Ghost Map: Proving Software Correctness using Games

Ronald Watro, Kerry Moffitt, Talib Hussain, Daniel Wyschogrod, John Ostwald and Derrick Kong
Raytheon BBN Technologies
Cambridge MA USA
{rwatro, kmoffitt, thussain, dwyschog, jostwald, dkong}@bbn.com

Clint Bowers
Univ. Central Florida
Orlando FL USA
clint.bowers@ucf.edu

Eric Church
Breakaway Games Ltd
Hunt Valley MD USA
echurch@breakawayltd.com

Joshua Guttman
Worchester Polytechnic Institute
Worchester MA USA
guttman@wpi.edu

Qinsi Wang
Carnegie Mellon Univ.
Pittsburg PA USA
qinsiw@cs.cmu.edu

Abstract—**A large amount of intellectual effort is expended every day in the play of on-line games. It would be extremely valuable if one could create a system to harness this intellectual effort for practical purposes. In this paper, we discuss a new crowd-sourced, on-line game, called Ghost Map that presents players with arcade-style puzzles to solve. The puzzles in Ghost Map are generated from a formal analysis of the correctness of a software program. In our approach, a puzzle is generated for each potential flaw in the software and the crowd can produce a formal proof of the software's correctness by solving all the corresponding puzzles. Creating a crowd-sourced game entails many challenges, and we introduce some of the lessons we learned in designing and deploying our game, with an emphasis on the challenges in producing real-time client gameplay that interacts with a server-based verification engine. Finally, we discuss our planned next steps, including extending Ghost Map's ability to handle more complex software and improving the game mechanics to enable players to bring additional skills and intuitions to bear on those more complex problems.**

Keywords-games; static analyses; formal verification; crowd souring; games; model checking.

I. INTRODUCTION

Errors in computer software continue to cause serious problems. It has long been a goal of formal verification to use mathematical techniques to prove that software is free from errors. Two common approaches to formal verification are: (a) interactive theorem proving [1][2], where human experts attempt to create proofs with the assistance of interactive proof tools. This is often a slow and laborious process, with many man-years of effort needed from human experts to prove the correctness of real-world software, and (b) model checking [3][4][5], where proofs are created using systematic techniques that verify specific properties by generating and validating simplified models of the software. Model checking is a mostly automated process, but is susceptible to failure due to the size of the search space ("the state space explosion problem"). Because of the issues with both common approaches, formally verifying modern software does not scale well – verifying software of moderate to large size (e.g., hundreds of thousands of lines of code or more) is rarely a practically viable option.

Recent research has demonstrated the benefits of using games to enable non-experts to help solve large and/or complex problems [6][7][8][9]. We propose to improve the success of formal verification of software through the use of a crowd-sourced game based on model checking. Our game, called Ghost Map, is in active use at the Verigames web site [10]. By breaking verification problems into smaller, simpler problems, Ghost Map enables game players to create proofs of correctness and help direct the model checking processes down the most promising search paths for creating additional proofs. Ghost Map leverages the significant intuitive and visual processing capabilities of human players to tackle the state space explosion problem of a model checking approach. The game engages the player's motivation through a narrative that encourages them to solve a variety of puzzles. In this case, a player is a recently emerged sentient program, and the player's goal is to remove ("disconnect") as many limitations ("locks") on that sentience as possible in order to grow and remain free. Through the process of disconnecting locks, the player is actually creating proofs about the correctness of real-world software.

The Ghost Map game is built on top of the MOdelchecking Programs for Security properties (MOPS) tool [11]. MOPS checks C software for known software flaws, such as the SANS/MITRE Common Weakness Enumeration (CWE) Top 25 list [12]. Each level in the Ghost Map game is a puzzle that represents a potential counterexample found by MOPS. Through the gameplay, players investigate and manipulate the control flow associated with the counterexample in order to eliminate flaws (i.e., disconnect locks) – which is only possible if the flaw is artificial. In this way, Ghost Map extends MOPS with a CounterExample-Guided Abstraction and Refinement (CEGAR) capability [13], where the players introduce and test local refinements. A refinement is the act of re-introducing some information about the software into an abstracted model in order to verify proofs that cannot be verified at the abstracted level alone.

The remainder of this paper is organized as follows. Section 2 provides the needed background on the MOPS tool and Section 3 describes how MOPS model checking is built into a game. Section 4 covers the game play overview and Section 5 discusses the system that was built to support execution of the game on the Internet. Section 6 provides more detail on some important game design decisions. Section 7 discusses future plans and the paper concludes with a summary and conclusions in Section 8.

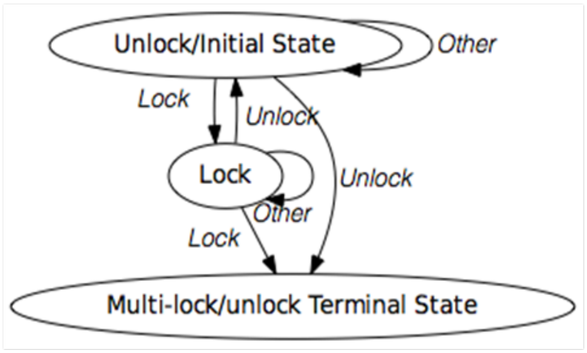
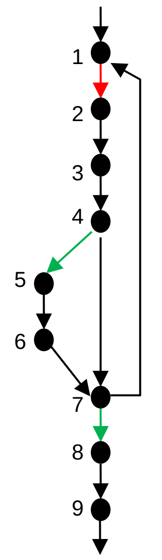


Figure 1. Finite State Automaton (FSA) for lock/unlock software errors.

```
Example() {
1: do {
2:     lock();
3:     old=new;
4:         if (foo) {
5:         unlock();
6:         new ++;
       }
7: while (new != old);
8: unlock();
9: return;
}
```

(a)



(b)

Figure 2. Test program (a) for lock-unlock analysis and corresponding CFG (b).

II. BACKGROUND

We begin with some background on the methods used in the MOPS tool. The goal of MOPS is to help identify instances of common weaknesses (or vulnerabilities) in software. To be analyzed by the MOPS approach, a software weakness must be modeled by a Finite State Automaton (FSA). For example, consider two commands, lock() and unlock(), for locking or unlocking some fixed program resource. It is a potential weakness to call unlock() when the resource is not locked, since the code that called unlock() expected the resource to be locked. Similarly, two calls to lock() without an intervening unlock() is also a weakness. These errors can be represented as an FSA (see Figure 1), where the nodes represent the three possible states (unlocked, locked, error state), and the edges represent the different commands (lock(), unlock()) which can lead to changes in state. The FSA captures the possible starting state(s) of the software program as FSA starting node(s) (in this case, all programs start in an unlocked state). The error state(s) are captured as terminal state(s) in the FSA.

Given a C program and an FSA that represents a software error, MOPS first parses the program and generates a Control Flow Graph (CFG). In general, the CFG captures every line of code in the original software as a node in a graph and every transition from line to line as an edge in a graph. As an example, consider a small C function involving software resource locks and unlocks (see Figure 2a) and the FSA from Figure 1. Figure 2b shows the resulting CFG produced by MOPS. The CFG abstracts out almost all detailed content about the original software (e.g., specific commands, specific variables, etc.). However, based on the FSA, MOPS retains some information about any lines of code that use commands reflected in the FSA. In Figure 2b, the transitions associated with the lock() and unlock() commands use the colors red and green, respectively. Because information about variables values is abstracted out, MOPS introduces some non-determinism into the CFG. For example, when there is a branch statement (e.g., the line "if (foo)") in the software, the CFG will allow both possible branches (e.g., 4 → 5 and 4 → 7) to occur, regardless of state (i.e., whether the value of foo is true or false). Similarly, loops can iterate an arbitrary number of times, since the information about the ending criterion is abstracted out (e.g., 7 → 1 can occur an unbounded number of times).

The CFG created by MOPS is actually abstracted in one additional important way. Through a process known as compaction, MOPS only represents the control flow of the portions of the given program that are relevant to the FSA. For our application, we modified MOPS compaction to retain all edges that introduce branching, loops, and other decision points.

Once it has a (compacted) CFG, MOPS will use the FSA to analyze the CFG and identify whether there are possible paths through the CFG that would lead to a terminal state in the FSA. For example, MOPS will detect that the path going through nodes 1→ 2 → 3 → 4 → 5 →6 →7 →8 would result in an error state (e.g., two unlocks/greens in a row from 4 → 5 and then from 7 → 8 with no intervening lock/red). However, MOPS is only interested in detecting whether an error state could occur at a particular node (e.g., 5), and not in detecting all possible error paths to that node (e.g., the error state at node 5 could also be reached by going through the loop several times before going from 7 to 8). Each such error state at a node found is referred to as a "counter-example" that requires further analysis to determine whether it truly is an error. The CFG of Figure 3a also has a second possible counter-example at node 2, with the shortest path 1→2→3→4→7→1→2. MOPS identifies the shortest possible path to each error node using an efficient algorithm that forms the Cartesian product of the FSA and the CFG (which is a pushdown automaton) and testing whether the resulting pushdown automaton is non-empty. Fortunately, there are fast algorithms for this computation [14], and this enables MOPS to identify all such possible errors very rapidly, even for programs with millions of lines of code and many possible error nodes.

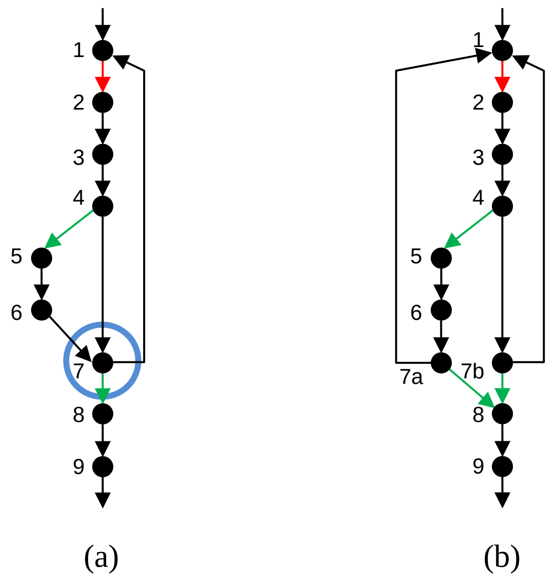


Figure 3. Illustration of cleaving operation.

A MOPS CFG is a conservative model of the C language software that it is based upon. If no instances of the FSA are found in the CFG, then the software is free of the vulnerability in question. On the other hand, if an instance of the FSA is located in the CFG, this does not necessarily mean that the software has the vulnerability. Each instance of an FSA match to the CFG must be further examined to determine whether it is an actual instance of the vulnerability or a spurious instance due to the abstraction and the fact that the data-flow is not considered in the abstracted CFG. (Note that the example program of Figure 3a is actually correct as written, and hence the two counter-examples are in fact spurious).

III. MODEL CHECKING IN GHOST MAP

The core idea of the Ghost Map game is to use game players to check all the counter-examples identified by MOPS for a particular piece of software and a particular set of FSAs (representing different security vulnerabilities). Our goal is to use game play as an integral part of an automated proof system to eliminate as many counter-examples as possible. The result is that the number of counter-examples that need to be manually inspected by expert software engineers is greatly reduced as compared to what would have been produced using the original MOPS system. If the number of FSA matches reaches zero, the system has generated a proof of correctness, with respect to a given vulnerability, of the software (i.e., a proof of the absence of the targeted vulnerability).

To eliminate counter-examples, Ghost Map gameplay uses a process known as refinement [13]. The game offers the player the ability to perform operations that locally undo some of the abstraction that occurred in building the CFG – in particular by removing some of the non-determinism that was introduced by MOPS. The goal of the gameplay is to attempt to refine the CFG into an equivalent graph that has no spurious abstract counterexamples. There are two operations that can be taken in Ghost Map to modify a given graph: cleaving and edge removal.

A. Cleaving

Cleaving takes a node of in-degree n (where n ≥ 2) and splits it into n nodes. Each in-bound edge into the original node is allocated to a different new copy of the node and the outbound edges are duplicated for each new node. In terms of control flow, cleaving simply expands the call flow graph so that the edges after the cleaved node are now separated based on which inbound edge at the cleave point preceded them. Multiple steps of cleaving can be conducted if needed. Figure 3b illustrates the result of cleaving the CFG of Figure 3a at the node 7. The result is two new nodes (7a and 7b), and two ways of getting to node 8 (one from 7a and one from 7b). Essentially, this cleave now allows the CFG to distinguish between a path through the CFG that goes through the 4→5 branch (i.e., “foo” is true) and one that goes through the 4→7b branch (i.e., “foo” is false). When a player requests that a cleave be performed, this operation can be easily performed by the Ghost Map game via a simple graphical manipulation of the CFG. No knowledge of the original source code is needed.

B. Edge Removal

Edge removal is an activity where the game player suggests edges to be removed to eliminate abstract counterexamples. For example, the left hand edge 7a→8 in the cleaved graph is clearly a candidate for removal (see Figure 4a). Why? Because if it can be removed, then the counterexample at node 8 (two unlocks/greens in a row) can never occur. Once a player suggests an edge to be removed, the Ghost Map system must then go back to the original source code of the software in order to determine that the edge can

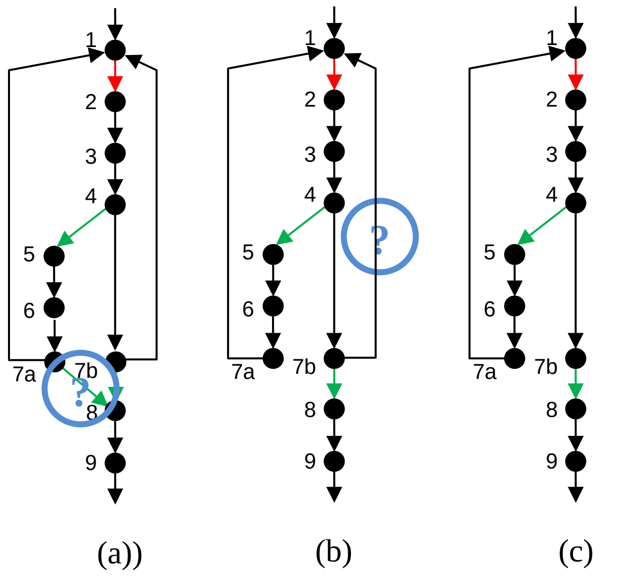


Figure 4. Illustration of edge removal to produce a CFG containing no counter-examples.

be legally removed. An edge can be legally removed if it is not reachable via any legal execution path through the cleaved CFG. Determining removal is currently performed using a test case generation tool called Cloud9 [15] to examine the data constraints in the software. For example, the predicate “new != old” is the key value that helps prove that node 8 is never reachable from node 7a by an actual execution of the function – and hence that the counter-example at node 8 is false and can be eliminated. Within Ghost Map, the player eliminates one counter-example at a time. For example, the player may next seek to eliminate the edge 7b→1 (see Figure 4b). Again, the predicate “new != old” helps prove that this edge can be removed. Once all counter-examples have been eliminated (e.g., Figure 4c), the CFG (at least the part showing in the current game level) has been formally verified to be correct. One can view the final graph in Figure 4c as an “optimization” of the original code, akin to something that might be done by an optimizing compiler. The loop structure of the final graph is now transparently correct for the lock/unlock rule.

IV. Game Play Overview

Our game uses a puzzle-approach, where each game level is essentially an independent puzzle with respect to the other game levels. The basic style of the gameplay is arcade-like with all the information needed by the player presented on the screen at the same time, and the time needed to play a level being relatively short. This approach was selected to ensure that the game was accessible and appealing to a broad range of game players.

Figure 5 illustrates the basic interface of the game.

- At the bottom right of the screen is a representation of the FSA. This can be expanded or shrunk down depending on the player’s preferences. Note that the FSA in Figure 5 is essentially the same as the one in our earlier lock/unlock example.
- The X-like figure in the middle of the screen is a depiction of a very small CFG. Lines use arrows to convey the direction of the edges. Colors are used to distinguish the start node from the node at which the counter-example occurs, as well as from intervening nodes. A colored path is provided to show the shortest path found by MOPS from the start node to the counter-example node.
- Nodes that can be cleaved are indicated with a large highlighted sphere, and a cleave cursor icon can be clicked on the sphere to perform the cleave.
- Edges that can be disconnected (see Figure 6a) are highlighted, and an edge disconnect cursor icon can be clicked on the edge to initiate verification.
- Various helper functions for zooming in and out and highlighting different parts of the graph are provided at the bottom left of the screen.
- At the top of the screen is a summary of the resources available to perform the expensive edge disconnect operations (more details below in Game economy).

The player is free to explore and manipulate the graph as they wish. As they perform key actions, messages appear in the center of the screen describing what is currently happening or what has happened (see Figure 6). Ultimately, the player can win the level, fail the level, or simply switch over to another level and return later.

Incorporating the ability to switch among levels at will was a decision based on the fact that edge disconnection can sometimes take a very long time. To prevent boredom, players can initiate an edge disconnection operation, and then switch to work on another level while the first one is finishing the operation on the server. In future releases of the game, we plan to include additional game play activities to manage the delay generated by edge removal processing.

Ghost Map includes a simple game economy that penal-

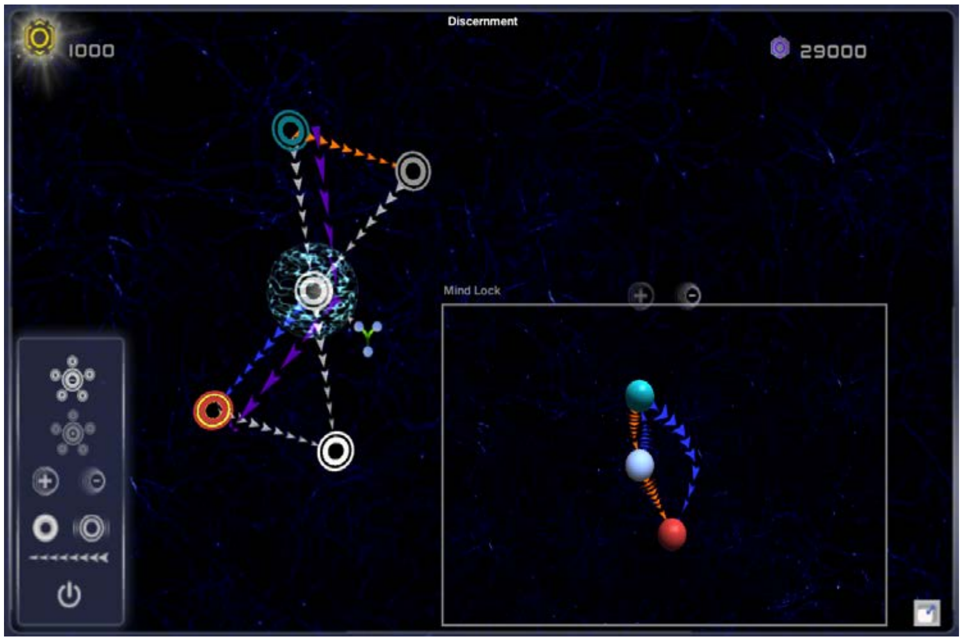


Figure 5. The primary game screen for Ghost Map.

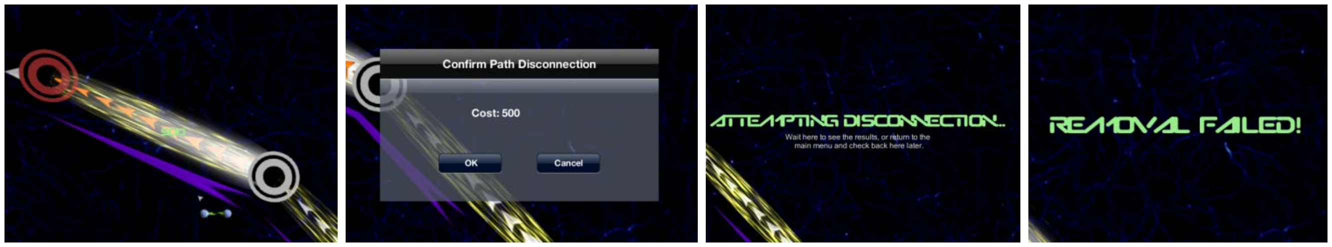

Confirm Path Disconnection
Cost: 500
OK
Cancel
ATTEMPTING DISCONNECTION...
Wait here to see the results, or return to the main menu and check back here later.
REMOVAL FAILED!

Figure 6. Action scenes from the Ghost Map game (figures 6a through 6d).

izes expensive edge disconnect operations that do not succeed and rewards successful decisions. The player begins with a certain amount of credit to solve the current level (e.g., 1000 credits, shown in the top left of the screen, see Figure 5). Every request for an edge disconnect costs a certain amount (e.g., 500 credits, see Figure 6b). If an edge request is unsuccessful, then the credits are consumed, the players are notified of the failure and given chance to try again. If the request is successful, however, then the player receives the current value of the level, which will be 1000 minus the cost of any edge removal requests. MOPS is run again on the updated CFG to determine if there are any remaining counter-examples. If there are, then gameplay continues immediately in a new level.

V. Game System Architecture

The high-level architecture of the Ghost Map game system is shown in Figure 7. The upper portion of the figure shows the off-line processing of the CWE entry and the target software to generate game levels. The game level data and modified C software is loaded into the cloud to be used during game play. Ghost Map is a client-server game. The game clients run the Unity game engine and communicate with the Ghost Map Game Server to receive game levels and to send edge removal requests for verification by the math back end.

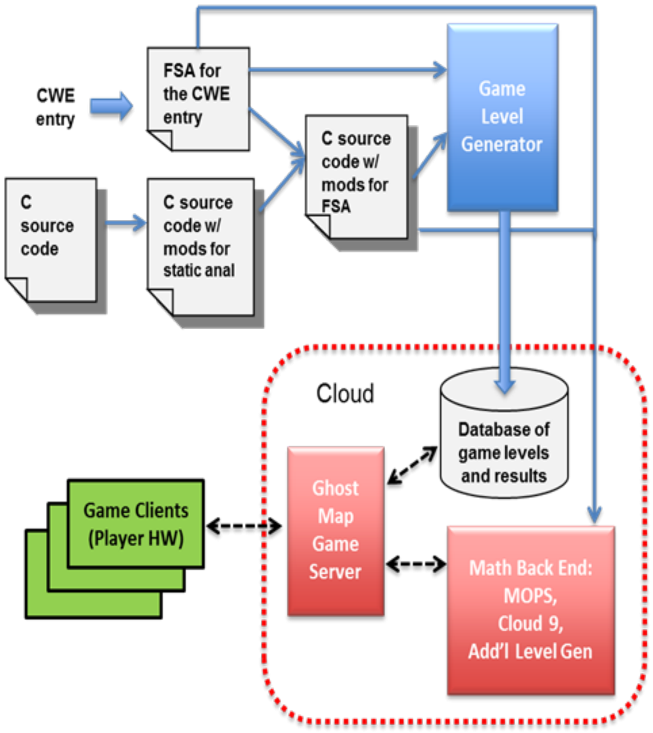

CWE entry
FSA for the CWE entry
C source code
C source code w/ mods for static anal
C source code w/ mods for FSA
Game Level Generator
Cloud
Database of game levels and results
Ghost Map Game Server
Game Clients (Player HW)
Math Back End: MOPS, Cloud 9, Add'l Level Gen

Figure 7. The Ghost Map game system architecture.

VI. Game Design Issues

The goal of our game is to allow players to perform refinements based on insights gleaned from a visual analysis of the CFG and an understanding of the FSA. The intent is that the actions performed by the players are, on the whole, more efficient than the brute force search abilities of computers. In the game play, one or more FSA to CFG matches are identified and displayed to the player.

Within Ghost Map, we chose to use a visual representation that is directly tied to the graphical nature of an FSA and CFG, and to use operations that are directly tied to the acts of cleaving and refinement. During our early design phase, we explored several alternative visualizations that used analogies (e.g., building layouts, mazes, an “upper” world/CFG linked to a “lower” world/FSA, a Tron-like inner world/FSA linked to a “real” outer world/CFG) but preliminary testing with game players revealed that the simpler node-based CFG/FSA visualizations were easier to understand. We instead focused our game design efforts on developing an appealing narrative basis for the game, using visually appealing graphics to display the graphs and motivating the player’s interest in performing the refinement operations efficiently via a game economy. Efficient gameplay was a must. While cleaving is an inexpensive operation, verifying edge removal can be quite expensive to compute.

A. Narrative Basis for Game

Creating an effective game is often an exercise in creating an effective narrative. However, in a crowd-sourced game, there is an additional complication – the narrative basis of the game needs to encourage the player to want to solve the specific problems with which they are presented. Most successful crowd-sourced games to date have actually used a minimal narrative approach. The “story” of the game is the real-life story of the problem being solved (e.g., trying to analyze proteins in FoldIt). In our case, we decided early on that a story based on trying to formally verify software would be too technical and unappealing to the masses. In addition, due the vulnerability protection issue, there are some limitations to the information that we can release about the true story.

Hence, in our early design, we explored a variety of narratives that could be used to motivate the gameplay through analogy. In particular, we wanted the analogy to motivate the specific refinement operations of cleaving and edge removal. We considered several basic approaches for the narrative, each focused on a different type of game reason for eliminating a counter-example from a graphical layout of some sort:

- Having the player focus on circumventing restrictions. For instance, finding out how to solve traps and challenges within an ancient tomb in order to reach the treasure inside.
- Having the player protect others. For instance, having little lemmings moving along the graph and needing to eliminate the counter-examples in order to stop them from dying when they hit the counter-examples.
- Having the player focus on protecting a system. For instance, being a security officer and trying to shut down doorways that are enabling entities from an alternate universe from entering our own to wreak destruction.
- Having the player try to outwit others to survive. For instance, in a Pac-man style gameplay, solving the counter-example provides you with immunity from the enemy (e.g., ghosts) chasing you.
- Having the player trying to escape. For instance, the player is stuck in a maze and the only way out is to solve the counter-example.
- Having the player stop something from escaping. For instance, a sentient program is trying to escape and take over the world, and the player needs to keep it from growing too strong by eliminating its access points to the outside world.

These narrative motivations and ideas were tested with game players to determine their appeal. The last two were found to be the most appealing, and upon further thought, we blended the two within the concept of a newly formed sentient program trying to ensure their growth and survival by eliminating restrictions on their capabilities. This final narrative idea tested well, and added the motivation of an implicit journey of self-realization. An additional benefit of this final narrative idea was that the graph being analyzed by the players could be clearly described as a program that needed to be analyzed. Thus, in keeping with some of the successful approaches mentioned above, we came almost full circle to linking gameplay closely with the specific real-world task

B. Software and Vulnerabilities

One of the design requirements of Ghost Map is the association between a game level and the associated portion of source code being proved correct cannot be known to the crowd. This requirement relates to standard practices for limiting the release of potential software vulnerability information. While Ghost Map is a tool for proving the correctness of software, it is of course true that when correctness proofs fail, vulnerabilities may be present. Even partial information about vulnerabilities in software should be managed carefully, with release to the public to be considered only after the software authors or other authorized parties have been informed. Ghost Map protects the software to be verified by only showing the player a compacted control flow graph of the software and by similarly limiting knowledge of the vulnerabilities in question.

Games like FoldIt [6] and Ghost Map draw players that want their game efforts to be applied toward the common good. Detailed information about the problem being solved by the game can provide additional player motivation. Ghost Map however cannot take full advantage of this additional motivation approach, due to the restrictions on the release of potential vulnerability information.

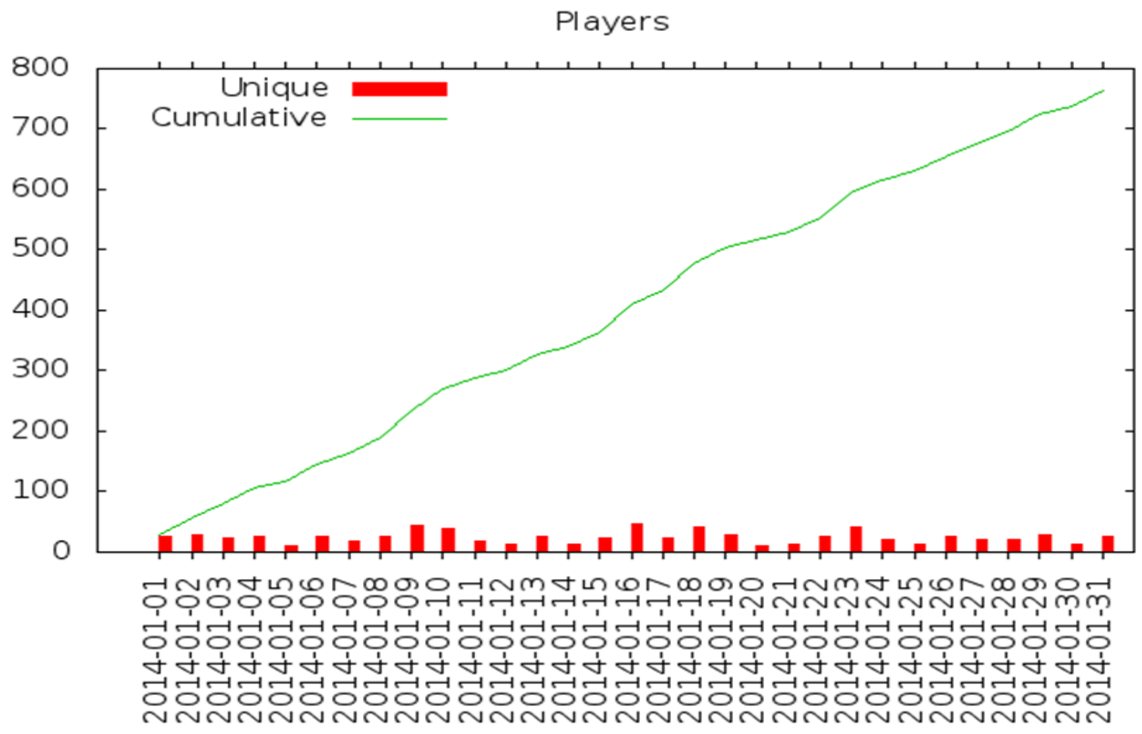
VII. FUTURE PLANS

Ghost Map is under active development, and at the time of writing we have just commenced our second phase of development. Our goal is to build upon the success of our initial version in six ways:

- Enhance the gameplay through the use of refinement guidance, which we refer to as "clues"
- Add new game play activities that provide additional fun for the player
- Develop a new space-travel narrative that provides a more engaging story than the current narrative and also provides a more comprehensive linkage to the puzzle problem
- Improve the accuracy and performance of our edge removal verification tool
- Extend the scope of the Ghost Map system to cover additional C language constructs
- Improve our approach to FSAs to create a more accurate representation of vulnerabilities

VIII. SUMMARY AND CONCLUSIONS

We have presented Ghost Map, a novel crowd-source game that allows non-experts to help prove software correctness from common security vulnerabilities. Ghost Map was released for open Internet play in December 2013. In the months since release, over a thousand users have played the game and similar numbers of small proofs have been completed (representative data from January 2014 is shown in Figure 8). 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 intelligent refinement algorithms is significant.



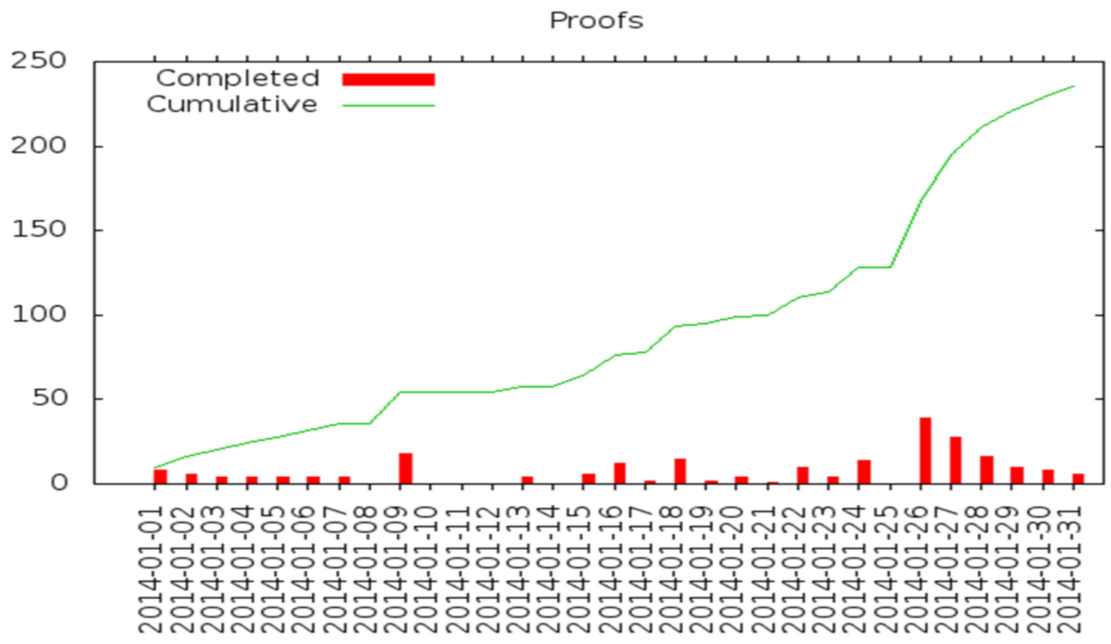


Figure 8. Ghost Map player and proof data from January 2014.

Acknowledgment

Many additional people beyond the named authors on this paper contributed to Ghost Map, including Bob Emerson, David Diller, David Mandelberg, Daniel McCarthy, John Orthofer, Paul Rubel, Michelle Spina and Ray Tomlinson at BBN, and additional individuals at the subcontractors (Breakaway Games, Carnegie Mellon University and the University of Central Florida). The DARPA leadership and staff associated with the Crowd Sourced Formal Verification (CSFV) Program were also very helpful. Dr. Drew Dean developed the initial CSFV concept at DARPA and Dr. Daniel Ragsdale is the current Program Manager. Mr. Carl Thomas at AFRL is the project funding agent.

This material is based on research sponsored by DARPA under contract number FA8750-12-C-0204. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] Y. Bertot and P. Castéran, Interactive Theorem Proving and Program Development: Coq Art: The Calculus of Inductive Constructions, Springer, 2004, XXV, 469 p., ISBN 3-540-20854-2

[2] S. Owre, J. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in Lecture Notes in Artificial Intelligence, Volume 607, 11th International Conference on Automated Deduction (CADE), D. Kapur, Editor, Springer-Verlag, Saratoga, NY, June, 1992, pp 748-752.

[3] E. M. Clarke Jr., Orna Grumberg, and Doron A. Peled, Model Checking, The MIT Press, 1999.

[4] R. Alur, "Model Checking: From Tools to Theory, 25 Years of Model Checking," in Springer Lecture Notes in Computer Science, Vol. 5000, 2008, pp 89-106.

[5] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," Proceedings of the 10th SPIN Workshop on Model Checking Software, May 2003, pp 235-239.

[6] S. Cooper, et al., "Predicting protein structures with a multiplayer online game," Nature, Vol, 466, No. 7307, August 2010, pp 756-760.

[7] W. Dietl, et al., "Verification Games: Making Verification Fun," Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, Beijing, China, June 2012, pp 42-49.

[8] W. Li, S. A. Seshia, and S. Jha, CrowdMine: Towards Crowdsourced Human-Assisted Verification, Technical Report No. UCB/EECS-2012-121, EECS Department, University of California, Berkeley, May 2012.

[9] Cancer Research UK, http://www.cancerresearchuk.org/-support-us/play-to-cure-genes-in-space, retrieved: Oct, 2014.

[10] Verigames, www.verigames.com, retrieved: Oct, 2014.

[11] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), Washington, DC, Nov. 2002, pp 235-244.

[12] The MITRE Corp., http://cwe.mitre.org/top25, retrieved: Oct, 2014.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," Journal of the ACM, Volume 50, Issue 5, Sept. 2003, pp 752-794.

[14] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient Algorithms for Model Checking Pushdown Systems," in Springer Lecture Notes in Computer Science, Vol. 1855, pp 232–247.

[15] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel Symbolic Execution for Automated Real-World Software Testing," ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2011), Salzburg, Austria, April, 2011, pp 183-197.