in its workflow, the fraction of the reward for performing the next task can be made to grow over time. This can also be used to feed into the global default fraction for that type of task. For instance, if many workflows stall in the testing phase, the fraction of the reward allocated for testing will go up over time.

4.3.1 Applicability to Closed Source Systems

A major question is whether the kind of market system we are proposing will work for closed-source software, where the producer maintains some form of monopoly control over the code.

While closed source reduces the opportunities for contributions, it does not preclude using the market. First of all, some tasks, like reporting bugs, creating test cases, and writing documentation can be handled by users with no access to source code.

Secondly, the market may be used to drive allocation of development resources within the company. This could range from direct control by managers who merely use the market data as useful input about what issues are important to the user base, and collect the payments on behalf of the company, to a system where developers receive a base salary and must earn the rest of their pay from the software market. In the latter case, the producer can seed the market with money that is intended primarily to flow back to its employees as incentive-driven compensation.

Finally, the if there is demand for work in the market which only the producer can perform (due to its monopoly status), the producer can hire contractors to perform the work, funding them out of the accumulated rewards in the market. In effect, the market provides a way for the demand to "tunnel through" the organizational boundary created by the producer's closed-source monopoly, and allow it to repond with a higher degree of elsasticity to the demand for work on the software.

4.4 Workflows

Different kinds of work have different *work flows* that require various types of contributors in order to complete.

4.4.1 Bug Workflow

We begin by describing the workflow associated with reporting and fixing a bug:

Report. Filing a report of a bug. This can be as simple as clicking a button on a popup box ("Send Report to Provider") when the application crashes, or filing an explicit report of a problem in a bug database. The former can be made available to anyone (except perhaps "spammers" with negative reputation ratings); the latter would typically require some non-zero rating.

Bid. Bidding an amount to contribute to the reward R_j for fixing the bug.

Organize. Determining when two reports are instances of the same bug, or when what appears to be one bug is in fact two different bugs, or that a bug has already been resolved, is sometimes a difficult task. Therefore it is explicitly recognized as an independent part of the workflow, and requires a relatively high rating. Organization can also include categorization of a bug, for instance as a security exposure.

Reproduce. Creating and filing a way of reproducing the bug, so that fixers can begin to work on it.

Fix. Develop a fix for the bug. This can be open to contributors with a wide range of ratings, but a bug fix submitted by a low-rated contributor will have to be committed by someone with a proportionately higher rating.

Test. Test the fix, either by providing an executable test case or by exercising the system. Since the latter requires a large degree of trust, it would typically require a high rating. A test that is failed by a fix receives a higher reward than one that passes.

Commit. Commit the fix to the canonical code repository. This will typically require a high rating.

Distribute Patch. In some systems, it may be desirable to make either source- or object-code patches available before the bug fix can be incorporated into a full release. Bidders could be allowed to specify whether they would pay for a patch, or would rather wait for a new release.

Some tasks in the workflow may be designated as being *antagonistic* (for instance fixing and testing). Antagonistic tasks will either be prevented from being performed by the same contibutor, or will require a relatively high rating to allow it.

4.4.2 *Other Workflows*

Various other kinds of workflows would address other parts of the overall software lifecycle. For the development of new features, TopCoder provides a good model. Their process includes *Conceptualization, Specification, Architecture, Component Design, Component Development*, and *Assembly*. Depending on their scope (minor localized features versus large-scale changes across the application) the workflow might be condensed or extended. Documentation could also be a workflow in which tasks might be *Outlining, Terminology, Section Writing, Integration*, and *Proofreading*.

4.5 Using the Demand Trajectory

An interesting aspect of the system is that the *demand trajectory*, that is how the demand (i.e. aggregate bid value) varies over time, can be used to extract information about how the market values bugs and how the rewards should be divided between various contributors to the workflow for a job.

Figure 1 tracks the aggregare reward for four different hypothetical bugs. A critical security bug due to a buffer over-run would be high priority (its reward would rise drammatically as soon as it was discovered), a fix would be submitted and validated quickly, and the fix would then be delivered immediately to the field. Thus the payouts would be made rapidly. On the other hand, a low priority bug that takes a lot of time to fix may never get fixed at all, and have a very slowly growing (or even shrinking) reward.

In Section 4.4 we described realistic scenario involving different players in a bug workflow: a developer would fix the bug and then testers would test the fix. Moreover the fix might only be accepted if some other worker of high reputation validates the final solution.

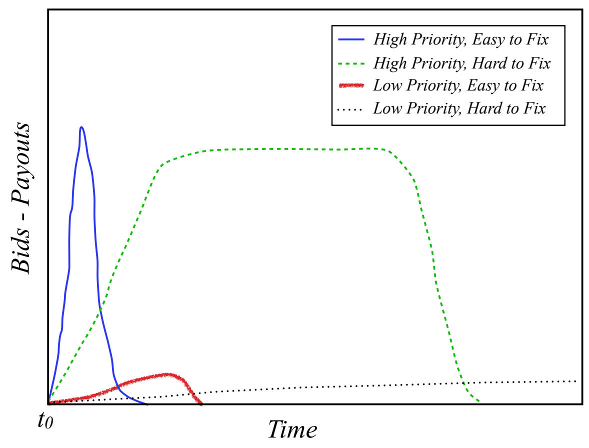

Figure 1: Characteristic Demand Trajectories of Various Types of Bugs

In the case of security vulnerabilities the mechanism must also incentivize users to report defects.

The dynamics of our proposed market mechanism function as follows: At time T_0 user U reports a bug B. The software producer (company or organization or open source community) pertinent information regarding B in a public database. Now users start to contribute money on B reflecting their value for a fix. At some time T_1 the reward riding on the bug has grown to an amount $R_B^{T_1}$. A developer, who decides that enough money has been contributed to make correcting B profitable, jumps in and begins work on B. Later at time T_2 the developer has completed his work and other users or testers test the software to see if the fix is indeed robust. If at some time T_3 flaws are found in the solution submitted by the developer then the persons responsible for the detection of the flaws share in the reward money $R_B^{T_3}$.

We can consider a payment scheme that distributes the reward money R_B among the bug finder, the bug fixer, and the bug tester in different proportions reflecting the strength of their relative contribution to the resolution of the bug. For example, the ratio $R_B/(T_1 - T_0)$ is computed when ascertaining the significance of the intial discovery of the bug. If this ratio is reltively high then it implies that the discovery was very useful as a large sum of money was accumulated in a very short time. Clearly some portion of the user base would derive significant utility from a fix for the bug. Similarly if the time taken to fix $T_2 - T_1$ is long then it tends to indicate that the fix was non-trivial and the bug fixer expended considerable effort (recall that there is a competitive process here, so it is not easy for one developer to artificially delay this window). Likewise the longer the period $T_3 - T_2$ the more robust the fix. Therefore the sooner testers can find a flaw the greater their share in the pie. The discoverer of the bug is determined to be the first person to find a particular bug that is verified to be genuine. Many people may work on fixing the bug; however only the first person to submit a working solution (including fixing any flaws) is paid. Any tester who either detects a new and valid flaw in the fix or devises a thorough acceptance test can share in the reward.

4.6 Design Challenges

A market structure seems better suited to address the problem of efficiently resolving neglected bugs than existing non-market systems. However bug management is a nontrivial problem with complex interdependencies and side-effects and there remain some interesting challenges, that point to the richness of the problem and are intrinsic to the problem. For example, the unintended side-effect of introducing new bugs while fixing an existing bug is a common occurrence in software development. Under current bug management systems these new bugs are simply discovered and then logged in the bug database for future examination. However with a market-based approach where a payment is made conditional on a fix, we can no longer afford to ignore these side-effects and a market-based approach will need to develop techniques to promote “good" fixes that don’t lead to lots of new unintended side-effects.

Challenge 1: Certifying fixes. A fundamental challenge is how to determine whether a fix has in fact thoroughly fixed the bug in question. This is important in order to ensure that the payment for a fix is fair and robust against manipulations. For example, users may initially be unable to reproduce the bug for *the workflows typically used by these users*. However some time later some user executing a different workflow may reproduce the “fixed” bug. If the payment for that bug has already been made there is little incentive for developers to prioritize a fix. Or else if the bug is treated as a "new" bug (thereby requiring that a further reward be provided for its fix) then the system’s efficiency deteriorates. Just allowing users time to validate a fix is unlikely to be enough. One idea is to incentivize testers to either provide a *certificate of fix* or show that the bug has not been properly fixed.

Challenge 2: Deep versus shallow fixes. A natural question that arises out of examining Challenge 1 is how to design incentives in order to obtain robust bug fixes and to preempt any strategizing on the part of the fixers. The system must incentivize “deep" fixes over “shallow" fixes. A discounted payment scheme might be one way of addressing this, so that follow-on fixes to augment an earlier fix are assigned rewards in a way that ensures that the aggregate payment is greater if a good, deep fix is completed initially.

Challenge 3: Externalities caused by bug fixes. One or more bug fixes may unintentionally fix a number of other reported bugs. Likewise one or more bug fixes may unintentionally introduce new bugs. Let us consider the following scenario: users have logged several bugs in the bug database. In response 3 developers decide to tackle bugs X, Y, and Z. They simultaneously submit fixes to their respective bugs. As a result several other bugs D,F,G,M, and Q appear to be resolved as well. Now we are left with the question of which of the original 3 bug fixes resolved the other bugs and how should the payment be determined. A preliminary solution might be to permit only one bug fix at a time to be entered into the software. Verification of the effects of a bug fix may require time and effort from testers who can certify that the bug fix is indeed correct, that other bugs were (not) fixed, and that new bugs were (not) introduced. However leaving a time gap between two fixes slows down the process and could cause bottlenecks. Alternatively could there be some kind of *automatic* market-driven method to verify the effects of a bug fix? For example, would it be possible to generate a bug dependency graph that could allow propagation of credit?

Challenge 4: Eliciting expressive bug reports. The system must elicit and aggregate expressive bug reports needed for robust fixes. We may want to examine the tradeoffs between private versus public disclosure of bugs. Publicly sharing information on bugs would help users to coordinate their efforts and exchange valuable information regarding the bug and potential workarounds. One way to accomplish this would be to set up a *Bug Information Exchange*. This would increase the quality of the bug reports and reduce the number of duplicate bugs. In the event that a reasonable workaround is found users would then have a better sense of how much they value an immediate fix for that bug.

Challenge 5: The public goods problem. A user who chooses not to invest money or effort on a bug which is fixed by others nonetheless benefits from the fix. As the user base grows the public goods or free-riding problem appears to be unavoidable. Peer-production systems also face similar issues. Some funding models, such as “escrow from sale" already address this problem. But the problem will likely still remain in most such systems in some form or other. Future work should explore this challenge; e.g., how can its effect can be minimized, or can bounds be established on its impact on social welfare?

5. ACKNOWLEDGEMENTS

We thank Karim Lakhani, Ben Lubin, Rob O’Callahan, and the Harvard EconCS group for valuable feedback and discussions.

6. REFERENCES

[1] Adobe bug and issue management system. *http://bugs.adobe.com/flex/*.

[2] BENNETT, A. Itworld: Finding freelance jobs: 6 sites for talented techies. *http://www.itworld.com/career/65739/finding-freelance-jobs-6-sites-talented-techies*, April 2009.

[3] BOUDREAU, K. J., LACETERA, N., AND LAKHANI, K. R. Parallel search, incentives and problem type: Revisiting the competition and innovation link. working paper, no. 09-041. Tech. rep., Harvard Business School, September 2008.

[4] FLANDEZ, R. The wall street journal: Help wanted – and found. *http://online.wsj.com/article/SB122347721312915407.html*, October 2008.

[5] HOWE, J. Wired magazine’s jeff howe talks about crowdsourcing and topcoder. *http://www.topcoder.com/direct/blogs/?p=174*, February 2009.

[6] Jira bug and issue tracker. *http://www.atlassian.com/software/jira/*.

[7] LAKHANI, K. R., AND PANETTA, J. A. The principles of distributed innovation. *Innovations: Technology, Governance, Globalization 2*, 3 (2007), 97–112.

[8] Mozilla website. *https://www.mozilla.org/security/bug-bounty.html*.

[9] NIZOVTSEV, D., AND THURSBY, M. Economic analysis of incentives to disclose software vulnerabilities. In *In Fourth Workshop on the Economics of Information Security* (2005).

[10] OZMENT, A. Bug auctions: Vulnerability markets reconsidered. In *In Third Workshop on the Economics of Information Security* (2004).

[11] RAYMOND, E. S. *The Cathedral and the Bazaar*, 2^{nd} ed. O’Reilly & Associates, 2001.

[12] Rentacoder inc. website. *http://www.rentacoder.com*.

[13] RINARD, M. Acceptability-oriented computing. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Anaheim, CA, USA, 2003), pp. 221–239.

[14] SCHECHTER, S. E. How to buy better testing: using competition to get the most security and robustness for your dollar. In *In Infrastructure Security Conference* (2002).