Lessons Learned in Game Development for Crowdsourced Software Formal Verification

Drew Dean, *SRI International* Sean Guarino, *Charles River Analytics*
Leonard Eusebi, *Charles River Analytics* Andrew Keplinger, *Left Brain Games*
Tim Pavlik, *University of Washington* Ronald Watro, *Raytheon BBN*
Aaron Cammarata, *VoidALPHA* John Murrary, *SRI International*
Kelly McLaughlin, *XPD Analytics* John Cheng, *Veracient LLC*
Thomas Maddern, *Veracient LLC*

Introduction

The history of formal methods and computer security research is long and intertwined. Program logics that were in theory capable of proving security properties of software were developed by the early 1970s [1]. The development of the first security models [2-4] gave rise to a desire to prove that the models did, in fact, enforce the properties that they claimed to, and that an actual implementation of the model was correct with respect to its specification [5; 6]. Optimism reached its peak in the early to mid-1980s [7-11], and the peak of formal methods for security was reached shortly before the publication of the Orange Book [12], where the certification of a system at class A1 required formal methods. Formal verification of software was considered the gold standard evidence that the software enforced a particular set of properties. Soon afterwards, the costs of formal methods, in both time and money, became all too apparent. Mainstream computer security research shifted focus to analysis of cryptographic protocols (e.g. [13; 14]), policies around cryptographic key management [15], and clever fixes for security problems found in contemporary systems [16-19].

Our appetite for formal verification historically has been insufficient to limit our appetite to build ever larger operating systems. In the 1980s, it was possible to verify a few hundred to a few thousand lines of code. By comparison, the 1986 release of the 4.3BSD Unix operating system had a kernel of approximately 50,000 lines of code. From the 1980s to present, there have been numerous advances in formal verification technology, for example, the introduction of software model checkers, (mostly) practical satisfiability solvers, and SMT solvers. The seL4 project [20] remains a highlight of modern operating system verification, with a microkernel of approximately 9,000 lines, took 11 person-years, plus an additional 9 person-years of tool development. For comparison, due primarily to the large number of devices supported, the 2013 Linux 3.10 kernel has 15.8 million lines of code[1].

While the seL4 project is justifiably celebrated as a success, it also unfortunately reinforces the message that formal verification has scaling challenges. Based on the seL4 data, if one optimistically assumed linear scaling of effort vs. lines of code in formal verification, verifying Linux 2.6.24 with 8.9 million lines of code[2] from January 2008 would take 11,000 person-years, or nearly 3 years if all of the world's estimated 4,000 formal methods experts [21] productively working together on a single project. With the average salary of a software engineer being approximately $93,000 in 2013[3], we derive a direct cost of $1 billion for the verification effort. In those intervening 3 years, Linux had advanced to version 2.6.36, with an additional 4.5 million lines of code. It is easy to see that this process will never converge, even with unrealistically optimistic assumptions!

The time and cost of formal verification appeared to be an intractable problem outside of very specialized domains, where cost and long development times could be tolerated for improved safety and security. If one examines the situation a little closer, the key to the problem is that the size of the available talent pool is limited by today's formal verification tools, complete with user interfaces that can be described charitably as obscure. It is often said that an advanced degree in Computer Science is necessary to use formal verification tools. If, however, this talent pool could be expanded, the key bottleneck to effective formal verification could be removed. We note that automation, while proven very helpful by the seL4 effort, cannot provide a full solution

[1] http://www.h-online.com/open/features/What-s-new-in-Linux-3-10-1902270.html
[2] http://royal.pingdom.com/2012/04/16/linux-kernel-development-numbers/
[3] http://money.usnews.com/careers/best-jobs/salary

due to Rice's Theorem [22], which established that most common questions about software are algorithmically undecidable. Given that we cannot fully automate the verification problem, it is natural to attempt to add aspects of human intuition to the solution.

Towards the goal of human-assisted verification, two remarkable circumstances converged: (1) the then director of DARPA, Dr. Regina Dugan, expressed interest in applying crowdsourcing to computer security; and (2) a set of enlightening discussions with Michael Ernst and Jeannette Wing, starting at the November 2010 Usable Verification workshop hosted by Microsoft Research, led to the idea of applying gamification to the formal verification domain. If formal verification problems could be turned into entertaining video games, those games could be crowd-sourced to a large audience. At first, this seemed like an impossible challenge: how do you define a puzzle that encodes a formal verification problem in a way such that a solution to the puzzle can be mapped usefully back to the underlying verification problem, while simultaneously be entertaining to solve? The remainder of this paper describes five remarkable solutions to this challenge developed under the aegis of DARPA's Crowd-Sourced Formal Verification (CSFV) program, identifying numerous lessons that can be carried improve the success of future citizen science and gamification efforts.

Solution 1: Circuitbot and Dynamakr

Andrew Keplinger, *Left Brain Games* Mathew Barry, *Kestrel Technology*
J. Nelson Rushton, *Texas Tech University* Greg Izzo, *Left Brain Games*
Qianji Zheng, *Texas Tech University*

1. Introduction

The Circuitbot and Dynamakr games provide a crowd-sourced contribution to the verification of C-language programs. In particular, the player-provided solutions of these puzzle games contribute so called "points-to graphs", which represent information about which memory locations may hold the addresses of other memory locations as the program runs. Nodes in the graph correspond to memory locations, and an arc from node *x* to node *y* represents that *x may hold the address of y at some point during program execution*. This is classically known as the "pointer analysis problem" and has many variations. The variation we treat takes account of offsets in memory, but abstracts away control flow from the program. Even this simplified version of the problem is undecidable, and its solution or sound partial solution contributes substantially to program verification. These two factors make this version of the pointer-analysis problem a good candidate for the application of human intelligence through game play.

There are three steps to our program verification approach. In the first step, the CodeHawk static analyzer creates a set of constraints on the points-to graph of the given source program. These constraints are partitioned into sets corresponding roughly to functions in the source code, which are then transformed into game levels. In the second step, the game players solve these levels by making moves in a judicious order. Each move in the game consists of adding arcs to the graph that result in satisfying a single constraint. Eventually, as the players complete the levels and satisfy all of the constraints, the gameplay yields a fixpoint solution -- but the time required to reach this solution, and whether the process halts, depends on triggering constraints in a wise order, as well as performing operations that lose information but speed up the solution process or allow it to halt. In the third step, CodeHawk uses the information derived from the points-to arcs to detect buffer overflow and underflow errors, or (more hopefully) verify their absence.

1.1. General Game Play

The challenge is to create an engaging game from the constraints on the points-to graph of a software program. Our player-engineers are actually receiving information about a section of the program to be verified, in the form of game levels. The information takes the form of constraints defining when connections ("arcs") must be added between elements to satisfy the constraint, at least temporarily. Once all rules are satisfied simultaneously, the level is solved (corresponding to a local fixpoint).

From the player's perspective, the tricky part is this: arcs added to satisfy a constraint may cause another constraint to become unsatisfied. Indeed, a brute force auto solver could spend an infinite amount of time attempting to complete all the connections. In practice, the size and connectedness of the graph grow as the game progresses, resulting in ever-more complex interactions between constraints. Eventually as a fixpoint is neared, some sections of information become idle. Our autosolver uses a divide-and-conquer approach, but the current strategy did not become apparent until after a great deal of experimentation.

1.2. Game Play Evolution

Although our core game concept has remained unchanged throughout the CSFV program, our approach to crowd contribution has changed substantially. Our present game-play approach is to present essential elements of the graph to the player in very large chunks, then prompt him to steer the autosolver in exploring the graph.

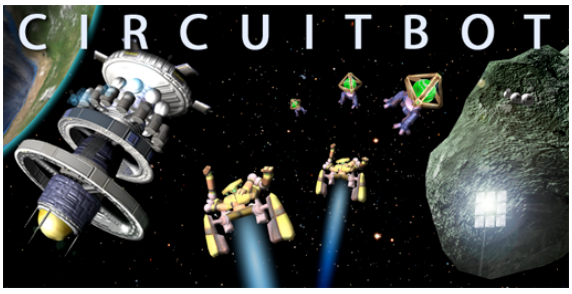
Since we focus on the creation of a points-to graph, our key heuristic for player productivity is the number of

arcs added to the graph. The source of the name "Circuitbot" was a game concept where constraints were represented directly and individually on the screen, and robotic spiders traveled from one to another in a specific order carrying information, like an assembly line changing with each rule application. A potential problem present in this early version was Circuitbot going into a trivial infinite loop due to incompatible rules. We developed art for this concept, and created some cartoonish Acme-Labs style gates that would destroy the Circuitbots.

As the game evolved we found no good strategies for constraint ordering that worked significantly faster than brute force, and we found that constraints needed to be represented in a different way. We also found that, as the concept matured, we were uncertain about the number of total constraints and how often they would be applied. So we had to change our game concept into something that would work regardless of the number of constraints. In the end we discovered through experimentation that some rules can produce thousands or tens of thousands of arcs in a single pass, and we had to adapt to this.

Since the game model hit a technical bottleneck, while work was being done on the backend server we had to base our game on speculation and some sample data. There were many unknowns from a game-making perspective, which made it difficult to predict how much fun -- or how much work -- the resulting game would provide the player. We considered it likely that some of the work could be automated, so we needed a game concept that would maintain user engagement and also could adapt to some automation.

2. Circuitbot



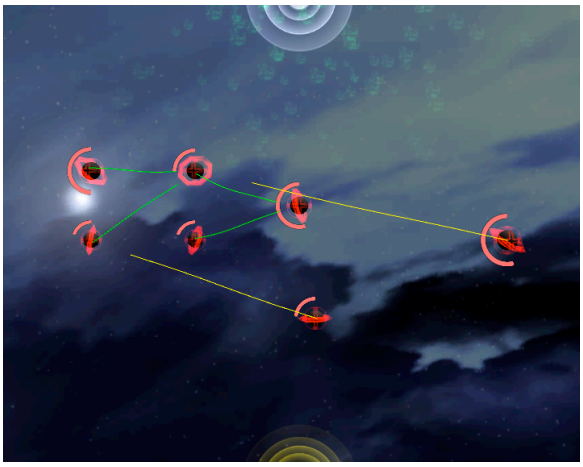
The Circuitbot game employs a turn-based strategy in which the motivational system drives the player back to the "work" part we want accomplished for verification. The universe of Circuitbot is the near future exploration and exploitation of near Earth asteroids, along with the development of a space program. We took many liberties with physics in favor of directing the player toward rapidly expanding his supplies of critical resources. The landing sequence, in which robots arrive on the surface of some far-flung location, requires the player to develop connections (arcs) in order to program them so they can complete the automated process of building a support facility. This is the "work" that we are asking the player to accomplish.

After launching the game to the public and supplying data from actual to-be-verified software, we began analysis of player-generated results back into the verification backend. After much analysis and some reworking on the software analysis side, we realized that we were looking at the information too narrowly. We would receive a game level that represents too small a portion of the software; and focusing the player on individual constraints inside each level was not accounting for a sufficiently wide view of the target program. This led to the development of Dynamakr, which better leverages the respective capabilities of the human player and the autosolver.

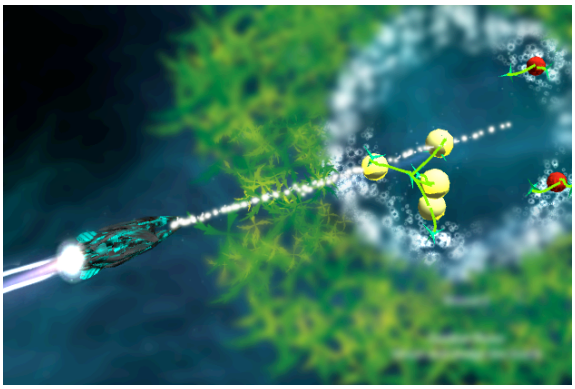
3. Dynamakr

Though the mathematical game model for Dynamakr is the same as for Circuitbot, game play is very different. Dynamakr presents sets of game levels and the player manages them on the global level. This allows automation to solve each individual game level and present the player with the goal of finding the right sets of levels to solve in order. The player's objective is to reach a fixpoint quickly while minimizing information loss. We also discovered that we could display this solution process, showing the individual arcs, and this would make interesting knots of interconnected arcs. We then developed an arcade-style game around this concept as a reward game that challenge players to find connecting game levels.

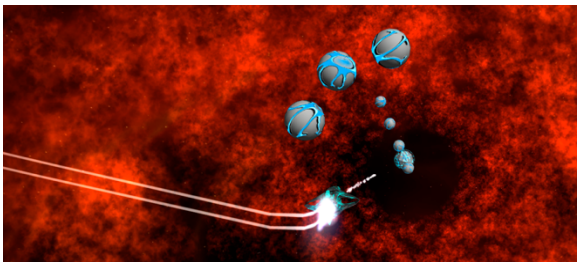
The reward game became Dyna-makr. Conceptually, Dynamakr is a quantum level 3D printer. Inside the Dynamakr the player examines patterns and feeds them into the Dynamo. The player first takes on the chal-



lenge of finding patterns that will produce the most energy in the Dynamo, as sometimes patterns will amplify each other's energy. Once they generate enough energy from the patterns, the player feeds the patterns into the Dynamo and launches the arcade game. The player's success in the first game determines his points and power-ups available in the second game. To help the player search for higher-valued patterns we provide him with a set of tools. Each pattern yields some energy by itself, but when joined with the energy from other patterns its energy can multiply by many times. The game rules govern the search space and the energy value. The player cannot feed a pattern into the Dynamo until it has joined its energy with the energy design. Moreover, we provide the player with search tools in the solution space to discover related patterns based on various relationships. These patterns have a value based on their composition and the past game activity. If a pattern produced energy recently it is likely to produce energy again so we encourage the player to find related patterns and then join these results with the energy design. The tools the player deploys correspond to parameters used in heuristics by the auto-solver. We had to learn to set these parameters effectively, based on what we saw in the solution process, to find fixpoint solutions quickly. The players perform the same task within the abstraction of the game.



The Dynamakr arcade game shows the same information that is displayed as robots in the Circuitbot game, but in Dynamakr there are many times more instances involved and they fly past the player in an infinite-runner style game. The player has to dodge and shoot the bad elements, which are constraints that have not yet been triggered, and has to collect the energy generated by triggering the active constraints. This feature is meant to reward the player for generating maximal energy during the first phase of the game. The energy elements arrive at the player in waves, with each wave associated with one of the patterns he fed into the Dynamo.



The design effort for Dynamakr required the game developers to understand the underlying logic of the verification method and game rules. In essence, the game development team had to become familiar with pointer analysis, especially as represented through the abstraction of the game. We experimented with various manual and auto-solving strategies, processing candidate constraints sets to better understand how the player would best provide assistance for verification. A combination of auto-solving and manual play turns out to be most useful, where we auto-solve much of the game set prior to releasing it to the crowd who complete the iteration. This final step of the procedure is where the human game players in the crowd add the most value.

Solution 2: Flow Jam and Paradox

Tim Pavlik, *University of Washington*
Craig Conner, *University of Washington*
Jonathan Burke, *University of Washington*
Mathew Burns, *University of Washington*
Werner Dietl, *University of Waterloo*
Seth Cooper, *Northeastern University*
Michael Ernst, *University of Washington*
Zoran Popović, *University of Washington*

1. Introduction

Paradox is a game designed for *crowd-sourced formal verification* [23], in which the actions of ordinary people assist in the production of a proof of correctness for a computer program. Paradox is a puzzle game with levels that resemble branching tree-like structures. Each level of Paradox corresponds to Java code that has been converted into a constraint graph via a type analysis system. A level solved with all constraints satisfied the game corresponds to a proof that some code satisfies a security property. The player may not be able to fully solve a level; however, a partial solution will reduce the amount of work necessary for a skilled programmer to complete the proof.

This section discusses the design of Paradox and the application of lessons learned from a previous version of the game called Flow Jam. Paradox and Flow Jam were developed at the University of Washington Department of Computer Science and Engineering, as a collaboration between the Programming Languages & Software Engineering Group and the Center for Game Science.

2. Verification Approach

Our verification approach is based on type theory. To verify a security property, the types in a program must satisfy certain type constraints. As a simple example, if the program contains the assignment statement "x = y", then the type of x must be a supertype of the type of y. Therefore a proof of correctness can be thought of as a set of constraints involving the statements of the program.

A Paradox game level can also be thought of as a set of constraints that a player is trying to solve. Like many puzzle games, in order to complete a Paradox game level, the player must find consistent settings for all the game elements.

Because both Paradox and type-checking are based on constraints, it is possible to create a Paradox level that corresponds to a given piece of code. Specifically, our type analysis system takes as input a Java program and a security property, and it generates as output a set of type constraints that the Paradox game presents to players as a puzzle to solve. When a player adjusts a game element, this corresponds to selecting a different type for a variable. Because the actual type system constraints are displayed as simple game mechanics, players can help perform verification tasks without needing any prior knowledge of software verification.

If the player is able to solve a given level, the player has also generated a proof that the input piece of code is free from vulnerabilities for the given security property. If the level cannot be fully solved, the constraint graph must contain certain inconsistencies that correspond to type-checking errors for the program -- potential security vulnerabilities that can be examined by a verification expert.

3. Paradox Game Play

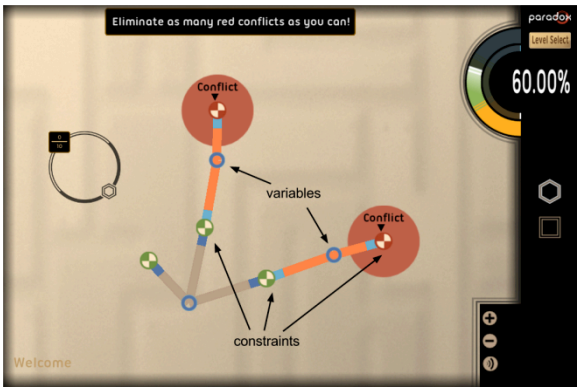


Figure 2-1: Paradox variables, constraints, and conflicts.

A Paradox level's elements represent variables and constraints from the underlying constraint problem (see Figure 2-1). A variable node is either light blue or dark blue, representing type qualifiers or their absence in the code being verified. A constraint node requires that at least one of the connected variables has a certain value. If none of the variables for a given constraint are the correct value, then the constraint is marked as a conflict. Edges are the connections between a variable and a constraint when a constraint contains a given variable.

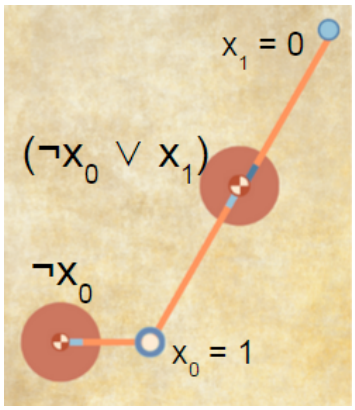


Figure 2-2: A Paradox level representing the formula: ¬x0 ∧ (¬x0∨ x1). The red circles represent conflicts are shown for the unsatisfied constraints involving variables x0 and x1.

The player's goal is to find a setting for the variables that minimizes the number of conflicts. Currently, we represent the variables as boolean values and the constraints as disjunctions over variables or their negations, making the problem the players are solving a maximum satisfiability problem (MAX-SAT) (Figure 2-2). Exposing MAX-SAT problems to human players is similar to the approach taken by the game FunSAT [24; 25].

4. Maximizing Human Contribution

In order to maximize the contribution that untrained human players can make to the verification process, players should focus on the portion of problem that is least solvable by automated methods. Up to a certain size, constraint graphs can be solved rapidly by automated solvers and are not challenging for human players. Very large constraint graphs, however -- corresponding to real-world programs such as Hadoop -- can be difficult to understand and present multiple problems for user interface design. A previous version of this game, Flow Jam, required players to toggle variables (in that game called "widgets") individually, which did not scale well to larger levels where humans were most needed.

To address this, Paradox provides a "paintbrush" mechanism that allows the player to select arbitrary groups of variables. The player can change them all at once, or the computer can automatically solve them (for groups up to a predetermined limit). Different paintbrushes can allow the player to apply different automated algorithms to their selection. Thus, the main feature of Paradox gameplay is the player guiding the automated methods: deciding which areas of the graph to solve and in what order. Currently players have access to four paintbrushes that have the following effects on the selected variables: set to true, set to false, launch an exact DPLL optimization [26; 27]or launch a heuristic GSAT optimization [27]. These optimizations are the two phases of the maximum (MAX-SAT) solving algorithm suggested by Borchers and Furman [28]. New optimization algorithms can be added to the game as additional paintbrushes.

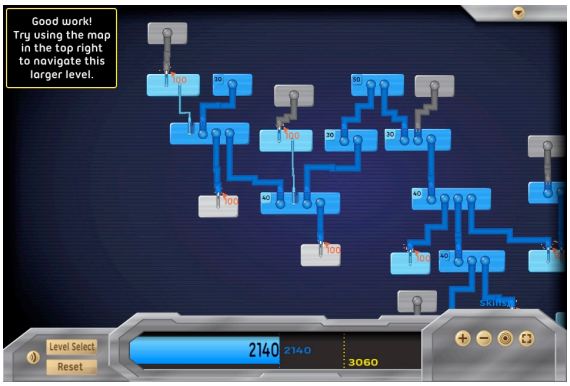


Figure 2-3: A previous version of the game, Flow Jam, required players to adjust variables individually.

Additionally, in Paradox, human players are never given small optimization problems (for example, toggling the values of 50 variables to get the optimal score) since automated methods can solve that scale of problem. Instead, they are consistently provided with large and challenging problems that are computationally intractable to solve in an automated manner.

5. Maintaining Player Interest

In a normal game, levels are created by a game designer with the aim of creating a fun and engaging experience for players. In a formal verification game, however, the levels that are most valuable for players to solve are those generated from the code that is being verified. Since the code in question was most likely created for a very different purpose than making an interesting game level, sometimes levels contain oddities such as enormous sections that are not integral to the solution. Worse, some levels are very large but consist only of repeating structures, resulting in puzzles that are not interesting or challenging for human players.

To study player preferences, a comparable batch of levels was synthesized -- that is, generated randomly and not based on real-world Java code. Using Flow Jam (the previous version of Paradox), real versus synthesized levels were compared by surveying players to see which type of levels were found enjoyable. Synthesized levels designed to maximize complexity were clearly preferred, with an average 65% preference rating, over real levels, which averaged a 30% preference rating.

Although not a rigorous comparison, this indicates that there is room for improving levels generated from real code. We do not yet know whether this preference for synthesized levels in Flow Jam carries over to levels in Paradox.

To ensure that levels generated from real-world code are interesting enough to entice non-expert human players to solve them, our system adjusts the constraint graphs before they are served to players. For example, irrelevant parts are removed, and a level is broken down into independent levels when possible. If a level can be automatically solved, then it is never given to human players. Subparts of a level may be solved before the player ever sees it. We plan to perform a study comparing levels directly from Java code to levels optimized for human engagement.

6. Solution Submission and Sharing

Game players on the Internet are not obligated to persist in playing until a level is solved. We found that many players of Flow Jam would make some amount of progress, but very few of them would follow through and submit or share their results. Before changing our submission process, there were only about 3,300 submissions compared to about 100,000 levels played (note that players could make multiple submissions on an individual level if desired). Players would often quit midway through without returning to their current state, or fail to notice the level submission/sharing functionality even though they were making progress on the levels.

To address this, Paradox automatically submits level configurations to a central server whenever the player's score increases. This takes the burden off of players to manually submit their solutions for evaluation. By adding these submissions back into the system as new level starting points, it also allows future players of a given level to begin with the progress that prior players have made, without requiring them to proactively share solutions with each other.

7. Sense of Purpose

Another aspect of working with a human population of solvers is motivation. Playtesting has shown that, if players do not understand what they are doing and why they are doing it, they quickly lose interest in the task. In early versions of Paradox, players were given the optimizer brush and tasked with painting around conflicts to solve them, leaving them with no sense of what they were actually doing to solve the levels. To fix this, the tutorial now includes a few levels where players must change variables manually. Playtest feedback indicates a much better understanding of the underlying problem and a general sense of purpose when players are required to adjust individual variables in tutorials before using optimizer brushes.

8. Results

Since the public launch of the combined verigames.com portal in December 2013, over 6,000 unique players have played Flow Jam for a combined total of over 7,500 hours of play and over 34,000 level submissions.

In addition, we completed an experiment on Hadoop to test how much expert analysis time is saved using inference and Verigames. Two developers annotated a program, one starting from unannotated source code and one starting from game results (inference). Each continued manually until the program type-checked.

There were a total of 23 annotations required. Of the two conditions, unannotated code required 45 minutes total time (7 minutes of type checking and 38 minutes of manual effort) versus 4 minutes total when starting with game results (3 minutes of type-checking and 1 minute of manual effort).

Not included in these timing were the annotation of APIs (determining the proof goal, required in both cases), and gameplay (crowd time, machine time to generate levels). Note that the game computed correct annotations in this case (the human merely verified them).

9. Conclusions

Due to its crowd-sourcing approach, the CSFV program is as much about game design, human-computer interaction, and human behavior as it is about formal verification of software. The lessons that have guided development from the earlier game Flow Jam to the current game Paradox naturally point towards future areas of study. These topics include player performance versus fully automated methods, player effectiveness with different graph representations and groupings, and differences between volunteer players and compensated players. Also, given its general nature, problems from other domains that can be encoded as maximum satisfiability problems (MAX-SAT) could be used to create levels in Paradox. The game design may also extend to other types of constraint satisfaction problems that can be visualized as a factor graph.

Solution 3: Ghost Map and Hyperspace

Ronald Watro, *Raytheon BBN* Kerry Moffit, *Raytheon BBN*
John Ostwalk, *Raytheon BBN* Eric Church, *BreakAway Games*
Dan Wyschogrod, *Raytheon BBN* Andrei Lapets, *Raytheon BBN*
Linsey Kennard, *Raytheon BBN*

1. Introduction and Approach

The Ghost Map project is led by Raytheon BBN Technologies with support from Breakaway Games, the University of Central Florida, and Carnegie Mellon University. Ghost Map uses model checking as its software verification technique. The fundamental concept of model checking is that properties of a complex system can sometimes be most effectively deduced by creating and reasoning about a simplified model of the system rather than the system itself. For software, the control flow graph (CFG) of a program is a simplified model of the program's actual executions. Many of the software correctness properties from the SANS/MITRE Common Weakness Enumeration (CWE) list [29] can be associated with a set of control flow patterns. The Ghost Map underlying mathematical engine takes a program and a CWE and identifies any paths through the program's CFG that have the potential to violate the correctness property. Each such path is built into a level in the Ghost Map game. During game play, the player performs actions that attempt to resolve the potential violation path, that is, to establish that the path is not realizable in the program. If all the levels for a program and a CWE are resolved by game play, then we have a proof that the program is free from the CWE vulnerability. In model checking terms, Ghost Map game players perform counterexample-guided abstraction refinement (CEGAR), in that they extend the CFG to a more precise model as necessary to verify the correctness of the software with respect to the CWE in question. The verification approach used by Ghost Map is based on the MOPS tool, which was shown successful over a series of papers [30; 31]. Ghost Map game play attempts to resolve the potential violations identified by MOPS, with the goal of reducing the numbers of false alarms that waste the time of programmers and verification experts. In the future, the Ghost Map approach could potentially be combined with commercial tools that generate vulnerability warnings, such as Coverity and HP Fortify.

2. Ghost Map

The high-level theme of Ghost Map is that the player is a cybernetic entity attempting to achieve consciousness. The software CFGs are described as aspects of the cybernetic entities own programming and the potential violation paths in the CFG are called locks, meaning obstacles to consciousness. The player/entity resolves the paths in order to break the locks and achieve its goal. The cybernetic entity theme is not deeply developed in the initial game and it is possible for players to ignore the theme and play purely abstractly if they so choose.

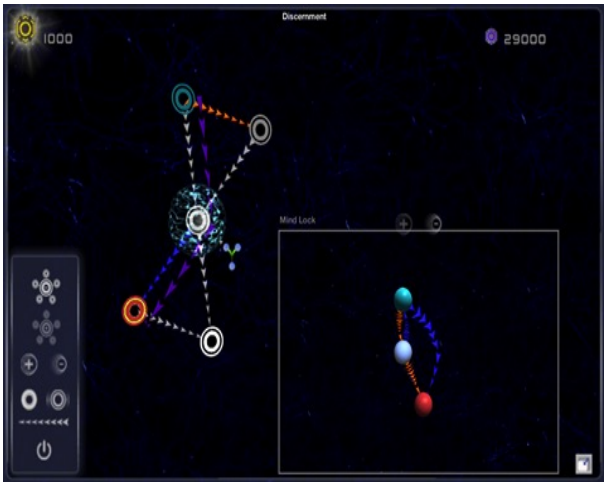


Figure 3-1: Simple Example of Ghost Map Level

A simple example of a Ghost Map game level is shown in Figure 3-1. The software CFG is the X-like pattern in the middle of the figure. The three node graph in the box in the lower right is a representation of the software vulnerability being addressed. The purple arrows on the CFG show a potential violation path that must be addressed. The player uses game tools to "cleave" the haloed node into two nodes. After the cleaving operation, a modified CFG will appear and each of the new nodes will have just one incoming edge. The player then is able to propose the elimination of the new path that contains the blue edge. More details on the game play of Ghost Map are available in Watro et al. [32] and at the Verigames web site.

3. Ghost Map Hypersapce

For the second game, the team decided to retain the underlying mathematical approach but to update the

game. The new game, called Ghost Map Hyperspace, addresses several observations from early play testing. First, initial play testing showed that players lacked the needed information to make informed choices on path elimination proposals. The vulnerability pattern window in the game did allow users to infer that certain paths would be valuable to eliminate, but nowhere in the game was their data that suggested that a path could be successfully eliminated. The Hyperspace game attempts to resolve this issue with the use of "energy analysis," discussed below.

Another observation from Ghost Map was that cybernetic organism theme was confusing at times, as the game narrative concepts such as the organism's software overlapped with the underlying verification concepts, such as the software being proved correct. Also, the theme did not seem to foster engagement from players. For Ghost Map Hyperspace, we adopted a "space opera" theme that we believe will be more engaging, less confusing, and will allow easy expansion of the narrative to cover the new data that supports path decisions.

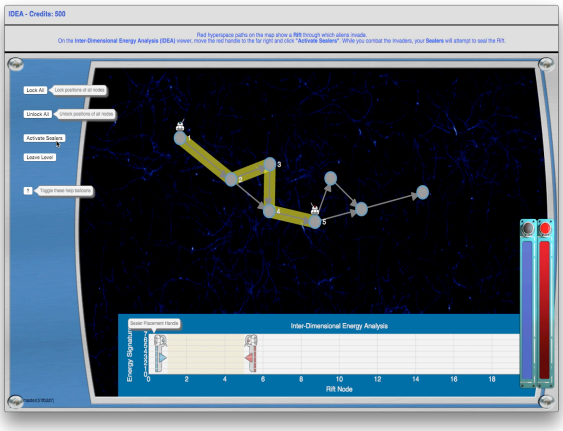


Figure 3-2: Example of a Ghost Map Hyperspace Level

Finally, one of the issues with Ghost Map is the significant delay required to process the path elimination input. In Ghost Map Hyperspace, we include additional game play activities that are integrated with the overall theme and occur while the path elimination process is running. We are hopeful that this new feature will support a more balanced game play experience.

Figure 3-2 shows a screen shots from Ghost Map Hyperspace. The potential violation path is shown as a highlighted segment of a portion of the CFG, much as in Phase 1. In the new narrative, the potential violation path is a rift in hyperspace that the player is attempting to seal. In Figure 3-3, we see a second example where variable reads and writes in the software have been modeled as energy exchanges and displayed in the chart at the bottom of the game window. These energy analysis readings allow the game player to make better path removal suggestions since they reflect actual data exchanges in the software. Once the elimination suggestion is completed, a combat game begins that represents alien ships slipping through the rift to attack. Points scored in the combat game add to the players total and the rift sealing results (determined by the math backend) are released at a later point in game play. More information on the player engagement strategy in Ghost Map Hyperspace can be found in Moffitt et al [33].

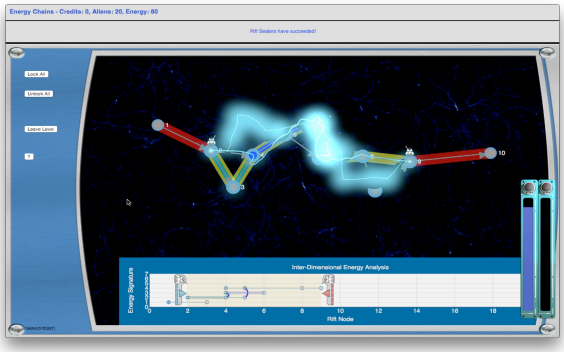


Figure 3-3: Using energy clues to seal rifts

4. Ghost Map Summary

Since the initial release in December 2013, more than a thousand users have played Ghost Map and hundreds of small proofs have been completed. Ghost Map demonstrates the basic feasibility of using games to generate proofs and provides a new approach to performing refinement for model-checking approaches. In addition to the immediate benefits of verifying software using games, we also anticipate that the Ghost Map approach may enable new automated methods as well. Through the intermediate representations we have developed and the proof tools we have created for validating edge removals, we believe the possibility of creating novel refinement algorithms is significant.

Solution 4: StormBound and Monster Proof

Aaron Cammarata, *VoidALPHA* Aaron Tomb, *Galoi, Inc.*

1. Introduction

Our team is Galois, specialists in formal methods, and voidALPHA, a videogame studio. We first built StormBound, which challenged players to find patterns in magical energy and save their planet. Based on lessons learned from StormBound, we are building Monster Proof, in which players solve puzzles to gather resources and become wealthy beyond desire.

2. Verification Approach

Our games used two different implementations of the same verification approach. In the games, players use their intuition and insight to generate assertions about the code being verified. The verification back end creates individual puzzles, which are then presented in-game. It assembles player answers (logical assertions), and tries to perform an end-to-end verification.

In StormBound, our approach was to instrument the code being verified, and take snapshots of the software during execution. This generated 'trace data', which captured the values of in-scope variables at key program points. The players identified patterns in those data, for example noting the relationship between an integer function parameter and the size of a local array. Taken collectively, these player-generated assertions sketched out a spec for 'normal operation' of the program, which in turn acted as hints for the verification solvers.

In Monster Proof, we establish the weakest precondition under which a desired property holds for a block of code. We then ask the player to discover invariants that prove the preconditions by using pre-defined rules to transform or supplement those preconditions. For a trivial example, a player may be tasked with proving the precondition "a < c", by identifying the invariant "a < b" in a context where "b < c" is already known.

3. Game Descriptions

StormBound is:

- Story-driven engagement
- In a "Magepunk" universe, a blend of brass/steam and glowing magical runes
- Designed to "completely hide the math": allow players to make assertions without any math or numbers in-game

Figure 4-1:StormBound play screen

- Targeted to a broader, casual audience
- Created with Unity Webplayer, embedded in a MeteorJS web page

Figure 4-2: Monster Proof Game Screen

Monster Proof is:

- A Resource-gathering and collection
- Utilizes cute cartoon monsters, with an emphasis on tongue-in-cheek humor
- Designed to to "completely show the math": give players tons of context, and focus on efficiency and comprehension
- Targeted to a focused puzzle-game audience
- Created with Famo.us for HTML/CSS Sprites, and MeteorJS for web page / server

4. Game Results

The audience of the StormBound followed a typical industry adoption curve – numerous players up front at launch, tapering off to a steady state, trailing off over

time. All told, 10,650 players tried the game, 7,264 in the three weeks after launch in December 2013. The game continued to attract about 150 players / week until June, then dropped to near zero.

We received 142,711 valid assertions – successful solutions – generated over 2,919.2 hours. Note: levels can have multiple solutions. (All figures exclude CSFV team members.)

In order for a level to be verified, it must have at least one player-generated answer. By the end of the active play period, players had contributed to 4,361 out of 6,523 levels (66.8%).

When we began, automated tools could discharge about 19% of the work with no human input. Improvements to automated tools done under the CSFV program resolved an additional 15%, and player-assisted levels solved an additional 15%, totaling about 49%. Once automated tools remove some of the workload, players completed 22.3% of the remaining work. Note that all of these measures apply to verifying program properties in isolation rather than across the entire code base—a weakness we are addressing in the Phase 2 game.

The original code base was about 300,000 lines of code (LOC), so players touched about 103 LOC per hour of gameplay, and contributed to verifying 15.4 LOC per hour. The reason these differ is because as you'll see, in StormBound it was possible to give us an answer that isn't useful for making verification progress – players could easily 'waste' effort.

As with any free-to-play offering, players dropped off quickly as they went through our tutorials. Of the 10,650 registered players who watched the intro story cutscene, only 2,048 (19.2%) completed the sixth tutorial, which is when the player begins contributing to verification progress. This is analogous to the "conversion rate" – the percentage of players who convert to paying customers. Since this is a research effort, we define 'conversion' as 'contributing to the problem'. Standard industry conversion rates are often in the 3% range, so 19% might indicate that players motivated by "contributing to science" are more invested in sticking with the game.

5. Assessment & Lessons Learned

According to Flow Theory, much of a game's enjoyment comes from a delicate balance between a player feeling competent and feeling challenged. Game designers craft complex game systems that aim to self-regulate and adapt to player skills, or at least provide a measured, reasonable path of progression.

The biggest challenge in a 'real science game' is that the solutions for levels are by definition unknown, and unknowable - if the answer could be computed, the system would not need the players. This means there is no reliable predictor of level difficulty. A 'small' level can be impossible to resolve, while a very large level with lots of data might require only a single action to solve, like collapsing a house of cards with a gentle tap. In StormBound, this was exacerbated by the fact that even after we got a player's solution, we didn't know if it would help verification. It may have been an interesting fact, and true, but not necessary to construct a proof. The analogy we used was 'shooting mosquitos with a shotgun'. Players could generate lots of true assertions, but determining their usefulness could take days. Not being able to give players immediate feedback really hamstrung our ability to use common game feedback mechanisms.

In Monster Proof, we are addressing these issues by putting the verification engine closer to the player. As you play a level, you know what it is you're trying to build (there is a clear 'goal' for each level), and you know unequivocally whether you solved it or not. It is still possible to do a certain amount of 'solution by intuition', but generally you know which pieces of the puzzle are relevant and which are not. We are investigating if this improves two metrics. First, we believe that it will result in better retention. The highly math-centric style might discourage some users, resulting in a smaller audience, but we theorize that the players who do continue with the game will find it far more satisfying than those who started StormBound thinking they'd be playing a cool space RPG and found only an unsatisfying make-work task. Second, we feel that the increased context and transparency within the core game will greatly reduce 'effort waste'. That is, we are replacing the player's shotgun with a (figurative) set of building blocks and a target shape. It's then up to the player to assemble the blocks, using known and teachable rules, into the desired shape. Players should be able to address the complete problem more quickly, and produce more verification progress during an equivalent amount of gameplay.

Another challenge of designing these games is something we have come to call "The Bump". That is, the transition between custom tutorial levels, designed for clarity and pedagogy, into 'real' levels derived from the code. Because there is no way to classify level difficulty, players are effectively 'thrown into the deep end' – because all of the actual problems are deep end. The only remotely effective solution we identified was to make players fairly skillful before letting them into the 'real data' pool. This results in a long ramp-up time

before you can contribute, and feeling like a 'citizen scientist' is a key motivator for people who play these games. Requiring 30-60 minutes of tutorials before you can help is frustrating, and leads to churn (player departure).

Worse, it's possible that a level is, in fact, unsolvable – and it is impossible to know this in advance. To account for this, designers need to provide a way to 'win' even unwinnable levels. In StormBound, this could only be detected if players made every possible assertion through the game UI (which could take hours or even days). In Monster Proof, a player can demonstrate that a level is, in fact, unsolvable. They can then "bang a gavel" to assert that the level is unsolvable (possibly indicating that the code is in fact unverifiable), and place a bet on that assertion. If someone else is later able to solve the level, the first player loses her bet, while the second collects it. If three players report that a level is unsolvable, we set it aside for expert review, and reward players. It is important that, again, since gameplay emerges from data over which you have no control, players have a way to feel successful in all cases.

Tutorial design was also challenging – we struggled to find the best 'voice' for the narrator / instructor. Since our tutorials needed to teach more than just basic game mechanics, we vacillated between speaking "game" and "science". In StormBound, because we were math-phobic, we twisted and contorted our script to fit into the game universe's vocabulary. Our intent was to allow players to relax into the game narrative and not break the 'fourth wall'. Instead, it frustrated players, who just wanted to know what everything actually "was", so they could work with it. In Monster Proof, we are using a lot less game language, and while we have not completely eliminated such language, we are being a lot more cautious and intentional to use game-themed language only where it affects the resource collection meta-game, and not the core logic problem.

As we designed the games, we thought quite a bit about "griefing" – cheating or interfering with other players. This did not happen, but sometimes players gave us lots of useless answers (and scored tons of points) because they game told them they were doing well. The key takeaway is that players want to help, so you need to give clear feedback about what you need.

Thematically, we found that the primary motivator for players was in fact the 'citizen scientist' role. It's important to give them feedback about their effort in terms they can understand, preferably in the language of the underlying science.

We found that although players wanted to contribute to science, they didn't want to learn it. Many players dismissed or skimmed tutorials, then complained they didn't understand the game. This remains a point of design friction for which we do not have a great solution.

Finally, as development unfolded we discovered how to automate certain classes of solution. In StormBound, we did not do very much automated solving. In Monster Proof, we are automating everything we can, so players will not be given 'busy work'. We do have a concern that this leaves only very challenging levels, which will exacerbate the issue with level difficulty.

5. Conclusions & Future Work

We feel the key takeaway from projects like CSFV is that 'utilitainment' is here to stay. Games and applications like these are the very first, unstable steps of a new industry, in which high-cost, high-skill, low-supply work is done by a low-cost, low-skill, high-supply crowd. As game designers, we are only just beginning to understand how to craft a satisfying, entertaining experience that produces useful results. We believe that with continued work, game-based work on problems that require human intuition (i.e. are not easily automated) could be a viable industry within the next 10 years.

Solution 5: Xylem and Binary Fission

John Murray, *SRI International* Heather Logas, *University of California, Santa Cruz*
Jim Whitehead, *University of California, Santa Cruz*

1. Introduction

In this section, we describe two games developed: *Xylem: The Code of Plants* and *Binary Fission*. *Xylem* is a logical induction puzzle game where the player plays a botanist exploring and discovering new forms of plant life on a mysterious island. Players observe patterns in the way a plant grows, and then construct mathematical equations to express the observations they make. In doing so, players work in concert with the game's mechanics to perform loop invariant synthesis.

Xylem was designed with a "casual niche" audience in mind. The idea was to appeal to as wide a player base as possible, while addressing the concern that including mathematical game play would somewhat limit the audience. To that end, the game design team chose to use a visual metaphor (plants, for their representational flexibility) and make the gameplay as light on math as possible while still supporting the underlying formal verification problem. Focus was given to creating a smooth player experience in a typical casual game to avoid confusing players. However, this approach proved to be largely ineffective in addressing the broader task of crowd-sourcing formal verification. Casual players were not interested in the math oriented gameplay, while those who enjoyed the science goals were frustrated by the lack of more advanced math tools with which to describe patterns.

The second game, *Binary Fission*, sought to address these problems by taking the project in a new direction. Instead of addressing pure game players, we instead focused on a "citizen science" audience. Player reports from *Xylem* indicated that those most engaged in the game were also those who were interested in the actual CSFV program goal, i.e. formal software verification.

The project is led by SRI International, a non-profit research institute based in Menlo Park CA. *Xylem* and *Binary Fission* were both designed and developed at the University of California at Santa Cruz. The verification infrastructure is provided by CEA, the research arm of the Atomic Energy Commission in France.

2. Verification Strategy

Xylem problems were generated from source code using Frama-C, with an additional value analysis module. Sets of variable values were delivered to players as game instances. A fast response to players' proposed solutions is key for reward and retention. However, traditional confirmatory analysis of invariants can take many hours of CPU time, and is thus impractical in a game environment. Using a Hasse partial ordering approach, in conjunction with our backend verification modules, enables us to sieve play results and enables an initial coarse ranking of candidate invariant solutions.

For progress metrics, we use abstract interpretation-based software analysis to determine the overall potential state space. We propagate states to encompass all possible execution paths. State space management is a key issue for industrial-strength software analysis. It triggers non-termination, over-widening, and false alarms during the analysis process. Frama-C/Value Analysis takes advantage of crowd-sourced candidate invariants to significantly reduce its state space.

3. Game Descriptions

Xylem is a logical induction puzzle game where players are botanists exploring the strange island of Miraflora.

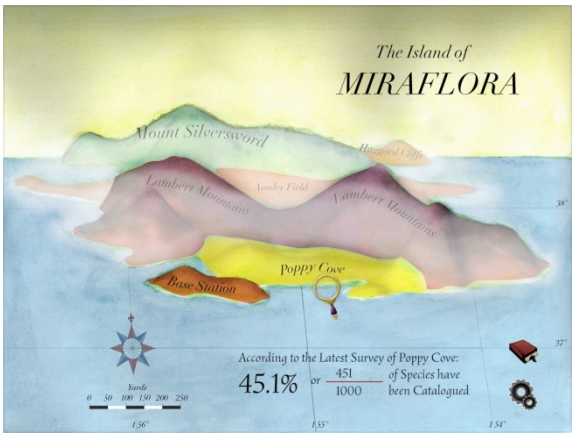


Figure 6-1:Xylem: Miraflora Island

Players are tasked with observing and comparing the growth patterns of the plants they discover, as they travel around the island. The Floraphase Comparator is used for this purpose. In describing the growth patterns, the players also provide candidate loop invariants for the CSFV verification task.

Figure 6-2:Xylem: Floraphase Comparator

Each region of Miraflora contains increasingly hard problems. Access to interior regions is granted only when the entire player base has collectively solve a certain number of problems in earlier areas.

In the second game, *Binary Fission*, players still work with loop invariants, but now they refine searches performed by an automated system instead of creating simple invariants from observations of data changes over time. *Binary Fission* presents players with an abstract tree-like structure of nodes. Each node contains a number of "bits" (or "atoms", as players like to call them) in either purple or green. The player's job is to sort the bits using provided filters, in an attempt to create "clean sets" -- that is, nodes which contain only one color of bits. As an additional challenge, players must create these clean sets while using as few nodes as possible (i.e. performing as few as possible sorts).

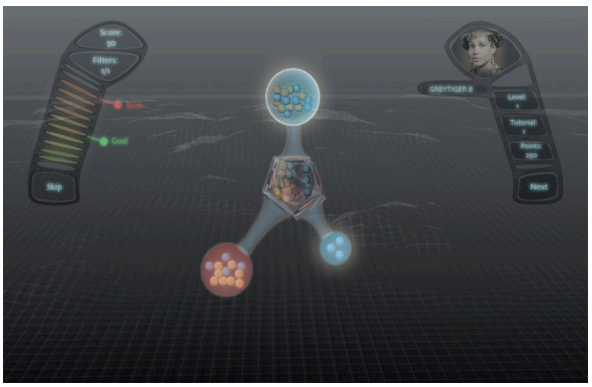


Figure 6-3:Binary Fission: Tree Structure

For each node, the game provides up to a hundred filters to choose from. The filters are presented as small spheres set in a circular container. As players move their mouse cursor over the spheres, they are shown in real time how that particular filter would sort the node. This takes advantage of a key thing humans can do better than computers - visual pattern recognition. Players can additionally save filters for later in case the one they have chosen doesn't produce the results they would like later in the filtering process.

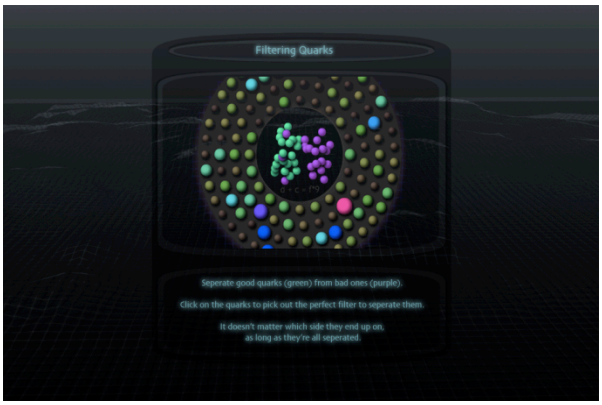


Figure 6-4:Binary Fission: Fliter Selection

The auxiliary *Binary Fission* feature set is very light, since our goal is to keep players focused on solving problems. The game features live chat, in order to foster a sense of community among players and help with player retention. *Binary Fission* also clearly shows community progress in the form of number of problems solved on the main menu screen, in order to reinforce the sense of collaborative citizen science.

4. Lessons Learned

Xylem: The Code of Plants was designed with a "casual niche" audience in mind. Our concept was that, even though we could not legitimately pursue a truly "casual" audience (by game industry definitions) due to the math gameplay inherent in the core game design, it would still be worthwhile to pursue as "casual an audience as possible." This was important in order to bring in more players, which we believed would best take advantage of the crowd-sourcing nature of the application. To attract and keep this audience, we created a game around math-based puzzle solving, but with as lightweight math as we could manage (while still keeping the integrity of the science task) and within the bounds of a narrative-oriented casual puzzle game.

Xylem turned out to attract a much smaller audience than we would have preferred. The math oriented game play was not (for the most part) appealing to the larger puzzle game audience. Instead, we found that the players who most enjoyed *Xylem* were most likely to be people who came to our game with an already established interest in math and computer science, and were

drawn by the stated science objectives. During the first nine months of gameplay, our top 20 players submitted a total of 1754 invariant solutions.

In designing *Binary Fission*, we decided to change our tactics. Instead of attempting to bring in the largest crowd possible, we decided to focus on pulling in a high quality crowd. We changed our approach completely in order to attract and maintain a different sort of audience - citizen scientists who are interested in the science problem being solved.

Building off the lessons learned from our experience with *Xylem*, as well as additional research into automated invariant synthesis and design principles from other successful citizen science projects, we believe that *Binary Fission* will provides better CSFV results than *Xylem* for several reasons. For example, as a citizen science project, our recruitment policy draws in players who are interested in cybersecurity, many of whom are less likely to have conflicts with mathematical gameplay. Also, our science goals are transparent within the game itself and in all marketing materials.

Binary Fission partners with other methods of crowdsourced synthesis of candidate invariants, such as *Xylem* and similar CSFV games, as well as automated generation of candidates. Thus, players are asked to guide searches through suites of potential invariants, rather than produce invariants from scratch (although players are able to do this too). The game thus integrates the best skills of both the human and computer partners. *Binary Fission* enables the creation of disjunctive invariants, which is a key advantage over traditional automated systems.

Binary Fission emphasizes community, an important aspect of successful citizen science projects, through better-integrated chat, active community management, and regular community events. The game also allows for more player choice by allowing them to select puzzles to work on from a visible group of problems every time they play. The *Binary Fission* tutorial assumes a higher level of sophistication in players, and therefore focuses on teaching the game interface rather than teaching about the game. The tutorial is much shorter, allowing players to reach productive ability levels much faster.

5. Conclusions & Future Work

Our vision of appealing to a less-math-literate audience with *Xylem* was not as successful as we anticipated, primarily because of the complexity of some solutions and/or the potential lack of clear answers for certain problems. In addition, the nature of the verification challenge made it difficult to consistently assign levels of difficulty to problem instances. We nonetheless were able to make a largely inaccessible task accessible to a wide variety of people, making it instantly understandable to advanced players and less alienating to those who will not necessarily become experts but want to try the game. Discovering the characteristics of our true audience helped to drive the design of updates to *Xylem* and to inform the strategy for *Binary Fission*.

Looking beyond the first release of *Binary Fission*, we plan to support different levels and styles of play, with at least two distinct play styles that are interdependent on each other. These roles will allow for different expenditures of cognitive energy; less-math-literate players who are interested in contributing to the science goals of the project can contribute alongside those who are more math-sophisticated. Further, players can switch freely between roles as they see fit. *Binary Fission* will also offer more player choice by allowing them to select from a visible group of problems every time they play. Solutions will also be forkable, so that multiple players can take a single problem in several different directions.

Conclusions and Lessons Learned

Overall, across the development of these five efforts, the crowd-sourced formal verification has shown mixed success in demonstrating the potential for crowdsourcing to enrich the formal verification process. In each effort, solutions have been collected from numerous players, providing significant progress towards formal verification proofs. Furthermore, these efforts provide several critical lessons that drove the development of the second set of formal verification games that are now being tested, and that can be readily extended to other citizen science and game-based crowdsourcing efforts.

One key lesson learned across several of these efforts is to know the player population. At the start of the program, a key focus was to develop games that would be engaging enough to bring crowds of players with no significant mathematical background. We quickly learned that this was not the best way to motivate high-contributing players. Rather than drive a general population, each of these games was better served by citizen scientists with a strong interest in the underlying science and outcome of the effort (e.g., players with a mathematical and computational interest and/or background). While it is important for the games to be engaging for citizen scientists, it is perhaps more important that these players understand the types of contributions they are making and the impact they are having on addressing the scientific problem. That combination of intrinsic and extrinsic value to the player has been the greater focus for the second round of games, which will be tested over the summer of 2015.

Scientific tasks, such as those performed in the course of formal verification, often involve both complex logical or abstract problem-solving and simple, rote repetition of previously learned strategies. The most valuable work on these problems can only be done once the repetitive solutions have been exhausted. This pushes the creators of a game-based task to teach concepts to the player in rapid succession, in hopes that the player will learn enough to contribute meaningfully before walking away from the game. With so many concepts to teach, it becomes difficult to keep the terminology simple and accessible and to give the player enough of an opportunity to practice and grasp a concept before the next one is introduced. Our teams took several approaches to solve this problem in the second round of games, from progressions of tools that teach the player key concepts when they are unlocked to video tutorials using humorous in-game characters to keep the player entertained while learning to play.

Related to this, a key challenge in any citizen science gamification effort is navigating the tradeoff between making a game *engaging* and making the game *address critical problems*. When the game is being designed for a very specific purpose, game designers have a limited ability to modify game elements to drive a more engaging experience. Rather, the game must capture and address a specific, structured problem—and cannot stray too far from the structure of that problem in the process. One way to address this issue is to separate the puzzle-solving process (related to addressing actual citizen science problems) from a game section that is focused on fun and accomplishment. While this can be a successful approach to make the games more engaging, providing that engaging game can limit the contributions that are made by the game players (who may wish to spend more time on the fun game than on the puzzle-solving process). Our teams took a variety of approaches to address this problem, ranging from targeting citizen science audiences (as described above) to incorporating the engagement elements during downtime in the puzzle-solving process to maximizing the use of human intuition and insight for problem-solving, which makes the problems more fun to solve.

Related to this latter element, many of the games benefited strongly from incorporating an autosolver to address wide segments of the problem. Rather than having the human address every element of the computational problem, humans were focused on either guiding the autosolver (e.g., in the case of *Paradox* and *Dyanamkr*) or addressing only the complex problems that need human insights. When there are numerous tedious problems that need to be solved on the way to addressing a larger computational problem—as is the case in formal verification proofs—autosolvers can be extremely useful to manage the work that must be addressed by citizen scientists. However, they pose a number of challenges as well. For example, overusing automation can lead players to question whether the computer is really doing all the work and if so, why they should bother to play at all. In addition, if players have a limited understanding of what the automation is doing, and, because of that, a limited understanding of what *they* are doing, it will lead to errors, frustration, and attrition. This is further exasperated by the bump in complexity from training levels to live levels (which are often a lot more complex than the levels used to train players on the game concept). Ultimately, judicious use of an autosolver that allows citizen scientists to focus on the problem aspects where they can make the greatest contributions and learn the details as they need them can make the game more fun and more accessible.

Across all of these individual points we find that the main lesson has been the challenge of turning a task into a game without sacrificing too much of the player's

time on pure engagement mechanics and without compromising the value of the task. It is easy to focus too heavily on the constraints of the task and to lose focus on the things that constrain good games: clarity (of goals and the consequences of actions) and value to the player (through entertainment, improvement, social rewards, etc). Without these things, the game fails to motivate play and the opportunity to leverage leisure time to accomplish scientific goals can be lost.

Disclaimed and Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] Hoare, C. A. R. (2002). Proof of correctness of data representations.: Springer.

[2] Lampson, B. W. (1974). Protection. ACM SIGOPS Operating Systems Review, 8, 18-24.

[3] Bell, D. E. and LaPadula, L. J. (1973). Secure computer systems: Mathematical foundations. DTIC Document.

[4] Lipton, R. J. and Snyder, L. (1977). A linear time algorithm for deciding subject security. Journal of the ACM (JACM), 24, 455-464.

[5] Neumann, P., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L. (1980). A provably secure operating system: The system, its applications, and proofs.: SRI International.

[6] Feiertag, R. J. (1980). A technique for proving specifications are multilevel secure. DTIC Document.

[7] Rushby, J. M. (1981). Design and verification of secure systems. ACM SIGOPS Operating Systems Review, 15, 12-21.

[8] Levitt, K. N., Crocker, S., and Craigen, D. (1985). VERkshop III: Verification workshop. ACM SIGSOFT Software Engineering Notes, 10, 1-136.

[9] Neumann, P. G. (1981). VERkshop II: Verification Workshop. ACM SIGSOFT Software Engineering Notes, 6, 1-63.

[10] Landwehr, C. E. (1981). Formal models for computer security. ACM Computing Surveys (CSUR), 13, 247-278.

[11] Young, W. D., Boebert, W. E., and Kain, R. Y. (1988). Proving a computer system secure. ADVANCES IN COMPUTER SYSTEM SECURITY., 1988, 3.

[12] U.S. Department of Defense (1985). Department of Defense trusted computer system evaluation criteria. DoD 5200.28-STD.

[13] Lowe, G. (1996). Breaking and fixing the Needham-Schroeder public-key protocol using FDR. Tools and Algorithms for the Construction and Analysis of Systems, 147-166.

[14] Mitchell, J. C., Mitchell, M., and Stern, U. (1997). Automated analysis of cryptographic protocols using Murφ. Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on, 141-151.

[15] Blaze, M., Feigenbaum, J., and Lacy, J. (1996). Decentralized trust management. Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on, 164-173.

[16] Hu, W.-M. (1992). Reducing timing channels with fuzzy time. Journal of computer security, 1, 233-254.

[17] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. (1998). StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Usenix Security, 98, 63-78.

[18] Toth, T. and Kruegel, C. (2002). Accurate buffer overflow detection via abstract pay load execution. Recent Advances in Intrusion Detection, 274-291.

[19] Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. (2001). RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. USENIX Security Symposium, 165-176.

[20] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., and Norrish, M. (2009). seL4: Formal verification of an OS kernel. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 207-220.

[21] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. ACM Computing Surveys (CSUR), 41, 19.

[22] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, 358-366.

[23] DeOrio, A. and Bertacco, V. (2009). Human computing for EDA. Proceedings of the 46th annual design automation conference, 621-622.

[24] Bertacco, V. (2012). Humans for EDA and EDA for humans. Proceedings of the 49th Annual Design Automation Conference, 729-733.

[25] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. Journal of the ACM (JACM), 7, 201-215.

[26] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. Communications of the ACM, 5, 394-397.

[27] Jiang, Y., Kautz, H., and Selman, B. (1995). Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. 1st International Joint Workshop on Artificial Intelligence and Operations Research.

[28] Borchers, B. and Furman, J. (1998). A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Journal of Combinatorial Optimization, 2, 299-306.

[29] 2011 CWE/SANS Top 25 Most Dangerous Sofwtare Errors, http://cwe.mitre.org/top25/index.html.

[30] Chen, H. and Wagner, D. (2002). MOPS: an infrastructure for examing security properties of software. Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), Washington, DC.

[31] Chen, H., Wagner, D. and Dean, D. (2004). Model Checking One Million Lines of C Code. Proceedings of the Network and Distributed Security Symposium (NDSS). San Diego, CA.

[32] Watro, W., Moffitt, K., Hussain, T., Wyshcogrod, D., Ostwald, J., Kong, D., Bowers, C., Church, E., Guttman, J., and Wang, Q (2014). Ghost Map: Proving software correctness using games. Proceedings of the Eight International COnference on Emerging Security Information, Systems, and Tecnologies (SECUREWARE). Lisbon, Portugal.

[33] Moffitt, K., Ostwald, J., Watro, R., and Church, E. (2015). Making Hard Fun in Crowdsourced Model Checking: Balancing Crowd Engagement and Efficiency to Maximize Output in Proof by Games. Proceedings of the Second INternational Workshop on Crowdsourcing in Sofware Engineering. Florence, Italy.