Gamification of Loop-Invariant Discovery from Code

Andrew T. Walter, Benjamin Boskin, Seth Cooper, Panagiotis Manolios
Khoury College of Computer Sciences
360 Huntington Avenue
Boston MA, 02115
{walter.a, boskin.b}@husky.neu.edu, se.cooper@northeastern.edu, pete@ccs.neu.edu

Abstract

Software verification addresses the important societal problem of software correctness by using tools to mechanically prove that software is free of errors. Since the software verification problem is undecidable, automated tools have limited capabilities; hence, to verify non-trivial software, engineers use human-in-the-loop theorem provers that depend on human-provided insights such as loop invariants. The effective use of modern theorem provers requires significant expertise and recent work has explored the possibility of creating human computation games that enable non-experts to find useful loop invariants. A common feature of these games is that they do not show the code to be verified. We present and evaluate a game which does show players code. Showing code poses a number of design challenges, such as avoiding cognitive overload, but, as our experimental evaluation confirms, also provides an opportunity for richer human-computer interactions that lead to more effective human-in-the-loop systems which augment the ability of programmers who are not verification experts to find loop invariants.

1 Introduction

Society is crucially dependent on software, which is used to control critical infrastructure, medical devices, power plants, trains, automobiles, planes, stock markets and sensitive data such as medical and financial records. Bugs in software have lead to deaths, the loss of massive amounts of money and numerous security vulnerabilities (RTI 2002).

The field of software verification is concerned with tackling the problem of erroneous software in a foundational way, by proving that software is free of errors. This is done with the aid of reasoning tools such as theorem provers, interactive tools that help humans construct mechanically-checked proofs. Theorem provers are capable of automatically generating simple proofs, but since the verification problem is undecidable, significant human interaction by experts is typically needed to effectively use such tools. A brief overview of software verification appears in Section 2.

Recent work has shown that it is possible to create human computation games where non-expert players find loop invariants that allow theorem provers to reason about programs that are too complex to automatically analyze. As discussed in Sections 3 and 4, these games are based on abstractions such as traces and do not show players code.

We challenge the assumption that showing code is a bad idea. In fact, we propose that the opposite is true and to test our hypothesis, we present a game where players are shown code in Sections 5 and 6. Showing code can lead to cognitive overload, and requires new kinds of human-computer interfaces and interactions. We discuss how our game overcomes these challenges, allowing it to give players with programming experience, but no verification expertise, actionable feedback. An in-depth evaluation of our game, in Section 7, shows that our work leads to more effective human-in-the-loop systems that significantly augment the ability of non-experts to find loop invariants. For increased confidence in our evaluation, we decided to implement our own version of the state-of-the-art in loop invariant discovery games (in Section 4). This allows us to account for differences in player populations between our players and previous work, determine if we can replicate previous results, enforce definitions, and use the same metrics and statistical tests. We end with design insights, limitations, future work and conclusions. Our games are available for play online. [1]

2 Software Verification

In this section, we present some key concepts and terminology from the field of software verification, which will be used throughout the rest of the paper. As a working example, we use the program MULTIPLY, shown in Figure 1, which takes two natural numbers as input and multiplies them together. The program is written in a Simple Imperative Programming language (SIP), where the notation $[\![\texttt{A}]\!]$ denotes that `A` is an *invariant*, a Boolean-valued expression that is always true when program execution reaches it. $[\![\texttt{G}]\!]$ is the *guarantee*, a statement that we expect to be true at the end of the program, stating that `res` is the product of the inputs, `n` and `m`. Guarantees are used to specify and characterize program behavior: MULTIPLY is a program that given two natural numbers, multiplies them.

One way to check MULTIPLY is to test it, but, as is well known, while testing can reveal the existence of errors, it cannot prove that no errors exist. The most foundational way

[1] See `http://invgame.atwalter.com/dashboard1` and `http://invgame.atwalter.com/dashboard2`.

```
main(n, m: nat)
{ var res, cnt: int;

  res := 0;
  cnt := 0;

  while ⟦I⟧ (cnt < m)
  { print(n, m, cnt, res);
    cnt  := cnt+1;
    res  := res+n;
    ⟦I⟧
  }
  print(n, m, cnt, res);
  ⟦G⟧
}
```

Figure 1: MULTIPLY: Our running example program.

to reason about computation is to use *formal methods*, which allow us to prove correctness. This task is in general undecidable, but software verification tools, such as theorem provers, can be used to automate parts of the process.

One of the most difficult areas of software verification is reasoning about loops (Gleiss, Kovács, and Robillard 2018). The standard way of doing this is to use *loop invariants*, denoted ⟦I⟧ in program MULTIPLY, to characterize the behavior of loops. A loop invariant is an expression that holds at the start and end of every iteration of a given loop. An example of a loop invariant for the `while` loop in MULTIPLY is `res=n*cnt`: at every iteration of the loop `cnt` is increased by 1, and `res` is increased by `n`. Loop invariants must be *inductive*, which means:

- the expression holds at the initial entry of the loop, and
- if the expression is true at the beginning of an iteration of the loop, and the loop condition is satisfied, then the expression remains true after the loop body is executed.

The inductivity requirement for loop invariants is what allows us to go from reasoning about *unbounded* program behavior to reasoning about *bounded* program behavior. For example, we can prove that `res=n*cnt` is a loop invariant, by verifying the above two conditions which only involve reasoning about a bounded number of instructions. The points marked ⟦I⟧ in MULTIPLY mark the points of the program significant for inductivity: the start and end of a loop iteration. The values of `cnt` and `res` are both 0 when the loop begins, so `res=n*cnt` holds at the initial entry of the loop. If `res=n*cnt` and the loop condition holds at the first ⟦I⟧, then `res+n=n*(1+cnt)` holds at the second ⟦I⟧ (via simple algebra). In this way, we can show that `res=n*cnt` is a loop invariant, so it holds when program execution reaches ⟦G⟧. Also notice that after a loop finishes, its loop condition must be false. So, when the loop of MULTIPLY is finished, both `res=n*cnt` and `cnt>=m` hold. Unfortunately, `res=n*cnt` is not enough to prove that MULTIPLY satisfies its guarantee. We also have to establish that at ⟦G⟧, `cnt=m` holds, which requires a second loop invariant, `cnt<=m`.

For the rest of this paper, when we say that some set of inductive invariants *proves the guarantee*, or *implies the guarantee*, we mean that the conjunction of a) the types of the variables in the program, b) the set of inductive invariants and c) the negation of the loop condition, logically imply that the guarantee holds. For example, we know that the following statements are true at ⟦G⟧:

- `res=n*cnt`, `cnt<=m` (our loop invariants)
- `cnt>=m` (the negation of the loop condition)

Hence, we can prove that `G` (`res=n*m`) holds.

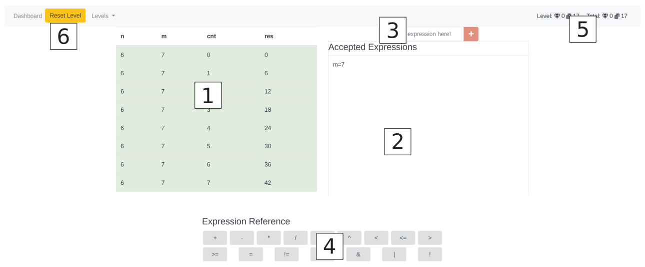
Loop invariants are useful because they characterize a superset of the *reachable states* of a program that may be realized at the start or end of any loop iteration using any input, where a state is an assignment of type-preserving values to program variables. For MULTIPLY, an example of a reachable state is `n=2, m=3, cnt=2, res=4`. An example of a *non-reachable state* is `n=2, m=2, cnt=3, res=5`. We can determine that this state is not reachable using the inductive invariant `res=n*cnt`, which says that at every loop iteration, `res` is a multiple of `n`. Notice that, in general, not every state that satisfies a set of loop invariants is reachable, *e.g.*, the state `n=2, m=3, cnt=-2, res=-4` satisfies the loop invariants, but is not reachable (notice that `cnt, res` are of type `int`). This indicates that we can strengthen the set of loop invariants until we obtain the *strongest set of loop invariants*, which exactly characterize the set of reachable states.

Not every invariant is inductive, *e.g.*, $\neg$(`cnt < m`) $\Rightarrow$ (`res=n*m`) is an invariant that holds when program execution reaches ⟦I⟧, and can be used to prove `G`, but it is not inductive and hence not a loop invariant. Coming up with loop invariants requires insight and is difficult for modern theorem provers, but perhaps regular programmers are better able to discover interesting loop invariants. On the other hand, checking loop invariants is something that modern theorem provers are good at, allowing us to automate the proofs, which is not something regular programmers are trained to do.

Many fine textbooks provide an in-depth overview of formal methods and software verification (Kaufmann, Manolios, and Moore 2000; Bradley and Manna 2007; Dijkstra and Scholten 2011).

3 Related Work in Games

Early work in games dealing with invariants explored different representations of program execution traces and other program elements. These representations were selected to present a more game-like experience, hiding numbers and code, while still asking players to use these representations to help discover invariants or prove properties of the code. Xylem (Logas et al. 2014; 2015; Dean et al. 2015) presented traces to players as plants and flowers, also asking them to discover loop invariants. StormBound (Dean et al. 2015) presented traces as collections of magic symbols and runes. Binary Fission similarly had players work with invariants, but through a higher-level sorting interface informed by an automated system (Fava et al. 2016; Dean et al. 2015). Monster Proof (Dean et al. 2015) presented information as cartoon monsters. These games all made the design decision to hide the code from the players and rely primarily on the presentation of program elements as game pieces.



n	m	cnt	res
6	7	0	0
6	7	1	6
6	7	[illegible]	12
6	7	3	18
6	7	4	24
6	7	5	30
6	7	6	36
6	7	7	42

m=7

Figure 2: Trace Version of the Invariant Discovery Game

InvGame is the most recent and effective game for invariant discovery; empirical evaluation showed that players were able to solve several benchmark problems that were unsolved by leading fully-automated tools (Bounov et al. 2018). InvGame is a puzzle game that challenges players to help prove properties of short pieces of code containing loops. InvGame exposes numerical information about the underlying program by showing the player traces of program execution. Players are asked to enter mathematical expressions that hold true for all entries in the trace. These expressions may turn out to be loop invariants that can be used to prove guarantees. Similar to prior games, InvGame hides the code (as well as the guarantee) from players, relying on the trace to give players enough information about the program. The authors of that tool claimed that hiding code and only showing traces "reduces cognitive burden and ... can allow non-experts to achieve expert tasks." [2]

4 Discovering Invariants from Traces

The fundamental difference between our Invariant Discovery Game (IDG from now on), and InvGame, the state-of-the-art in gamified invariant discovery, is that IDG shows both the program and guarantee to the player, while InvGame abstracts these components of the problem away. In order to effectively compare IDG to InvGame, we created IDG-T, our own version of InvGame, which has a similar user interface and data collection model to IDG. By using the same metrics, player population, definitions, analysis tools and backend solvers for both games, we can fairly evaluate the merit of showing code as a gamification strategy and whether or not the resulting cognitive overhead is justified.

User Interface

In Figure 2, we show a screenshot of the IDG-T game, which essentially mirrors the InvGame layout, aside from cosmetic differences. In this section we discuss the purpose of each screen component, numbered 1 through 6. Every component in the IDG-T screen has a counterpart in the IDG screen, where the IDG counterpart has possibly become interactive (as described in Section 6).

[2]Two days before the camera-ready manuscript submission deadline, we discovered FlowGame, a game for loop invariant discovery that also shows code (Bounov 2018). Our work and FlowGame were developed independently.

The key component in IDG-T is the Trace Window (1), which displays the result of running a (hidden) program on a single, unchangeable, input. The rows reflect the result of printing every local variable, at each iteration of the loop, as well as at the end of the loop. Next, we have the Accepted Expressions box (2). This is where all the expressions proposed by the player and accepted by the game are accumulated. Players enter expressions in the Entry Box (3). There are three requirements which an expression must satisfy in order to be accepted. An expression must be true at every row of the trace, cannot be a tautology and cannot be implied by the conjunction of the other expressions that have already been accepted. If an expression is rejected a succinct explanation is provided.

Finally, because the expression language includes potentially unfamiliar Boolean and arithmetic operators, we provide an Expression Reference (4), where players can read definitions of the meanings of each symbol available, which mirrors a similar feature of InvGame.

Gamification

There are several features of IDG-T which help establish it as a *gamified* (Deterding et al. 2011) approach to loop invariant discovery, including the scoring system, tutorials and levels. The Score Panel (5) lets players see their progress on the level being displayed, as well as their total score across all attempted levels. Players can receive two types of points: coins and gems. A player receives coins when they submit an expression that is accepted; this is intended to be a frequent source of positive reinforcement for players. A player receives gems when they complete a level, either by submitting 6 accepted expressions or submitting loop invariants that imply the guarantee. Each level is worth a different amount of gems and coins depending on its difficulty. The tutorials for both IDG-T and IDG are videos, where an expert player plays the game, explains the game features, describes his thinking process, explores different strategies and shows how to complete one level. Finally, we give players the ability to reset and skip levels (6), so that players who want to explore or get frustrated with a level and want to move on, can freely do so.

5 Limitations of Traces

A single trace can convey a significant amount of information about a program and is a useful abstraction that allows players to discover invariants. There are, however, several issues in current games that arise when traces are the only information available to players. The root of the problem is that the trace abstraction is lossy because there are an infinite number of programs that can produce a given trace. This leads to a sizable semantic gap between the reward system in trace-based games and the underlying goal of invariant discovery, a gap that players can easily exploit to *cheese* these games *i.e.*, to use tactics that make it easy to make game progress without providing any useful invariants. The goal of gamification is to design games which elicit the desired behavior from players: ideal gamifications are cheesing-resistant. Unfortunately, there are several cheesing strategies that can be used with trace-based games.

Trace-specific expressions

The first cheesing strategy is to submit expressions which are trace specific. Consider the trace (1) shown in Figure 2, which is for the multiplication program shown in Figure 1, where the inputs `n` and `m` are 6 and 7. Now consider the expressions `n=6`, `n≠0`, `m=7` and `cnt<=7`, none of which are invariants (inductive or otherwise), but all of which hold for the trace. Since the trace abstraction does not identify input variables, players have no idea that `n` and `m` are inputs whose values can be any natural number. Therefore, these games give players credit for such expressions, even though they are clearly of no help to the verification process. Finally, notice that players may be unaware that they are cheesing the game, because there is a sizable semantic gap between the reward system in the trace game and the underlying goal of invariant discovery.

Backward redundancy checking

The order in which expressions are submitted can be used to cheese trace-based games. When a new expression is submitted, checks are performed to ensure that the new expression is not subsumed by the already accepted expressions. Consider the following sequences of proposed expressions:

```
1. cnt >= -2,  2. cnt >= -1,  3. cnt >= 0
1. cnt >= 0,   2. cnt >= -1,  3. cnt >= -2
```

When the expressions in the first line are submitted all expressions are accepted, although the last expression implies the rest. When the expressions in the second line are submitted, however, only the first expression is accepted; the rest are rejected because they are subsumed. If the only way to complete a level is to prove the guarantee, this is not an issue: redundant expressions cannot help players at all. When players can complete levels by only providing a finite number of useless expressions, as is the case in InvGame, this becomes a problem, as players can pick a variable, find the minimum value it has in the trace and can then easily generate an arbitrarily long sequence of incrementally stronger expressions. One might consider adding backward redundancy checks, but that would make the game seem arbitrary because players could make "negative progress" by submitting a strong expression that subsumes several of their previously submitted expressions.

Inductivity

A final issue is that concepts such as inductivity are out-of-scope, because the location, loop condition and body of the loop in question have been abstracted away. For example, consider the expression `res<=n*m`. This does, in fact, hold for all reachable program states, but proving that it is an invariant requires other inductive invariants. In general, players may submit expressions that are invariants but which do not help prove program correctness because they are not inductive. Trace-based games are not able to provide players with meaningful feedback in such situations, which leads to a space of acceptable expressions that is much larger than the space of expressions which are actually helpful for proving correctness. As the simplest acceptable expressions tend not to be the expressions which are useful for proving correctness, only a fraction of player effort is useful in tackling the underlying software verification problem.

6 Invariant Discovery Game

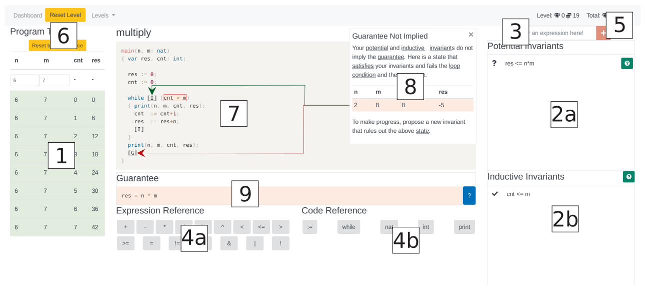
Having shown the limitations of invariant discovery games based on traces, we turn our focus to IDG, where players are shown code and guarantees. Key verification concepts such as input variables, types, program structure, loop invariants and guarantees become transparent entities which can be referred to by the player and the game. Resolving the challenges that arose from the decision to show code enabled us to design a game with more sophisticated gameplay flow and increased interactivity. The design balanced the competing goals of providing as much actionable information as possible while minimizing cognitive overhead and technical jargon. By showing players program behavior not covered by existing invariants, our game helps players discover invariants, leading to a higher rate of successful proof attempts, as shown in Figure 4. Our evaluation provides compelling evidence that showing code in verification games is worth the increased cognitive overhead, as it leads to more effective human-in-the-loop verification systems.

User Interface

Now, we describe the user interface of IDG, shown in Figure 3, describing changes made to the features present in IDG-T (components 1-6) and the new features (components 7-9). The changes are to facilitate a player's understanding of the feedback that they receive and to enable players to explore the program. Code is shown in the code window (7) and the guarantee (9) is shown separately, below the program text. Highlighting the guarantee helps focus players on this objective. The message window (8) is where players receive actionable feedback regarding the expressions they submitted, typically consisting of a concrete program state showing players program behavior not characterized by the current set of invariants. In order to help players build an accurate mental model of the critical program points used to judge invariants, we incorporated supplemental visual stimuli to the message window in the form of arrows which point from the message window to the points in the program relevant to that message. The arrows are colored if they correspond to a location where a proposed expression, the guarantee, or loop condition, holds (green) or doesn't (red). Otherwise, arrows are black. These arrows help to both explain the feedback players receive and to further reinforce unfamiliar verification concepts.

As in IDG-T, there are references to the symbols that players can use in expressions (4a), as well as to forms that appear in programs (4b). In addition, any technical terms used in feedback messages can be hovered over, which shows a short definition. These definitions can remind players of the content of the tutorial video and are a good way of teaching players about loop invariants while they play.

The remaining features of IDG allow players to more fully explore the program's state space. One key change is that in IDG, the trace window (1) is an interactive component, al-



n	m	cnt	res
6	7	-	-
6	7	0	0
6	7	1	6
6	7	2	12
6	7	3	18
6	7	4	24
6	7	5	30
6	7	6	36
6	7	7	42

multiply

```
main(n, m: nat)
{ var res, cnt: int;

  res := 0;
  cnt := 0;

  while ⟦I⟧ (cnt < m)
  { print(n, m, cnt, res);
    cnt  := cnt+1;
    res  := res+n;
    ⟦I⟧
  }
  print(n, m, cnt, res);
  ⟦G⟧
}
```

Guarantee

```
res = n * m
```

Guarantee Not Implied

Your potential and inductive invariants do not imply the guarantee. Here is a state that satisfies your invariants and fails the loop condition and the guarantee.

n	m		res
2	8	8	-5

To make progress, propose a new invariant that rules out the above state.

? res <= n*m

✓ cnt <= m

Figure 3: Code Version of the Invariant Discovery Game

lowing players to enter inputs and generate traces. This capability gives players access to all reachable program states.

Gameplay and Invariant Categorization

IDG is a human computation game that consists of a sequence of interactions between a player and a reasoning engine, which is based on the ACL2s theorem prover (Chamarthi et al. 2011; Chamarthi and Manolios 2019). The reasoning engine presents a program state which satisfies all of the player's accepted invariants but is a counterexample to the claim that the program satisfies the guarantee. The player responds by providing a loop invariant that rules out this counterexample. Since all of the programs used in the game satisfy their guarantees, it is always possible to make progress. This process is repeated until the player provides enough loop invariants to prove the guarantee. A full description of the reasoning engine is beyond the scope of this paper, but we provide a player-level overview.

There are several fundamental categories of proposed invariants. We start by describing the useless categories. Proposed invariants may have reachable counterexamples. If the existing trace provides a counterexample, players get immediate feedback: the rows consisting of counterexamples are shown in red. Otherwise, the reasoning engine generates a new trace showing why the proposed expression is not an invariant. Proposed invariants may be tautologous, so they do not add any information and cannot help prove the guarantee. In such cases, a useful explanation is provided to the player. Proposed invariants may be implied by the existing inductive invariants, so they also do not add any information and a useful explanation is provided to the player. All of these useless invariants are discarded and the player, armed with new information, is asked to propose a new invariant.

Proposed invariants which are not useless and which can be proven inductive when conjoined with existing inductive invariants are added to the Inductive Invariants window (2b). Expressions for which neither of the above cases apply are added to the Potential Invariants window (2a). Potential invariants may get *promoted* to inductive invariants as new inductive invariants are discovered. Recall that we check inductivity by conjoining all known inductive invariants, so as the set of inductive invariants increases, we identify and promote all promotable potential invariants. Finally, a new inductive invariant may subsume existing invariants, in which case we report this to the player and remove the redundant invariants (in an effort to minimize the player's cognitive load). If the removed invariant was inductive, it is called an *displaced inductive* invariant; otherwise if it was potential it is called a *redundant* invariant. After a newly proposed invariant is processed, we check if the current level has been completed, by checking whether the known inductive invariants imply the guarantee. All of the reasoning is done by the reasoning engine, so players never see proofs, nor are they expected to understand proofs. If the level has not been completed, a new counterexample is generated by the reasoning engine and shown to the player.

Players can click on any buttons labeled with a **?** to explore a level. Clicking on the button appearing next to the guarantee (9) produces a counterexample to the claim that the conjunction of the player's inductive and potential invariants imply the guarantee, if such an example exists. Such counterexamples have high utility. Their existence indicates that more invariants are definitely needed. In addition, these counterexamples provide players with a high-quality immediate goal, which is to propose a new invariant that rules out the counterexamples. Clicking on a button in (2a) appearing next to a potential invariant, say p, provides counterexamples to the claim that p, conjoined with the current set of invariants, is inductive. Two states are produced in this case. The first state satisfies all of the invariants and the loop condition. The second state is produced by executing the body of the loop and violates p, showing that if the first state is reachable, then p is not inductive. Players use such counterexamples to generate expressions that when conjoined with existing invariants are strong enough to be inductive.

To summarize, the reasoning engine continuously challenges the player to find invariants, relationships between the program variables, that bound the set of known reachable states until we have enough bounds to establish the guarantee. So, the player provides insight and the reasoning engine turns this insight into (unseen) proofs.

Gamification

The key features of IDG which establish it as a *gamified* (Deterding et al. 2011) approach to loop invariant discovery include the features of IDG-T, using the same scoring system, tutorials and discrimination of levels, but also include interactive interfaces and a richer game-flow consisting of actionable, interactive feedback provided by the analysis engine in response to player actions. IDG uses the same two-tier point system as IDG-T, where accepted invariants lead to coins, and level-completions lead to gems. The difference, however, is that the only way to complete a level is to provide invariants whose conjunction implies the guarantee, which means that the user's gem count reflects the value of the discovered invariants and is not susceptible to the cheesing attacks of trace-based games. The game-flow of IDG attempts to minimize the time players spend uncertain of what to do next. Since players are not expected to understand formal methods, the reasoning engine generates concrete states and messages that provide players with immediate subgoals, playing the role that road signs, objects, or insightful characters play in conventional video games.

		Difficulty	Proved by game?		
Program Name	InvGame		IDG-T	IDG	InvGame
multiply	-	tutorial	✓	✓	-
mult-by-1000	nl-eq-3	easy	✓	✓	✓
square-times-2	nl-eq-1	easy	✓	✓	✓
square-times-const	nl-eq-2	easy	✓	✓	✓
cube	cube-1	medium	✓	✓	✗
mult-by-add	nl-ineq-4	medium	✓	✓	✗
summation	gauss-1	medium	✗	✗	✓
summation2	gauss-2	medium	✚	✓	✚
binary-product	prod-bin	hard	✓	✓	✗
cube2	cube-2	hard	✚	✓	✗
int-square-root	sqrt	hard	✗	✓	✓
mult-of-6	-	hard	✗	✓	-

Figure 4: Benchmark programs. ✚ indicates that the level was not proved by an individual player, but was proved collectively.

7 Evaluation

In this section, we describe the experimental setup for our evaluation of IDG. The major criteria we use when comparing IDG with IDG-T are: (a) number of levels proved, (b) the ratio of the number of users who proved a level to the number of users who loaded it, (c) the distribution of submitted expression types, (d) cheesability, (e) player skill data, and (f) player feedback.

Experimental Setup

Programs Used Many of the benchmark programs we used were derived from programs used to evaluate InvGame (Bounov et al. 2018). We included all 4 of the programs that InvGame players were unable to prove guarantees of, as well as representatives from each class of program (such as CUBE, NL-EQ and NL-INEQ). There were, in addition, two new levels: MULTIPLY, the running example for this paper, and MULT-OF-6, which computes the sum of the squares of the first n natural numbers. For programs derived from InvGame, the name of the corresponding InvGame program name is displayed in the InvGame column in Figure 4. These programs were shown to be "unsolved by leading [software verification] tools" in Bounov et al. (2018).

Each benchmark program consists of variable initializations followed by a loop, in which variables are updated using arithmetic operations and conditionals. The InvGame programs were slightly modified to include print statements and to rename some variables.

Mechanical Turk We ran our experiment on Amazon Mechanical Turk and recruited 300 players. The task description contained the text "PROGRAMMING EXPERIENCE REQUIRED" in a large font.

Each task consisted of the tutorial video, the tutorial level, 1 randomly selected easy level and a random permutation of 2 medium and 2 hard levels. Upon completing the consent form, each player was randomly assigned to either IDG or IDG-T. The probabilities with which each game variant was assigned were tweaked over the course of the study in order to ensure sufficient attempts of each level for both games.

Players received a $2.00 base payment for completing the end-of-game survey, which they could access at any point.

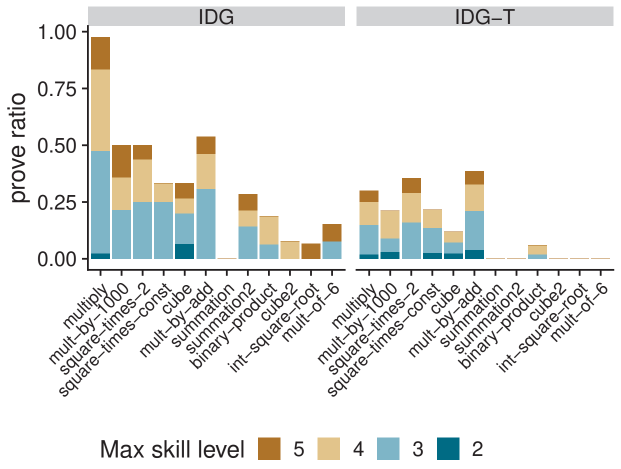


Figure 5: Ratio of the number of users who proved a level to the number of users who loaded it, broken down by the maximum of the player's self-reported programming and math skill levels.

Players also received a $0.80 bonus per completed level (including the tutorial). We included the bonus to incentivize players to attempt levels, while the base payment was intended to ensure that players who put in effort but were unable to complete any levels were compensated for their time. The survey included a question about programming experience with the options "None," "Novice," "Intermediate," "Advanced," "Professional" and a question about the highest level math course taken, with the options "Middle school or less," "High school," "Bachelors," "Masters" and "PhD." Skill levels are assigned numbers, with the lowest skills assigned the value 1. A player's max skill level is the maximum of their programming and math skill levels.

Experimental Comparison of IDG and IDG-T

The data presented here only includes those players who completed at least one level. The tutorial video walks players step-by-step through the solution of the tutorial level, which players then immediately play. To satisfy the condition to be included in the data analysis, players only had to repeat the steps shown in the tutorial video. However, of players who accepted the consent form, only 16.4% of IDG players and 51.5% of IDG-T completed a level. This may be due to participants trying to cheese Mechanical Turk, as 52.2% of IDG and 22.8% of IDG-T players accepted the consent form but did not even load a level.

Data Analysis Notes The "Chi-squared test" discussed below is R's built-in "Pearson's Chi-squared test". If Yates' continuity correction is applied, it is noted. The reported effect size for Pearson's Chi-squared tests is Cramer's V, as calculated by R's lsr package (Navarro 2015). A Holm correction was applied whenever multiple Chi-squared tests were performed on the same data. The "Wilcoxon rank sum test" discussed below is an "Exact Wilcoxon-Mann-Whitney Test" computed using R's coin package (Hothorn et al. 2006). The effect size for Wilcoxon rank sum tests discussed below is computed as $Z/\sqrt{n}$.

Terminology We use the following definitions to characterize progress in completing levels:

- ***loaded***: the player loaded the level's page
- ***attempted***: the player submitted at least one expression that is not disproved by the current trace
- ***completed***: the player received a gem for the level, *i.e.*, they proved the level or, for IDG-T, they submitted 6 accepted expressions
- ***proved***: the player's invariants are sufficient to prove the guarantee (coincides with completed for IDG)
- ***proved collectively***: the conjunction of the invariants of all players for that level is sufficient to prove the level, but the level is not proved by any individual player

Principal Experiments Figure 4 shows which levels were proved by players of InvGame, IDG-T and IDG. The only level that IDG players were unable to prove was SUMMATION, which was also not proved in IDG-T, but was proven in InvGame (by a single player). We believe that the main reason why this level was not proved is the small number of attempts: 7 IDG players attempted it. In addition to SUMMATION, IDG-T players were not able to prove INT-SQUARE-ROOT or MULT-OF-6. SUMMATION2 and CUBE2 were proved collectively.

Since IDG-T is our implementation of InvGame, we expected that the two games would have similar outcomes, but of the 10 levels that InvGame and IDG-T shared, they differed on 6 of them. IDG-T was more effective in the sense that its players were able to prove more levels and IDG-T players proved 4 levels that InvGame players did not, whereas InvGame players proved 2 levels that IDG-T players did not. The differences between the outcomes of the InvGame and IDG-T experiments validate our decision to implement our own version of InvGame for this study; differences in Mechanical Turk task descriptions and changes in the population of Mechanical Turk's users over time are two potential explanations for the difference in outcomes.

In Figure 5, we compare the *prove ratio*, the ratio of the number of players who proved the level to those that loaded the level, for each level. IDG levels have a significantly higher prove ratio than IDG-T levels, which is compelling evidence that showing code helps with loop invariant discovery. Notice that for 4 out of the 12 game levels, the prove ratio is 0 for all skill levels using IDG-T, but non-0 for IDG. For the remaining 8 game levels the average IDG ratio is more than twice as large as the average IDG-T ratio. The figure also shows that for IDG, players of skill levels 3, 4 and 5 solved 9, 9 and 8 game levels, respectively, whereas only 7 game levels were proved in total for IDG-T. Hence, more levels are solved with IDG even if we throw out all of the IDG players, except those with skill level 3, but keep all the IDG-T players of any skill level.

A Wilcoxon rank sum test comparing the level prove ratios of IDG players versus IDG-T players gives $p < 0.001$ and an effect size of 0.62, strong statistical evidence that the distribution of prove ratios for levels in IDG is different than that distribution for IDG-T.

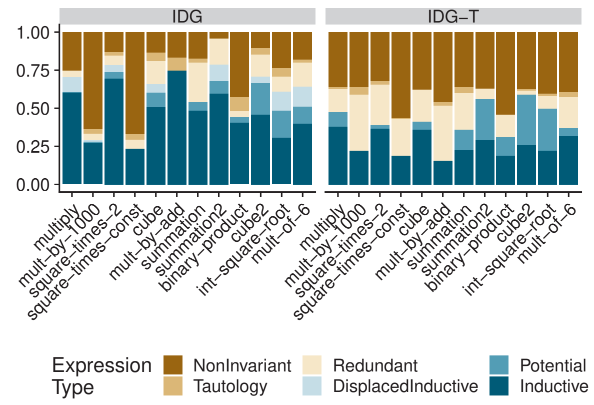


Figure 6: Distribution of the types of submitted expressions, by level. Types are ordered by usefulness, with the most useful type (inductive) at the bottom of the chart.

We analyzed the data presented in Figure 5 for only players with a max skill of 3 or above, players with a programming skill of "Intermediate" or above and players with math education of "Bachelors" or above. These graphs were not notably different and Wilcoxon rank sum tests as described above all gave $p < 0.001$. We also compared prove ratios within each programming skill level between IDG and IDG-T. We ran Wilcox signed rank tests on the prove ratios of each level between IDG and IDG-T for each programming skill level, with the alternative hypothesis being that IDG-T prove ratios are shifted to the left (closer to 0) than IDG prove ratios. The Holm-adjusted p-values are 0.56 for "Novice", 0.0098 for "Intermediate" and 0.031 for both "Advanced" and "Professional". This indicates that there is statistically significant evidence that IDG levels overall had better prove ratios than IDG-T levels for "Intermediate," "Advanced" and "Professional" programming skill players.

Distribution of Proposed Expressions Figure 6 shows that IDG players submitted proportionally more inductive invariants than IDG-T players, who submitted a greater proportion of non-invariants. Chi-squared tests of the number of expressions of each type versus game variant show a statistically significant correlation between the two variables ($p < 0.001$, effect size 0.29). Post-hoc chi-squared tests comparing each expression type versus the total number of all other expression types across games indicates that there is a significant difference across game variants and the distribution of each type of expression (the maximum p-value was 0.036 and effect sizes ranged from 0.21 to 0.03).

Cheesability A major goal of gamification is to design games which elicit desired behavior from players. In our context, we want players to discover expressions that are potentially useful for proving the guarantee. This includes inductive and potentially inductive invariants. Noninvariant, redundant, tautologous and displaced expressions are all useless because they cannot contribute to proofs and, as described in Section 5, there are several cheesing strategies based on useless expressions. The graph shown in Figure 7

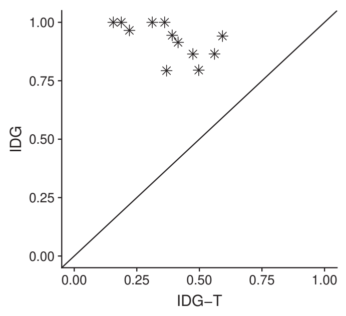


Figure 7: Proportion of submitted expressions which were useful, paired by level, for each game.

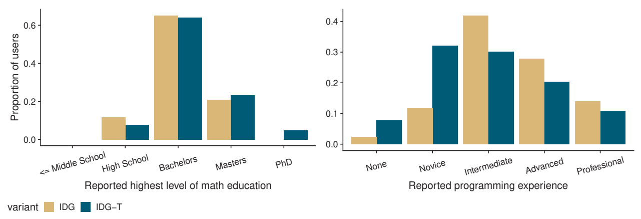


Figure 8: Self-reported skill levels for math and programming from the exit survey.

compares the percentage of accepted expressions that are useful between the two games. This demonstrates that IDG is more effective than IDG-T at eliciting useful expressions from players.

Player Skill & Completion Data The distribution of self-reported programming and math skill levels from an exit survey can be seen in Figure 8. Since players were randomly assigned to game variants, we expect that the distribution of self-reported programming and mathematics skills to be similar. Chi-squared tests showed no significant difference between IDG and IDG-T players in terms of self-reported mathematics skill level ($p = 0.834$), but found a significant correlation between game variant and self-reported programming skill level ($p = 0.013$, effect size 0.22). IDG players reported higher programming skill levels than IDG-T players, with proportionally fewer IDG players reporting novice or below programming skill level and proportionally more reporting skill levels of intermediate or above.

The average task time was 48.7 minutes for IDG and 42.4 minutes for IDG-T; this was found not significant by a Wilcoxon rank sum test ($p = 0.230$).

Player Feedback The message window and actionable feedback in IDG spurred interest for some players, who asked questions and expressed enthusiasm surrounding game mechanics in our exit survey. We even received a job request! Player responses suggest that, in addition to holding value as a verification tool, games like IDG may have educational merit, allowing novice programmers to casually learn about software verification.

8 Discussion, Limitations and Future Work

In this paper, we discussed the concepts underlying IDG and the experimental results of a comparison between IDG and IDG-T. In future work we will describe the reasoning engine and related algorithms, which involve many interesting design decisions. These algorithms maximize the utility of all submitted expressions as well as attempt to provide players with the most significant, insightful, goal-directed advice possible.

While our results seem compelling, it is important to identify and clarify potential limitations. The first limitation is that we used a small set of simple benchmark programs was used to evaluate the two games. Most of these programs were also used to evaluate InvGame because current software verification tools could not solve them, an indicator of the difficulty of loop invariant discovery. Nevertheless, it would be interesting to use more complex programs, possibly written in standard programming languages. Doing so would also enable the study of how well IDG scales to large codebases.

It is unclear how different the population of Mechanical Turk players who played our game is from other populations that might play the game, including volunteers for a citizen science project, students or contractors for a large company. All of these caveats are shared with InvGame's evaluation. Our expectation is that there are significant differences, as informally several undergraduate students were able to solve all the levels in under an hour and many Mechanical Turk players seemed to be most interested in finding ways to make money quickly with minimal effort.

The player completion data suggests that IDG deters lower-skilled players, as the IDG players who completed the exit survey had a higher self-reported programming skill level than those who completed the IDG-T exit survey. Some of the exit survey responses requested a more in-depth tutorial. Perhaps an interactive, in-depth, tutorial will make the game accessible to more players. It would be interesting to design such a tutorial and to evaluate whether it can effective train low-skill players to effectively play the game.

Organizations interested in using a game like IDG to verify their code may not want to make their code public. An interesting research question is whether one can design games that obfuscate the code being verified, but are as effective as IDG in terms of eliciting loop invariants from players.

IDG and IDG-T can be thought of as two extremes on a spectrum of program abstractions, neither of which may be ideal, and there are many hypothetical games in between. For example, consider a game that resembles IDG-T, but where input variables are exposed and new traces can be generated. It may be possible to design such a game in a way that rules out the cheesing strategies that IDG-T suffers from. This raises many questions: will the types of input variables be exposed, since the player can now enter new inputs? If so, what about the other variables' types? Is there a way to provide actionable information and clear goals when the loops and the guarantee are not player viewable? Discovering useful abstractions of programs that are effective in eliciting loop invariants from players and are amenable to gamification remains an interesting, unanswered question, to be explored in future work.

There are many potential applications of software verification games. One idea is to design games for specific programming languages that allow regular programmers, without expertise in formal verification, to effectively reason about programs. Another idea is to design educational games that target various student populations, starting with games that allows players familiar with programming to learn about verification and develop experience with loop invariants through gameplay. We had several undergraduates play the game and some of them were able to complete all of the levels. A more in-depth interactive tutorial would be especially important for educational applications.

We are interested in exploring the design and evaluation of AI agents who can play IDG and other verification games, using machine learning, logic programming, deep learning, etc. Notice that our design of IDG has the very useful property that the proposed invariants can be anything at all. That is, the reasoning engine is what guarantees correctness and nothing an AI agent proposes can lead to unsoundness. Therefore, AI agents that mostly generate useless suggestions, can be used to construct powerful software verification tools, as long as every once in a while, one of the AI agents generates useful invariants.

9 Conclusions

There has been recent interest in the design and evaluation of human computation games where players find loop invariants. In fact, recent games have allowed players to find loop invariants that automated tools cannot. None of these games show players code and the common wisdom is that showing code is a bad idea. We introduced IDG, an invariant discovery game which does show players code and our experimental evaluation shows that players of IDG discover higher quality invariants that prove more programs correct than players of previous games. IDG supports richer human-computer interactions, leading to effective human-in-the-loop systems that are able to augment the ability of non-experts to find loop invariants. We hope that these results will encourage future investigation into creating effective program verification games that can be used to improve the quality of software.

Acknowledgments

The work discussed in this paper was inspired by the work of Bounov, DeRossi, Menarini, Griswold and Lerner (Bounov et al. 2018). We thank them for sharing the code used for that project.

References

Bounov, D.; DeRossi, A.; Menarini, M.; Griswold, W. G.; and Lerner, S. 2018. Inferring loop invariants through gamification. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–13. ACM Press.

Bounov, D. 2018. *Toward Gamification and Crowdsourcing of Software Verification*. Ph.D. Dissertation, University of California San Diego, USA.

Bradley, A. R., and Manna, Z. 2007. *The calculus of computation: decision procedures with applications to verification*. Springer.

Chamarthi, H., and Manolios, P. 2019. ACL2s homepage. `http://acl2s.ccs.neu.edu/acl2s`.

Chamarthi, H. R.; Dillinger, P. C.; Manolios, P.; and Vroon, D. 2011. The ACL2 Sedan theorem proving system. *TACAS*.

Dean, D.; Gaurino, S.; Eusebi, L.; Keplinger, A.; Pavlik, T.; Watro, R.; Cammarata, A.; Murray, J.; McLaughlin, K.; Cheng, J.; and Maddern, T. 2015. Lessons learned in game development for crowdsourced software formal verification. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education*. USENIX Association.

Deterding, S.; Dixon, D.; Khaled, R.; and Nacke, L. 2011. From game design elements to gamefulness: defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, 9–15. ACM.

Dijkstra, E. W., and Scholten, C. S. 2011. *Predicate calculus and program semantics*. Springer.

Fava, D.; Shapiro, D.; Osborn, J.; Schäef, M.; and Whitehead, Jr., E. J. 2016. Crowdsourcing program preconditions via a classification game. In *Proceedings of the 38th International Conference on Software Engineering*, 1086–1096. ACM.

Gleiss, B.; Kovács, L.; and Robillard, S. 2018. Loop analysis by quantification over iterations. In *LPAR*, 381–399.

Hothorn, T.; Hornik, K.; van de Wiel, M. A.; and Zeileis, A. 2006. A Lego system for conditional inference. *The American Statistician* 60(3):257–263.

Kaufmann, M.; Manolios, P.; and Moore, J. S. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Logas, H.; Whitehead, J.; Mateas, M.; Vallejos, R.; Scott, L.; Murray, J. T.; Compton, K.; Osborn, J. C.; Salvatore, O.; Shapiro, D. G.; Lin, Z.; Sanchez, H.; Shavlovsky, M.; Lewis, C.; Cetina, D.; and Clementi, S. 2014. Xylem: The Code of Plants. In *Proceedings of the 9th International Conference on the Foundations of Digital Games*.

Logas, H.; Vallejos, R.; Osborn, J.; Compton, K.; and Whitehead, J. 2015. Visualizing loops and data structures in Xylem: The Code of Plants. In *2015 IEEE/ACM 4th International Workshop on Games and Software Engineering*, 50–56.

Navarro, D. 2015. *Learning statistics with R: A tutorial for psychology students and other beginners.* University of Adelaide, Adelaide, Australia. R package version 0.5.

RTI. 2002. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology.