Incentivizing Deep Fixes in Software Economies

Malvika Rao, David F. Bacon, David C. Parkes, and Margo I. Seltzer

Abstract—An important question in a software economy is how to incentivize deep rather than shallow fixes. A deep fix corrects the root cause of a bug instead of suppressing the symptoms. This paper initiates the study of the problem of incentive design for open workflows in fixing code. We model the dynamics of the software ecosystem and introduce *subsumption mechanisms*. These mechanisms only make use of externally observable information in determining payments and promote competition between workers. We use a *mean field equilibrium* methodology to evaluate the performance of these mechanisms, demonstrating in simulation that subsumption mechanisms perform robustly across various environment configurations and satisfy important criteria for market design.

Index Terms—Market design, mean field equilibrium, software engineering, payment mechanisms

1 Introduction

Rather than the Turing machine "input tape to output tape" model of computation, today's software systems have evolved to be those of continuous interaction with other large systems and with the physical world. The size and complexity of software systems have increased to such an extent that it is straining our ability to effectively manage them. This is illustrated by Fig. 1 which depicts the dependency graph of an open source software.[1]

In the meantime, traditional software engineering techniques have failed to scale accordingly, leading to increased inefficiencies and errors. A study commissioned by the U.S. National Institute of Standards and Technology concluded that software errors alone cost the U.S. economy approximately \$59.5 billion annually [1]. What is more, software is often deployed with discovered bugs (as well as undiscovered ones) because there are simply not enough resources to address all issues [2], [3], [4]. Through an empirical study of 277 coding projects in 15 companies, Wright and Zia [5] determine that software maintenance actually introduces more bugs: each subsequent iteration of fixes has a 20-50 percent chance of creating new bugs. All of this points to the need for a new paradigm in creating and evolving such systems.

In fact software systems have come to resemble economic systems, where behaviour is decentralized, interdependent, and dynamic. This suggests that the principles of market design and mechanism design have an important role in complementing traditional engineering techniques. We see this in market-based platforms such as Bountysource [6] and TopCoder [7]. Markets enable us to directly target incentives issues as well as elicit valuations from users so that they can influence the direction of software development. Moreover markets can aggregate supply and demand, thereby providing the scale needed to solve long neglected engineering problems, with prices guiding the efficient allocation of resources.

Software economies refer to a vision for a software development process in which supply and demand drive the allocation of work and the evolution of the system [8]. The idea of a software economy is to promote equilibria in which all fixes and features for which there is enough value have been implemented. A private software economy deals with the internal incentives of managers and employees [9]. A public software economy involves end users, who are able to express value for bug fixes as well as missing features. A public software economy must contend with a much larger scale than a private economy, including a large user base, a large number of potential workers, and conceivably a large number of bugs and missing features.

We are interested in a specific question arising in the public software economy. We want to understand how to design incentives to obtain deep rather than shallow fixes to bugs, at least where the disutility of users warrants a deep fix. A deep fix attempts to correct the root cause of the problem so that another bug with the same root cause is found only after a long time or not at all. In contrast a shallow fix suppresses the symptoms of a bug at a superficial level and other bugs with the same root cause may appear soon after. While this presents a known problem in software engineering, there has been little prior work on improving incentives in the literature. This paper initiates the first study of this problem, proposing *subsumption mechanisms*, where deeper fixes can *subsume* or replace shallower fixes, and where a worker's payoff increases if a fix subsumes other fixes. A subsumption mechanism employs an installment-based payment rule that stops paying the worker when a fix is subsumed, transferring the remaining reward to the

1. Austin Meyer and Adil Yalcin created this network visualization using NodeXL (www.nodexlgraphgallery.org) for a graduate computer science class project "Analyzing Open Source Project Networks" at the University of Maryland College Park taught by Prof. Ben Shneiderman during Spring 2011, https://wiki.cs.umd.edu/cmsc734_11/index.php?title=Analyzing_Open_Source_Project_Networks

• *M. Rao, D. C. Parkes, and M. I. Seltzer are with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138. E-mail: {malvika, parkes, margo}@eecs.harvard.edu.*
• *D. F. Bacon is with Google. E-mail: dfb@google.com.*

Manuscript received 27 May 2017; revised 31 Jan. 2018; accepted 4 May 2018. Date of publication 13 June 2018; date of current version 9 Jan. 2020. (Corresponding author: Malvika Rao.) Recommended for acceptance by T. Zimmermann. Digital Object Identifier no. 10.1109/TSE.2018.2842188

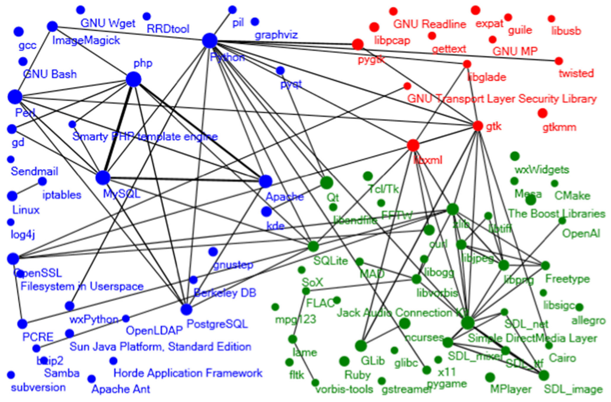


Fig. 1. Dependency graph of an open source software.

contributor of a deeper fix. The idea is that deeper fixes are less likely to be subsumed and more likely to subsume prior fixes, thereby enabling the worker to earn a higher payment and promoting competition for deep fixes.

We study a stylized, dynamic model of the software engineering ecosystem comprising workers, users, root causes of bugs, bugs and fixes, user values, and worker costs. The user base reports bugs and offers money for suitable fixes. Workers are tentatively matched to bugs, and decide which fix to submit given considerations about cost, payment, and market dynamics. The market design determines how money from users is allocated to fixes. Crucially, we insist for practical reasons that the market process can only use information that is externally observable. For example, we allow for regression tests on code that has been corrected with submitted fixes, without allowing the possibility of code inspection. Some examples of externally observable information include the time taken for the next bug to appear, and the number of bug reports that are addressed by a fix. Our model is flexible and its elements can be viewed as building blocks that can be reconfigured to encode different issues in the software economy.

Current practice in software testing is consistent with our paradigm. For example, black box testing tests the functionality of the software without examining its internal workings. Regression testing tests whether a new fix addresses bug reports without reintroducing old bugs. Tools such as SVN and Github provide operations such as forking, merging, and syncing. In a similar spirit, a subsumption test in our work requires that past versions of the software are merged with the new fix and tested to see which bugs are fixed.

Examples of market-based platforms for software development include Bountysource, which is a funding platform for open-source software with 39,699 community members. Here users post bounties on issues they want solved while developers devise solutions and claim rewards. The minimum reward that can be contributed is \$5 and a reward paid for a solution can amount to over \$5000. Once a bounty is posted, a developer chooses the issue and begins work on it. Upon completion the developer submits a claim. During a two week verification period, backers vote to accept or reject the claim. If the claim is accepted the developer is paid the bounty. If an issue is closed without any resolution the bounty is refunded. Bountysource also organizes fundraisers to raise money for new features or costly issues requiring a significant amount of work.

A somewhat different type of platform, based on crowdsourcing contests, is Topcoder with over a million active members. Here companies that require software solutions are matched to a global community of programmers, and the latter compete in a contest with cash awards to provide the best solution that can address a specific client request. The Topcoder community works on a variety of products, including web and mobile application development, user experience design, predictive analytics, algorithm optimization, technical prototyping, as well as regular coding tasks (e.g., bug fixes, feature requests, testing, etc.). Bountify [10] is also a platform based on crowdsourcing contests. It focuses on coding tasks and questions–the website states, “The best tasks to post on Bountify are those with verifiable solutions. Clearly define your problem and specify the deliverables. Be specific about what you want and ensure that you can verify the correctness of the possible solutions you receive.” [11]. The reward range that can be offered is \$1 to \$100. A client posts the task and the attached bounty which expires after a week. Programmers must submit solutions within this timeframe. The client then chooses the best solution and awards the bounty to the winner. The client pays the full amount of the bounty to Bountify immediately upon posting the task. Interestingly the client is not refunded if none of the solutions submitted is acceptable. Instead the bounty is awarded to charity. Rather than providing fixes or features, Bugcrowd [12] is a security platform that has at its disposal a crowd of workers that includes security researchers and hackers. The crowd works to discover vulnerabilities in a client’s software. The client only pays for those vulnerabilites judged to be valid, and not for the effort expended to find them. Further, clients can specify a budget and reward range for continuous testing of their software. Bugcrowd helps clients set reward ranges by offering a pricing model assembled from historical data. For a comprehensive survey of crowdsourced platforms for software development as well as the associated literature, see [13].

The aforementioned platforms and practices are natural precursors to the market-based system proposed in this paper. Although we are inspired by these platforms, our design departs from them in several ways. Current platforms tend to address software tasks that are small and modular. The solutions tend to be human-verified and payment schemes consist of simple one-shot payments. As the growth in crowdfunded and crowdsourced models of production continues, it seems likely that platforms will become more sophisticated, addressing complex design issues such as incentivizing high quality work, handling inter-dependent tasks, and automating the market. In this paper we take that next step and design a system that incentivizes deep fixes using externally observable information only.

Open source development underlies billions of dollars in software production [14]. However, it is becoming increasingly clear that more resources are needed to support open source infrastructure. A recent study [15] points out that the boom in startups relies on digital infrastructure made possible by open source software. A lack of adequate funding and support can result in security breaches and interruptions in service. This real-world problem was anticipated in earlier economics research. Kooths et al. [16] argue that the lack of price signals in open source software production means that developers do not know users' valuations. Therefore the supply of open source goods does not fully align with users' needs. This raises the question of what are ways to design market mechanisms to fund digital infrastructure. These issues are further motivating factors for the line of research in the present paper.

In studying software economies we adopt a *mean field equilibrium* (MFE) methodology, which is an approximation methodology that is useful for analyzing the behaviour of large systems populated with self-interested participants. In large markets, it is intractable and implausible for individuals (in this case workers) to best-respond to competitors' exact strategies [17]. Instead the MFE approach assumes that in the long run the temporal patterns in agents' actions average out. Hence, agents can optimize with respect to long run estimates of the marginal distribution of other agents' actions. Using this methodology, we study the software engineering ecosystem under different parameterizations of the environment and different incentive designs. Our simulation results demonstrate that the basic premise in this work has merit: by using externally observable comparisons we can drive competition and get deeper fixes and higher utility. We conclude by drawing lessons for market design.

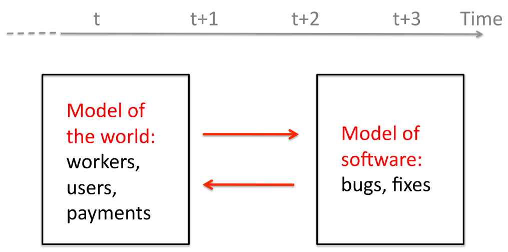
1.1 Main Contributions

- We contribute a dynamic model of the software ecosystem. In addition, we introduce the bit string language, a mathematical language that encodes the relationship between bugs and fixes. This is the first work to study incentives design for deep fixes.
- We introduce subsumption mechanisms–mechanisms that incentivize deep fixes by using externally observable information only. We design and analyze variants on subsumption mechanisms, namely *eager subsumption*, *lazy subsumption*, and *eager with reuse*. These mechanisms are all based on the underlying principle of checking whether submitted fixes subsume prior fixes, but with variations in the timing of checks and in whether replaced fixes are kept around, checked again later and possibly reused.
- We frame the problem in the context of the mean field methodology and obtain convergence computationally under different parameterizations of the environment. This work is the first to apply the mean field equilibrium concept to the software economy.
- We develop models of worker and user utility. The expected worker utility is estimated by look-ahead sampling and by using the mean field distribution to understand how other workers may act in the future. User utility is a metric that unifies several performance measures, such as a user's cost, wait time for a fix, and the depth of a fix.
- Our main result via the simulation study is that subsumption mechanisms perform robustly across a variety of environment configurations and satisfy important criteria for market design. In contrast, this is not the case for mechanisms without subsumption. In general, we find that installment-based payment schemes (including subsumption mechanisms) create incentives for deep fixes. Installment-based payment schemes that allow *transfers*, that is, unpaid installments from a fix are added to the reward for a later fix, perform even better. Transfers augment a low payment for a fix, making deeper fixes more profitable. Subsumption mechanisms pay in installments and allow transfers conditional on subsumption. The use of subsumption and fix-to-fix comparisons introduces competition amongst workers and produces deeper fixes and higher user utility. Subsumption mechanisms that *reuse* fixes achieve higher user utility because they allow for fixes without new payments. Surprisingly, mechanisms without installment payments are also able to produce deep fixes, but at low rewards only and with a high wait time.
- In conclusion we draw lessons for market design, both from the simulation study as well as from qualitative analysis of the mechanisms, and make recommendations regarding the suitability of the different mechanisms for various market design criteria.

2 Related Work

Prior literature has considered economic approaches towards related aspects of the software engineering industry, but with a different focus from this paper. For instance, the use of market-based approaches in improving vulnerability-reporting systems has been explored. Schechter [18] describes a vulnerability market where a reward is offered to the first tester that reports a specific security flaw. The reward grows if no one claims it. At any point in time the product can be considered secure enough to protect information worth the total value of all such rewards being offered. Ozment [19] likens this type of vulnerability market to an open first price ascending auction. While vulnerability markets as well as existing bug bounty programs (e.g., the Mozilla security bug bounty, and the recently launched "Hack the Pentagon" bug bounty program [20]) motivate testers to report flaws, such systems do not capture users' valuations for fixes. In particular this literature does not consider how to incentivize deep fixes–our work is the first to study this problem. Other papers examine the role of incentives in information security and privacy, and propose policies to improve the level of security and privacy [21], [22], [23], [24], [25], [26], [27], [28]. None of the aforementioned papers study large market dynamics or use the MFE methodology.

Hosseini et al. [29] address the problem of efficient bug assignment. They model the bug triager as an auctioneer and programmers as bidding agents in a first-price sealed bid auction. Agents bid on bug reports, where a bid value is based on the developer's past history of fixing bugs, the developer's expertise and speed, severity of the bug, and so forth. The bug is allocated to the agent with the highest bid. Le Goues and colleagues [3], [30], [31] share our view of software as an evolving, dynamic process. However where we approach software engineering from a market design perspective, they are influenced by biological systems and apply genetic programming for automated code repair. Several papers [32], [33], [34], [35] have studied open-source movements from the perspective of community formation and contribution. For instance, Athey and Ellison [32] look at the dynamics of how open-source projects begin, grow, and decline, addressing issues such as the role of altruists, the evolution of quality, and the pricing policy. In contrast,



t
t+1
t+2
t+3
Time
Model of the world: workers, users, payments
Model of software: bugs, fixes

Fig. 2. Overview of our model of the software ecosystem.

our work focuses on solving incentives problems that arise in the production of fixes. Other research has examined software from the point of view of technological innovation [36], [37], as well as from the point of view of the design and modularity of software [38], [39].

Mean field equilibrium derives its name from mean field models in physics, where large systems display macroscopic behaviour that is easier to analyze than their microscopic behaviour. MFE have been studied in a variety of settings in economics [40], [41], [42], [43] and control theory [44], [45], [46].

Our work is most closely related, from a methodological perspective, to a series of recent papers that analyze MFE in various market settings. Motivated by sponsored search, Iyer et al. [47] consider a sequence of single-item second price auctions where bidders learn their private valuations as they win auctions. The authors show that the agent's optimal strategy in an MFE is to bid truthfully according to a function of the expected utility. In a related paper, Gummadi et al. [48] examine both repeated second price and generalized second price auctions when bidders are budget constrained. The authors show that the effect of long-term budget constraints is an optimal bidding strategy where agents bid below their true valuation by a shading factor. Other settings have also been analyzed in the mean field context [17], [49], [50], [51]. These papers present a theoretical analysis whereas we take a computational approach and study a richer domain model.

The structure and performance of different contest architectures have been widely studied [52], [53], [54]. A series of papers [55], [56], [57] model crowdsourcing contests as all-pay auctions. However crowdsourcing contests are an altogether different scenario to ours. In our model, submissions are not simultaneous, and nor can we directly observe the quality of submissions and judge which submission is best. Instead the incentives in place ensure that the system evolves over time based only on competition and externally available information in a way that retains deeper fixes.

This work is a significantly extended version of a preliminary contribution presented at a workshop [58].

3 The Model of the Software Ecosystem

The model of the software ecosystem consists of two parts that interact with each other: a model of software bugs and fixes and a model of the world (see Fig. 2). We first describe the system of bugs and fixes, before presenting the dynamics of the entire ecosystem.

3.1 Modeling Bugs and Fixes

We consider an abstract model of software as a set R of independent *root causes* of some fixed cardinality, where each root cause generates a series of related bugs (see

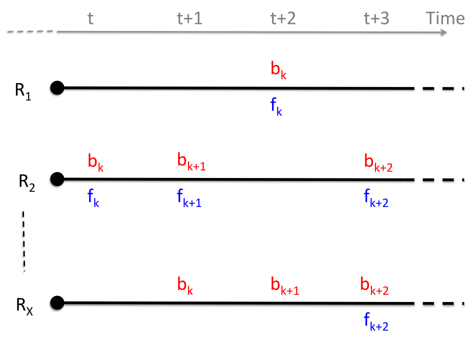


Fig. 3. Each root cause in set R generates bugs b, which in turn may receive fixes f.

Fig. 3). To draw an analogy with a real world scenario, a root cause may be thought of as a specific component or functionality of a software; for example, one root cause might be a synchronization error in the user interface component, while another root cause might be a buffer overflow in the graphics component. We develop a *bit string* model that captures how a particular root cause can generate bugs, and that encodes the relationships amongst bugs and fixes.

Each root cause is associated with a *bit string length* $l > 0$. The set of bugs belonging to this root cause are modeled through the set of $2^l - 1$ non-zero bit strings of length l. The set of *permissible fixes* for this set of bugs is modeled as the set of 2^l bit strings, including the $2^l - 1$ non-zero bit strings that address the bug as well as the zero bit string that models a worker who chooses not to submit a fix (referred to as the *null fix*). Each bit string encodes a different bug, and similarly a different fix (see Fig. 4). A larger length l signifies a root cause that generates more bugs.

The bit string representation of bugs and fixes combined with the rules defining their relationships gives us a compact mathematical language that we can use to capture a complex, interdependent phenomenon such as the one studied in this work. In our model, fixes pertaining to a particular root cause cannot be used to fix bugs generated by other root causes. Thus all relationships and properties are relevant for only those bugs and fixes that belong to the same root cause. In what follows, we refer to a bit whose value is 1 as an *ON-bit*.

Definition 1 (Valid fix). *A fix $f \in \{0,1\}^l$ is* valid *for bug $b \in \{0,1\}^l$ if it includes all the ON-bits in b, i.e., $f \geq b$ bit-wise. Thus an AND operation between f and b must result in b.*

We refer to the set of valid fixes for a bug plus the null fix as the set of *feasible* fixes for the bug. Different bugs can have different numbers of feasible fixes. Bug 1,110 has only 3 feasible fixes. In contrast bug 0001 has $2^3 + 1$ feasible fixes.

Example 1. A root cause with $l = 4$ can generate the set of bugs $\{0001, 0010, \ldots, 1111\}$. Consider bug $b = 1110$. Bug b is fixed by two fixes: $f_1 = 1110$ and $f_2 = 1111$. Fix 0111 cannot fix b. The set of feasible fixes for b is $\{1110, 1111, 0000\}$.

Definition 2 (Fix depth). *The* fix depth *of fix f refers to the number of ON-bits in the bit string of $f \in \{0,1\}^l$, and is denoted $|f|$.*

Continuing with the above example, f_1 and f_2 have fix depths equal to 3 and 4 respectively. We can now define, in

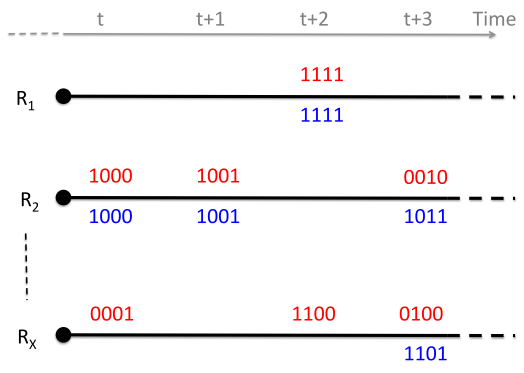


Fig. 4. Root causes R generate bugs, which may receive fixes. Bugs and fixes are represented as bit strings.

the context of the bit string representation, what constitutes a shallow or deep fix with respect to a given bug.

Definition 3 (Shallow fix). *Given a bug b, a* shallow fix *f is a valid fix for which $|f| = |b|$.*

Definition 4 (Deep fix). *Given a bug b, a* deep fix *f is a valid fix with $|f| > |b|$.*

In other words, a shallow fix is the fix that meets the minimal requirement of having the same ON-bits as the bug being fixed and no more. A deep fix is a valid fix that is not a shallow fix. The deepest fix not only fixes the bug in question but all bugs of that root cause.

Example 2. Consider bug $b = 1100$ generated by a root cause with bit string length $l = 4$. A shallow fix for b is $f_1 = 1100$. A deep fix for b is $f_2 = 1101$ or $f_3 = 1110$. The deepest possible fix is $f_4 = 1111$.

Next we define an ordering relationship between fixes. This will be crucial for reasoning about subsumption mechanisms.

Definition 5 (Ordering relation). *A fix f_k is* deeper *than a fix f_i, written $f_k \succ f_i$, if $f_k \geq f_i$ interpreted bitwise, with $f_k[j] > f_i[j]$ for at least some $j \in \{0, \ldots, l-1\}$.*

The ordering relation provides a partial order, as shown in Example 2, where we have $f_4 \succ \{f_2, f_3\} \succ f_1$. Here f_2 and f_3 are incomparable.

The bit string language is an abstraction of the ways in which bugs and fixes may relate and interact with one another in our setting. There are several plausible practical interpretations of the bit string model. For example, one interpretation may consider each ON-bit as corresponding to a location in code, and therefore a fix with more ON-bits would fix more code locations than one with fewer ON-bits. In this interpretation, a bug with a particular ON-bit could imply that a problem has occurred in that specific location and the stipulation that a valid fix must at least include the same ON-bits as the bug it addresses makes sense–a fix must at least fix the code location where the problem has occurred.

So far we have described how bugs and fixes relate to one another in the context of the bit string model. Moving on, we consider those properties that are externally observable, and to that end, introduce a key concept, that of subsumption.

Definition 6 [Subsumption relation]. *A fix f_k* subsumes *another fix f_i with respect to a set of bugs B, written $f_k \succsim f_i$, if the set of bugs fixed in B by f_k strictly contains the set of bugs fixed in B by f_i.*

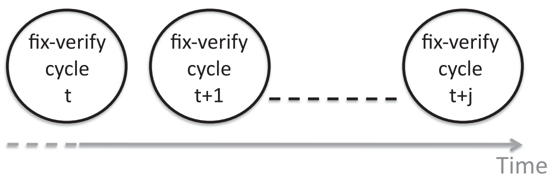


Fig. 5. A sequence of fix-verify cycles.

Continuing with Example 2, suppose bug b receives fix f_2. Now suppose the root cause generates another bug $b' = 0110$ which receives the fix $f' = 1,110$. Fix f' subsumes f_2, since f' fixes all the bugs fixed by f_2 as well as b'. However $f' \succsim f_2$ does not imply $f' \succ f_2$. Clearly, we have $f_2 \not\succsim f'$. But it may be that $f' \succ f_2$ or that f' and f_2 are incomparable, as is the case here.

3.2 Temporal Dynamics

We study a setting with discrete time periods, and a large population of workers who submit fixes. Users discover and report bugs, and may offer payment for fixes.

In each time period and for a given root cause, we randomly sample from all $2^l - 1$ bugs associated with the root cause, regardless of whether some of those bugs might be already fixed (i.e., sampling with replacement). A new bug enters the system only if the sampled bug is as yet unfixed and unreported. Note that an un-reported bug may be preemptively fixed by a fix already submitted for a different reported bug. This reflects a natural situation where a root cause may generate several bugs initially, but fewer and fewer as fixes accumulate because an increasing number of bugs are preemptively fixed.

Example 3. Consider a fix 1001, which fixes the set of bugs $\{1001, 1000, 0001\}$. Any bug that is not fixed by fix 1001 may appear after this fix is submitted on a particular root cause, such as a bug 1110.

Upon generation, a bug is associated with a total amount of reward that is provided by the user or users who report the bug (see Section 3.3.) A root cause $x \in R$ that has not generated a new bug in some period of time (this time period is a parameter of the model) is called *inactive* and is regenerated with probability $\beta_x \in (0, 1]$. When a root cause is regenerated, it is removed and replaced with a new root cause (i.e., one with all bugs yet to be generated) of the same bit string length. This removal and replacement models the root cause as having received deep enough fixes that it is now unlikely to generate more bugs, and that the user base shifts its attention to a new set of bugs. Moreover it may be that certain modules are no longer the focus of the user base and therefore retired, eliminating that root cause, and allowing for new modules and root causes to be created.[2]

The market dynamics are simulated through a sequence of *fix-verify cycles* that occur over time (see Fig. 5). Each fix-verify cycle takes place in a single time period t at a given root cause. Thus at each time period t we go round-robin

2. The presence of multiple, regenerating root causes ensures an infinite stream of bugs, also supports the stationarity in long-run statistics that motivate the MFE approach.

through root causes, executing fix-verify cycles at each root cause. A fix-verify cycle for a particular root cause proceeds as follows:

1) The root cause is queried once to see if it generates a new bug, this bug also associated with a total available payment as provided by the associated user.
2) A worker is selected at random and tentatively assigned to an unfixed or newly generated bug.
3) The worker submits a feasible fix, maximizing expected utility given a model of the future.
4) The fix is verified by the market infrastructure in regard to the assigned bug.
5) The total amount to be paid to the worker over some time horizon is calculated. The worker is paid in a single step, or in installments, depending on the specifics of the payment rule.

We make four main modeling assumptions. First, a total payment amount is associated with a bug at the time the bug is generated. No further payments are added to this bug later in time. This is a reasonable modeling assumption since different payments can be flexibly associated with a bug when generated, capturing user base disutility. Second, only one new bug is introduced per root cause per period and only one worker is matched to a bug for any given root cause. These two assumptions simplify the analysis, avoiding the possibility of two fixes being submitted on the same root cause in any one period. While work within a particular root proceeds in sequential order, work across different roots may happen in parallel. Third, a matched worker submits a fix in the same time period that the worker is assigned the bug. This is a reasonable modeling assumption given that time periods are long enough. Fourth, the likelihood that the same worker is repeatedly assigned bugs belonging to the same root cause is low, hence the worker considers only those scenarios where future work on the same root cause is performed by other workers. In settings with a large number of workers and root causes, this assumption is sound.

3.3 Users and Workers

The users in our model are not strategic. Rather, users are modeled as being willing to make a payment that corresponds to a disutility for a bug or a value for a fix. This disutility is associated with a specific bug and thus associated with a root cause that the user cares about. A user may represent the aggregate utility of the user base for a fix but for simplicity we refer to a single user.

Let $r \in [r_{min}, r_{max}]$ denote a user's *instantaneous disutility* associated with a bug. A user's *realized value* for the sequence of fixes that occur following the report of a bug is a function $v(r, \hat{t})$ of the disutility r and the wait time $\hat{t}$, where v is decreasing in $\hat{t}$ and $v(r, 0) = r$, and is given by $v(r, \hat{t}) = \delta^{\hat{t}} r$ for discount factor $\delta < 1$. Thus the longer a user must wait for a fix, the lower the realized value. A user's realized value is at most r.

If disutility $r = r_{min}$ then the user is called a *short-term* user, whereas if $r > r_{min}$ then the user is a *long-term* user. In the time period that a fix is first submitted, both short-term and long-term users get an initial installment of realized value, defined as $v(r_{min}, \hat{t})$. This represents the value of any fix, even a shallow fix. This is also the total value of a short-term user, modeling the idea that the user was only interested in a shallow fix. A long-term user receives the remaining value $v(r, \hat{t}) - v(r_{min}, \hat{t})$ in h^* equal installments, or fewer since payments are halted if an additional bug appears on this root cause subsequent to $\hat{t}$.

The *utility* of a user is the total value net the cost of payment. Since users are not strategic payment is assumed to equal the amount of the instantaneous disutility r. The maximum utility a user can get is zero (representing realized value r, net payment r), and the utility is generally negative.

The utility model has the following properties:

1) If the fix never occurs, the user utility is $-r$.
2) If the fix occurs right away and all installments are received, the user utility is $-r + r = 0$.
3) Fixes that occur after a user's bug report have some associated disutility, and thus the user utility is negative.

In this way, the user utility is a metric that unifies several performance measures, including the wait time for a fix, the side-effects of a fix (a deep fix pre-empts future bugs), and the loss in value in any given period without a fix. In our model the user commits to a payment even if the work is not done.

A worker, chosen at random, is given the option of working on a bug, in what is referred to as a *tentative assignment* (or tentative match). The worker's decision problem is to determine which fix to submit, if any. A decision about whether or not to invest effort now does not foreclose the ability to work in the future because we assume that a fix (if any) is produced within the current period and because we assume a worker ignores the possibility of being matched to the same root cause in the future.

To produce a fix, worker w_j incurs cost $c_j(|f_k|)$, which is a non-decreasing function of the number of ON-bits in f_k. A simple example is a linear cost function, with $c_j = |f_k|$. We assume the worker discounts any future payments according to discount factor, $\delta < 1$, indicating a preference for earlier payments over later ones. The utility derived by a worker from submitting a fix on a bug is equal to the discounted sum of the payments received, starting from the time period in which the fix is submitted, net the worker's cost for the fix f_k.

Given a model of the behaviour of other workers, a model of the environment by which bugs and user payments are generated, and a particular design for an incentive mechanism, a worker chooses a fix (perhaps null) to maximize total expected, discounted utility. In what follows, we describe the principal mechanisms that we study, and make explicit the way in which the worker computes the expected utility.

4 Subsumption Mechanisms

Consider an instantiation of the model where Step 4 of the fix-verify cycle involves a specific kind of correctness check. In a *subsumption mechanism*, this step involves a check for whether the current fix subsumes any previous fixes. If so, the subsumed fixes are replaced by the current fix. In addition, the worker's payment may increase if the fix subsumes previous fixes. In this way, a worker now competes with other workers who fix subsequent bugs on the same root cause. Thus the model captures indirect competition because a later worker might produce a deeper fix for a subsequent bug k, that not only fixes k but also subsumes some other worker's fix for an earlier bug on the same root cause. To allow for this take-over of earlier payment, the payments in a subsumption mechanism occur over a fixed number of installments over time.

Generally, a subsumption mechanism is defined as a mechanism that uses subsumption relations in at least one of two ways:

i to make externally observable comparisons between fixes, where one fix may subsume another;

ii to determine payments according to the subsumption ordering of fixes.

We focus on three variations on subsumption mechanism design, referred to as *eager*, *lazy*, and *eager with reuse*.

4.1 Eager Subsumption

Eager subsumption performs externally observable checks for subsumption using only the bugs and fixes seen *so far*. Consider a fix f_k for bug k at time t. Fix f_k subsumes a prior fix f_i if it fixes all bugs fixed by f_i from the set of bugs seen until time t. If fix f_i is subsumed then it is discarded. Since subsumption is concluded without waiting to see the entire range of bugs and fixes, eager subsumption only approximates the ordering relationship between fixes.

For eager subsumption, we use a simple payment rule that pays out equal installments of the total payment over a fixed number of time periods, h^*. Let $i < k$ be the index into all fixes f_i that occurred at a time prior to f_k on this root cause. Let $Q(t)$ represent the set of all bugs at time t on this root cause that were generated, remained unfixed, and are now fixed by f_k. Let $\hat{r}_i(t)$ denote the total payment still to be made to the worker associated with fix f_i in periods t and forward.

The total payment remaining to be made for a fix f_k for bug k submitted at time t is,

$$\hat{r}_k(t) = \sum_{i=1}^{k-1} \hat{r}_i(t) I_{\{f_k \succsim f_i\}} + \sum_{q \in Q(t)} r_q + r_k, \tag{1}$$

where $\hat{r}_i(t)$ is the remaining unpaid payment (if any) at time t of a previous fix f_i subsumed by f_k, r_q is the payment associated with an unfixed bug q that f_k fixes, and r_k is the payment associated with bug k. This total payment is made in installments over h^* periods. The payment in period t', with $t \leq t' \leq t + h^*$, is,

$$p_k(t') = \frac{\hat{r}_k(t)}{h^*}. \tag{2}$$

In particular, the worker is paid an installment each time period until all h^* installments are exhausted or until the worker's fix is subsumed by a new fix, whichever occurs sooner. In the latter case, the remainder of the worker's payment is transferred to the subsuming fix. Hence if the worker's fix is only subsumed after h^* time periods have passed, or not subsumed at all during the lifetime of the root cause, the worker is paid $\hat{r}_k$ in its entirety. If a root cause is regenerated while a worker still has outstanding payments, the worker is paid the remainder in full.

Eager subsumption can give false positives in regard to the ordering of fixes.

Example 4. Suppose bug $b_1 = 0001$ is reported, attracting fix $f_1 = 0011$. Next suppose bug $b_2 = 1000$ comes along on the same root cause, with fix $f_2 = 1001$. The eager mechanism concludes that f_2 subsumes f_1 as it fixes all bugs seen so far. Thus f_1 is discarded and replaced by f_2. f_2 receives payments from the reward on b_2 plus transfer from f_1. Now suppose bug $b_3 = 0010$ arrives. f_1 can fix b_3 but not f_2. Had f_1 been retained it would have precluded the appearance of b_3. Instead b_3 enters the market and awaits a new fix.

4.2 Lazy Subsumption

Lazy subsumption decreases the number of false positives by delaying judgement. Although a check for subsumption is performed when a new fix is submitted by a worker, subsumption relations are finalized only *when a root cause regenerates*. Thus, given the same instance, the partial ordering of fixes ultimately obtained with lazy subsumption may be different than the one obtained with eager subsumption.

To illustrate, let us consider a single fix f_k for bug k submitted at time t. Suppose f_k fixes all bugs fixed by a prior fix f_i that have appeared until time t. At this point eager subsumption would decide that f_k has subsumed f_i, whereas with lazy subsumption this decision is deferred. Continuing, assume that f_i has not subsumed any fixes. Payments towards f_i are stopped and f_i is replaced by f_k, whose payments begin. But in contrast to eager subsumption, f_i is not discarded permanently, but is instead removed from the system and placed on a *waiting list*. Moreover f_k's payments do not yet include the transfer of the remaining unpaid amount $\hat{r}_i(t)$ from f_i.

The total payment with lazy subsumption is also given by Equation (1). At time t the worker responsible for f_k begins to receive installments of the portion of the total payment represented by the second and third terms in Equation (1), that is, $\sum_{q \in Q(t)} r_q + r_k$. However while installments for the rest of the payment begins, the payments associated with the first term in Equation (1) are frozen until the root cause regenerates at time t^*. Only then does the mechanism determine, based on all bugs generated and fixes submitted up until time t^*, whether fix f_k subsumes f_i. At that point only, the lazy subsumption scheme redistributes the payment $\hat{r}_i(t)$ to the right party: if f_k is found to have subsumed f_i at the time the root is regenerated, then $\hat{r}_i(t)$ is transferred to the worker who submitted f_k, otherwise it is retained by the worker who submitted f_i.

Example 5. Suppose bug $b_1 = 0001$ is reported, attracting fix $f_1 = 0011$. Next suppose bug $b_2 = 1000$ arrives, attracting fix $f_2 = 1001$. The lazy mechanism considers that f_2 may have subsumed f_1. Thus f_1 is placed on a waiting list. Payments to f_1 are frozen and f_2 receives payments from the reward on b_2 but not the transfer from f_1. Now suppose that bug $b_3 = 0010$ arrives. f_1 is reused to fix b_3. At the end of the root's lifetime, lazy determines that f_2 did not subsume f_1 and f_1 gets the remaining payments that were frozen.

An interesting feature of lazy subsumption is that it *reuses* fixes that are on the waiting list. In Example 5, lazy subsumption would attempt to reuse f_1 when b_3 appears. If f_1 can be successfully reused, then it is simply reinstated into the current, working set of fixes and the bug does not enter into the economy and no new payment amount is brought into the market. Hence, only if none of the fixes on the waiting list can fix b_3 does the mechanism revert to the standard scenario where the bug is listed and a new payment is associated with it. It is via reuse that subsumption relations, and consequently the partial ordering of fixes, are updated during a root's lifetime.

Thus, the process of determining subsumption in the lazy mechanisms proceeds in two stages, consisting of an *eager check* followed by a *lazy check*:

1) When a new fix is submitted a check is done, as in eager subsumption, to ascertain if the new fix subsumed any prior fixes based on the bugs and fixes seen so far. The purpose of this check is to freeze payments that may be subsumption transfers–if the new fix has subsumed prior fixes then their payments are frozen. Although any subsumption transfers (the first term in Equation (1)) are frozen, the new fix starts to receive installments from payments associated with any previously unfixed bugs (the second and third terms in Equation (1)).
2) When a root regenerates, subsumption relations are finalized by examining the final partial ordering of fixes. The purpose of this check is to determine how to distribute the frozen payments. This step is carried out by retrospectively stepping through the arrival times of fixes and computing payments according to the subsumption relations revealed by the final, partial ordering across fixes.

The difference with the eager mechanism, given the same system state, is twofold: i) the stricter criteria of judging subsumption means that the first term in Equation (1) might lead to a smaller payment, and ii) there is a delay in paying out subsumption transfers, thus the per time period payment does not include the first term in Equation (1) until the root regenerates. Note that subsumption checks are performed only in the case where a new fix is submitted by a worker. No such checks are performed when a fix from the waiting list is reused to fix a bug and reinstated into the current, working set of fixes. The goal here is simply to alleviate the mistakes that can be made by eager subsumption in determining subsumption without having seen more of the set of bugs that a root cause may generate. More than one fix in the waiting list may be able to fix a new bug– in this instance we use a tie-breaking rule, giving priority to the most recent waiting fix. Because lazy subsumption finalizes subsumption relations when a root cause regenerates, which may happen before all bugs are generated, the subsumption relations concluded by the lazy approach remain an approximation of the true ordering relation amongst fixes.

4.3 Eager with Reuse

This hybrid mechanism works in a similar way to the eager subsumption mechanism and uses the eager payment rule, with total payments given by Equation (1). However it deviates from the eager mechanism in that subsumed fixes are not discarded. Instead they are kept on a waiting list and reused as is done in lazy subsumption. What this means is that there are time periods when a new bug is fixed via a reused fix and the bug does not enter the market or get associated with a new payment amount. The effect is as if a new bug was not generated at all, and instead was preemptively fixed by an existing fix in the system (which would have been the case had the reused fix not been removed from the set of current fixes in the first place). Accordingly, any eager installments that are being paid out proceed as if a bug was not generated in this time period.

Example 6. Continuing with Example 4, f_2's payments include the remaining unpaid payment of f_1 as well as the payment for b_2. Payments are made as per the eager subsumption rule. However in eager with reuse, fix f_1 is not discarded, instead it is reused when b_3 appears and therefore a new payment is not associated with bug b_3.

4.4 Baseline, Non-Subsumption Mechanisms

In this section, we present instantiations of the fix-verify cycle that do not involve subsumption. These provide baselines against which to compare the subsumption mechanisms.

- *Myopic mechanism*. The worker is paid in full as soon as submitting a valid fix. Let $Q(t)$ represent the set of all generated bugs at time t on this root cause that are now fixed by f_k. The total payment for a fix f_k for bug k submitted at time t is,

$$\hat{r}_k^m(t) = \sum_{q \in Q(t)} r_q + r_k, \tag{3}$$

where r_q is the payment of a formerly unfixed bug q that f_k has now fixed, and r_k is k's reward.
- *Installment mechanism*. The total payment available to be paid for a fix is described by Equation 3. However the worker is paid in a fixed number, h^*, of equal installments that stop as soon as a new bug of the same root cause appears. Any remaining payment goes unused and can be considered to be "wasted". The payment in time period t', where $t \leq t' \leq t + h^*$, is equal to $\hat{r}_k^m(t)/h^*$.
- *Installment with transfer*. This mechanism functions like the installment mechanism, but takes one step closer to eager subsumption by allowing transfers of payments from one fix to another. Installments that remain unpaid when a new bug appears are not thrown away. Rather, they are added to the payment for the new bug. The total payment available to be paid for a fix f_k for bug k submitted at time t is,

$$\hat{r}_k^n(t) = \hat{r}_i(t) + \sum_{q \in Q(t)} r_q + r_k, \tag{4}$$

where $\hat{r}_i(t)$ is the remaining unpaid payment at time t of the previous fix f_i whose installments were interrupted by k, and the other terms are as defined in Equation (3). Because bugs appear in sequence, and payments are interrupted as soon as the next bug appears, there can only be one such earlier fix that is currently receiving payments. The payment in time period t', where $t \leq t' \leq t + h^*$, is equal to $\hat{r}_k^n(t)/h^*$.

4.5 Discussion

Recall that a new, unreported bug may be preemptively fixed by a fix already existing in the system for a reported bug. We refer to the set of submitted fixes that are currently present in the system as the set of *active* fixes. The composition of the set of active fixes differs amongst the different mechanisms.

Because the myopic and installment mechanisms do not perform fix-to-fix comparisons and thus do not eliminate any submitted fixes from the system, the set of active fixes comprises all fixes submitted so far on a root cause. Hence at time t all fixes submitted at a root cause up until time t are used to preemptively fix new bugs in the myopic and installment mechanisms.

In the case of eager subsumption, subsumed fixes are permanently deleted from the system. Therefore the set of active fixes at time t is equal to the set of all fixes submitted so far minus those fixes that have been subsumed by time t. As a result, the preemptive power of eager subsumption may be lower than that of the myopic and installment mechanisms, specifically when there are false positives as shown in Example 4.

Turning to eager with reuse and lazy subsumption, although subsumed fixes are removed from the system, they are not permanently deleted as they are in eager subsumption. Rather, subsumed fixes are stored in a waiting list. The set of active fixes at time t is the same as in eager subsumption: it comprises the set of all fixes submitted so far minus those fixes that have been subsumed by time t. However, when a new bug is generated that could not be preemptively fixed by any active fix, the eager with reuse and lazy mechanisms check whether any fixes stored in the waiting list may be reused. The effect is that new bugs are introduced into the market at time t only if no fix in the set of all fixes submitted at a root up until time t (i.e., the set of active fixes and the set of subsumed fixes) has already fixed them.

5 Mean Field Equilibrium

In a subsumption mechanism, workers must contend with competing workers who may subsume their fixes, thereby curtailing their payment installments. Thus the worker faces a naturally strategic situation. The deeper the fix submitted the less likely it is to be subsumed and payments curtailed. But, deeper fixes cost the worker more. A worker in this market must decide how to best-respond to competitors' play in order to maximize expected utility.

We study subsumption mechanisms in a *mean field equilibrium*. MFE is an approximation methodology suited to large market settings, where keeping track of the strategies of individual agents becomes prohibitive. As the number of agents grows large, it is reasonable to assume that agents optimize instead with respect to long run estimates of the marginal distribution of other agents' actions. In particular, each worker best-responds as if in a stationary environment. The equilibrium aspect of MFE is that it insists on a consistency check: the statistics of the marginal distribution must arise as a result of agents' optimal strategies.

5.1 Adapting MFE to Our Setting

Recall that a worker assigned a bug must decide which fix to submit, if any. In a subsumption mechanism this strategic decision amounts to estimating the subsumption time of each valid fix, since this affects the distribution over future payments as a result of submitting a fix now.

Each worker models the future as though there is an i.i.d. distribution D of fix depths submitted by others, where the set of possible fix depths is $\{0, \dots, l\}$ (including null fixes). In addition, a worker assumes that all fixes associated with a particular fix depth occur with equal probability. This induces a probability distribution over the set of all 2^l possible fixes. Finally, a worker infers the conditional distribution on feasible fixes for a specific bug (noting that only some fixes can be valid).

For concreteness, we can consider eager subsumption, and normalize time, so that a fix is considered to be submitted in period $t' = 0$. Let y_{jk} denote a worker's utility for submitting f_k to fix bug k. The worker's expected utility is

$$\mathbb{E}[y_{jk}] = \mathbb{E}_h\left[\sum_{t'=0}^{h} \delta^{t'} p_k(t') - c_j(|f_k|)\right], \tag{5}$$

where $p_k(t')$ is the payment in time period t', and $h = min(h^*, H)$ where H is a random variable representing *subsumption time*, defined as the number of time periods before f_k is subsumed.

Subsumption times vary amongst the fixes submitted, because each fix precludes a different set of bugs, which affects the time till the next bug (and its fix) appears. Moreover each fix permits a different set of future fixes that can subsume the fix. For any particular system state, the worker chooses a fix f^* such that,

$$f^* = \arg\max_{f_k} \mathbb{E}_h\left[\sum_{t'=0}^{h} \delta^{t'} p_k(t') - c_j(|f_k|)\right]. \tag{6}$$

The fix that maximizes the worker's expected utility is determined by estimating the distribution on subsumption time, and thus estimating Equation (5). To estimate the utility of each feasible fix, a worker samples possible future trajectories assuming that workers behave according to belief D, given the environment model that dictates how bugs and new root causes are generated, and arrives at a sample over possible subsumption times and thus an estimated, total expected discounted utility.

In particular, given a fix f^* submitted in the current time period, bugs not fixed by f^* might appear in future time periods. A future fix to one of these bugs might subsume f^*, thereby curtailing the number of payment installments. In order to estimate the expected utility of a fix f^*, the worker simulates the software economy environment according to the MFE a number of times. Several trajectories are sampled in order to realize subsumption time and arrive at an estimate of the utility the worker can expect when submitting fix f^* (see Fig. 6). We stop sampling trajectories once the worker's estimated utility converges to within a confidence interval. This look-ahead sampling technique is inspired by the sparse sampling algorithm for estimating optimal policies in Markov decision processes [59].

Let $\Phi(D)$ be the long-term, empirical distribution given that workers assume model D, and apply their optimal strategy, given the dynamic model of the root cause and bug environment.

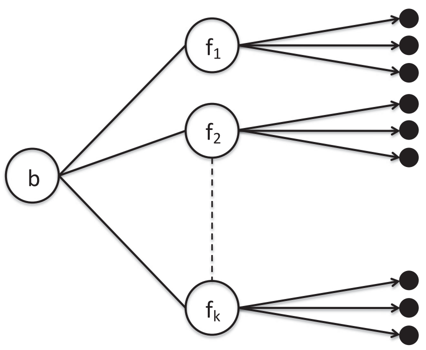


Fig. 6. Look-ahead sampling to compute an approximately optimal, best response given a worker is matched with bug b. Fixes f_1 through f_k are possible fixes by this worker, and for each one the worker rolls out possible futures.

Definition 7. *An MFE in the software economy is a distribution D on fix depths such that $\Phi(D) = D$.*

Continuing, for a given environment and mechanism, we estimate the MFE following the approach described in the algorithm in Fig. 7. To determine convergence of the system to a MFE we compare distributions using a likelihood ratio test (details are in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2018.2842188). The test for convergence is performed once after letting the algorithm run for a long period. This period of time is determined at design time, where we look for a number of periods such that reliably, over and over again when we rerun the experiment, the test passes. This avoids the multiple testing problem.

```
1: Initialize time period $t = 0$.
2: Initial distribution on fix depths (worker beliefs), $D_0$.
3: while convergence not reached do
4:    $t = t + 1$.
5:    for each root cause do
6:       Regenerate root cause according to its regeneration
         probability.
7:       Try to generate a new bug.
8:       if bug generated then
9:          Generate payment by sampling from reward distribution.
10:         Assign a worker at random to bug.
11:         Verify fix submitted by worker, if any.
12:         Update the empirical distribution on fix depths
            and compute $D_{t+1}$.
13:         Make payment installments on this root cause.
14:      end if
15:      Update user's realized utility.
16:   end for
17: end while
```

Fig. 7. Algorithm to estimate MFE.

6 Simulation Study

The goal of the simulation is to understand the characteristics of the model and mechanisms proposed in this paper. We hope to draw lessons for market design and distill useful insights for practitioners. Some questions that motivate our study are as follows: How do the simple mechanisms of myopic and installment fare relative to eager subsumption? Does reuse improve performance? What is the relative performance of the three subsumption mechanisms?

Because there are several parameters in the model of the software ecosystem, where each can take on a range of values, and because any combination of parameter settings yields a different environment, there are a large number of environments that can be used for simulations. We hope to study a more comprehensive set of environments in future work. For the present paper, we consider the following environment parameters: Root causes are associated with bit string length 4. Bugs are generated from a uniform or geometric distribution. The worker uses either a linear or exponential cost function, and the worker's cost c for the first ON-bit is sampled uniformly at random from different ranges. The number of installments h^* is either 5 or 10. Unless otherwise stated, the discount factor is 0.8. Users are either all short-term, or a mixture of short-term and long-term with user disutility sampled uniformly at random from different ranges.

The performance of the mechanisms is measured according to the following metrics, which are averaged over all observations for each root cause over a number of periods after the system has converged in simulation to an approximate MFE:

- *Percentage of immediate fixes*: percentage of all bugs generated at a particular root cause that receive any non-null fix in the same time period that they appear;
- *Percentage of immediate deep fixes*: percentage of all generated bugs that receive a deep fix in the same time period that they appear;
- *User cost*: total payment (equivalent to instantaneous disutility r) paid out towards fixes by all users of a particular root cause;
- *User utility*: average user utility per bug generated, averaged over all bugs seen during a length of time;
- *Variance in worker utility*: variance in the worker utility per bug fixed, averaged over all bugs seen during a length of time.

A good mechanism produces a high percentage of immediate and deep fixes, provides high user utility, involves low levels of instantaneous disutility, and exhibits low variance in worker utility which guarantees that the utility workers get from participating is stable.

6.1 Simple Mechanisms

We first compare the performance of eager subsumption, which is the simplest subsumption mechanism, against the myopic and the installment mechanisms. In this simulation, we adopt the following *Environment I*:

Bugs are generated uniformly at random, the number of installments is 10, and the worker cost function is linear, with each worker randomly sampling cost for initial bit, $c \sim uniform(1, 2)$. This environment is parameterized by the instantaneous disutility r for users.

To understand the ratio of the instantaneous disutility to cost in this environment, suppose the disutility is 10. Then, for an average initial bit cost of 1.5, we have a ratio of $10/1.5 = 6.67$.

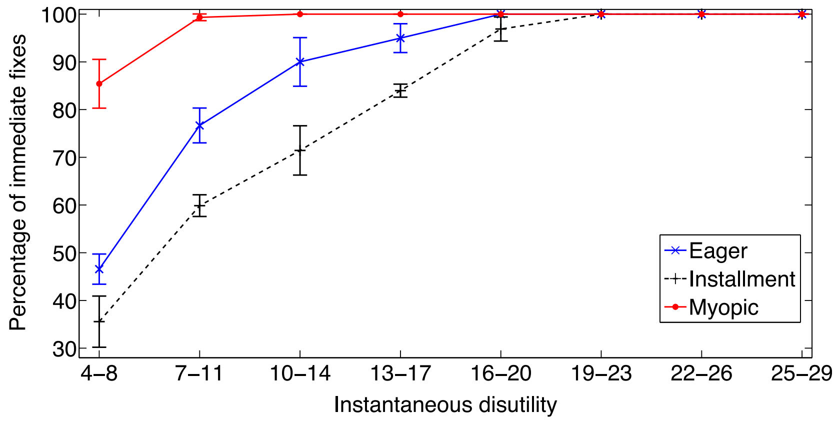


Fig. 8. The percentage of immediate fixes in simple mechanisms in Environment I, for different instantaneous disutility ranges.

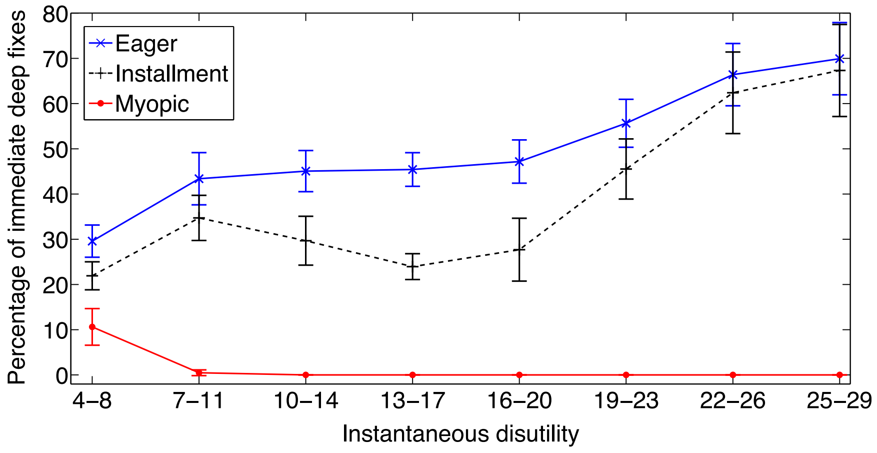


Fig. 9. The percentage of immediate deep fixes in simple mechanisms in Environment I, for different instantaneous disutility ranges.

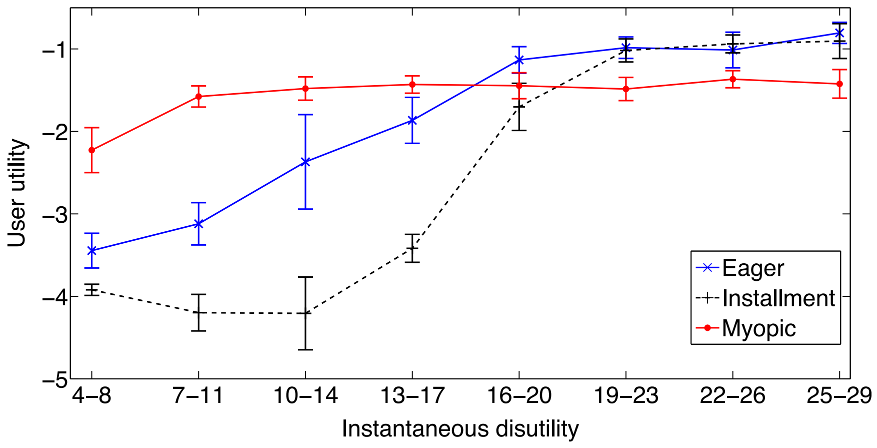


Fig. 10. The user utility in simple mechanisms in Environment I, for different instantaneous disutility ranges.

At relatively low levels of instantaneous disutility (and thus low payments), the eager and installment mechanisms, which make payments for a valid fix over time, do not cover the worker's cost. This is because installments are discounted over time, and because payments may be interrupted early due to a new bug appearing or when a fix is subsumed. Bugs accumulate until the combined payment for covering multiple bugs with a single fix is high enough. That bugs linger is seen in Fig. 8, which shows that less than 100 percent of bugs receive immediate fixes at lower payment values. In comparison, myopic pays the entire payment when a valid fix is submitted. This leads to more immediate fixes.

On the other hand, the equilibrium strategy of a worker in the myopic mechanism is to produce the shallowest possible fix. This is a best response because the worker is paid in full as soon as he submits any valid fix. Because of this, the performance of the mechanism levels off once fixes are affordable, and remains unresponsive to a further increase in payment (see Figs. 9 and 10). The installment mechanism fares better than the myopic mechanism when the payment is high but lags behind eager subsumption at lower

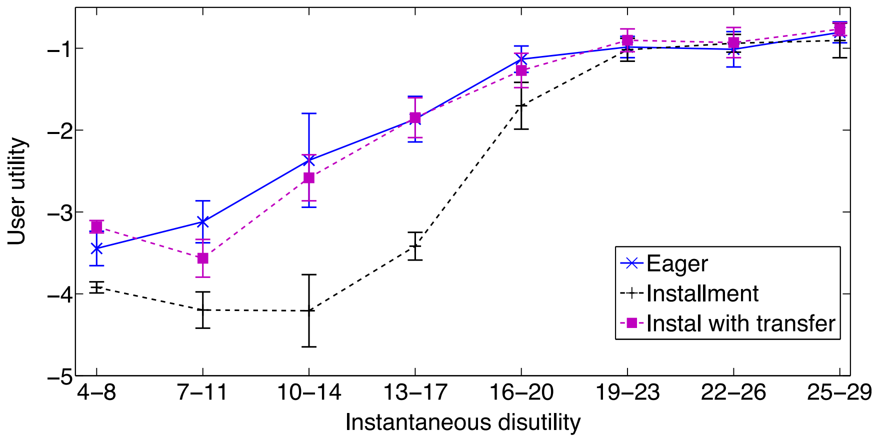


Fig. 11. User utility in Environment I for the eager, installment, and installment with transfer mechanisms.

payments. Without transfers, the installment mechanism cannot augment the low payment for a current fix with any remaining unpaid payment from a past fix. These results show the need for something more than the basic myopic and installment mechanisms.

6.2 Installment with Transfer and Eager with Reuse

Fig. 11 shows the results when we include installment with transfer in place of myopic. Installment with transfer provides higher user utility than the installment mechanism. This is as expected, because the amount transferred can accumulate for bugs appearing later on a root cause. For the parameter settings considered here, the performance of installment with transfer bridges the gap between that of installment and eager subsumption.

In order to create opportunities for the reuse of subsumed fixes, we modify the environment. Environment II is defined as:

> *Bugs are generated according to a geometric distribution, and worker cost is an exponential function in the depth of a fix. When bugs are generated uniformly at random and worker cost is linear, reuse is often precluded because we get bugs of a high bit string value, receiving deeper fixes, arising earlier in a root's lifetime. For reuse to occur, we require fixes that are deep enough to be reused, but not so deep that they are hard to subsume. The worker's cost for the first ON-bit is $c = 1$, and we set the number of installments to be $h^* = 10$. Bug generation is sped up, so that a root cause is queried five times, to see if it generates a new bug in a time period, instead of just once. This captures a scenario where a technology is newly adopted and may contain a large quantity of bugs initially. This environment is parameterized by the instantaneous disutility range for users.*

See Figs. 12 and 13. Installment with transfer produces a lower percentage of deep fixes than the other mechanisms. Like the installment mechanism, installment with transfer stops paying a worker as soon as the next bug appears. It follows that the portion of the payment that the worker can expect to keep is partly influenced by the rate of arrival of bugs. Moreover submitting the deepest possible fix on a root cause may not be profitable given the worker's cost and available payment. Hence the worker's expected payoff for submitting deeper fixes is reduced.

Environment III is defined as:

> *Workers have an exponential cost function, with each worker's cost $c \sim uniform(2.5, 3)$, bugs generated from a geometric distribution, and the number of installments is 10. In this environment we consider a higher worker cost than heretofore. This environment is parameterized by the instantaneous disutility range for users.*

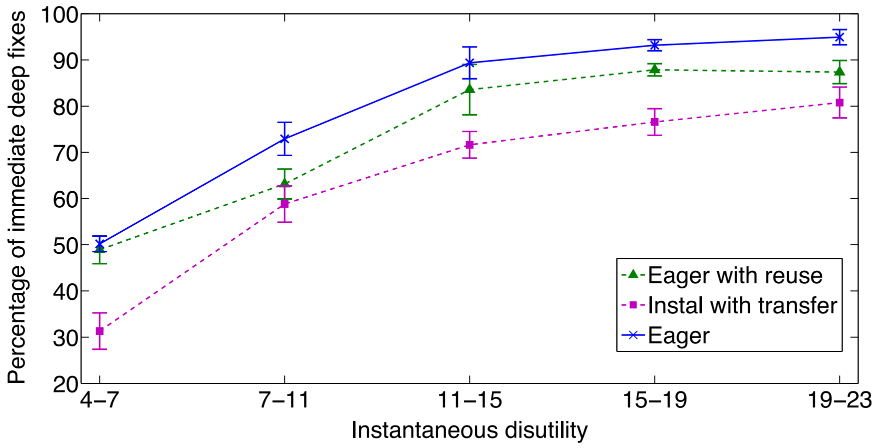


Fig. 12. The percentage of immediate deep fixes in Environment II, for different instantaneous disutility ranges, and comparing eager with reuse, instalment with transfer, and eager.

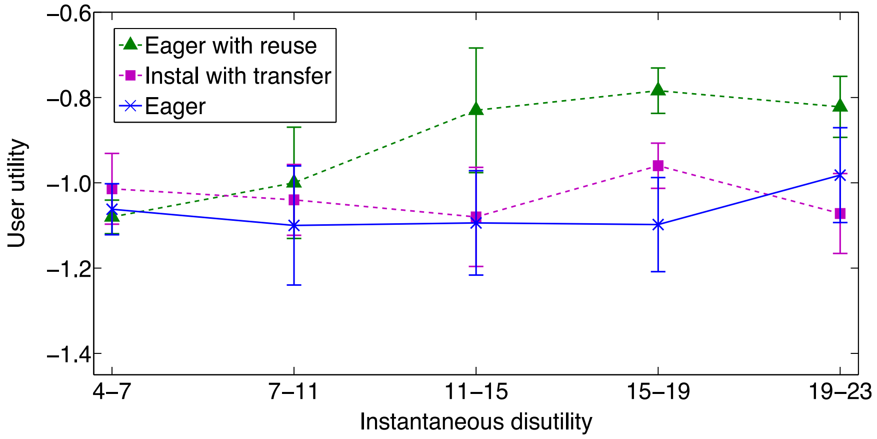


Fig. 13. The user utility in Environment II, for different instantaneous disutility ranges, and comparing eager with reuse, instalment with transfer, and eager.

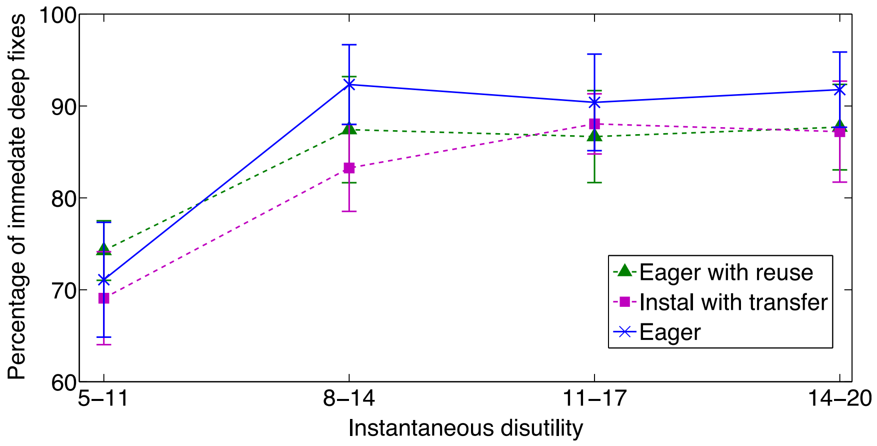


Fig. 14. The percentage of immediate deep fixes in Environment III, varing the instantaneous disutility range, and comparing eager, installment with transfer, and eager with reuse.

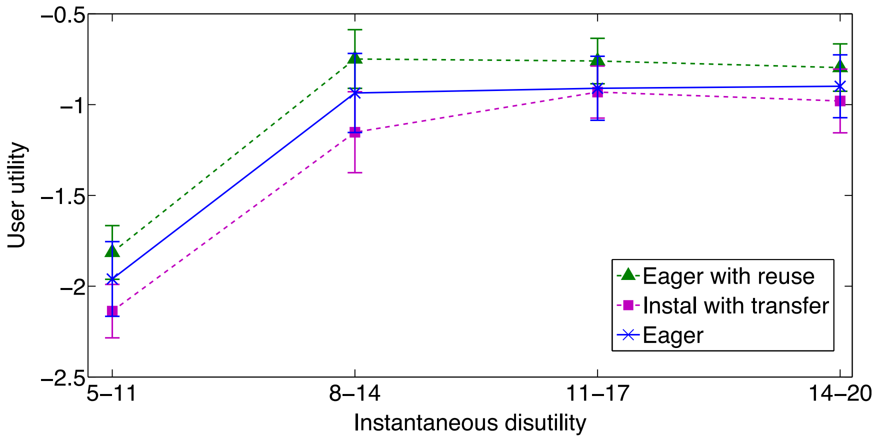


Fig. 15. User utility in Environment III, varying the instantaneous disutility range, and comparing eager, installment with transfer, and eager with reuse.

See Figs. 14 and 15. In this setting, eager with reuse does best with regard to user utility, whereas the eager mechanism performs better in terms of percentage of deep fixes. By deleting fixes that turn out to be false positives, eager reduces its ability to preemptively fix new bugs instead of generating them. This places eager subsumption at a disadvantage with respect to metrics such as user utility. Eager with reuse enhances the eager mechanism's performance with the ability to reuse subsumed fixes instead of discarding them.

6.3 Lazy Subsumption

For this simulation we work with environment IV:

Bugs are generated from a geometric distribution, the number of installments is 10, and the worker cost model is an exponential function, with each worker randomly sampling $c \sim uniform(3, 4)$. To compare the three subsumption variants, we consider an environment with relatively high worker cost. This environment is parametrized by the user instantaneous disutility range.

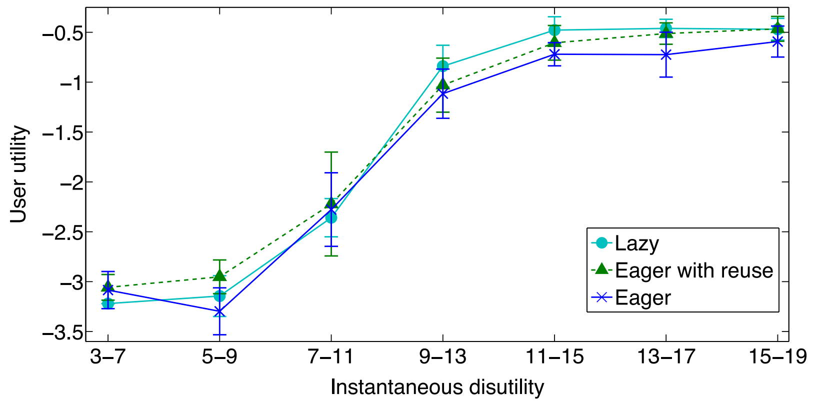


Fig. 16. User utility in Environment IV, varying the instantaneous disutility range, and comparing the lazy, eager, and eager with reuse mechanisms.

See Figs. 16 and 17. As a consequence of reusing subsumed fixes, the eager with reuse and lazy subsumption mechanisms achieve lower user cost, resulting in higher user utility. Recall that the lazy mechanism finalizes subsumption relations only when a root cause regenerates, and then redistributes payments. In order to better understand the performance of lazy subsumption, we adjust for the effect of discounting on later payments (by increasing redistributed amounts by a small quantity), thereby levelling the playing field. In general lazy subsumption performs as well as eager with reuse for the metrics examined in this study. Because the lazy mechanism uses more conservative checks, it is harder to subsume fixes and likewise harder to be subsumed. As a result lazy subsumption exhibits a smaller variance in worker utility than eager with reuse. For instantaneous disutility ranges 4-9, 6-11, and 8-13, lazy subsumption has variance 0.3, 0.4, and 0.29 while eager with reuse has variance 1.0, 1.75, and 2.18 respectively.

6.4 Robustness Across Environments

We also conducted a test of the robustness of the different designs across ten different environments, E1, E2,..., E10, ordered from simple to more complicated environments. These environments can be viewed in the context of different software engineering scenarios. For instance, environment E1 might represent academic researchers who need help with issues in a scientific software that they use for experiments. They are short term users, as fixes are only needed for a specific set of experiments. The workers may be research assistants who have fixed low salaries. Moreover the disutility of bugs is low compared to safety critical systems. Environments E2 and E3, involving higher user disutility and variable worker cost respectively, might represent short-term or seasonal needs of businesses. One example of this is web compatibility issues where, for example, retailers would like their websites to appear the same way in different web browsers. To this end they might need bugs to be fixed but are not concerned with the long-term evolution of the web browser code. Environment E6, with a fast rate of bug generation, might represent the scenario where a new technology is adopted. Initially the technology gets a lot of attention and many issues are uncovered. Over time most of the bugs are fixed, the technology matures, and is eventually overtaken by competition, leading to decreased activity. Environments E8, E9, and E10 might represent large-scale, critical systems with high cost workers. Examples include medical visualization software where patients' health is at stake, airline booking software where a bug could leave thousands stranded, banking and financial software where a glitch could affect millions across the world, and so forth. The environment settings are as follows:

E1. Short-term users, fixed instantaneous disutility chosen in the range $[2, 5]$, same cost workers with $c = 1, h^* = 5$.

E2. Short-term users, fixed instantaneous disutility chosen in the range $[11, 15]$, same cost workers with $c = 1, h^* = 5$.

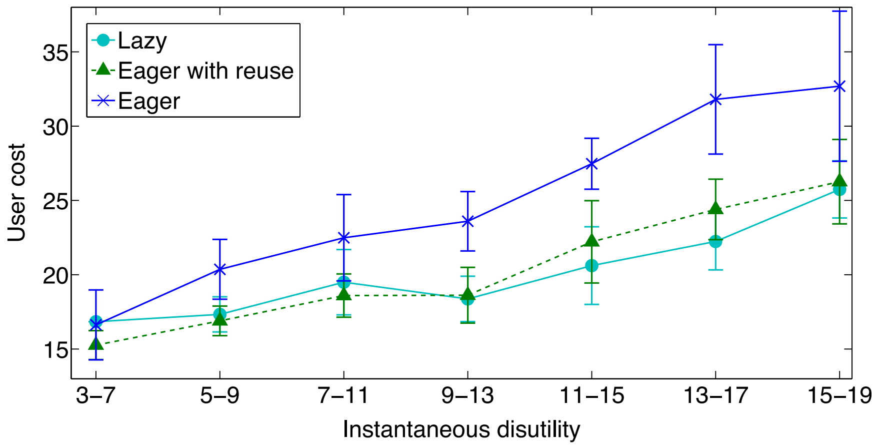


Fig. 17. User cost in Environment IV, varying the instantaneous disutility range, and comparing the lazy, eager, and eager with reuse mechanisms.

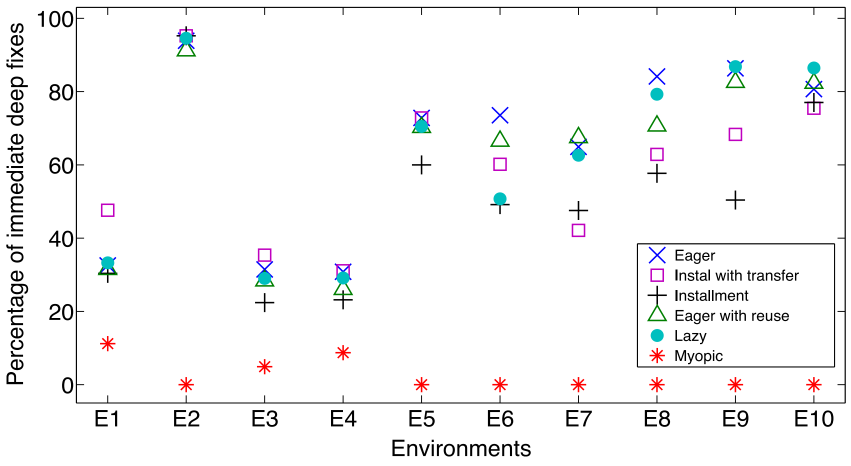


Fig. 18. Robustness check: Percentage of deep fixes across the ten different environments.

E3. Short-term users, fixed instantaneous disutility chosen in the range $[5, 7]$, worker randomly samples $c \sim uniform(1, 2)$, $h^* = 5$.

E4. Long-term as well as short-term users, with instantaneous disutility sampled according to $r \sim uniform(4, 8)$, worker randomly samples $c \sim uniform(1, 2)$, $h^* = 10$.

E5. Long-term as well as short-term users, with instantaneous disutility sampled according to $r \sim uniform(25, 29)$, worker randomly samples $c \sim uniform(1, 2)$, $h^* = 10$.

E6. Faster rate of bug generation.

E7. Low cost workers, where each worker randomly samples $c \sim uniform(1, 1.5)$, long-term and short-term users with instantaneous disutility sampled according to $r \sim uniform(5, 11)$, $h^* = 5$.

E8. High cost workers, where each worker randomly samples $c \sim uniform(2, 2.5)$, long-term and short-term users with instantaneous disutility sampled according to $r \sim uniform(8, 14)$, $h^* = 5$.

E9. High cost workers, where each worker randomly samples $c \sim uniform(2.5, 3)$, long-term and short-term users with instantaneous disutility sampled according to $r \sim uniform(11, 17)$, $h^* = 5$.

E10. High cost workers, where each worker randomly samples $c \sim uniform(3, 4)$, long-term and short-term users with instantaneous disutility sampled according to $r \sim uniform(9, 13)$, $h^* = 10$.

See Figs. 18 and 19. As the environments get more complex we see a shift in the kinds of mechanisms that dominate with respect to percentage of immediate deep fixes as well as user utility. The myopic mechanism performs well in the first few environments, but later the installment with transfer and the subsumption mechanisms take over. The utility of long-term users depends on receiving both immediate and deep fixes. As we have seen, simple mechanisms such as myopic and installment do not provide the right incentives. Hence mechanisms that permit transfers, and reuse subsumed fixes, perform best in environments 5-10, in particular the eager with reuse mechanism.

6.5 Myopic Mechanism and Short-Term Users

In a counterintuitive result, the myopic mechanism gives deep fixes at low payments. For this, we consider Environment V:

Bugs are generated uniformly at random, the worker cost function is linear with $c \sim uniform(1, 2)$ for all workers, the number of installments is 10, and the instantaneous disutility is a constant (and varied in different trials). In this regime all users are short-term. This environment is parameterized by the fixed instantaneous disutility of users.

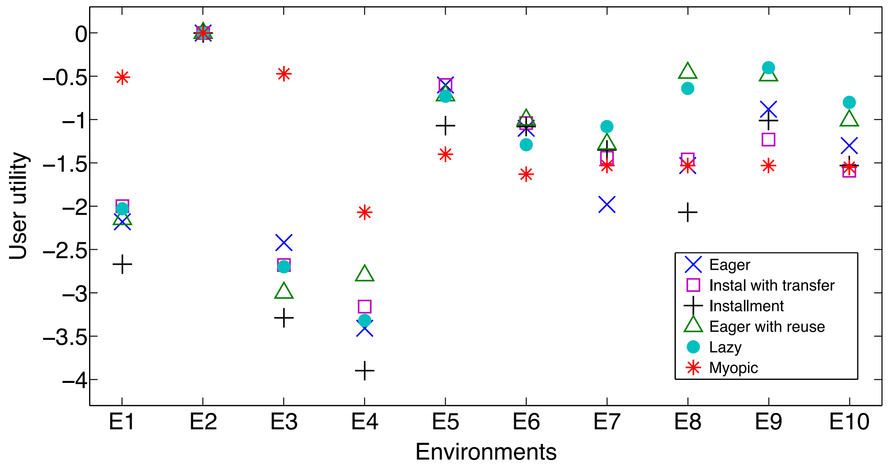


Fig. 19. Robustness check: User utility across the ten different environments.

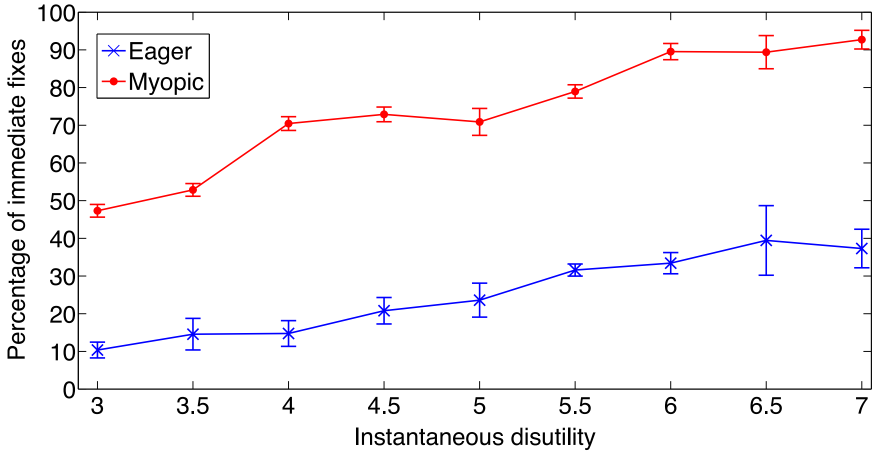


Fig. 20. Percentage of immediate fixes in Environment V, varying the fixed instantaneous disutilities, comparing the eager and myopic mechanisms.

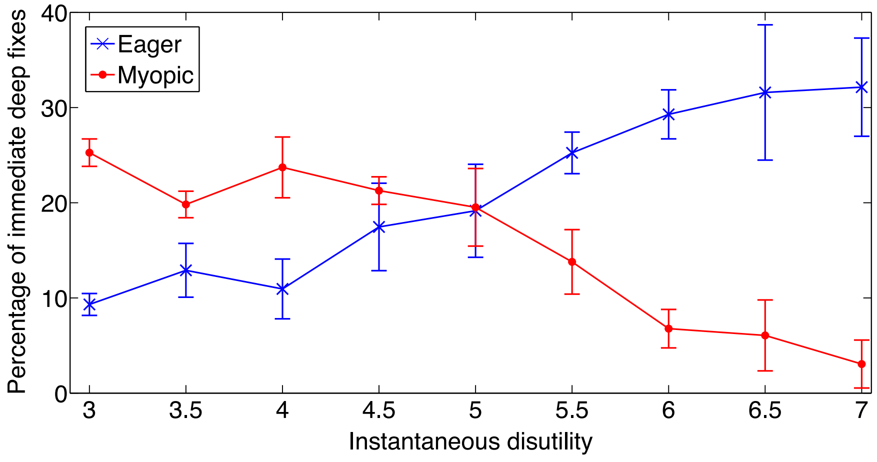


Fig. 21. Percentage of immediate deep fixes in Environment V, varying the fixed instantaneous disutilities, and comparing the eager and myopic mechanisms.

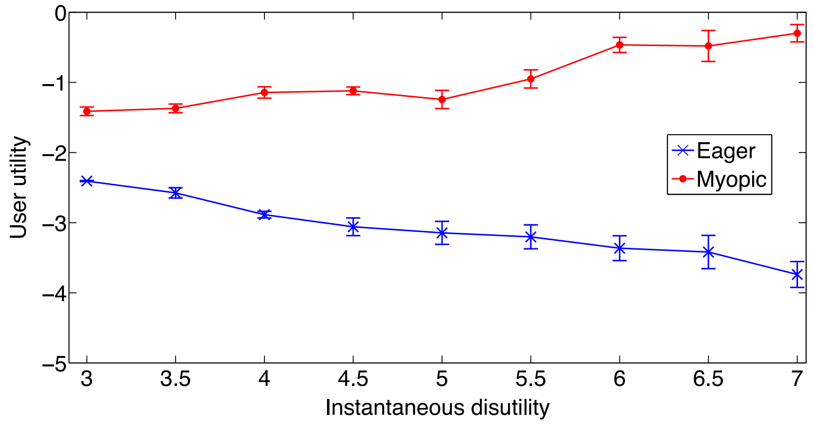


Fig. 22. User utility in Environment V, varying the fixed instantaneous disutilities, and comparing the eager and myopic mechanisms.

See Figs. 20, 21 and 22. In these figures, we "zoom in" to observe what happens at low payments with short-term users. Because the myopic mechanism pays the entire payment at once it promotes immediate fixes (see Fig. 20). Because this environment has only short-term users, the myopic mechanism achieves the maximum user utility when all bugs receive immediate fixes (see Fig. 22).

Further, accumulated payments from cases where the fix remains too costly provide an incentive for the worker to produce a deep enough fix in order to collect the accumulated amount. Thus, we also see deep fixes with the myopic mechanism. However, as the payment increases and completely covers the worker's cost for a fix, the incentives for deep fixes go away. The worker simply gives the shallowest possible fix to claim the payment, and there is no accumulation. In particular, as the payment increases beyond the amount shown in these figures, we revert to the case examined earlier in Figs. 8, 9 and 10, where the eager mechanism was dominant.

TABLE 1
Comparing Mechanisms by the Set of Fixes Used to Preempt Bugs from Entering the Market

All fixes submitted so far (equal the set of active fixes)	Myopic Installment Installment with transfer
All fixes submitted so far (subsumed fixes are reused)	Eager with reuse Lazy subsumption
Set of active fixes (subsumed fixes are deleted)	Eager subsumption

TABLE 3
Comparing the Essential Features of the Payment Rule in Each Mechanism

	Fix to fix comparisons	No fix comparisons
Transfers	Eager subsumption Eager with reuse Lazy subsumption	Installment with transfer
No transfers	N/A	Myopic Installment

7 Conclusion

A market mechanism for bug fixes must take into consideration some important design criteria. These are:

1) Robustness of performance in regard to user utility and percentage of deep fixes, submitted quickly, across different software environments.
2) Selecting a small, good set of fixes, and eliminating redundant fixes.
3) Making judgments about the relative quality of different contributions with minimum knowledge about the domain.

Subsumption mechanisms satisfy all three design criteria. Table 1 compares the mechanism designs according to the set of submitted fixes that are checked when determining whether a new bug already has a fix. Table 2 orders the mechanisms from simple designs without competition between workers for payments to the subsumption mechanisms, which promote competition. Table 3 compares the different designs according to whether there is a transfer of payments, and whether fixes are compared against one another.

In regard to performance, the eager subsumption mechanism is of particular merit if we care primarily about the percentage of bugs that receive immediate and deep fixes. On the other hand, if user utility is more important, then eager with reuse and the lazy mechanism are good choices. At low reward levels, and with short-term users, the myopic mechanism performs best with respect to the metrics, percentage of immediate fixes, percentage of immediate deep fixes, and user utility.

As described in Table 1, subsumption mechanisms identify and remove fixes that have been judged to be subsumed, thereby striving to eliminate redundant submissions. As a result, a system evolves under the subsumption designs to a minimal yet good set of fixes. By comparison, the installment mechanisms do not identify and remove redundant fixes. This can have a negative side effect of leading to a bloated system that keeps around all fixes ever submitted.

The different mechanisms give rise to different types of competition dynamic, as shown in Table 2. Workers in the myopic mechanism produce deep fixes at low payment levels only. Beyond these levels workers are unresponsive and simply submit the shallowest possible fix since no competition is involved. The installment and installment with transfer mechanisms stop paying when the next bug appears. Hence workers in these mechanisms must contend with aspects of the environment such as bug generation rate. Because subsumption mechanisms involve competition amongst workers as well as transfers and fix-to-fix comparisons (Table 3), their performance is robust across different environments.

Regarding knowledge requirements, the check for whether one fix subsumes another in the subsumption mechanisms does not strictly require the assumption of different underlying root causes–rather, it relies on making externally observable comparisons. In contrast, the installment mechanisms have a high knowledge requirement because they need to know that a set of bugs belongs to the same root. Curtailing workers' payments without a priori knowledge that a set of bugs belongs to the same root cause does not seem reasonable in practice.

We further consider the performance of our mechanisms in environments (E1, E2, . . ., E10) that can be mapped to realistic software engineering settings. Our results show that as environments get more complicated, involving more critical systems and higher development costs, there is a shift in the types of mechanisms that function well according to metrics such as percentage of immediate deep fixes and user utility. In these more complex environments, mechanisms that permit transfers and reuse subsumed fixes perform best.

In conclusion, subsumption mechanisms satisfy all three market design criteria and perform robustly across all environment configurations examined in the empirical analysis.

7.1 Limitations and Future Work

This work initializes the study of how to incentivize deep fixes. The present paper evaluates subsumption mechanisms by comparing their performance in simulation with respect to a set of metrics. It would be interesting to consider, for a simpler model, the design of the optimal mechanism, where optimality is defined with respect to a suitable measure or a specific class.

We make several simplifying assumptions to obtain a minimal, tractable model and to focus on capturing the essence of the problem. Building on this work, an interesting direction would be to relax these assumptions. For

TABLE 2
Comparing Mechanisms by the Kind of Competition Dynamic That They Create

No competition	Myopic
Environment as adversary	Installment Installment with transfer
Competition and MFE	Eager subsumption Eager with reuse Lazy subsumption

instance a worker might be allowed to choose what bug to fix instead of being assigned one at random. Making users strategic and allowing payments to change over time would also be an interesting path to pursue. Another experiment could treat the length of time for which payments are deferred as a parameter, thereby characterizing a tradeoff between the eager mechanism at one end and the lazy mechanism at the other end of the spectrum. Future work might consider a family of subsumption mechanisms parameterized by time period, and such that subsumption relations are concluded after t time periods.

Because of the complexity of the design space, we have evaluated the model and mechanisms in simulation. One of us [60] has also carried out theoretical analysis of a simplified variation of the computational model presented here. It would be fruitful to further explore a theoretical approach in order to derive theoretical characterizations regarding the properties of the equilibrium, the performance of the mechanisms, market conditions, and so forth.

Equally interesting would be to take the abstract model presented in this paper to a more practical context. For example, future work might try to apply subsumption mechanisms to an existing system or use real-world data to validate our model. This would shed light on what are useful settings for the parameters in our model as well as highlight instances of the problem that might arise more frequently in practice. The model of the software ecosystem consists of several parameters and each of these parameters can take on a range of values. Various combinations of parameter settings can produce a large number of environments to use for simulations. Building on the foundation laid in this paper, we hope to conduct a wide-ranging series of simulations in future work, focusing on those parameter settings inferred through real-world data.

Because the problem addressed and the system designed in this paper depart from existing systems in several ways, obtaining relevant real-world data would require that we adapt and deploy subsumption mechanisms to work in practice. Deployment raises several interesting challenges. For instance, fixes addressing various reported issues may not always be completely independent and may build on one another in layers of work carried out over time [61]. How to adapt subsumption mechanisms to take into account different contributions that comprise a single solution is an interesting problem left to future work. Another consideration is a fast-changing code base which might make it difficult to compare the effect of a new fix on past versions. In addition a real-world implementation would need to map the parameters in this abstract model to equivalent concrete values. For example, the lazy mechanism defers payments until the end of a root cause's lifetime. In reality this period of deferral could be interpreted in different ways with different consequences: it could be a predetermined period of time such as two weeks, or when the rate of bug generation in the relevant module falls below a certain threshold indicating that the root cause has been mostly addressed and alternatively is no longer much in use, or when the next version of the software is released, and so forth. Preliminary experiments with real users and workers might help to inform what values to set in a final deployment.

In practice bug reports consist of reproducible steps and test cases. If a bug report describes a specific bug then a developer might not realize that a deeper fix is possible in this instance. Incentivizing testers to write failing test cases for more general bugs would help to shift attention to the possibility of deeper fixes. That is, if bug X is a subset of bug Y then we would like developers to focus efforts on bug Y. This would create a test-driven environment where subsumption mechanisms might be implemented more effectively.

In this work we have focused on financial incentives. However in peer production environments, such as that of open source software, volunteers self-organize and contribute to projects without being paid. Studies have shown the open source developers are driven by a diverse set of motivations, monetary and non-monetary in nature [62]. In fact it has been argued that it is the presence of these diverse motivations that results in peer production communities successfully completing complex projects [63]. It would be interesting to consider the implications for the incentives design presented in this paper in the context of peer production. How can financial incentives be introduced without affecting the characteristics of peer production? How can we adapt our model to work with non-monetary incentives, or otherwise to work with a mix of monetary and non-monetary incentives? A reconciliation of the different types of incentives is left to future work.

In general, the problem of incentivizing deep, or equivalently high quality and lasting, fixes is not unique to software engineering. Subsumption mechanisms may also find application in settings other than software engineering that share the aspect of a modular and open design structure, where the quality of the system may evolve over time.

Acknowledgments

The authors thank Edo Airoldi, Eric Balkanski, Yiling Chen, Thibaut Horel, John Lai, Benjamin Lubin, Hongyao Ma, Don Marti, Greg Stoddard, Ming Yin, the participants of AAAI WIT-EC 2014, the participants of the 5th Workshop on Dynamic Games in Management Science 2014, and the anonymous reviewers for useful feedback on this work.

References

[1] NIST, The economic impacts of inadequate infrastructure for software testing. Planning report 02–3. 2002. [Online]. Available: http://www.nist.gov/director/planning/upload/report02–3.pdf
[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 361–370.
[3] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 3–13.
[4] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2003, pp. 141–154.
[5] C. S. Wright and T. A. Zia, "A quantitative analysis into the economics of correcting software bugs," in *Proc. 4th Int. Conf. Comput. Intell. Security Inf. Syst.*, 2011, pp. 198–205.
[6] Bountysource inc. website, 2013. [Online]. Available: https://www.bountysource.com
[7] Topcoder inc. website, 2001. [Online]. Available: https://www.topcoder.com
[8] D. F. Bacon, E. Bokelberg, Y. Chen, I. A. Kash, D. C. Parkes, M. Rao, and M. Sridharan, "Software economies," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res.*, 2010, pp. 7–12.

[9] D. F. Bacon, Y. Chen, I. A. Kash, D. C. Parkes, M. Rao, and M. Sridharan, "Predicting your own effort," in *Proc. 11th Int. Conf. Auton. Agents Multiagent Syst.*, 2012, pp. 695–702.

[10] Bountify inc. website, 2012. [Online]. Available: https://bountify.co

[11] Bountify inc. faq website, 2012. [Online]. Available: https://bountify.co/faq

[12] Bugcrowd inc. website, 2012. [Online]. Available: https://bugcrowd.com

[13] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *J. Syst. Softw.*, vol. 126, pp. 57–84, Apr. 2017.

[14] The linux foundation releases first-ever value of collaborative development report, 2015. [Online]. Available: https://www.linuxfoundation.org/press-release/the-linux-foundation-releases-first-ever-value-of-collaborative-development-report/

[15] N. Eghbal, *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure*. New York, NY, USA: Ford Foundation, 2016.

[16] S. Kooths, M. Langenfurth, and N. Kalwey, "Open-source software—An economic assessment," *MICE Econ. Res. Stud.*, vol. 4, 2003.

[17] S. Adlakha, R. Johari, and G. Y. Weintraub, "Equilibria of dynamic games with many players: Existence, approximation, and market structure," *J. Economic Theory*, vol. 156, pp. 269–316, Mar. 2015.

[18] S. E. Schechter, "How to buy better testing: Using competition to get the most security and robustness for your dollar," in *Proc. Infrastructure Security Conf.*, 2002, pp. 73–87.

[19] A. Ozment, "Bug auctions: Vulnerability markets reconsidered," in *Proc. 3rd Workshop Econ. Inf. Security*, 2004.

[20] Statement by pentagon press secretary peter cook on dod's partnership with hackerone on the hack the pentagon security initiative, 2016. [Online]. Available: http://www.defense.gov/News/News-Releases/News-Release-View/Article/709818/statement-by-pentagon-press-secretary-peter-cook-on-dods-partnership-with-hacke

[21] R. Anderson, "Why information security is hard–An economic perspective," in *Proc. 17th Annu. Comput. Security Appl. Conf.*, 2001, pp. 358–365.

[22] R. Anderson, R. Bohme, R. Clayton, and T. Moore, "Security economics and the internal market," *Tech. Report Eur. Netw. Inf. Security Agency*, 2008.

[23] R. Anderson and T. Moore, "The economics of information security," *Sci.*, vol. 314, pp. 610–613, 2006.

[24] I. Cofone, "The value of privacy: Keeping the money where the mouth is," in *Proc. 14th Annu. Workshop Econ. Inf. Security*, 2015.

[25] K. Kannan and R. Telang, "Market for software vulnerabilities? think again," *Manage. Sci.*, vol. 51, no. 5, pp. 726–740, 2005.

[26] S. Laube and R. Bohme, "The economics of mandatory security breach reporting to authorities," in *Proc. 14th Annu. Workshop Econ. Inf. Security*, 2015.

[27] T. Maillart, M. Zhao, J. Grossklags, and J. Chuang, "Given enough eyeballs, all bugs shallow? revisiting Eric Raymond with bug bounty markets," *J. Cybersecurity*, vol. 3, no. 2, pp. 81–90, Jun. 2017.

[28] S. E. Schechter, "Toward econometric models of the security risk from remote attack," *IEEE Security Privacy*, vol. 3, no. 1, pp. 40–44, Jan./Feb. 2005.

[29] H. Hosseini, R. Nguyen, and M. W. Godfrey, "A market-based bug allocation mechanism using predictive bug lifetime," in *Proc. 16th Eur. Conf. Softw. Maintenance Re-Eng.*, 2012, pp. 149–158.

[30] C. L. Goues, S. Forrest, and W. Weimer, "The case for software evolution," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res.*, 2010, pp. 205–210.

[31] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[32] S. Athey and G. Ellison, "Dynamics of open source movements," *J. Econ. Manage. Strategy*, vol. 23, no. 2, pp. 294–316, 2014.

[33] J. Johnson, "Open source software: Private provision of a public good," *J. Econ. Manage. Strategy*, vol. 11, no. 4, pp. 637–662, 2002.

[34] J. Johnson, "Collaboration, peer review and open source software," *Inf. Econ. Policy*, vol. 18, no. 4, pp. 477–497, 2006.

[35] J. Lerner, P. Pathak, and J. Tirole, "The dynamics of open source contributors," *Amer. Econ. Rev. Paper Proc.*, vol. 96, no. 2, pp. 114–118, 2006.

[36] K. J. Boudreau, N. Lacetera, and K. R. Lakhani, "Parallel search, incentives and problem type: Revisiting the competition and innovation link. working paper," Harvard Business School, Boston, MA, HBS Working Paper Number: 09-041,Sep. 2008.

[37] K. R. Lakhani and J. A. Panetta, "The principles of distributed innovation," *Innovations: Technol. Governance Globalization*, vol. 2, no. 3, pp. 97–112, 2007.

[38] C. Y. Baldwin and K. B. Clark, *The Power of Modularity. Vol. 1, Design Rules*. Cambridge, MA, USA: MIT Press, 2000.

[39] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, 2006.

[40] D. Acemoglu and M. K. Jensen, "Aggregate comparative statics," *Games Econ. Behaviour*, vol. 81, pp. 27–49, 2013.

[41] A. Bodoh-Creed, "Approximation of large dynamic games," Working Paper, 2012.

[42] G. Y. Weintraub, C. L. Benkard, and B. V. Roy, "Markov perfect industry dynamics with many firms," *Econometrica*, vol. 76, no. 6, pp. 1375–1411, 2008.

[43] G. Y. Weintraub, C. L. Benkard, and B. V. Roy, "Industry dynamics: Foundations for models with an infinite number of firms," *J. Econ. Theory*, vol. 146, pp. 1965–1994, 2011.

[44] M. Huang, P. E. Caines, and R. P. Malhame, "Social optima in mean field LQG control: Centralized and decentralized strategies," *IEEE Trans. Autom. Control*, vol. 57, no. 7, pp. 1736–1751, Jul. 2012.

[45] M. Huang, R. P. Malhame, and P. E. Caines, "Large population stochastic dynamic games: Closed-loop McKean-Vlasov systems and the nash certainty equivalence principle," *Commun. Inf. Syst.*, vol. 6, no. 3, pp. 221–251, 2006.

[46] J.-M. Lasry and P.-L. Lions, "Mean field games," *Japanese J. Math.*, vol. 2, pp. 229–260, 2007.

[47] K. Iyer, R. Johari, and M. Sundararajan, "Mean field equilibria of dynamic auctions with learning," *Manage. Sci.*, vol. 60, no. 12, pp. 2949–2970, 2014.

[48] R. Gummadi, P. Key, and A. Proutiere, "Optimal bidding strategies and equilibria in repeated auctions with budget constraints," in *Proc. Allerton Annu. Conf. Commun. Control Comput.*, 2011.

[49] S. Adlakha and R. Johari, "Mean field equilibrium in dynamic games with complementarities," in *Proc. IEEE Conf. Decision Control*, 2010, pp. 6633–6638.

[50] N. Arnosti, R. Johari, and Y. Kanoria, "Managing congestion in decentralized matching markets," in *Proc. Extended Abstract ACM Conf. Econ. Comput.*, 2014, pp. 451–451.

[51] R. Gummadi, R. Johari, and J. Y. Yu, "Mean field equilibria of multiarmed bandit games," in *Proc. ACM Conf. Electron. Commerce*, 2012, pp. 655–655.

[52] B. Moldovanu and A. Sela, "The optimal allocation of prizes in contests," *Amer. Econ. Rev.*, vol. 91, no. 3, pp. 542–558, 2001.

[53] B. Moldovanu and A. Sela, "Contest architecture," *J. Econ. Theory*, vol. 126, no. 1, pp. 70–97, 2006.

[54] G. Tullock, "Efficient rent seeking," in *Towards a Theory of the Rent Seeking Society*. James M. Buchanan, Robert D. Tollison, Gordon Tullock Eds., College Station, Texas, USA: Texas A&M University Press, 2001, pp. 97–112.

[55] N. Archak and A. Sundararajan, "Optimal design of crowdsourcing contests," in *Proc. Int. Conf. Inf. Syst.*, 2009.

[56] S. Chawla, J. D. Hartline, and B. Sivan, "Optimal crowdsourcing contests," in *Proc. 23rd Annu. ACM-SIAM Symp. Discr. Algorithms*, 2012, pp. 856–868.

[57] D. DiPalantino and M. Vojnovic, "Crowdsourcing and all-pay auctions," in *Proc. 10th ACM Conf. Electron. Commerce*, 2009, pp. 119–128.

[58] M. Rao, D. C. Parkes, M. Seltzer, and D. F. Bacon, "A framework for incentivizing deep fixes," in *Proc. AAAI Workshop Incentives Trust E-Communities*, 2014.

[59] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large Markov decision processes," in *Proc. Int. Joint Conf. Artif. Intell.*, 1999, pp. 1324–1231.

[60] M. Rao, "Incentives design in the presence of externalities," PhD Dissertation, The School Eng. Appl. Sci., Harvard Univ., Cambridge, MA, 2015.

[61] J. Howison and K. Crowston, "Collaboration through open superposition: A theory of the open source way," *MIS Quart.*, vol. 38, no. 1, pp. 29–50, 2014.

[62] J. A. Roberts, I.-H. Hann, and S. A. Slaughter, "Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects," *Manage. Sci.*, vol. 52, no. 7, pp. 984–999, 2006.

[63] Y. Benkler, "Coases penguin, or, linux and the nature of the firm," *Yale Law J.*, vol. 112, no. 3, pp. 369–446, 2002.

Malvika Rao received the PhD degree in computer science from Harvard University, in 2015. Her research considers the design of incentives in systems that exhibit both computational and economic characteristics. Currently, she is studying the incentives in peer production.

David F. Bacon received the PhD degree in computer science from the University of California, Berkeley, in 1997. His research lies in the areas of programming languages, parallel computing, garbage collection, and hardware compilation. At present he is with Google. Previously, he was a principal research staff member with IBM's Thomas J. Watson Research Center. He took a sabbatical, in 2009 as a visiting professor of computer science with Harvard. In 2009 he was inducted as an ACM fellow for contributions to real-time systems and to object-oriented language design and implementation.

David C. Parkes received the PhD degree in computer science from the University of Pennsylvania, in 2001. He is George F. Colony professor of computer science and co-director of the Harvard Data Science Initiative with Harvard University, where he founded the EconCS research group and leads research with a focus on market design, artificial intelligence, and machine learning. He is fellow of the Association for the Advancement of Artificial Intelligence (AAAI), and recipient of the 2017 ACM/SIGAI Autonomous Agents Research Award.

Margo I. Seltzer received the PhD degree in computer science from the University of California, Berkeley, in 1992. She is the Herchel Smith professor of computer science and the faculty director of the Center for Research on Computation and Society (CRCS) in Harvard's John A. Paulson School of Engineering and Applied Sciences. Her research interests are in systems for capturing and accessing data provenance, file systems, databases, transaction processing systems, storage and analysis of graph-structured data, new architectures for parallelizing execution, and systems that apply technology to problems in healthcare. She was a founder and CTO of Sleepycat Software, the makers of Berkeley DB and is now an architect for Oracle Corporation. She is the USENIX representative to the Computing Research Association Board of Directors, a member of the Computer Science and Telecommunications Board of the National Academies, and a past president of the USENIX Assocation. She is a Sloan Foundation fellow in computer science, an ACM fellow, a Bunting fellow, and was the recipient of the 1996 Radcliffe junior faculty fellowship, and the University of California Microelectronics Scholarship. She is recognized as an outstanding teacher and won the Phi Beta Kappa Teaching award in 1996 and the Abrahmson Teaching Award in 1999.