Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game

José Miguel Rojas*, Thomas D. White†, Benjamin S. Clegg‡, Gordon Fraser§
Department of Computer Science, The University of Sheffield, Sheffield, United Kingdom
Email: {*j.rojas, †tdwhite1, ‡bsclegg1, §gordon.fraser}@sheffield.ac.uk

***Abstract*—Writing good software tests is difficult and not every developer's favorite occupation. Mutation testing aims to help by seeding artificial faults (mutants) that good tests should identify, and test generation tools help by providing automatically generated tests. However, mutation tools tend to produce huge numbers of mutants, many of which are trivial, redundant, or semantically equivalent to the original program; automated test generation tools tend to produce tests that achieve good code coverage, but are otherwise weak and have no clear purpose. In this paper, we present an approach based on gamification and crowdsourcing to produce better software tests and mutants: The** CODE DEFENDERS **web-based game lets teams of players compete over a program, where attackers try to create subtle mutants, which the defenders try to counter by writing strong tests. Evaluation in controlled and crowdsourced scenarios reveals that writing tests as part of the game is more enjoyable, and playing** CODE DEFENDERS **results in test suites and mutants that are stronger than those produced by automated tools.**

***Keywords*-gamification; crowdsourcing; mutation testing;**

I. Introduction

Software needs to be thoroughly tested in order to remove bugs. To evaluate how thoroughly a program has been tested, the idea of mutation testing is to measure the number of seeded artificial bugs (mutants) a test suite can distinguish from the original program. Testers can then be guided to improve test suites by writing new tests that target previously undetected mutants. In contrast to more basic code coverage criteria such as statement coverage [22], the ability of a test suite to detect mutants is correlated with detecting real faults [25].

However, writing good tests is difficult and developers are often reluctant to do so [7]. They are even less likely to write tests for mutants: Mutation tools tend to produce huge amounts of mutants, and many of these mutants are trivial or redundant, and sometimes even semantically equivalent to the original program, in which case time spent trying to write a test is time wasted. One possible solution lies in also generating the tests automatically, but humans tend to write tests that are stronger, have a clear meaning, and are typically more readable.

The difficulties of writing good tests and using automated mutation tools are similar in nature to those generally targeted by gamification and crowdsourcing: Gamification [13] is the approach of converting tasks to components of entertaining gameplay. The competitive nature of humans is exploited to motivate them to compete and excel at these activities by applying their creativity. Crowdsourcing is a problem solving strategy [21] where a difficult problem is encoded and assigned to an undefined group of workers (the crowd), who provide their solutions back to the requester; the requester can then derive the final solution from the solutions collected from the workers, who are usually rewarded, e.g., with cash or prizes.

In this paper, we describe an approach to generate good software tests and mutants using gamification and crowdsourcing with the CODE DEFENDERS game. Testing activities are gamified by having players compete over a program under test: *Attackers* try to create subtle, hard to kill mutants, while *defenders* try to create tests that can detect and counter these attacks. In order to crowdsource sets of good tests and mutants, CODE DEFENDERS is played as a multi-player game, where teams of attackers and defenders compete to defeat the opposing team, and to score the most points within their own team.

In detail, the contributions of this paper are as follows:

- We introduce the CODE DEFENDERS multi-player game, its players' actions, and its balanced scoring system aiming to make the gameplay enjoyable for both player roles.
- We evaluate the gamification aspects of CODE DEFENDERS and present the results of a controlled study comparing it to traditional unit testing in terms of the objective performance and subjective perception of 41 participants.
- We evaluate the application of CODE DEFENDERS in a crowdsourcing scenario and present the results of 20 multi-player games played on open source classes, comparing the tests and mutants to those generated by automated tools.

All participants of our experiments confirmed that playing the game is fun, and that writing tests as part of CODE DEFENDERS is more enjoyable than doing so outside the game. Code coverage and mutation scores are higher compared to tests (a) written outside the game and (b) generated by automated tools (on average, 28% higher mutation score than Randoop [28], and 25% higher mutation score than EvoSuite [17]). Mutants created by attackers are significantly harder to kill than those created by the Major mutation tool [23].

In this paper, we target the crowdsourcing aspect of CODE DEFENDERS; however, the game is also naturally suited for educational purposes. Our initial findings for educational applications are documented elsewhere [35]. To support educational use, CODE DEFENDERS also provides a single-player mode, where players compete against an automated attacker (the Major mutation tool) or an automated defender (the EvoSuite test generation tool), and a two-player mode. We have made CODE DEFENDERS freely available to play online at http://www.code-defenders.org.

II. Background

A. Unit Test Generation

Developers frequently execute unit tests to guard their programs against software bugs. As writing a good test suite can be difficult and tedious, there is a range of different tools to support this activity by automatically generating tests.

A basic approach to generating tests is to do so randomly. For example, Randoop [28] is a mature test generation tool for Java that produces random sequences of calls for a given list of classes; violations of code contracts are reported as bugs, and tests that do not reveal bugs are equipped with regression oracles that capture the current program state for regression testing. Because random test generation tends to result in very large test suites and may struggle to cover corner cases, search-based testing has been suggested as an alternative. For example, EvoSuite [17] generates test suites using a genetic algorithm which aims to maximize code coverage. Test suites are minimized with respect to the target criteria, thus resulting in far fewer tests than random testing would produce. Approaches based on symbolic execution can be effective for certain types of problems that are particularly amenable to the power of modern constraint solvers. For example, EvoSuite implements an experimental extension [19] that uses dynamic symbolic execution to generate primitive input values, and the Pex [43] tool uses dynamic symbolic execution to instantiate parameterized unit tests for C#.

The annual unit test generation tool competition [37] compares different unit test generation tools for Java, and although tools have made substantial progress in recent years, there remain several challenges. Xusheng et al. [49] identify different challenges that hinder test generation tools in reaching code (e.g., object mutation, complex constraints, etc.), and Shamshiri et al. [40] identified several problems that hinder automatically generated unit tests from finding real faults. Pavlov and Fraser [32] demonstrated that some of these can be overcome by including human intelligence by using an interactive genetic algorithm in the EvoSuite tool.

B. Mutation Testing

In order to evaluate test suites and to guide selection of new tests, mutation testing has been proposed as an alternative to traditional code coverage metrics. Mutation testing consists of seeding artificial faults ("mutants") in a program, and then measuring how many of them are found ("killed") by the test suite. The mutation score, i.e., the ratio of mutants killed, provides an indication of the test suite quality, while mutants that remain "alive" provide hints on where to add new tests. There is evidence [2], [25] that test suites that are good at finding mutants are also good at finding real faults.

One of the main advantages of mutation testing over code coverage is that code coverage does not consider the quality of test oracles, i.e., how the correctness of the test execution is checked. However, the practical application of mutation testing is hindered by two significant problems: First, non-trivial code results in large numbers of mutants. Mutants are generated using different *mutation operators*, which systematically perform simple modifications (e.g., replace an operator), and each application of an operator results in a new mutant. Despite many efforts to reduce the number of mutants produced (e.g., [24]) the number remains large, which is not only a problem for scalability, but also because many mutants are either trivial or subsumed by other mutants [29].

The second problem is that some mutants are semantically equivalent to the original program, such that there exists no killing test. Detecting equivalent mutants is an undecidable problem [9], [27], and effort on trying to derive such a test is likely wasted. Different techniques and systems have been developed to detect equivalent mutants (e.g., [1], [30], [39]), but they are generally limited to certain types of mutants. Thus, human intervention is still required to discern hard-to-kill (or "stubborn") mutants from equivalent ones [50].

One insight underlying this paper is that these two main problems of mutation testing, designing good mutants and deciding equivalence, both require human intelligence. This leads us to investigate the use of gamification and crowdsourcing.

C. Crowdsourcing and Gamification

Problems that are hard to solve computationally but can be effectively solved by humans can be amenable to crowdsourcing [21]. The general principle is to identify and extract tasks that require human intelligence, and then to present these "human intelligence tasks" to "crowd workers". Additional computational effort is usually necessary to assemble the individual task solutions to solve the overall problem. In software engineering, crowdsourcing platforms such as Amazon Mechanical Turk, where crowd workers are paid small fees for completed tasks, are often used for empirical studies [42], but there are attempts to crowdsource various parts of the software development process [26].

Gamification uses game design elements (competitions with other players, game rules, point scoring, fantasy scenarios, etc.) to make unpleasant or dull tasks more entertaining and rewarding [13]. It is often applied in education settings, but has also been useful for improving how people engage with aspects of their work, even in software engineering [33]. A particular form of gamification are "games with a purpose", where players of the game solve underlying computational problems (sometimes without being aware of this). In other words, games with a purpose are a form of crowdsourcing, where the incentive for workers is provided in terms of the gameplay. Famous examples include ReCaptcha [47] or DuoLingo [46].

III. The Code Defenders Game

A. Gameplay

Code Defenders is a competitive game where two teams compete over a Java class under test (CUT) and its test suite; one team leads an "attack" on the CUT, whereas the other team tries to defend it. Attackers aim to create variants of the CUT, i.e., *mutants*, with which they "attack" the fault-detection capability of the associated test suite. Defenders aim to protect the CUT by writing unit tests that detect, i.e., *kill* the mutants.

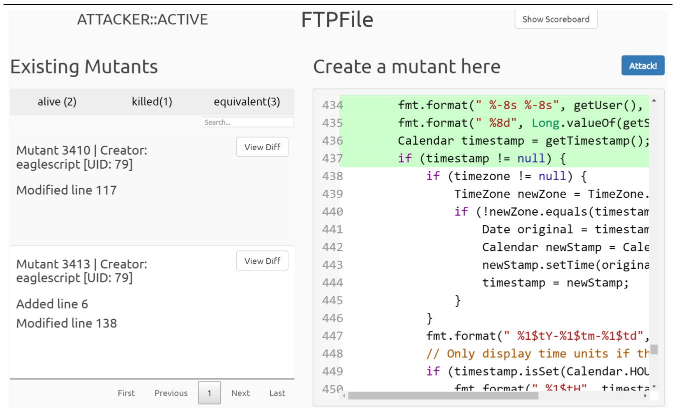


Fig. 1: The Attacker's View.

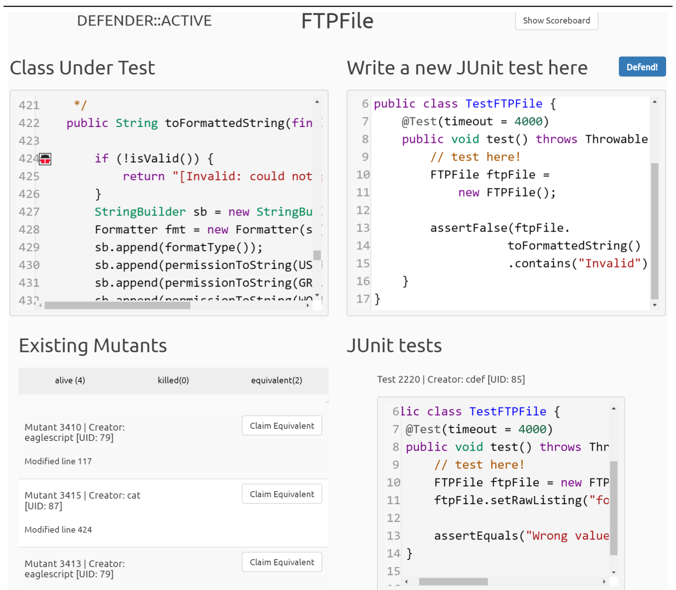


Fig. 2: The Defender's View.

Two difficulty levels are available in the game. In the *easy'* level, attackers and defenders see all submitted mutants and tests. In the default *hard* level, the information presented to players is restricted to balance the gameplay and make it more interesting for both roles. Attackers have a code editor where they create mutants by modifying the CUT (Figure 1). They see all mutants in the game including their code diffs, and the code editor highlights the line coverage achieved by the tests submitted to the game so far. The highlighting reflects how often lines are covered; the more often a line is covered, the darker the highlighting is. Defenders (Figure 2) see the source code of the CUT together with the locations of live and dead mutants. In their code editor, they are given a template to write a unit test for the CUT, and they also see previous tests as well as their coverage. Unlike the round-based gameplay of our preliminary version of CODE DEFENDERS [34], attackers and defenders can submit mutants and tests at any time and do not need to wait for other players to act.

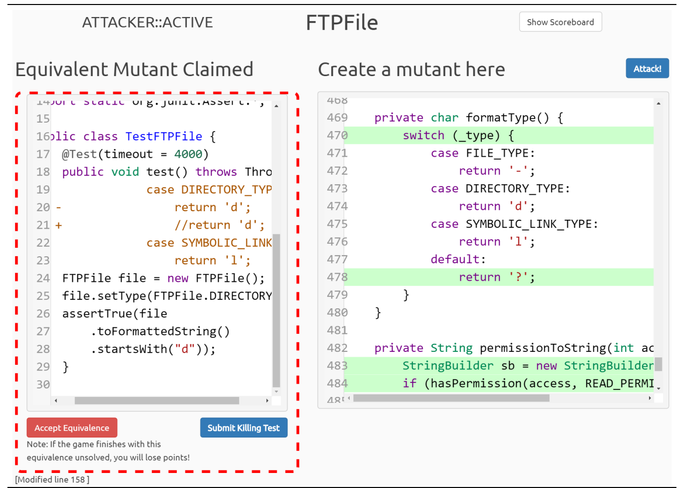


Fig. 3: The Equivalence Duel View.

B. Equivalence Duels

The mutants that attackers create in the game may be equivalent, whether on purpose or not. The gameplay integrates *duels* that allow players to decide on equivalent mutants. If a defender suspects a mutant to be equivalent, for example because the mutant is still alive after several failed attempts at killing it, then he/she can challenge the attacker by starting an equivalence duel. The onus is then on the attacker either to write a test that kills the mutant, proving it is not equivalent, or to confirm that the mutant is indeed equivalent (Figure 3).

C. The Multiplayer Scoring System

The point scoring system is based on assigning each mutant and test a number of points that can change as the game unfolds. In particular, mutant points are calculated as follows:

- If a mutant is killed by an existing test when it is created (i.e., a stillborn mutant), then it receives no points.
- A mutant gains a point for every test that covers any of the mutated lines but still passes. Thus, surviving mutants created on heavily tested lines, although risky, will result in more points.
- If a mutant is created and not killed by any existing tests, then it receives one point (in addition to points gained from tests that cover it but do not fail). This is to encourage creation of mutants also for code not yet covered by tests.
- Once a mutant is killed, its score is no longer increased.

Test points are calculated as follows:

- For each mutant that a test kills, the test gains points equal to the score of the mutant plus one. This applies to mutants that already existed at the time the test was created, as well as mutants added to the game later.
- A test gains one point for killing a newly created mutant.
- When a mutant is submitted, tests are executed in the order of their creation. Thus, the oldest test that kills a mutant receives the point, and no other tests receive points for the same mutant.

The score of an attacker is the sum of the points of her mutants; the score of a defender is the sum of the points of her tests. Equivalence duels can update both players' scores: If a defender claims a mutant as equivalent but the attacker proves non-equivalence with a test, then the attacker keeps the mutant's points and the mutant is killed. However, if the attacker accepts that the mutant is equivalent, or the game ends, then she loses all the points she scored with that mutant and the defender who had claimed equivalence gains one point. While an equivalence duel is active, the mutant remains alive and can be killed by other defenders (which would cancel the duel); the mutant can still gain points for surviving newly submitted tests until the equivalence duel is resolved. If the attacker submits a test which compiles but fails to kill the mutant, they lose the duel and the mutant is assumed equivalent. An elaborate example of the scoring system can be found on the CODE DEFENDERS webpage.

One potential issue with the scoring system is that in the last few minutes of a game, a defender could flag all mutants as equivalent leaving no time for attackers to resolve the equivalence; this would mean that all mutants are penalized and lose their points. Similarly, attackers could submit equivalent mutants in the last few minutes, leaving defenders no time to react. We prevent this from happening by introducing a grace period of configurable duration at the end of each game (e.g., one hour). In this grace period, no new mutants or defender tests can be submitted. in the first part of the grace period (e.g., 15 min.) defenders can flag mutants as equivalent while attackers wait; the remaining time of the grace period can only be used by attackers to resolve pending equivalence duels.

D. Code Editing Restrictions

Whenever humans engage in competitive games, there is the possibility of cheating and unfair behaviour, and this also holds in gamified software engineering tasks [15]. In particular, once players understand the scoring system, there will likely be some players who try to create mutants or tests in a way that benefits their score without providing a useful improvement in terms of the mutants or tests generated in the game. For example, an attacker could add an if-condition of the type `if(x == 2355235)` which could only be killed by a test that happens to use the arbitrary input data 2355235 – which is very unlikely. This mutant would increase the attacker's score, but it may misdirect the effort of the defenders and likely does not resemble a real fault.

To reduce the possibility of such behaviour, we implemented a number of restrictions on the modifications that attackers can perform, and the tests that defenders can create. In particular, the following restrictions apply when creating mutants:

- If-conditions, the ternary operator (`?`), loops (e.g., `for`, `while`, ...), and Boolean operators (e.g., `&&`) cannot be added. This prevents mutants as shown in the example above, which are near impossible for defenders to kill, but easy for an attacker to prove non-equivalent.
- Calls to `java.util.System.*` cannot be added: This is to restrict access to system information (e.g., environment variables) and to prevent calls (e.g., `System.exit`) that would affect the security of the web system. For security purposes, tests are also executed in a sandbox using a custom security manager.
- Calls to `java.util.Random` cannot be added to avoid flaky or impossible to kill mutants.

The following restrictions are applied when writing tests, mainly to prevent defenders from writing huge tests with "mega"-assertions, which would discourage other defenders, reduce points of surviving mutants and is contrary to the idea of a good unit test:

- Only two assertions can be used in a single test.
- Test classes can not define helper methods, and must not use conditionals (if-statements, ternary operators, logical connectives) and loops, which would allow sidestepping the 2-assertion limitation.
- Tests cannot make calls to `System.*` (see above).

IV. DOES GAMIFICATION IMPROVE TESTING?

Before evaluating the applicability of CODE DEFENDERS as a crowdsourcing solution for test generation, we investigated its general feasibility as a gamification approach to software testing. To this end, we used the two-player version [34], where one attacker plays against one defender in a round-based mode, and designed a controlled empirical study to answer the following research questions:

RQ1: Do testers produce better tests when playing a game?
RQ2: Do testers enjoy writing tests more when playing a game?

A. Experiment Setup

We conducted this controlled study in a computer lab at the University of Sheffield. We invited undergraduate and postgraduate students, researchers and professional developers by email. Student candidates were required to have completed at least one Java course in their degree and all candidates were asked to complete an online Java qualification quiz to demonstrate their Java skills. We selected all 41 candidates who answered at least 3 out of the 5 questions correctly. 52% of the participants were undergraduate students, 37% were Master's or PhD students and the rest were either professional developers or academics. All participants were in Computer Science or Software Engineering-related fields, had a diverse degree of experience programming in Java but generally little or no industrial work experience (66%). The majority (76%) had used JUnit or a similar testing framework before and understood well or very well the concept and usage of mutation testing, although most admitted to only rarely or occasionally writing unit tests when programming.

Prior to the experiment, participants attended a training session consisting of a brief tutorial on unit and mutation testing and an introduction to CODE DEFENDERS. They familiarized themselves with the web interface of the game through short, guided tasks. To conclude the training session, all participants played an actual CODE DEFENDERS game on a simple class. When asked in the exit survey whether they understood the gameplay, only 3 participants partially disagreed and 3 further participants neither agreed nor disagreed.

The actual experiment consisted of two 30-minute tasks per participant. The three possible tasks were: (1) Writing unit tests manually; (2) playing CODE DEFENDERS as an attacker; or (3) playing CODE DEFENDERS as a defender. We selected two classes under test: SortedIntegerList, a standard implementation of a data structure for sorted list of integers, and IBAN, a validator for International Banking Account Numbers from the swift-wife open source project. Each participant performed one task on each of the two classes. The manual testing tasks serve as the baseline of regular testing behavior and were also performed using the CODE DEFENDERS web interface; we asked participants to test the class as well as possible to guard against potential faults, but we did not explicitly ask them to optimize for coverage or other metrics. A pre-created assigment determined the two tasks for each participant. The assignment was designed to balance tasks for the two classes, the order in which participants performed each task, and the order in which participants played as attackers or defenders for each class. The assignment further ensured that the attacker and the defender in each game did not sit next to each other. Participants were randomly assigned usernames based on the assignment and did not get to know who they were playing against. The experiment, including training, lasted two hours, and each participant was paid GBP20 for their involvement.

In total, 28 games were played and 26 manual testing tasks were completed. On average, each game lasted 3.8 rounds, and a total of 72 valid unit tests were produced by the game players. Manual testers were not bound to the round-based setting of the game, and produced 93 valid tests (a test is valid if it compiles and passes on the original class under test). To answer RQ1, we compare the tests written by participants playing as defenders with tests written by participants doing manual (unguided) unit testing. We measure the standard quality attributes of code coverage and mutation scores using Jacoco[1] to measure coverage, and Major [23] to calculate mutation scores.

After the experiment, all participants were asked to fill out an exit survey which consisted of standard demographic questions, 10 questions of agreement on aspects of the gameplay with 5-value Likert-scale responses, 8 questions where we asked users to state their agreement with possible improvements, and free-text questions to comment on the user interface, the point scoring system, and the overall game. To answer RQ2, we use the data on five questions that directly asked the participants whether they preferred playing the game to writing tests.

B. Threats to Validity

Construct: We used mutation scores and branch coverage to compare tests, but it may be that other quality attributes (e.g., readability) are affected by the gameplay. We countered this threat by adding restrictions on the tests (e.g., maximum number of assertions). While evidence supports that real faults are correlated with mutants [2], [25], it is possible that the use of faults created by developers may yield different results.

Internal: To prepare the study and to process the results we used automation extensively, and faults in the automation may have an influence on the results of the study. To counter this threat, we tested all our software, and make all data and scripts available. To avoid bias we assigned tasks to participants randomly, based on a pre-created balanced assignment. This assignment ensures that no two neighbouring participants would work on the same class or treatment at the same time. Participants without sufficient knowledge of Java and JUnit may affect the results; therefore, we only accepted participants who correctly answered at least three out of five question of a qualification quiz. We also provided a tutorial on unit and mutation testing before the experiment. To ensure that experiment objectives are not unclear we tested and revised our material on a pilot study with PhD students. We also interacted with the participants throughout the experiment to ensure they understood their tasks; in the exit survey participants confirmed they understood the objectives.

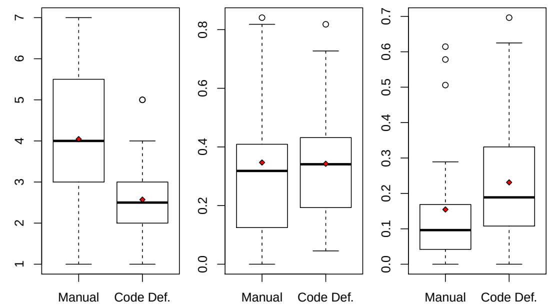
As each participant performed two tasks, it is possible that those playing as a defender in the first session could grasp insight on how tests should be written to kill mutants if they are given manual testing as their second task. To lessen the impact of this learning effect, our assignment of objects to participants ensures that each pair of classes/treatments occurs in all possible orders. To counter fatigue effects we restricted the tasks to 30 minutes, included short breaks after the training session and between the two main sessions, and also provided light refreshments. In order to minimize participants' communication, we imposed exam conditions and explicitly asked participants not to exchange information or discuss experiment details during the breaks.

External: Most participants of our study are students, which is a much debated topic in the literature (e.g., [10], [20]). However, we draw no conclusions from absolute performance, and see no reason why students' experience of playing CODE DEFENDERS should be different from other types of players. The classes used in the experiment are small to allow understanding and testing within the short duration of the experiment. Although object oriented classes are often small, it may be that larger classes with more dependencies affect the gameplay. Thus, to which extent our findings can be generalized to arbitrary classes remains an open question.

C. Results

1) RQ1: Do testers produce better tests when playing a game?: Figure 4(a) shows that participants performing the manual testing task wrote more tests than participants playing CODE DEFENDERS as defenders; this is expected as the two-player mode is turn-based, and after submitting a test defenders have to wait for the attacker to create a new mutant. Figure 4(b) compares the resulting test suites in terms of branch coverage, measured with Jacoco. Interestingly, the branch coverage achieved by these tests is nevertheless similar (Mann-Whitney U test with $p = 0.81$, Vargha-Delaney effect size of $\bar{A}_{12} = 0.52$, where $\bar{A}_{12} = 0.5$ means there is no difference, and $\bar{A}_{12} > 0.5$ means higher values for game players): On average, the tests written by CODE DEFENDERS players achieved 34.3% branch coverage, while manual testers achieved 34.7%. In terms of

[1] http://www.eclemma.org/jacoco, accessed August 2016



(a) Number of Tests (b) Branch Cov. (c) Mutation Score

Fig. 4: Boxplots comparing the number of tests created, branch coverage and mutation scores achieved when using Code Defenders vs manual testing (Means indicated with red dots).

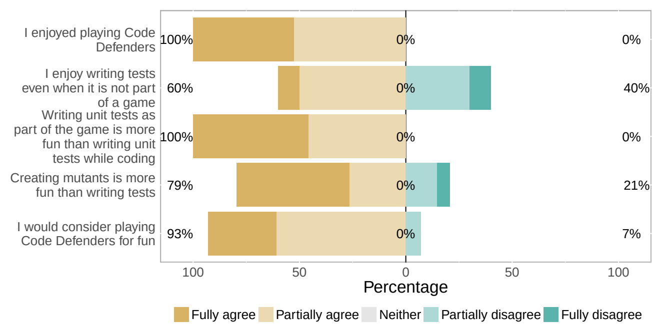


Fig. 5: Exit survey results

mutation score (Figure 4(c)) the tests written by players are clearly stronger, with an average mutation score of 23.1% vs. 15.5% for tests written by manual testers. The difference in mutation score is statistically significant ($\bar{A}_{12} = 0.68$, p =0.03).

> *RQ1: In our experiment, participants playing* CODE DEFENDERS *wrote stronger tests than those not playing.*

2) RQ2: Do testers enjoy writing tests more when playing a game?: For space reasons we cannot provide the complete survey results. To answer RQ2, we focus on the level of agreement expressed by the participants of the experiment with the following five statements: (i) I enjoyed playing Code Defenders; (ii) I enjoy writing tests even when it is not part of a game; (iii) Writing unit tests as part of the game is more fun than writing unit tests while coding; (iv) Creating mutants is more fun than writing tests; (v) I would consider playing Code Defenders for fun. Data on all other questions (most of which are related to the game experience and possible improvements) is available at **http://study.code-defenders.org**. Figure 5 shows that all participants enjoyed playing the game. 60% at least partially agree that writing tests in general can be fun, but all participants agree that writing tests was more fun as part of CODE DEFENDERS. All but 3 participants also claimed they would consider playing CODE DEFENDERS again. Overall, these responses indicate that the testing task is more engaging for participants when performed in a gamified scenario.

> *RQ2: Participants of our experiments claim they enjoyed writing tests more when playing* CODE DEFENDERS.

Figure 5 also shows a strong tendency that creating mutants is more enjoyable than creating tests. This is not surprising; however, the short duration of the game did not allow many equivalence duels to take place, in which case the attacker also has to write tests. The answers to the survey suggested a range of improvements to game, mainly related to the user interface and the point scoring system, which we considered when designing the multi-player version of CODE DEFENDERS.

V. CAN WE CROWDSOURCE TESTS AND MUTANTS?

Having established that players engage well with CODE DEFENDERS and produce useful tests, the question now is whether we can make use of the game and apply it in a crowdsourcing scenario, where multiple players compete and deliver good test suites and mutants. To this end, we implemented the multi-player version of CODE DEFENDERS described in this paper, and collected data from a number of games in order to answer the following research questions:

RQ3: Does crowdsourcing lead to stronger test suites than automated test generation?

RQ4: Does crowdsourcing lead to stronger mutants than automatically generated mutants?

RQ5: How well does the ratio of killed to total manual mutants predict the mutation score?

A. Experiment setup

We followed a systematic procedure to select 20 classes from the SF110 [18] repository, which consists of randomly sampled SourceForge projects as well as the top ten most popular ones, and the Apache Commons (AC) libraries [3]. First, because CODE DEFENDERS currently lacks support for browsing source code trees, we selected classes which compile in isolation (i.e., no dependencies). Next, we restricted the search by size and identified all classes with 100–300 non-commenting lines of code[2]. Our experience from previous user studies suggests that classes in this size range tend to be suitable for experimental unit testing tasks [36]. These two automated filters narrowed the search down to 169 classes (57 from AC and 112 from SF110), which were then independently inspected and ranked by the first three authors of this paper. The ranking criteria included complexity (e.g., does the class implement interesting logic?), purpose (e.g., is the class understandable without context?) and testability (e.g., does the class contain public observers?). Finally, the rankings were combined to choose twenty of the top ranked classes while preserving some diversity in the final selection. Table I lists the classes together with their size (in NCSS), their project, and the number of mutants created by Major (as an indicator of complexity).

Twenty games were then scheduled over the course of 15 days, one per selected class. Participants of the first study were invited to play the games, and the invitation was extended to academic and industrial contacts via direct emails, email lists and social media. In total, 35 unique participants signed up and took part in at least one game. Participants were free to

[2]Measured by JavaNCSS (https://github.com/codehaus/javancss)

TABLE I: Classes selected for crowdsourcing experiment.

Class	Project	NCSS	Major Mutants
ByteArrayHashMap	sf-vuze	179	174
ByteVector	sf-jiprof	128	311
ChunkedLongArray	sf-summa	102	230
FTPFile	ac-net	158	111
FontInfo	sf-squirrel-sql	104	66
HierarchyPropertyParser*	sf-weka	261	286
HSLColor	sf-vuze	160	651
ImprovedStreamTokenizer*	sf-caloriecount	128	111
ImprovedTokenizer	sf-caloriecount	163	77
Inflection	sf-schemaspy	112	105
IntHashMap	sf-vuze	113	145
ParameterParser	ac-fileupload	108	172
Range	ac-lang3	128	158
RationalNumber	ac-imaging	108	286
SubjectParser	sf-newzgrabber	117	136
TimeStamp	ac-net	103	209
VCardBean	sf-heal	188	184
WeakHashtable	ac-logging	168	93
XmlElement	sf-inspirento	196	166
XMLParser	sf-fim1	162	76

∗ Hereinafter abbreviated HPropertyParser and IStreamTokenizer.

TABLE II: Details of the 20 multi-player games played.

Class	Att.	Def.	Mut.	Tests	Killed	Equiv.	Score (A-D)
ByteArrayHashMap	5	4	126	46	73	0	**877** - 206
ByteVector	4	3	57	55	44	0	**136** - 90
ChunkedLongArray	5	5	94	16	41	8	**583** - 77
FontInfo	3	3	33	68	26	1	14 - **50**
FTPFile	4	4	34	66	29	0	31 - **52**
HPropertyParser	3	4	66	23	53	1	**178** - 174
HSLColor	5	5	50	15	33	2	18 - **68**
IStreamTokenizer	5	3	83	32	73	5	221 - **252**
ImprovedTokenizer	5	3	129	26	107	2	346 - **348**
Inflection	4	3	13	26	9	1	**65** - 56
IntHashMap	4	4	71	83	45	2	**742** - 201
ParameterParser	4	4	68	47	45	0	**678** - 117
Range	4	3	154	35	114	1	**232** - 226
RationalNumber	3	5	60	54	32	6	**242** - 117
SubjectParser	4	5	28	17	14	2	**40** - 16
TimeStamp	4	4	32	15	31	0	32 - **50**
VCardBean	4	4	174	123	141	7	501 - **923**
WeakHashtable	3	4	41	11	9	0	**50** - 40
XmlElement	4	3	177	49	134	4	315 - **372**
XMLParser	4	5	27	24	21	1	50 - **90**
Mean	4.05	3.90	41.55	75.85	53.70	2.15	

chose which games to play and which team to join in each game (attackers or defenders). In order to start, at least three attackers and three defenders were required and at most five players could join each team. Games started on their scheduled date and time, or were delayed until the minimum number of players was met, and lasted for 24 hours from its starting time. As incentive to play, the best attacker and defender in each game were awarded GBP10 in shopping vouchers.

Each game resulted in a set of mutants and a test suite containing all tests created in the game. To answer RQ3, we compared these test suites with automatically generated test suites in terms of branch coverage (using Jacoco) and mutation score (using Major). We chose EvoSuite and Randoop as representatives of state-of-the-art test generation tools for Java [37] and run them with default configurations and a one minute time budget to generate 30 test suites per class per tool (to account for the randomized algorithms they implement). This mutation analysis was performed at a *test-suite level*.

To answer RQ4, we run Randoop again on each game class to generate one single test suite with up to 1,000 random tests with a 10-minute time budget. We executed each of these tests *individually* on all the mutants generated by Major and on all the mutants created in CODE DEFENDERS. For each of these mutants, we calculated the number of Randoop tests that kill it. Intuitively, the ratio of random tests that kills a mutant indicates how "difficult" the mutant is.

Finally, to answer RQ5, we calculated the mutation scores of the test suites generated for RQ3 and RQ4 on all Major mutants as well as all mutants generated in the game, and investigated the correlation between these scores.

B. Threats to validity

Threats to validity caused by our object selection, automation, and metrics are similar to what is described in Section IV-B.

The crowdsourcing nature of this second experiment affects the participant selection. We advertised the experiment among the participants of our first study as well as standard email channels and social media; 17 participants of the original study took part, and 18 new external participants were recruited. External participants did not receive the same training participants of the first study received, but instead learned about the game purely from the help page on the website and by playing practice games on their own. It is possible that in practice participants may have more diverse qualifications and skills. However, the multi-player nature of the game means that the results are not dependent on the skills of individual players, and remuneration based on contribution would pose no financial risk to including worse players. Nevertheless, finding qualified participants is a general concern in crowdsourcing and requires careful planning of incentives. Participants chose the games and their roles without our influence. All classes originate from open source projects; to prevent players searching for existing tests for them, we anonymized all classes by removing all project-specific details, including package declarations. As games were run in sequence and participants were allowed to join more than one game, there may be learning effects between games. To reduce these effects, we only ran one game per class, which avoids learning effects on the CUTs.

Test suites are compared against those produced by Randoop and EvoSuite using the default configuration of the tools. This means that the time spent on automatically generating tests is not comparable to that spent on playing the game. It is possible that running the tools longer would improve their test suites; however, there are fundamental limitations in the tools [32], [40], [49] that our approach aims to overcome.

C. Results

Table II summarizes the 20 games that were played. On average, there were 4.05 attackers, submitting a mean of 75.85 mutants. The average number of defenders was 3.9, submitting a mean of 41.55 tests per game. Out of the 20 games, 12 were won by the defending teams and 8 by the attacking teams, suggesting that overall the scoring is well balanced.

TABLE III: Comparison of test suites generated with CODE DEFENDERS with automatically generated test suites.

Class	Branch Coverage					Mutation Score (Major)					Mutation Score (Code Defenders)				
	Code D.	Randoop		EvoSuite		Code D.	Randoop		EvoSuite		Code D.	Randoop		EvoSuite	
		Cov.	A12	Cov.	A12		Score	A12	Score	A12		Score	A12	Score	A12
ByteArrayHashMap	86.49%	**78.02%**	**0.97**	**33.69%**	**1.00**	67.82%	50.85%	1.00	49.31%	0.97	57.94%	18.52%	1.00	12.41%	1.00
ByteVector	100.00%	**45.45%**	**1.00**	**55.68%**	**1.00**	72.35%	25.22%	1.00	28.89%	1.00	77.19%	0.58%	1.00	8.60%	1.00
ChunkedLongArray	100.00%	94.07%	0.73	94.81%	0.78	68.26%	77.24%	0.07	30.38%	1.00	43.62%	89.72%	0.00	25.96%	0.80
FontInfo	88.00%	**77.53%**	**1.00**	**90.87%**	**0.12**	84.68%	49.64%	1.00	42.49%	1.00	78.79%	55.56%	1.00	57.88%	1.00
FTPFile	89.47%	**57.02%**	**1.00**	86.14%	0.75	86.36%	62.59%	1.00	83.69%	0.83	85.29%	50.00%	1.00	27.65%	1.00
HPropertyParser	78.00%	**60.70%**	**1.00**	**91.77%**	**0.00**	56.64%	42.10%	0.97	40.56%	1.00	80.30%	42.42%	1.00	33.48%	1.00
HSLColor	98.28%	**96.26%**	**1.00**	**97.13%**	**0.83**	89.71%	83.23%	1.00	45.74%	1.00	66.00%	70.67%	0.00	43.80%	0.95
IStreamTokenizer	100.00%	**14.81%**	**1.00**	**78.27%**	**1.00**	88.29%	**21.62%**	**1.00**	35.66%	1.00	87.95%	6.02%	1.00	14.46%	1.00
ImprovedTokenizer	95.00%	**91.92%**	**1.00**	**90.17%**	**0.87**	68.83%	**71.38%**	**0.00**	38.10%	1.00	82.95%	60.47%	1.00	26.20%	1.00
Inflection	85.00%	**80.00%**	**1.00**	**76.00%**	**0.85**	42.86%	**27.62%**	**1.00**	22.48%	1.00	69.23%	33.33%	1.00	26.15%	1.00
IntHashMap	90.48%	**98.57%**	**0.00**	**97.70%**	**0.00**	71.03%	78.02%	0.00	62.55%	0.97	63.38%	56.34%	1.00	24.08%	1.00
ParameterParser	89.74%	**58.63%**	**1.00**	88.59%	0.68	66.86%	27.41%	1.00	36.43%	1.00	66.18%	3.43%	1.00	10.15%	1.00
Range	96.15%	**0.00%**	**1.00**	97.18%	0.27	84.81%	0.00%	1.00	64.81%	1.00	74.03%	0.00%	1.00	0.00%	1.00
RationalNumber	83.33%	**65.00%**	**1.00**	77.17%	0.67	52.10%	55.94%	0.00	54.85%	0.25	53.33%	47.50%	1.00	45.83%	1.00
SubjectParser	85.71%	**25.00%**	**1.00**	**81.07%**	**0.88**	69.85%	**19.85%**	**1.00**	41.18%	1.00	50.00%	7.14%	1.00	**21.79%**	**1.00**
TimeStamp	100.00%	**93.33%**	**1.00**	100.00%	0.50	88.04%	**95.12%**	**0.07**	85.33%	1.00	96.88%	100.00%	0.00	62.50%	1.00
VCardBean	95.45%	**87.46%**	**1.00**	**71.36%**	**0.90**	82.61%	70.99%	1.00	44.84%	1.00	81.03%	38.22%	1.00	35.00%	1.00
WeakHashtable	53.33%	**0.00%**	**1.00**	**75.33%**	**0.00**	6.45%	0.00%	1.00	9.93%	0.00	21.95%	0.00%	1.00	17.32%	0.85
XmlElement	80.00%	**68.62%**	**1.00**	76.81%	0.63	69.28%	35.65%	1.00	35.54%	1.00	75.71%	33.05%	1.00	27.40%	1.00
XMLParser	86.11%	**13.89%**	**1.00**	**48.06%**	**1.00**	73.68%	**14.47%**	**1.00**	31.71%	1.00	77.78%	3.70%	1.00	15.93%	1.00
Mean	89.03%	60.32%	0.94	80.39%	0.64	69.53%	45.45%	0.76	44.22%	0.90	69.48%	35.83%	0.85	26.83%	0.98

1) RQ3: Does crowdsourcing lead to stronger test suites than automated test generation?: Table III compares the tests written by players of CODE DEFENDERS with those generated with Randoop and EvoSuite. On average, the CODE DEFENDERS test suites achieved 89.03% branch coverage, whereas Randoop achieved 60.32% and EvoSuite 80.39%. The branch coverage achieved by Randoop was lower in 19 of 20 cases, and significantly so in 18 cases; note that Randoop could not generate any tests for class WeakHashtable and produced only non-compilable tests for class Range (both cases due to Java generics). Randoop achieved a significantly higher branch coverage for class IntHashMap. On closer look at how the game for this class evolved, we observed that the in-game tests missed 4 branches that the Randoop test suites did cover. A plausible conjecture, that also applies for the rest of the games, is that the CODE DEFENDERS highlighting feature, which currently only shows line coverage rather than branch coverage, may have misled defenders into thinking some parts of the code were fully tested, when in reality they were not. The average effect size of $\bar{A}_{12} = 0.94$ confirms that the CODE DEFENDERS tests indeed achieve substantially higher coverage. For 14 classes coverage is also higher than that of the test suites generated by EvoSuite, with 9 being significant. However, there are also 5 classes (4 significant) where EvoSuite achieved higher coverage, and one where the coverage is identical.

The average mutation score calculated by Major on the CODE DEFENDERS test suites is 69.53%, which is again substantially higher than that achieved by Randoop (45.45% on average) and EvoSuite (44.22% on average). There are 16 classes where the mutation score is higher than Randoop's (significant in 4 cases), but there are also 4 cases where the mutation score is lower (significant in 2 cases). Compared to EvoSuite there are no cases with significant differences, but the mutation score of the CODE DEFENDERS test suites is higher in 18 cases.

Finally, we also calculated the mutation scores based on the mutants generated during the gameplay. A similar pattern is revealed here: For ChunkedLongArray, HSLColor, and TimeStamp the Randoop test suites have higher mutation scores, but for all other comparisons the CODE DEFENDERS test suites have higher scores. On average, CODE DEFENDERS tests achieve a mutation score of 69.48%, whereas Randoop and EvoSuite tests only achieve 35.83% and 26.83%, respectively.

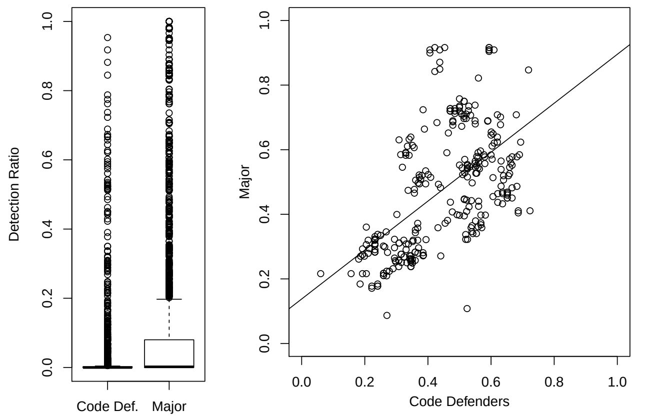
RQ3: Crowdsourcing achieves higher coverage and mutation scores than state-of-the-art test generation tools.

Example. The following test, created in the game played on class WeakHashtable, illustrates how players use stronger assertions than the regression assertions that automated tools are able to generate [40] (for example, by asserting on chains of calls, and using observers that take parameters):

```java
java.util.HashMap foo = new java.util.HashMap();
WeakHashtable w = new WeakHashtable();
foo.put("a","b");
w.putAll(foo);
assertTrue(w.keySet().contains("a"));
assertTrue(w.containsKey("a"));
```

However good for coverage and fault-detection, tests created in CODE DEFENDERS may require post-processing: some players used profane words in string literals and used esoteric stratagems to bypass our test code restrictions.

2) RQ4: Does crowdsourcing lead to stronger mutants than automatically generated mutants?: Figure 6(a) shows the detection rates for mutants resulting from CODE DEFENDERS and those generated by Major. The detection rate is the ratio of 1,000 randomly generated tests that detects a mutant; the lower it is, the harder the mutant is to detect. As we do not know which Major mutants are equivalent, we calculate hardness on *all* mutants; results are similar if considering only mutants killed by the random tests. On average, the detection rate is 0.04 for CODE DEFENDERS mutants, and 0.09 for Major mutants.



(a) Detection Rate (b) Score Correlation

Fig. 6: Comparison of Code Defenders and Major mutants

The difference is significant according to a Mann-Whitney U test at $p < 0.001$ with a medium effect size of $\bar{A}_{12} = 0.37$. Consequently, CODE DEFENDERS mutants are harder to kill.

> *RQ4: Mutants created by* CODE DEFENDERS *are harder to kill than mutants created by Major.*

Examples. The highest-scoring mutant in the study is a subtle replacement of a 16-base number in class IntHashMap, which was ultimately killed in the game, but survived all automatically generated tests ("-" means original class; "+" means mutant):

```
-   int index = (hash & 0x7FFFFFFF) % tab.length;
+   int index = (hash & 0x7EFFFFFF) % tab.length;
```

While this mutant is similar in nature to the common constant replacement mutation operator, it does suggest that players can identify subtle mutants. Other mutants are different to standard operators, for example by replacing references:

```
-   return this;
+   return new ByteVector();
```

Again, this mutant was not killed by any generated tests, although it was killed in the game. Finally, string modifications were common throughout the games, supporting recent evidence that such operators are missing in standard mutation tools [25]. For example, the following mutant for ImprovedTokenizer requires testers to use different delimiters, including one with a capital letter. This mutant was killed in CODE DEFENDERS, but not by any generated tests.

```
-   myDelimiters = delimiters;
+   myDelimiters = delimiters.toLowerCase();
```

3) RQ5: How well does the ratio of killed manual mutants predict the mutation score?: While having strong mutants is helpful for guiding test generation, mutants are also typically used to assess the quality of a test suite. It has been shown that mutation scores (on mutants generated with Major) correlate to real fault detection [25]; thus we would like to see whether mutation scores calculated using CODE DEFENDERS mutants are similar to mutation scores calculated on standard mutants. Figure 6(b) plots the relation of the mutation scores: There is a moderate positive correlation (Spearman 0.59, $p < 0.001$; Pearson's r 0.57, $p < 0.001$, Kendall's tau 0.39, $p < 0.001$) between the two, suggesting that CODE DEFENDERS mutants are suitable for calculating mutation scores. The slightly lower scores suggest that CODE DEFENDERS leads to less inflated scores [29] than mutation tools.

> *RQ5: There is moderate positive correlation between* CODE DEFENDERS *and Major mutation scores.*

D. Discussion

Some aspects more intrinsic to the dynamics of the game only surfaced as a result of observing the games played during our experiments. We observed that if defenders or attackers do not engage in the game early after it starts, they play in disadvantage and may feel discouraged to submit new mutants or tests, and therefore negatively affect the final outcome of the game. This undesirable effect is notorious in the games played on classes SubjectParser and TimeStamp, where one single defender submitted strong sets of tests early in the game, such that the rest of defenders remained inactive throughout those games. Alternatives to prevent games from early stagnation could involve ranking tests by non-functional properties (e.g., length or readability [12]) such that defenders have the chance to catch up if they submit shorter, more readable tests, possibly even *stealing* points from other team members. In general, an open challenge is to foster the creation of mutants and tests that are not only strong, but also of high quality.

Player motivation and engagement are key factors to the success of the CODE DEFENDERS crowdsourcing approach. The game played on class WeakHashtable showcases this problem: Players simply did not engage with this game and created only 41 mutants and merely 11 tests, achieving the lowest code coverage and mutation score in the experiment. It is worth noting, however, that WeakHashtable is likely one of the most complex classes in our experiment.

The last game of our experiment (class XMLParser) illustrates a scenario where human-written tests are unmatched by state-of-the-art test generation tools. Based on our limited empirical evidence, we speculate that the CODE DEFENDERS approach could be particularly apt and worthwhile for testing code with more complex logic involved, on which automated test generation tools often struggle.

VI. RELATED WORK

There are several successful examples of gamification for software engineering, where the methodology has been applied mostly to increase the motivation and performance of people participating in software engineering activities [33], such as removing static analysis warnings [4] or committing changes often to version control [41]. In contrast, CODE DEFENDERS is intended for outsourcing some of the developers work, rather than getting them more engaged with the testing tasks.

Chen and Kim [11] designed a game to support automated test generation, by letting humans solve puzzles that represent object mutation or constraint solving problems. While the aim of improving test generation is similar to ours, the approach is purely based on puzzle-solving, and does not make use of competitive or cooperative elements. However, it might be possible to involve automated test generation tools in the CODE

DEFENDERS game, in order to drive players to focus on areas where the tools struggled.

Dietl et al. [14] gamified the verification of certain program properties. Players of these games are not directly aware of the underlying verification task that is being solved. In contrast, CODE DEFENDERS makes explicit use of the coding skills of participants. However, abstracting away from code is something that might enable the application of CODE DEFENDERS to other types of testing in the future.

CodeHunt [8], based on the earlier Pex4Fun [44], is an online game that integrates coding and test generation. In contrast to CODE DEFENDERS, testing is not an active task in itself in CodeHunt but is used to help players find the solution to coding puzzles. The aim is programming education, and players do not contribute to writing tests.

There have been some attempts to use gamification in an educational setting to better engage students with software testing. For example, Elbaum et al. [16] developed Bug Hunt, a web-based tutorial system where students have to apply different testing techniques to solve challenges. Bell et al. [6] use storylines and quests to gradually introduce students to testing without explicitly telling them. While this paper explicitly focuses on the crowdsourcing aspect of CODE DEFENDERS, we are also considering an educational angle [35].

Crowdsourcing has also been used in relation to software testing without gamification elements. Crowdsourced testing is now a common practice in industry, but unlike CODE DEFENDERS the focus is mainly testing of mobile and web applications [51]. Testing has also been considered [45] as part of a general collaborative and crowdsourced approach to software engineering [26]. Pastore et al. [31] used crowdsourcing on Amazon Mechanical Turk in order to have crowdworkers confirm test oracles with respect to API documentation. Tests were generated automatically using automated unit test generation tools. CODE DEFENDERS currently does not address the test oracle problem, and this approach is thus complementary.

VII. CONCLUSIONS AND FUTURE WORK

Writing good tests and good mutants are hard tasks, and automated tools often reach the limits of their capabilities in practice. In this paper, we proposed an alternative approach based on gamification and crowdsourcing: Teams of players compete by attacking a program under test with subtle mutants, and defending the program with tests. At the end of a game there are sets of strong tests and mutants. Our evaluation on 20 open source Java classes shows that the CODE DEFENDERS game achieved higher coverage and mutation scores than state-of-the-art test generation tools, confirming that this is a promising avenue of research.

There remains, however, much to be done as future work:

Collaboration: In its current form, the game is based on the competitive nature of players, not only across but also within teams: an attacker wants to defeat the defenders, but also wants to score more points than the rest of attackers. However, stronger tests and mutants might result if players could team up and take on testing challenges involving working together to fully test a program (defenders) or to try to break an existing test suite (attackers).

Abstraction: The gameplay is currently based on writing and modifying program code directly, and is thus very similar in nature to the underlying problem that we are trying to solve with the game. While there are successful crowdsourcing models based on coding tasks (e.g., TopCoder[3]), games are often successful when played with more graphical interactions and tasks that are more abstract than actual coding. Research on code visualization, for example the city metaphor [48], may be well suited for this, in particular as Balogh et al. [5] recently demonstrated how to include test related metrics.

Dependencies: When selecting classes for our experiments, we deliberately used a minimal number of dependencies as one of the criteria. Including more complex dependency classes might require players to get additional information (e.g., API docs) which would have implications about the user interface and playability. Thus, more effective ways to handle complex dependency objects will need to be developed.

Test oracles: CODE DEFENDERS currently produces regression tests and mutants, like automated tools also do. However, it would be even better to have players provide real test oracles. For example, testers could base their assertions on API specifications rather than source code, similar to the CrowdOracles approach [31]. However, the gameplay would need to be adapted, as tests would then also fail on the program under test if a real bug is discovered.

Testing aspects: CODE DEFENDERS currently targets unit testing of Java classes, and rewards tests that are good at detecting faults. A main reason for this lies in the comparability to automated tools. It will be of interest to transfer the game to other languages, other types of testing (e.g., GUI testing), and to optimize other attributes of tests (e.g., readability).

Tool integration: The starting point of a game currently is an empty test suite and no mutants. Artefacts generated by tools may offer a different starting point, to focus the game on those aspects the tools struggle with. It may also be possible to integrate these tools as further incentive mechanisms (e.g., by trading points against automatically generated tests).

Incentives: Besides the general competitive nature of the game, in our experiments we used prizes (Amazon vouchers) for the winner of each team as incentive. While this may be a suitable approach for conducting a research study, in practice more refined strategies will be required, for example where each participant receives payment proportional to their contribution (e.g. based on points). Existing research on incentive mechanisms (e.g., [38]) may help to identify improvements.

Application: While our experiments have demonstrated the general feasibility of the idea, we have not yet explored how the game would be applied in practice. There remain open questions, such as how long games should last, how many players they need, and what the costs would be. Furthermore, there are open questions around when and on which code one would apply such an approach rather than automated tools.

[3]https://www.topcoder.com, accessed August 2016

References

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1338–1349. Springer Berlin Heidelberg, 2004.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 402–411. ACM, 2005.

[3] Apache Software Foundation. Apache Commons Libraries, 2016. http://commons.apache.org/.

[4] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa. A gamified tool for motivating developers to remove warnings of bug pattern tools. In *Int. Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 37–42. IEEE, 2014.

[5] G. Balogh, T. Gergely, Á. Beszédes, and T. Gyimóthy. Using the city metaphor for visualizing test-related metrics. In *Int. Conference on Software Analysis, Evolution, and Reengineering, the First International Workshop on Validating Software Tests (VST)*, pages 17–20, 2016.

[6] J. Bell, S. Sheth, and G. Kaiser. Secret ninja testing with halo software engineering. In *Int. workshop on Social software engineering*, pages 43–47. ACM, 2011.

[7] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[8] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux. Code hunt: Experience with coding contests at scale. *ACM/IEEE Int. Conference on Software Engineering (ICSE)(JSEET track)*, pages 398–407, 2015.

[9] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Inf.*, 18:31–45, 1982.

[10] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *IEEE Int. Software Metrics Symposium*, pages 239–249, 2003.

[11] N. Chen and S. Kim. Puzzle-based automatic testing: bringing humans into the loop by solving puzzles. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 140–149. ACM, 2012.

[12] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 107–118. ACM, 2015.

[13] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. From game design elements to gamefulness: Defining "gamification". In *Int. Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek)*, pages 9–15. ACM, 2011.

[14] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović. Verification games: Making verification fun. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, pages 42–49. ACM, 2012.

[15] D. J. Dubois and G. Tamburrelli. Understanding gamification mechanisms for software development. In *Joint Meeting on Foundations of Software Engineering*, pages 659–662. ACM, 2013.

[16] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 688–697, 2007.

[17] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

[18] G. Fraser and A. Arcuri. A large scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[19] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Int. Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369. IEEE, 2013.

[20] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects—A comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[21] J. Howe. The rise of crowdsourcing. *Wired*, 14(6), June 2006.

[22] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 435–445. ACM, 2014.

[23] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, 2014.

[24] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 315–326. ACM, 2014.

[25] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 654–665. ACM, 2014.

[26] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *RN*, 15(01), 2015.

[27] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability (STVR)*, 7(3):165–192, 1997.

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 75–84. IEEE, 2007.

[29] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon. Threats to the validity of mutation-based test assessment. In *Int. Symposium on Software Testing and Analysis*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM.

[30] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 936–946. IEEE Press, 2015.

[31] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *IEEE Int. Conference on Software Testing, Verification and Validation*, pages 342–351. IEEE, 2013.

[32] Y. Pavlov and G. Fraser. Semi-automatic search-based test generation. In *IEEE Int. Conference on Software Testing, Verification and Validation*, pages 777–784. IEEE, 2012.

[33] O. Pedreira, F. Garca, N. Brisaboa, and M. Piattini. Gamification in software engineering a systematic mapping. *Information and Software Technology (IST)*, 57:157 – 168, 2015.

[34] J. M. Rojas and G. Fraser. Code Defenders: A Mutation Testing Game. In *Int. Workshop on Mutation Analysis (ICSTW)*, pages 162–167. IEEE, 2016.

[35] J. M. Rojas and G. Fraser. Teaching Mutation Testing using Gamification. In *European Conference on Software Engineering Education (ECSEE)*, 2016.

[36] J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 338–349. ACM, 2015.

[37] U. Rueda, R. Just, J. P. Galeotti, and T. E. Vos. Unit testing tool competition: round four. In *Int. Workshop on Search-Based Software Testing*, pages 19–28. ACM, 2016.

[38] O. Scekic, H.-L. Truong, and S. Dustdar. Incentives and rewarding in social computing. *Communications of the ACM*, 56(6):72–82, 2013.

[39] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability (STVR)*, 23(5):353–374, 2013.

[40] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[41] L. Singer and K. Schneider. It was a bit of a race: Gamification of version control. In *Int. Workshop on Games and Software Engineering (GAS)*, pages 5–8. IEEE, 2012.

[42] K. T. Stolee and S. Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *The ACM-IEEE Int. Symposium on Empirical Software Engineering and Measurement*, page 35. ACM, 2010.

[43] N. Tillmann and J. De Halleux. Pex–white box test generation for .NET. In *Int. Conference on Tests and Proofs (TAP)*, pages 134–153. Springer, 2008.

[44] N. Tillmann, J. De Halleux, T. Xie, and J. Bishop. Pex4fun: Teaching and learning computer science via social gaming. In *Conference on Software Engineering Education and Training*, pages 90–91. IEEE, 2012.

[45] Y.-H. Tung and S.-S. Tseng. A novel approach to collaborative testing in a crowdsourcing environment. *Journal of Systems and Software*, 86(8):2143–2153, 2013.

[46] L. von Ahn. Duolingo: learn a language for free while helping to translate the web. In *Int. Conference on Intelligent User Interfaces (IUI)*, pages 1–2. ACM, 2013.

[47] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[48] R. Wettel and M. Lanza. Visualizing software systems as cities. In *IEEE Int. Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. IEEE, 2007.

[49] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux. Precise identification of problems for structural test generation. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 611–620. ACM, 2011.

[50] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 919–930. ACM, 2014.

[51] S. Zogaj, U. Bretschneider, and J. M. Leimeister. Managing crowdsourced software testing: a case study based insight on the challenges of a crowdsourcing intermediary. *Journal of Business Economics*, 84(3):375–405, 2014.