Inferring Loop Invariants through Gamification

Dimitar Bounov Anthony DeRossi Massimiliano Menarini William G. Griswold Sorin Lerner
University of California, San Diego
{ dbounov, aderossi, mamenari, wgg, lerner }@cs.ucsd.edu

ABSTRACT
In today's modern world, bugs in software systems incur significant costs. One promising approach to improve software quality is *automated software verification*. In this approach, an automated tool tries to prove the software correct once and for all. Although significant progress has been made in this direction, there are still many cases where automated tools fail. We focus specifically on one aspect of software verification that has been notoriously hard to automate: inferring loop invariants that are strong enough to enable verification. In this paper, we propose a solution to this problem through gamification and crowdsourcing. In particular, we present a puzzle game where players find loop invariants without being aware of it, and without requiring any expertise on software verification. We show through an experiment with Mechanical Turk users that players enjoy the game, and are able to solve verification tasks that automated state-of-the-art tools cannot.

Author Keywords
Games; Program Verification; Loop Invariants.

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation (e.g. HCI).

INTRODUCTION
Software is increasingly intertwined in our daily lives: it manages sensitive information like medical records and banking information; it controls physical devices like cars, planes and power plants; and it is the gateway to the wealth of information on the web. Unfortunately, sometimes software can also be unreliable. Toyota's unintended acceleration bug was caused by software errors [24]. Furthermore, cars were found vulnerable to attacks that can take over key parts of the control software, allowing attackers to even disable the brakes remotely [25]. Pacemakers have also been found vulnerable to attacks that can cause deadly consequences for the patient [31]. Finally, faulty software has caused massive credit card and personal information leaks, for example the Equifax leak [18] that affected 143 million people.

One approach for improving software reliability and reducing bugs in software is called *software verification*. Software verification tries to *prove* properties about a program once and for all, accounting for any possible execution of the program.

CHI 2018, April 21–26, 2018, Montreal, QC, Canada

ISBN 978-1-4503-5620-6/18/04…$15.00

DOI: https://doi.org/10.1145/3173574.3173805

Typically, the programmer writes some form of *specification*, for example "foo returns a positive number", or "no buffer overflows". Automated and/or manual approaches are then used to show that the program satisfies this specification.

Over the last two decades, software verification has shown tremendous advances, such as techniques for: model checking realistic code [7, 8]; analyzing large code bases to prove properties about them [40, 43]; automating Hoare logic pre and post-condition reasoning using SMT solvers [11, 6]; fully verifying systems like compilers [27], database management systems [30], microkernels [23], and web browsers [22].

However, despite these advances, even state-of-the-art techniques are limited, and there are cases where fully automated verification fails, thereby requiring some form of human intervention for the verification to complete. For example, some verification tools require *annotations* to help the prover [6, 17]. In more extreme cases, the verification itself is a very manual process, for example when interactive theorem provers are used [23, 27].

In this paper, we focus on the particular problem of *inferring loop invariants*, an essential part of program verification that is hard to fully automate. Loop invariants are properties that hold at each iteration of a loop. They are needed to verify properties of loops, and as such are required in any realistic program verification effort. Despite decades of work, inferring good loop invariants is still a challenging and active area of research, and oftentimes invariants need to be provided manually as annotations.

In this paper, we will show how to leverage *gamification* and *crowdsourcing* to infer correct loop invariants that leading automated tools cannot. Our approach follows the high-level idea explored in prior citizen science projects, including Foldit [10] and, more closely related to our research, Verigames [13]: turn the program verification task into a puzzle game that players can solve online through gameplay. The game must be designed in such a way that the players can solve the tasks without having domain expertise.

In particular, we have designed a puzzle game called InvGame. Our game displays rows of numbers, and players must find patterns in these numbers. Without players knowing it, the rows in the game correspond to values of variables at each iteration of a loop in a program. Therefore, unbeknown to the player, finding patterns in these numbers corresponds to finding candidate loop invariants. As a result, by simply playing the game, players provide us with candidate loop invariants, without being aware of it, and without program verification expertise. InvGame then checks these candidates with a solver to determine which are indeed loop invariants.

This division of labor between humans and computers is beneficial because invariants are easier to for a tool to check than to generate. We let humans do the creative work of coming up with potential invariants, and we use automated tools to do the tedious work of proving that the invariant really holds.

While there has been prior work on inferring loop invariants through crowdsourcing and gamification [29], our work distinguishes itself in two ways (with more details in the Related Work section at the end of the paper). First, we have chosen a point in the design space that exposes numbers and math symbols directly to players in a minimalist game, thus leveraging human mathematical intuition. Second, we present a thorough empirical evaluation against state-of-the-art automated tools, and show how our gamification approach can solve problems that automated tools alone cannot.

We envision our gamification approach being used on the problems that automated tools cannot verify. To evaluate our game in this setting, we collected a large set of verification benchmarks from the literature and our own experiments, and ran four state-of-the-art preeminent verification tools on these benchmarks, without giving the tools any human-generated annotations. Of these benchmarks, all but 14 were solved by state-of-the-art tools. The remaining 14 benchmarks, which could *not* be verified by state-of-the-art tools, are precisely the benchmarks we tried to solve using our gamification approach. In particular, our game, played by Mechanical Turk users, was able to solve 10 of these 14.

In summary our contributions are:

- We present a game, INVGAME, in which players with no background in program verification can come up with candidate loop invariants
- We present a thorough empirical evaluation, showing that Mechanical Turk workers playing INVGAME can verify benchmarks that automated tools cannot.
- We discuss the design insights that made the INVGAME approach successful.

PROGRAM VERIFICATION AND LOOP INVARIANTS

As background, we present some standard material on program verification and loop invariants. Consider the code in Figure 1, which is a benchmark taken from the Software Verification Competition [1]. The code uses C syntax, and sums up the first `n` integers. The code is straightforward in its meaning, except for the `assume` and `assert` statements, which we describe shortly. In the context of the following discussion, we will assume that a tool called a *program verifier* will try to verify this benchmark.

Specifications. The first step in program verification is defining what we want to verify about the program. This is achieved through a *program specification*. There are many mechanisms for stating specifications, and here we use a standard and widely used approach: pre- and post-conditions expressed as assume and assert statements. This approach is taken by the benchmarks in the Software Verification Competition, and also by all preeminent general verification tools.

```
void foo(int n) {
  int sum,i;
  assume(1 <= n);
  sum = 0;
  i = 1;
  while (i <= n)
  // invariant: a condition that holds here
  // at each iteration
  {
    sum = sum + i;
    i = i + 1;
  }
  assert(2*sum == n*(n+1));
}
```

Figure 1. Code for our running example

The `assume` statement at the beginning of the code in Figure 1 states that the program verifier can assume that `1<=n` holds at the beginning of `foo`, without proving it. It will be the responsibility of the caller of `foo` to establish that the parameter passed to `foo` satisfies `1<=n`.

The `assert` statement at the end of the code states what we want the verifier to prove. In this case, we want the verifier to show that after the loop, the predicate `2*sum == n*(n+1)` holds.

Note that after the loop, the `sum` variable holds the sum of the first `n` integers. This benchmark is therefore asking the verifier to show Gauss's theorem about summing sequences of numbers, which states that the sum of the first n integers (n included) is equal to $n(n+1)/2$.

Loop Invariants. Reasoning about loops is one of the hardest part of program verification. The challenge is that the verifier must reason about an unbounded number of loop iterations in a bounded amount of time.

One prominent technique for verifying loops is to use *loop invariants*. A loop invariant is a predicate that holds at the beginning of each iteration of a loop. For example, `i >= 1` and `sum >= 0` are loop invariants of the loop in Figure 1. Not all loop invariants are useful for verification, and we'll soon see that the above two invariants are not useful in our example. However, before we can figure out which loop invariants are *useful*, we must first understand how to *establish* that a predicate is a loop invariant.

It is not possible to determine that a predicate is loop invariant by simply testing the program. Tests can tell us that a predicate holds on some runs of the program, but it cannot tell us that the predicate holds on all runs. To establish that a predicate is a loop invariant, we must establish that the predicate holds at the beginning of each loop iteration, *for all runs of the program*.

To show that a predicate is a loop invariant, the standard technique involves proving some simpler properties, which together imply that the predicate is loop invariant. For example, in our code, given a predicate $\mathcal{I}$, the following two conditions guarantee that $\mathcal{I}$ is a loop invariant:

1. **[I-ENTRY] $\mathcal{I}$ holds on entry to the loop:** If the code from our example starts executing with the assumption that `1 <= n`, then $\mathcal{I}$ will hold at the beginning of the loop. This can be encoded as the verification of the straight-line code shown on row 1 of Figure 2.

1: [I-ENTRY]	`assume(1 <= n);sum = 0;` `i = 1; assert(`$\mathcal{I}$`);`
2: [I-PRESERVE]	`assume(`$\mathcal{I}$ `&& i <= n);sum = sum + i;` `i = i + 1; assert(`$\mathcal{I}$`);`
3: [I-IMPLIES]	`assume(`$\mathcal{I}$ `&& i > n);` `assert(2*sum == n*(n+1));`

Figure 2. Straight-line code to verify for each condition

2. **[I-PRESERVE] $\mathcal{I}$ is preserved by the loop:** If an iteration of the loop starts executing in a state where $\mathcal{I}$ holds, then $\mathcal{I}$ will also hold at the end of the iteration. This can be encoded as the verification of the straight-line code shown on row 2 of Figure 2. Note that in this code, the `assume` statement also encodes the fact that, since an iteration of the loop is about to run, we know that the `while` condition in the original code, `i <= n`, must also hold.

Given a candidate loop invariant, there are known techniques for automatically checking the above two conditions. Since these conditions only require reasoning about straight-line code, they are much easier to establish than properties of programs with loops. Typically, one encodes the straight-line code as some formula that gets sent to a certain kind of theorem prover called an SMT solver [11]. As a result, given a candidate predicate, there are known techniques to automatically determine if the predicate is a loop invariant on all runs of the program.

Verification Invariants. The main reason for having loop invariants is to establish properties that hold *after* the loop. However, in general a loop invariant is not guaranteed to be useful for this purpose. For example, `true` is a loop invariant for any loop, but it does not help in proving asserts after the loop.

To be useful for verification, the loop invariant must satisfy an additional condition:

3. **[I-IMPLIES] $\mathcal{I}$ implies the post-loop assert:** If $\mathcal{I}$ holds at the end of the last iteration of the loop, then the assert after the loop will hold. This can be encoded as the verification of the straight-line code shown in row 3 of Figure 2. Note that in this code, the `assume` statement encodes the fact that, since we are done running the loop, we know that the `while` condition in the original code must be false, which gives us `i > n`.

We call a loop invariant that satisfies the above property a *verification invariant*. In other words: whereas a loop invariant holds at each iteration of a loop, a verification invariant additionally implies the post-loop assert.

Verification. Say we have found a verification invariant for a given loop. In this case, the post-loop assert immediately follows from I-ENTRY, I-PRESERVE, and I-IMPLIES. Thus, finding a verification invariant is the key to proving post-loop asserts, and as such they are used by many verification tools.

Some tools require the user to explicitly provide the verification invariant. However, this requires a significant amount of effort for the programmer, and typically also requires some kind of expertise in formal verification. As a result, tools and techniques have been developed to automatically infer invariants.

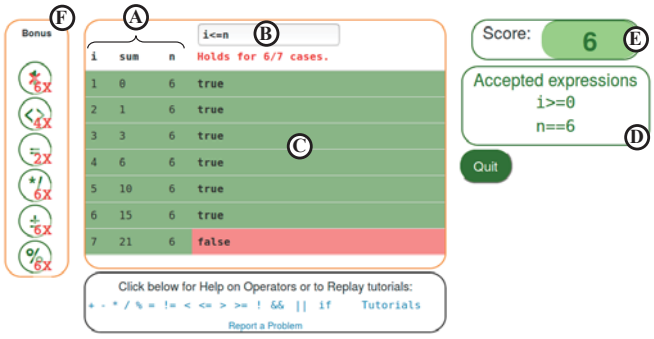

Figure 3. INVGAME play screen

As we will soon explain, automatically coming up with verification invariants is challenging. The tool must not only find a property that holds at each iteration of the loop, but is strong enough to show the post-loop assert.

Note that in our running example, the post-loop assert is *not* a loop invariant. Some readers might want to try to figure out the invariant by looking at the code before we give the invariant away.

Verification Invariant for our Example. The verification invariant for our example is:

```
2*sum == i*(i-1) && i <= n+1
```

The above invariant for our running example was not inferred by any of the five preeminent verification systems we tried. There are four main reasons why this invariant (and invariants in general) are difficult to infer automatically.

First, the expression `i-1` does not appear syntactically anywhere in the program (or in any assert or assume). In general, program verifiers must come up with invariants that are often quite removed from the actual statements, expressions, asserts and assumes in the program.

Second, the expression `2*sum == i*(i-1)` is not a common pattern. Some tools use patterns, for example $A = B * C$, to guess possible invariants. But the invariant in this case does not fit any of the common patterns encoded in preeminent verification tools. In general, program verifiers must come up with invariants that don't belong to pre-defined patterns.

Third, the verifier has to figure out the `i <= n+1` conjunct, which seems orthogonal to the original assert. In this case, the additional conjunct is somewhat straightforward to infer, because it comes from the loop termination condition and the single increment of `i` in the loop. However, in general, candidate loop invariants often need to be strengthened with additional conjuncts so that verifiers can establish that they are verification invariants.

Finally, the space of possible invariants is very large. In this example, if we consider just +, *, equality and inequality, there are over 10^{14} possible predicates whose number of symbols is equal to or smaller than the final verification invariant.

GAME DESIGN

We now describe our game, INVGAME, by showing how it allows us to verify the running example from Figure 1. Figure 3

shows a screenshot of INVGAME on a level that corresponds to our running example.

The area labeled (A) in the screenshot is a *data table*. This table shows rows of numbers, with columns labeled at the top. The player does not know this, but this table corresponds to the values of variables from a particular run of our example. In particular, the table displays the values of `i`, `sum` and `n` at the beginning of each loop iteration when the `foo` function from Figure 1 is called with `n = 6`.

The area labeled (B) in the screenshot is a *predicate input box*. The player enters boolean predicates in this box, with the goal of finding a predicate that evaluates to true for all rows.

The area labeled (C), which appears immediately below the input box, is the *result column*, which displays the result of evaluating the entered predicate on each row. The results in this column are displayed as "true" or "false", with "true" displayed in green, and "false" displayed in red. For example, in the screenshot of Figure 3, the player has entered `i<=n`. On the first six rows, this predicate evaluates to true, whereas it evaluates to false on the last row.

The result column updates automatically as soon as the player stops entering characters in the input box. More precisely, when the player hasn't entering a character in more than 500ms, the game tries to parse the text in the input box, and if it parses as a boolean expression, it evaluates the predicate on each row of the table, and displays the results in the results column.

In the screenshot of Figure 3, not all rows are green (true). Let's assume at this point the player enters the predicate `i<=n+1` (for example by simply adding "+1" at the end of the text box). The result column immediately updates to all green (true). At this point, the player is told that they can press "enter" to submit the predicate. If the player presses "enter" at this point (i.e., when all rows are green), four things happen.

First, the predicate in the textbox is added to area (D) on the screen, which contains a list of accepted expressions.

Second, the score in area (E) is updated. Each predicate is worth 1 point multiplied by a set of "power-up" multipliers. Area (F) displays the currently available multipliers. We discuss how the multipliers work in the next section.

Third, the multipliers in area (F) are updated to new power-ups that will be in effect for the next entered predicate.

Fourth (and finally), INVGAME records the entered predicate in a back-end server as a candidate loop invariant. Note that all rows being green does not necessarily mean that the entered predicate is an invariant, let alone a verification invariant. We will discuss later what INVGAME does with these candidates invariants, but at the very least it checks if some subset of the predicates entered by the player is a verification invariant for the level. INVGAME does this by checking, on the fly, conditions I-ENTRY, I-PRESERVE and I-IMPLIES.

If the player finds a verification invariant for the current level, the player has solved the level, and can go to the next level (which is generated from a different loop to verify). Also, if the player has not solved the level after entering a certain number of invariants, the player is advanced to the next level.

Note that INVGAME does not give the player any hint of the post-loop assert, or the code in the loop. As such, players are "blind" to the final goal of the invariant; they are just looking for patterns in what they see. While there are possibilities for adding such information to the game, our current design provides a more adversarial setting for getting results. Even in this more adversarial setting, we show that players can solve levels that leading automated tools cannot.

Scoring

One of the biggest challenges in developing a scoring strategy is that INVGAME does not know the final solution for a level. Thus, it is hard to devise a scoring mechanism that directly incentivizes getting closer a solution.

We address this problem by instead incentivizing *diversity*. We achieve this through "power-ups" that provide bonus points for entering certain kinds of predicates. The bonus points vary over time, and are bigger for symbols that the player has not entered in a while.

In particular, each accepted predicate earns one point, multiplied by the "power-ups" that apply to the predicate. The power-ups are shown in area (F). In order from top to bottom these power-ups apply in the following situations: the first applies if the predicate does not use any constants like 1, or 0; the second applies if the predicate uses inequality; the third applies if the predicate uses equality; the fourth applies if the predicate uses multiplication or division; the fifth applies if the predicate uses addition or subtraction; the sixth applies if the predicate uses the modulo operator.

Each power-up has an associated multiplier that varies over time. The multiplier is shown next to each power-up in area (F). For example, the top-most power-up in area (F) has multiplier 6X. At the beginning of a level, all multipliers are set to 2X. When a predicate is entered, the multipliers of all power-ups that apply are reset to 2X and the multipliers of all power-ups that do not apply are incremented by 2 (the sequence is: 2X, 4X, 6X, 8X, etc). As a result, the longer a power-up has not been used, the more valuable it becomes.

Power-ups compose, so that for example if two power-ups apply to a given predicate, with values 6X and 4X, then the score for that predicate is the base score of 1, multiplied by 6 and then by 4, giving 24.

Power-ups serve two purposes. First, they make the game more fun. In preliminary experiments with players, we often observed players show outward enjoyment from trying to hit multiple power-ups at once. The gratification of hitting a huge-scoring predicate was very apparent, something we also noticed when we as authors played the game. Second, power-ups incentivize players to enter a diversity of predicates. For example, if a player has not entered an inequality in a while, the inequality multiplier becomes large, incentivizing the player to enter an inequality.

Column Ordering

We found through preliminary trials that one important consideration is the order in which columns are displayed. In particular, certain column orderings made it more likely for players to discover certain invariants. For example, consider the screenshot in Figure 3, and recall the invariant in this case: `2*sum == i*(i-1) && i <= n+1`. The hardest part of this invariant is `2*sum == i*(i-1)`, which relates `sum` and `i`. This relationship is easier to see if the `sum` and `i` columns are displayed right next to each other, as in Figure 3. If the `sum` and `i` columns were further apart (for example by having the `n` column in between), then the pattern between `i` and `sum` would be harder to see, because the intervening columns would be cognitively distracting. The situation would be even worse if there were *two* intervening columns, containing unrelated numbers.

To address this problem, INVGAME serves the same level with different column orderings. In general, for n columns there are $n!$ possible orderings, which is quite large. To make this number more manageable, we take advantage of the fact that proximity between pairs of variables is the most important consideration. To this end, we define a notion of *adjacent-completeness*: we say that a set of orderings is *adjacent-complete* if all pairs of variables are adjacent in at least one of the orderings.

For 3 variables *a*,*b*,*c*, there are a total of 6 possible orderings, but only two orderings are needed for adjacent-completeness: *abc* and *acb*. For 4 variables *a*, *b*, *c*, *d*, there are a total of 24 possible orderings, but surprisingly only two orderings are needed to be adjacent-completeness: *abcd* and *cadb*. In general, adjacent-completeness provides a much smaller number of orderings to consider, while still giving us the benefit of proximity. INVGAME uses a pre-computed table of adjacent-complete orderings, and when a level is served, the least seen ordering for that level is served.

Generating data

To run an INVGAME level, we need to generate the data for table (A) in Figure 3. This is done automatically by guessing values for the input variables of the loop to be verified until we find input values that make the loop run for at least 7 iterations. The values of the variables from the first 7 iterations are instantiated as the 7 rows of data for the level. Note that the variable names that came with the benchmark are left intact, displayed at the top of the table, as seen in Figure 3. This means that in some cases some small hints of information might be communicated to the player. For example, a variable named `sum` might tell the player that there is some "summing" happening.

Back-end solver

A predicate that makes all rows true (green) is not necessarily an invariant, let alone a verification invariant. For example, in Figure 3, `n==6` would make all rows true, but it is not an invariant of the program. In other words, predicates entered by players are based on a particular run of the program, and might not be true in general.

Thus, we need a back-end solver that checks if the entered predicate is a verification invariant. This is straightforward for a given predicate, but there is a significant challenge in how to handle multiple predicates. For example, in Figure 3, the player might have entered `2*sum == i*(i-1)` and `i <= n+1` individually, but also other predicates in between, one of them being `n==6`, which is *not* a loop invariant. If we simply take the conjunction of all these predicates, it will not be a loop invariant (because of `n==6`), even though the player has in fact found the key invariants. Therefore, the INVGAME back-end needs to check if the conjunction of some *subset* of the entered predicates is an invariant. Furthermore, to enable collaborative solving, the back-end would need to consider the predicates from *all* players, which can lead to hundreds of predicates. Given 100 predicates there are 2^{100} possible subsets, which is for too many to test in practice.

To address this problem, we use an idea based on *predicate abstraction*, a widely investigated technique for doing program analysis [5]. In particular, given a set of predicates that are candidate loop invariants, the back-end first uses a theorem prover to prune out the ones that are not implied by the `assume` statements at the beginning of the function. For the remaining predicates, our back-end uses a kind of inductive reasoning. In particular, it first assumes that the remaining predicates all hold at the beginning of the loop. Then for each remaining predicate p the back-end asks a theorem prover whether p holds after the loop (under the assumption that all predicates hold at the beginning of the loop). The predicates p that pass the test are kept, and ones that don't are pruned, thus giving us a smaller set of remaining predicates. The process is repeated until the set of predicates does not change anymore, reaching what's called a fixed point. Interestingly enough, the theory on program analysis tells us that not only will this terminate, but the remaining set will be the maximal set of predicates whose conjunction is a loop invariant.

We run the solver in real-time, but with a short time-out, and leave the rest to an offline process. Predicates for an individual player are done quickly, in most cases at interactive speeds. However, checking the predicates of all players collectively (to check if a conjunction of predicates from different players solves a level) must often be left to an offline process.

Gamification Features

Our approach uses gamification, the idea of adding gaming elements to a task. InvGame has four gaming elements: (1) a scoring mechanism, (2) power-ups that guide players to higher scores, (3) a level structure, (4) and quick-paced rewards for accomplishments (finding invariants/finishing levels). While we didn't perform a randomized control study to measure the effects of these gamification elements, preliminary experiments with players during the design phase showed that scoring and power-ups significantly improved the enjoyment and engagement of players.

EVALUATION

We want to evaluate INVGAME along five dimensions, each leading to a research question: *(1) Comparison against Leading Tools*: Can INVGAME solve benchmarks that leading automated tools cannot? *(2) Player Skill*: What skills are needed to play INVGAME effectively? *(3) Solution Cost*: How much does it cost (time, money) to get a solution for a benchmark

using INVGAME? *(4) Player Creativity*: How creative are players at coming up with new semantically interesting predicates? *(5) Player Enjoyment*: How fun is INVGAME? Before exploring each of these questions in a separate subsection, we first start by describing our experimental setup.

Experimental Setup

We envision our gamification approach being used on problems that automated tools cannot verify on their own. To evaluate INVGAME in this setting, we collected a set of 243 verification benchmarks, made up of benchmarks from the literature and test cases for our system. Each benchmark is a function with preconditions, some loops and an assert after the loops. Functions range in size from 7 to 64 lines of code. Although these functions might seem small, it's important to realize that we have included all benchmarks from recent prominent papers on invariant generation [14, 20], and also benchmarks from recent Software Verification Competitions [1].

We removed from this set of benchmarks those that use language features that INVGAME doesn't handle, in particular doubly nested loops, arrays, the heap and disjunctive invariants (we discuss limitations of our approach later in the paper). This left us with 66 benchmarks. We ran four automated state-of-the-art verification tools on these benchmarks, namely: Daikon [15], CPAChecker [9], InvGen [21] and ICE-DT [20]. Daikon is a dynamic statistical invariant inference engine that generates invariants from a fixed set of templates. CPAChecker is a configurable static analysis tool based on abstract interpretation. InvGen is an automatic linear arithmetic invariant generator. Finally ICE-DT is an state-of-the art invariant generator based on counter-example guided machine learning.

We consider a benchmark as *solved by leading tools* if any of these four tools solved it. This left us with 14 benchmarks that were *unsolved by leading tools*. These 14 benchmarks, shown in Figure 4, are the ones we aimed to solve using INVGAME. In general, these benchmarks are challenging for automated tools because of the non-linear equalities and inequalities.

Here is the provenance of the 14 benchmarks that have resisted automated checking: `gauss-1` is from the Software Verification Competition suite [1]; `sqrt` is from the ICE-DT [20] benchmark suite; `gauss-2` is an elaborated version of `gauss-1`; `cube-1` is based on a clever technique for computing cubes [37]; `cube-2` and `prod-bin` are from an online repository of polynomial benchmarks [4]; the remaining 7 are from our test suite.

We use Mechanical Turk to run our experiment, as a set of human intelligence tasks (HITs). One HIT consists of playing at least two levels (benchmarks) of the game, but we let the players play more if they want. We paid participants because Mechanical Turk is a paying platform. Per our IRB protocol, we aimed to pay our players at a rate of about $10/hr. Participants were paid even if they didn't solve a level, and we didn't connect pay to score. Because we did not want users to purposely slow down to get paid more, we paid players by the level rather than by the hour. Preliminary trials showed that $0.75/level would lead a pay rate of $10/hr. However, players in experiments were quicker than in trials (avg of 109s/level), yielding an average pay rate of $24.77/hr.

Name	Invariant	Game
gauss-1	`2*sum==i*(i-1) && i<=n+1`	✓
gauss-2	`2*s==i*j && j==i-1 && i<=n+1`	✓
sqrt	`su==(a+1)*(a+1) && t==2*a+1`	✓
nl-eq-1	`i == 2*k*j`	✓
nl-eq-2	`i == k*j*l`	✓
nl-eq-3	`i == 10+k*j`	✓
nl-eq-4	`i == l+k*j`	✓
nl-ineq-1	`i*j <= k`	✓
nl-ineq-2	`i*j <= k`	✓
nl-ineq-3	`i*i <= k`	✓
nl-ineq-4	`i <= j*k`	✗
prod-bin	`z+x*y == a*b`	✗
cube-1	`i*(i+1)==2*a && c==i*i*i`	✗
cube-2	`z==6*(n+1) && y==3*n*(n+1)+1 && x==n*n*n`	✗

Figure 4. Benchmarks not solved by other tools

INVGAME dynamically serves the level (benchmark) that the current player has played the least. Although a production system would run a level until it was solved, our goal was to better understand how players interact with INVGAME. To this end, we ran experiments until we had at least 15 unique players for each level. In total, our data set includes 685 plays and 67 unique players, with each level played between 36 and 72 times. Some levels were also solved many times.

Comparison against Leading Tools

Figure 4 shows the benchmark name, the verification invariant, and whether our gamification approach was able to find the invariant using Mechanical Turk users. In short, our gamification approach worked on 10 out of the 14 benchmarks.

Individual vs. Collective Solving. One interesting question is whether solutions came from single plays or collections of plays. We define a *play* as a player playing a level once. We define an *individual solving play* as a play of a level that *by itself* finds the *entire* required invariant. In all cases but two, the solutions in our experiments came from individual solving plays. However, there can be cases where there is no individual solving play (because each player only finds part of the invariant), but putting the invariants together from all plays solves the benchmark. We call such solutions *collective solutions*; these solutions can only happen when the verification invariant has a conjunct, so in 5 of our 14 benchmarks. In our experiments, collective solutions happened *only twice*, for `sqrt` and `gauss-2` (for `sqrt`, an individual solution was also found). For *collective* solutions, because of the way our solver works, it is difficult to determined who contributed to solving the level. As a result, in all of the following discussions where we need to attribute the solution to a player, we only consider individual solving plays (and call them solving plays).

Player Skill

To better understand the relationship between a player's prior math/programming skills and their ability to solve levels, we asked players to rate themselves on a scale from 1 to

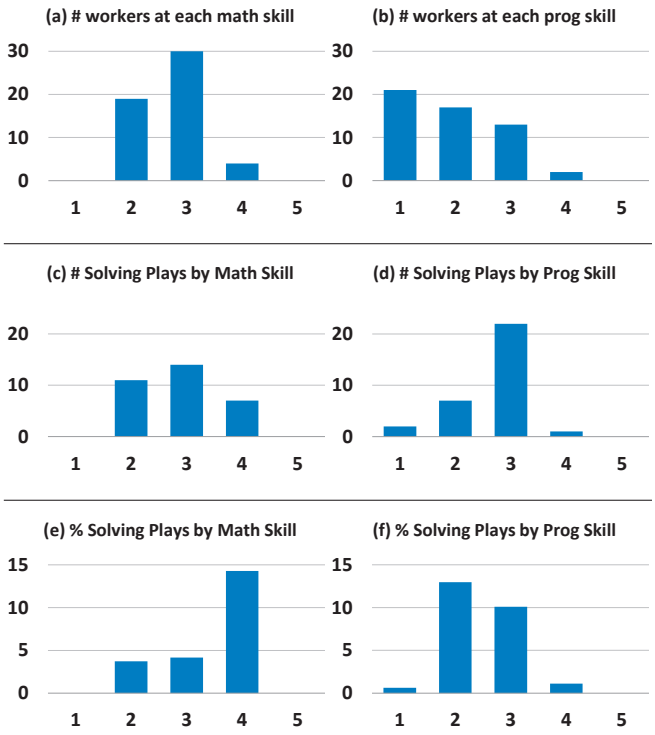

Figure 5. Number of players by math and programming skill

5 on math expertise and programming expertise. We use the term "math skill" and "programming skill" to refer to the math/programming skill rating that the player gave themselves. The skill ratings for math and programming were defined as follows in the survey:

Math 1	Middle school math	Prog 1	No experience
Math 2	High school math	Prog 2	Beginner
Math 3	College level math	Prog 3	Intermediate
Math 4	Masters level math	Prog 4	Advanced
Math 5	PhD level math	Prog 5	Professional

Bearing in mind that these are self-reported metrics, Figures 5(a) and (b) show the number of players at each math and programming skill, respectively. The majority of our players have not taken a math course beyond high-school or college and have either no programming experience, or are at best novice to intermediate. It's also worth noting that we have no players that only took math at the middle-school level and that we see a wider spread of skill levels in programming experience than math. Finally, we had no expert programmers (skill 5) in our study. Since verification is an advanced and specialized topic, we assume this also means that none of our players had verification expertise.

We define the math/programming skill of a *play* as the math/programming skill of the player who played the play. Figures 5(c) and (d) show the number of solving plays (recall, these are individual solving plays) categorized by the skill of the solving play — math in (c) and programming in (d). This shows us that in absolute numbers, plays at programming skill 3 contribute the most.

To better understand the performance of each skill rating, Figures 5(e) and (f) show the number of solving plays at a given skill divided by the total number of plays at that skill — math in (e) and programming in (f). For example, the bar at math skill 3 tells us that about 4% of plays at math skill 3 are solving plays. This gives us a sense of how productive plays are at math skill 3.

Looking at Figures 5(e) and (f), if we ignore programming skill 4 — which is difficult to judge because it has so few plays, as seen in Figure 5(b) — there is in general a trend, both in math and programming skill, that higher skill ratings lead to higher productivity.

In addition to looking individually at math and programming skill, we also looked at their combination, in particular: (1) the average of math and programming skill (2) the maximum of math and programming skill. Although we don't show all the charts here, one crucial observation emerged from examining the maximum of math and programming skill. To be more precise, we define the *mp-max* skill to be the maximum of the math skill and the programming skill. We noticed that *all* solving plays had an *mp-max* skill of 3 or above, in other words a skill of 3 or above in math *or* in programming. Furthermore, this was not because we lacked plays at lower *mp-max* skills: 35% of plays have an *mp-max* skill strictly less than 3, in other words a skill below 3 in *both* math and programming. These 35% of plays did not find any solutions. In short, InvGame does not enable players with low math *and* low programming skill to find loop invariants: instead the most effective players are those with at least a score of 3 in math or programming.

Difficulty of Benchmarks. Having focused on the effectiveness of players over *all* benchmarks, we now want to better understand what makes a benchmark difficult and this difficulty relates to what skill is required to solve it.

One clear indication of the difficulty is that some benchmarks were not solved. Looking more closely at the four benchmarks that were not solved by InvGame, we found that for `cube-1`, players found `c==i*i*i` many times, but not `i*(i+1)=2*a`; and for `cube-2` players found `6*(n+1)` and `x==n*n*n` but not `y==3*n*(n+1)+1`. From this we conclude that more complex polynomial relationships are harder for humans to see.

Focusing on levels that *were* solved, let's first consider `gauss-1`, which was only solved once in a preliminary trial with online Mechanical Turk workers. The player who solved `gauss-1` had a math skill of 2 and programming skill of 3, which is average for our population. This occurrence resembles cases observed in Foldit [10], where some of the biggest contributions were made by a small number of players with no formal training in the matter, but with a strong game intuition.

For the remaining benchmarks that were solved, a proxy measure for difficulty of a level might be the number of solving plays for that level. The more frequently a level is solved, the easier the level might be. Because there is variation in the number of times each level was played, we normalize the number of solving plays for a level by the total number of plays for that level. This gives us, for each level, the percentage of plays that were solving plays. Figure 6(a) and (b) show this

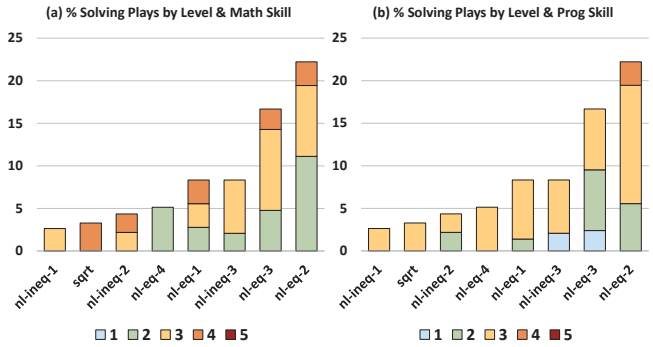

Figure 6. Percentage of solving plays (broken down by experience)

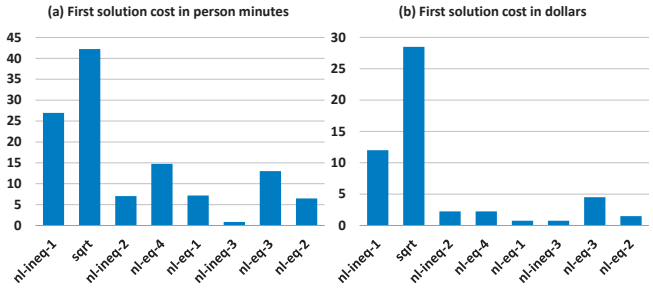

Figure 7. Cost of first solution in total player time and money

percentage for all solved levels — except for `gauss-2`, because it was solved by multiple plays, and `gauss-1` because as we mentioned earlier we don't include the data for that benchmark here. For example, we can see that for `nl-eq-2` about 22% of plays were solving plays, and for `nl-ineq-1` about 3% of plays were solving plays. The levels are ordered from hardest at the left (least number of solutions) to easiest at the right (most number of solutions).

Furthermore, we divided each bar into stacked bars based on math and programming skill of the solving plays. Figure 6(a) shows the chart divided by math skill, and Figure 6(b) shows the chart divided by programming skill.

Plotting the data in this way gives us a sense of how much each skill contributed to the solving plays for each level. We can notice, for example, that programming skill 3 appears on all benchmarks, meaning that if we only keep programming skill 3 plays, we still solve the same number of benchmarks.

We also expect that the hardest levels might require more skill to solve. Thus, moving from right to left in the bar chart, we would expect the bars to contain larger proportions of high skills. While not perfectly observed in Figure 6, we do see signs of this pattern. For example, scanning from right to left in the chart organized by math skill – 6(a) – we see that math skill 2 (green) disappears: the hardest three benchmarks are only solved by math skill 3 and 4. Scanning from right to left in the chart organized by programming skill – 6(b) – we see that programming skills 1 and 2 shrink and then disappear.

Solution Cost

In practice, if we were trying to solve verification problems with InvGame, we would have InvGame serve a given level until it was verified. Because all of our data has time-stamps, we can simulate this scenario from our data. In particular, given a level, we can measure the cost to get the first solution for that level, both in minutes of game play for that level, and in terms of money spent on that level. Figure 7(a) shows the first solution cost in minutes of gameplay for each level, and Figure 7(b) shows the first solution cost in dollars for each level. The benchmarks are ordered in the same way as in Figure 6. As expected, we notice a broad trend which is that benchmarks that were measured as harder according to Figure 6 tend to cost more to solve.

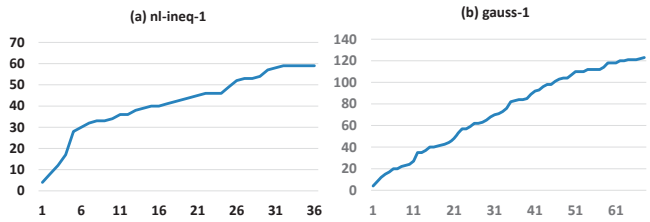

Figure 8. Cumulative # of semantically new predicates by # of plays

On average the first solution required 15 minutes of player time and cost $6.5. Also, although we don't show a chart with the number of unique players to the first solution, on average, the first solution required about 8 plays and 4 unique players.

Player Creativity

A key assumption of our game is that players are good at coming up with new predicates, in the hope that they will eventually hit upon a verification invariant. To get more detail on player creativity, we plot, over time, the cumulative number of predicates entered by players that were *semantically new*, meaning that they were not implied by any of the previously entered predicates. There are 14 such plots, two of which are shown in Figure 8(a) and (b).

To gain insight on whether these trends had flattened by the end of our experiments – signaling an exhaustion of what players are able to invent for accepted predicates – we computed the fits of two trend lines, one on the assumption that new predicates were continuing to come (a linear regression), and one that they were not (a logarithmic regression). Our intuition is that, given enough runs, we would see a leveling off in the discovery of semantically new predicates. This would mean that the logarithmic regression would be a clearly better fit. However, we are *not yet* seeing this for our benchmarks: the linear regression is a better fit (i.e., had a better R^2 value) in 12 of the 14 benchmarks. Furthermore, the R^2 values of *all* regressions, linear or logarithmic, were uniformly high, in the range 0.86 to 0.98 — with $R^2 = 1$ indicating a perfect fit. The most logarithmic plot is shown in Figure 8(a). This tells us that, despite having between 36 and 72 plays for each benchmark, we have not yet seen enough runs to detect a clear leveling off of semantically new predicates. This suggests that we have not yet exhausted the creativity of our players in terms of coming up with semantically new predicates. Besides being indicative of the creative potential of our players, it provides hope that further gameplay could solve more of our four remaining unsolved benchmarks.

Player Enjoyment

We asked players to rate the game from 1 to 5 on how much fun they had. The average rating for fun was 4.36 out of 5.

Also, we received several emails from users asking us when more HITs would be available. One user even referred to us as their favorite HIT. Note that our experiments do not show that participants are willing to play the game just for fun, without being paid. Still, even with players being paid, the rest of our data analysis is valid with regards to (1) how humans play INVGAME, (2) the solving power of humans vs. automated tools (3) the relation of player skill to solving potential.

The balance of intrinsic vs. extrinsic motivation in game design is an interesting topic of study [34]. Certainly, extrinsic motivation is not inconsistent with the idea of gamification, in that some citizen science projects use both [10, 39]. Recent research also suggests that hybrid schemes of motivation (mixing intrinsic and extrinsic) are worth studying in terms of player engagement, retainment, and effectiveness [34, 39, 44].

In addition to the "fun" rating, we also asked players to rate the game from 1 to 5 on how challenging they found it. The average rating was 4.06, suggesting users found this task fairly difficult. The fact that players found the game challenging and yet fun might suggest that INVGAME is a "Sudoku" style puzzle game, where players enjoy being engrossed in a mathematically challenging environment.

DESIGN INSIGHTS

We believe these were several important design insights that led to our approach being successful.

Less abstraction has benefits. In INVGAME we chose a point in the design space that exposes the math in a minimalist style, without providing a gamification story around it. As we will discuss more in related work, this is in contrast to other games like Xylem [29], which provide a much more polished and abstracted veneer on the game. Our minimalist user interface scales better with content size: our unencumbered table can convey a lot of information at once. Furthermore, we believe INVGAME was successful at solving problems in large part because of the lack of abstraction: we are directly leveraging the raw mathematical ability of players.

The diversity of human cognition is powerful. Players in INVGAME don't see code or asserts. Yet they are still able to come up with useful invariants. In fact, if players played perfectly, always finding the strongest invariants given the traces they saw, they would often miss the invariants we needed. In this sense, *the diversity of human cognition is extremely powerful*. Crowdsourcing, along with our scoring incentive of power-ups, together harness this diversity of cognition.

While looking at predicates entered by players, two cases stood out as exemplifying the diversity of human cognition.

First, a player found `su == (a+1)*(a+1) && su==t+a*a` for `sqrt`, and this solved the level. At first we were surprised, because this looks very different than the solution we expected from Figure 4, namely `su==(a+1)*(a+1) && t==2*a+1`. However, upon closer examination, the player's predicate is semantically equivalent to ours, even though syntactically different. The fact that a player found such a different expression of the same invariant, which required us to think twice about how it could actually solve the level, is a testament to the diversity in cognitive approaches and pattern recognition that different players bring to the table.

Second, in `nl-ineq-1`, many players found a compelling candidate invariant that we did not expect: `(i+1)*j==k`. This candidate invariant is stronger than (i.e., implies) the needed verification invariant, namely `i*j<=k`. The candidate invariant `(i+1)*j==k`, despite its compelling nature (upon first inspection we mistook it for another correct solution), is actually *not* a valid invariant, even though it held on the data shown in the game. What is interesting is that if all humans had played the game perfectly, finding the best possible invariant for the data shown, they would have only found the incorrect stronger `(i+1)*j==k`, which would have prevented them from entering the correct one, `i*j<=k`. In this sense, the diversity of human ability to see different patterns is crucial, as often the strongest conclusion given the limited data shown is not the one that generalizes well.

Law of Proximity. We found that column proximity affected the ability of players to see certain patterns: it was easier to discover patterns in closer columns. This insight is captured by the *Law of Proximity*, one of the Gestalt Laws of Grouping, from the Gestalt Psychology [41, 42]. The law of proximity states that objects that are close together appear to form groups. This law has a long history of being used in HCI design, including for web page design, menu design, and form design [41, 42]. In our setting, columns are perceived as grouped based on their proximity, which then affects the kinds of relations that players see.

Reducing the Two Gulfs by Changing the Task. One way of assessing the cognitive burden of a task is to consider the two well-known "gulfs": (1) *gulf of execution* [32]: the difficulty in understanding what actions should be taken to achieve a given goal; (2) *gulf of evaluation* [32]: the difficulty in understanding the state of a system, and how the system state evolves. When a human does program verification the traditional way, using loop invariants and a back-end theorem prover, both gulfs are large: it's hard to determine what invariant to use by looking at the program (gulf of execution); and it's also hard to predict how the back-end theorem prover will respond to a given invariant (gulf of evaluation). By transforming the task into one where humans look at run-time data, instead of programs, and try to come up with predicates that hold on this data, we significantly reduce *both* gulfs: it's easy to understand how to enter predicates that make all rows green to win points (gulf of execution); and it's easy to understand that the system just evaluates the predicates entered to either true (green) or false (red) on the given data (gulf of evaluation).

This reduction in cognitive burden, through a reduction in the two gulfs, explains in theoretical terms why players with no verification expertise can play INVGAME. However, these choices have also robbed the player of *fidelity* with respect to the original verification task: the player does not see the program, or the asserts, or any feedback from the theorem prover. This seems to indicate an *increase* in the gulf between our game and the *original verification* task. How is the game still successful at verification? This fidelity is in fact not always needed; and when not needed, it can be cognitively burden-

some, *especially* for non-experts. The ultimate insight is that a low fidelity proxy which reduces cognitive burden and also maintains enough connections to the original task can allow non-experts to achieve expert tasks. Also, from the perspective of game design, our choice of proxy provides another benefit: INVGAME's frequent positive feedback for entering novel predicates is likely more rewarding and motivating than the frequent feedback on incorrect verification invariants.

Some of the above design insights, individually, can certainly be connected to prior work (e.g.: some Verigames projects change the task [29], Foldit removes a lot of abstraction [10]). However, our work also provides empirical evidence that the above design insights can be incorporated together into a system that makes players effective at loop invariant inference compared to state-of-the-art automated tools.

LIMITATIONS AND FUTURE WORK

Arrays and the Heap. Our current version of INVGAME handles only arithmetic invariants. In the future we plan to investigate ways of incorporating arrays and the heap, for example simple numerical arrays, multi-dimensional arrays, linked-lists, and object-oriented structures. Doing so entails two challenges: (1) how to display these structures (2) how to enable players to express predicates over these structures.

Implication Invariants. Some benchmarks require verification invariants involving implications of the form `A => B`. One promising approach for adding such invariants to INVGAME involves splitter predicates [36]: if we can guess `A`, then the level along with its data can be split into two sub-levels, one for `A` and one for $\neg$`A`. Although guessing a correct `A` is challenging, we found in 38 out of 39 cases that `A` appears in the program text (usually in a branch), suggesting that there are promising syntactic heuristics for finding `A`.

RELATED WORK

Inspired by the success of projects such as Foldit [10] several lines of work have tried to apply gamification to the problem of Software Verification. One of the main differences between our work and these verification games is the extent of the empirical evaluation: we evaluate our system on a well-known set of benchmarks, showing that our system can outperform state-of-the-art automated tools. We can split existing work on gamification for Software Verification into games that expose math as part of the game play and those that conceal it.

Games that expose math. The closest approach to ours is Xylem [29, 28], a game where players are botanists trying to identify patterns in the numbers of growing plants. Like INVGAME, Xylem relies on players finding candidate invariants from run-time data. However, due to its focus on casual gaming, Xylem explores a different point in the design space than ours. Xylem is a touch-based game that uses a graphics-rich UI with many on-screen abstractions, including plants, flowers, and petals. Since these abstractions use space, Xylem is limited in the amount of data it can show onscreen, which in turn might hinder the ability of players to see patterns. For example, Xylem players can only see 2 data rows at a time, having to scroll to see other rows. In contrast, INVGAME tries to increase the ability of humans to spot patterns by: (1) showing the data matrix all at once (2) re-ordering columns. Xylem also restricts some aspects of predicate building, such as not allowing arbitrary numeric constants. In contrast, INVGAME has an unrestricted way for players to enter predicates, and provides immediate feedback on which rows are true/false, for quick refinement. Finally, although Xylem was played by many users, there is no systematic evaluation on a known benchmark set, or comparison to state-of-the-art tools.

Monster Proof [12] is another verification game that exposes math symbols to the player. However, Monster Proof is much closer in concept to a *proof assistant* [38, 2, 3], where the user is constructing a detailed proof using rules to manipulate expressions. In contrast, INVGAME uses humans to recognize candidate invariants, and automated solvers to perform much of the manipulations done by players in Monster Proof. Also, the published results on Monster Proof do not provide a systematic evaluation against state-of-the-art tools.

Games that conceal math. In Binary Fission [16], players build preconditions by composing primitive predicates generated by Daikon. The user is never exposed to the predicates and instead builds a decision tree out of black-box filters. Our benchmarks involve predicates that Daikon cannot infer, which makes them difficult to solve by Binary Fission. Binary Fission was also evaluated only on loop-free programs, whereas the goal of our work is to verify loops, which are one of the biggest challenges in program verification.

In Circuit Bot [12], Dynamaker [12] and Pipe-Jam/Flow-Jam/Paradox [13, 12] players resolves conflicts in constraints graphs derived from type/information flows. In Ghost Space [12] and Hyper Space [12] players manipulate an abstracted control-flow graph to perform counterexample-guided abstraction refinement under the hood. In StormBound [12] players build program predicates without seeing mathematical symbols or numbers. These games rely more on the player's visual and spatial recognition skills, rather than mathematical pattern recognition skills.

Machine Learning Invariant Generation. Another closely related line of work uses machine learning for invariant generation. Most approaches learn a boolean formula over a fixed set of predicate templates [19, 20, 26, 35]. Recent work by Padhi et. al. [33] extends this work by attempting to also learn the predicate features through counterexample guided synthesis. We believe crowdsourced gamification and machine learning based approaches are complimentary lines of research, as data gathered from the best players in a crowdsourced setting would be a useful training set for machine learning approaches.

CONCLUSION

We presented INVGAME, a game in which players guess loop invariants. We have shown that INVGAME allows players with no verification knowledge to verify benchmarks that leading automated tools cannot. We believe our minimalist design was one of the key factors contributing to its success: INVGAME directly leverages raw human mathematical skills. Our promising results lay the path for a larger exploration into program verification through gamification and crowdsourcing.

Acknowledgments. This work was funded by National Science Foundation grant #1423517.

REFERENCES

1. 2016. Competition on Software Verification (SV-COMP) benchmarks. https://sv-comp.sosy-lab.org/2016/. (2016).
2. 2017. The HOL Interactive Theorem Prover. https://hol-theorem-prover.org/. (2017).
3. 2017. Isabelle. https://www.cl.cam.ac.uk/research/hvg/Isabelle/. (2017).
4. 2017. Some programs that need polynomial invariants in order to be verified. http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html. (2017).
5. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. 203–213. DOI: http://dx.doi.org/10.1145/378795.378846
6. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 364–387. DOI:http://dx.doi.org/10.1007/11804192_17
7. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007a. The software model checker Blast. *STTT* 9, 5-6 (2007), 505–525. DOI: http://dx.doi.org/10.1007/s10009-007-0044-z
8. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007b. The software model checker Blast. *STTT* 9, 5-6 (2007), 505–525. DOI: http://dx.doi.org/10.1007/s10009-007-0044-z
9. Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 184–190. DOI: http://dx.doi.org/10.1007/978-3-642-22110-1_16
10. Seth Cooper, Adrien Treuille, Janos Barbero, Andrew Leaver-Fay, Kathleen Tuite, Firas Khatib, Alex Cho Snyder, Michael Beenen, David Salesin, David Baker, and Zoran Popovic. 2010. The challenge of designing scientific discovery games. In *International Conference on the Foundations of Digital Games, FDG '10, Pacific Grove, CA, USA, June 19-21, 2010*. 40–47. DOI: http://dx.doi.org/10.1145/1822348.1822354
11. Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. DOI: http://dx.doi.org/10.1007/978-3-540-78800-3_24
12. Drew Dean, Sean Gaurino, Leonard Eusebi, Andrew Keplinger, Tim Pavlik, Ronald Watro, Aaron Cammarata, John Murray, Kelly McLaughlin, John Cheng, and others. 2015. Lessons learned in game development for crowdsourced software formal verification. *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)* (2015).
13. Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popovic. 2012. Verification games: making verification fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*. 42–49. DOI: http://dx.doi.org/10.1145/2318202.2318210
14. Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 443–456. DOI: http://dx.doi.org/10.1145/2509136.2509511
15. Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.* 449–458. DOI:http://dx.doi.org/10.1145/337180.337240
16. Daniel Fava, Dan Shapiro, Joseph C. Osborn, Martin Schäf, and E. James Whitehead Jr. 2016. Crowdsourcing program preconditions via a classification game. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1086–1096. DOI: http://dx.doi.org/10.1145/2884781.2884865
17. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 234–245. DOI: http://dx.doi.org/10.1145/512529.512558
18. Forbes. 2017. How Hackers Broke Equifax: Exploiting A Patchable Vulnerability. https://www.forbes.com/sites/thomasbrewster/2017/09/14/equifax-hack-the-result-of-patched-vulnerability/#7b45bbf55cda. (2017).
19. Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 69–87. DOI: http://dx.doi.org/10.1007/978-3-319-08867-9_5

20. Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 499–512. DOI:http://dx.doi.org/10.1145/2837614.2837664

21. Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 634–640. DOI: http://dx.doi.org/10.1007/978-3-642-02658-4_48

22. Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2012. Establishing Browser Security Guarantees through Formal Shim Verification. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 113–128. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang

23. Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. DOI: http://dx.doi.org/10.1145/1743546.1743574

24. Phil Koopman. 2014. A Case Study of Toyota Unintended Acceleration and Software Safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf. (2014).

25. Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 447–462. DOI:http://dx.doi.org/10.1109/SP.2010.34

26. Siddharth Krishna, Christian Puhrsch, and Thomas Wies. 2015. Learning Invariants using Decision Trees. *CoRR* abs/1501.04725 (2015). http://arxiv.org/abs/1501.04725

27. Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 42–54. DOI: http://dx.doi.org/10.1145/1111037.1111042

28. Heather Logas, Richard Vallejos, Joseph C. Osborn, Kate Compton, and Jim Whitehead. 2015. Visualizing Loops and Data Structures in Xylem: The Code of Plants. In *4th IEEE/ACM International Workshop on Games and Software Engineering, GAS 2015, Florence, Italy, May 18, 2015*. 50–56. DOI: http://dx.doi.org/10.1109/GAS.2015.16

29. Heather Logas, Jim Whitehead, Michael Mateas, Richard Vallejos, Lauren Scott, Daniel G. Shapiro, John Murray, Kate Compton, Joseph C. Osborn, Orlando Salvatore, Zhongpeng Lin, Huascar Sanchez, Michael Shavlovsky, Daniel Cetina, Shayne Clementi, and Chris Lewis. 2014. Software verification games: Designing Xylem, The Code of Plants. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, FDG 2014, Liberty of the Seas, Caribbean, April 3-7, 2014.* http://www.fdg2014.org/papers/fdg2014_paper_17.pdf

30. J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 237–248. DOI: http://dx.doi.org/10.1145/1706299.1706329

31. Eduard Marin, Dave Singelée, Flavio D. Garcia, Tom Chothia, Rik Willems, and Bart Preneel. 2016. On the (in)security of the latest generation implantable cardiac defibrillators and how to secure them. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. 226–236. http://dl.acm.org/citation.cfm?id=2991094

32. Don Norman. 1988. *The Design of Everyday Things*.

33. Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 42–56. DOI:http://dx.doi.org/10.1145/2908080.2908099

34. Katie Seaborn and Deborah I. Fels. 2015. Gamification in theory and action: A survey. *International Journal of Human-Computer Studies* 74, Supplement C (2015), 14 – 31. DOI:http://dx.doi.org/https://doi.org/10.1016/j.ijhcs.2014.09.006

35. Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 88–105. DOI: http://dx.doi.org/10.1007/978-3-319-08867-9_6

36. Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 703–719. DOI: http://dx.doi.org/10.1007/978-3-642-22110-1_57

37. Rajat Tandon and Rajika Tandon. 2014. Article: Algorithm to Compute Cubes of 1st "N" Natural Numbers using Single Multiplication per Iteration. *International Journal of Computer Applications* 101, 15 (September 2014), 6–9.

38. The Coq Development Team. 2017. The Coq Proof Assistant Reference Manual. https://coq.inria.fr/refman/. (2017).

39. Ramine Tinati, Markus Luczak-Roesch, Elena Simperl, and Wendy Hall. 2017. An investigation of player motivations in Eyewire, a gamified citizen science project. *Computers in Human Behavior* 73, Supplement C (2017), 527 – 540. DOI:http://dx.doi.org/https://doi.org/10.1016/j.chb.2016.12.074

40. Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 205–214. DOI:http://dx.doi.org/10.1145/1287624.1287654

41. Johan Wagemans, James Elder, Michael Kubovy, Stephen Palmer, Mary Peterson, Manish Singh, and Rüdiger von der Heydt. 2012a. A Century of Gestalt Psychology in Visual Perception: I. Perceptual Grouping and Figure-Ground Organization. *Psychological Bulletin* 138, 06 (2012), 1172–1217.

42. Johan Wagemans, Jacob Feldman, Sergei Gepshtein, Ruth Kimchi, James R. Pomerantz, and Pater A. van der Helm. 2012b. A Century of Gestalt Psychology in Visual Perception: II. Conceptual and Theoretical Foundations. *Psychological Bulletin* 138, 07 (2012), 1218–1252.

43. Yichen Xie and Alexander Aiken. 2005. Saturn: A SAT-Based Tool for Bug Detection. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. 139–143. DOI:http://dx.doi.org/10.1007/11513988_13

44. G. Zichermann. 2011. Intrinsic and Extrinsic Motivation in Gamification. http://www.gamification.co/2011/10/27/intrinsic-and-extrinsic-motivation-in-gamification/. (2011).