AFRL-RI-RS-TR-2016-065

PROOF BY GAMES

RAYTHEON BBN TECHNOLOGIES

MARCH 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency (DARPA) Public Release Center and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-065 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
CARL THOMAS
Work Unit Manager

/ S /
JOSEPH CAROLI
Acting Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2016	FINAL TECHNICAL REPORT	JUN 2012 – OCT 2015

4. TITLE AND SUBTITLE

PROOF BY GAMES

5a. CONTRACT NUMBER
FA8750-12-C-0204

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Kerry Moffitt, Michelle Spina

5d. PROJECT NUMBER
CSFV

5e. TASK NUMBER
RB

5f. WORK UNIT NUMBER
BN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Raytheon BBN Technologies
10 Moulton Street
Waltham, MA 02451

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

Defense Advanced Research Project Agency
675 North Randolph Street
525 Brooks Road, Rome NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-065

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 25768
Date Cleared: 8 FEB 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Online gaming is popular and it would be extremely valuable if a system could harness this intellectual effort for practical purposes. In this report, we discuss two crowd-sourced, on-line games, that present players with arcade-style puzzles to solve. The puzzles in Ghost Map and Ghost Map Hyperspace are generated from a formal analysis of the correctness of a software program. In our approach, a puzzle is generated for potential flaws in the software and the crowd produces formal proofs of the software's correctness by solving the puzzles. This report documents the challenges, lessons learned and efficiency of producing formal verification proofs of software through crowd sourced game play.

15. SUBJECT TERMS
Formal Program Verification, Crowd Source, Games, Software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**CARL R. THOMAS**
U	U	U	UU	120	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

Preface

The Raytheon BBN team that conducted the work described in this report brought a remarkably diverse set of backgrounds and skills to the project, from mathematics to psychology to game design to cloud-based software development, just to name a few – and this says nothing about the greater Program team of which the BBN team was but a part. The authors wish to thank the Crowd-Sourced Formal Verification Integration and Compiler Teams for their significant contributions to the overall effort; the other Game Teams for being our best and most helpful critics; and, naturally, our friends at DARPA and AFRL for making this work possible in the first place.

1. SUMMARY

Errors in computer software continue to cause serious problems, yet generally decision-makers do not view formal verification of the correctness of software as a cost-effective investment.

DARPA's Crowd Sourced Formal Verification Program (CSFV) included BBN's Proof by Games (PBG) research project – the subject of this report. The goal of the CSFV program was to address this cost effectiveness problem by transforming the formal software verification process into computer games, which when played would effectively contribute to the software verification proof process. Since crowd-sourced gameplay can be arranged at relatively modest cost, CSFV games would have the potential to reduce the cost of formal verification.

PBG integrates gameplay with a *model checking* approach to program verification: given source code and a descriptor of a security property, the system automatically generates an abstract model of the software and a set of potential violations of the property. Player success in the game proves that these potential violations are false alarms, ultimately yielding a correctness proof.

The PBG project developed and deployed the *Ghost Map* and *Ghost Map: Hyperspace* games in support of the CSFV vision. These games were deployed at www.verigames.com and played by thousands of users, producing correctness artifacts for software that expedited its verification.

PBG prime contractor Raytheon BBN Technologies worked with subcontractors Carnegie Mellon University (CMU), Breakaway Games (BAG), and University of Central Florida (UCF).

In the course of development, the PBG team completed the following:

- ***Ghost Map* Development and Deployment** (including math and server infrastructure)
- ***Ghost Map: Hyperspace* Development and Deployment** (including new infrastructure)
- **Amazon Mechanical Turk (AMT) Deployment** (de-gameified test of paid user approach)
- **Academic Papers** (three published, two in progress)
- **Formal Usability Studies** (four internal, three external)
- **"Guru" Events** – conferences to celebrate and study high-performing players (two)

The work led to the following results and conclusions as shown in Table 1.

Table 1. Results and Conclusions

Result	Conclusion and Possible Future Research
Thousands of users played on the public Internet.	Certain segments of the general public willingly engage in even technically daunting problems, for fun and science.
Gurus perceive and play very differently from non-gurus.	At least for PBG – and perhaps for deeply technical problems in general – a guru-based crowd-sourcing model is ideal.
Non-essential combat play elements boosted user engagement.	Lowering real-world task efficiency per player-minute can actually increase overall system output.
Users on Amazon Mechanical Turk (AMT) very cheaply solved all PBG AMT content in one day.	Amazon Mechanical Turk is a strong potential alternative to games for some crowd-sourcing challenges. Moreover, the "de-gameified" PBG may prove useful in its own right.
Constraints requiring lossiness in problem transformation proved a serious challenge.	Early analysis of problem transformation is essential, and a general taxonomy of problem transformations may be useful.
Graph manipulations developed for PBG can also be used in conventional model checking.	Development of techniques for crowd-sourcing may yield mechanisms useful outside their original scope.

2. INTRODUCTION

This section lays out a map of this document as a whole, and provides a high-level overview of the Proof by Games project.

2.1 Document Overview

This report describes the technical details and results of the PBG project. In addition, it includes three published conference papers ([1],[2], and [3]), two technical manuscripts ([4] and [5]), and the most recent version of the PBG human subject experiment protocol, all of which are attached as appendices.

The main body of the report consists of these elements:

- Section 1 **("Summary"**, above) provides a top-level executive summary of the entire report.
- Section 2 (**"Introduction"** – this section) introduces the document and the project, providing this document map and a high-level summary of the project motivation and approach.
- Section 3 (**"Methods, Assumptions, and Procedures"**) describes in technical detail the mathematical background behind Proof by Games, and the implementation details of the math, game, and server systems.
- Section 4 (**"Results and Discussion"**) conveys and discusses the results we saw from usability studies and other user interactions, as well as the results from deployments on Amazon Mechanical Turk and the public Internet.
- Section 5 (**"Conclusions"**) draws conclusions from the work, discussing details of, and issues with, the kind of problem transformation employed by the system, and examining what we learned in the course of development.
- Finally, Section 6 (**"Recommendations"**) provides a collection of thoughts on where to go from here, and what might still be learned from related future work.

This is followed by lists of **acronyms** and **references** in the report, and then a set of **appendices** that includes three conference papers, two technical manuscripts, and the BBN IRB protocol.

2.2 Project Overview

Here we provide a high-level summary of the Proof by Games project and the Crowd Sourced Formal Verification (CSFV) program of which it was a part. We cover the 'Why' of the motivation, the 'How' of our basic approach and architecture, and the 'When' with a timeline of significant events that occurred throughout the course of development. Deeper coverage of the mathematical background underlying Proof by Games (PBG) and the details of the Proof by Games implementation appears in Section 3.

2.2.1 Motivation and Goals.

Historically, formal verification of the correctness of software has been viewed as a technology that is not cost effective; it may take many man-years of expensive verification expert time to prove the correctness of a common program.

Growing concerns over cyber security and the reliability of software applications have motivated stakeholders to become interested in finding vulnerabilities in software in a scalable way using automation, allowing those vulnerabilities to be eliminated at program design and

implementation time without such heavy reliance on human experts. There exist a large variety of automated tools and techniques for identifying potential vulnerabilities in software applications for which the original source code is available. However, these techniques often represent trade-offs due to the inherent difficulty of general-purpose program analysis: a specification of correct or incorrect behavior must be supplied, and this specification must be compared to the application itself in a necessarily incomplete or abstract manner due to fundamental limitations identified by the study of the theory of computation (such as the undecidability of the halting problem).

The Proof by Games approach involves integrating game play with a *model checking* approach to formal verification (See Sections 2.2.2 and 3.1.2) to create a hybrid system where gameplay results direct the model checking processes down the most promising search paths. The CSFV/PBG vision is that as players succeed at winning game levels, they generate artifacts that support the completion of correctness proofs for the software. Since crowd-sourced game play can be arranged at modest cost, CSFV/PBG games have the potential to significantly reduce the cost of formal verification.

2.2.2 Basic Approach and High-level Architecture.

In general, model checking involves the creation of a model of the software to be analyzed, and the encoding of specific properties that the software must exhibit. The system checks the model for, and ultimately seeks to ensure the absence of, *violations* of the property under consideration. This approach helps manage the vast complexity of modern software systems by transforming software into a simplified, *abstracted* form that represents the original software just well enough to perform suitable proof operations in a sound way, such that they still provably apply to the un-abstracted software.

The creation and use of such an abstract model brings the significant advantage that the check for violations becomes manageable – indeed, it can typically happen in a fully automated way. The problem with this sort of abstraction is that while the model is sound (meaning that an absence of violations in the model proves an absence of violations in the original code), the high level of abstraction typically implies that the system may discover potential violations in the model that do not represent real violations in the original code. We call these *false alarms*.

One approach to dispatching false alarms is termed *Counterexample-Guided Abstraction Refinement* (CEGAR). This method involves using potential violations reported from the model checker to guide *refinements* to the abstract model of the software. The system attempts to refine the model (i.e. un-abstract it, to bring it closer to the full level of detail that the original software provides) to the point where the violation (counterexample) no longer arises. The newly refined model is then re-checked against the property under consideration – if any violations remain, the process continues to iterate, whereas if no violations remain on the refined model, the proof is complete.

Several research efforts [6], [7] have investigated the possibility of implementing a CEGAR approach in a fully automated way. Although they show some promise, the challenges of managing the search for refinements and the complexity of suitably refined models remains daunting.

PBG sought to leverage the crowd to address these challenges. Figure 1 illustrates the essential process flow in PBG. "C" code and software properties encoded as FSAs (finite state automata) enter from the left. The encoded properties may be derived, e.g., from entries in the SANS/MITRE Common Weakness Enumeration (CWE) Top 25 list. If no violations of the properties are found even on the initial abstract model, the proof is instantly complete. More typically, though, some potential violations are reported, and one game level is generated per violation. Each of these levels includes a *control flow graph* (CFG) representing the structure of the program in its abstract form and the *execution trace* through that graph that represents the way of generating the violation. In attempting to solve the game level, the player performs game moves which actually map to selecting a region on that trace that is large enough to include sufficient information that the *solver* can prove its logical impossibility in the complete source code, yet small enough that the solver is not overwhelmed by complexity. If the player succeeds in this step – i.e. if the solver can prove that no solution is possible that would allow the violation to arise in the real-world software – the refined model is sent back to the violation generator and the next iteration of the process begins. If the player can bring the level to the point where no more potential violations are found, they have achieved victory in that level – and the system has generated a proof that the potential violation represented by that level is in fact a false alarm.

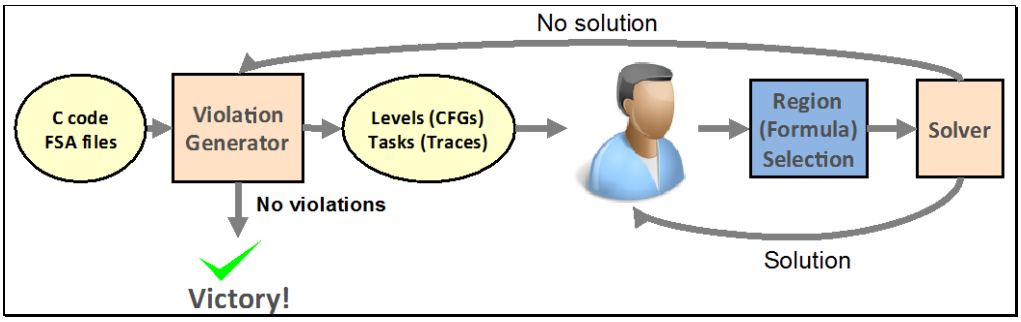


Figure 1. High-Level Process Flow in Proof by Games

2.2.3 Project Timeline.

Here we present a timeline of major events from the course of Proof by Games development. The project spanned two distinct phases, each of which involved the design, implementation, and deployment of a single game: Ghost Map in Phase One, and Ghost Map: Hyperspace in Phase Two.

2012 July 10-12 Program Kickoff, Monterey, CA

2013 Feb 5-7 PI Meeting, San Antonio, TX

2013 May 24 internal UCF Usability Report

2013 June 26 internal UCF Usability Report

2013 July 23-25 PI Meeting, Stevenson, WA

2013 September 24 I2O Demo Day

2013 November 15 Phase One Public Launch

2013 December 3-5 PI Meeting, San Antonio, TX (PM change: Drew Dean to Dan Ragsdale)

2014 May 21 DARPA Demo Day
2014 July 8-10 PI Meeting, Monterey, CA (including GameDocs usability testing)
2014 November 12 YouEye Usability Report
2014 December 16-17 PI Meeting, Orlando, FL (including GameDocs usability testing)

2015 March 6 PM change: Dan Ragsdale to Michael Hsieh
2015 March 16 UCF Usability Report
2015 May 9 Phase Two Public Launch
2015 May 15 YouEye Usability Report
2015 July 14 Mechanical Turk Public Launch
2015 July 31 UCF Usability Report
2015 August 10 PI Meeting, Washington, DC
2015 August 20 Guru Event, Menlo Park, CA
2015 September 24 Guru Event, Cambridge, MA

3. METHODS, ASSUMPTIONS, AND PROCEDURES

In this section, we describe the details of the math, game, and server implementations.

3.1 Background, Definitions and Theory

We begin with a review of some of the mathematics underlying the system as a whole.

3.1.1 Basic Concepts and Background.

In any programming language, a given program can be represented statically as an *abstract syntax tree* (AST). The AST is the data structure produced by the parser for that programming language, and it is the internal representation of a program used by interpreters and compilers for that language.

There is another important representation of a program that is derivable from the AST. A program in an imperative language such as C can be viewed as a collection of statements in which each statement can transition to one or more other statements during the execution of the program (according to transition rules determined by the semantics of the language). For a given program, the statements can be viewed as nodes in a directed graph in which an edge between two nodes *a* and *b* represents the possibility that control can transition from node *a* to node *b* during program execution. This graph is called *the control flow graph* (CFG) for a program. While programs in procedural languages such as C may actually consist of a collection of separate, modular CFGs (e.g., files, functions, and so on), for the purposes of analyzing an individual component or collection of components for vulnerabilities, it is reasonable to make the simplifying assumption that at any moment only a single CFG is being analyzed. Finally, note that any node in a CFG necessarily corresponds to a node in the AST for that program.

Assuming that a CFG consists of a directed graph with a distinguished root node, it can be converted into a corresponding *abstract reachability tree* (ART): the tree of all paths that originate at the root of the CFG and follows directed paths along the edges of the CFG. If a CFG has cycles, the corresponding ART will necessarily be infinitely large.

A path through the CFG (or the corresponding path through the ART) represents a potential execution of a program. We call such a path an *execution trace* or simply a *trace*. We call any subpath along a given trace a *trace region*.

Individual patterns of events (particular instructions or procedure calls) that can occur on paths within the CFG can be modeled using FSAs. An execution trace in which a sequence of events occurs that corresponds to an FSA representing a vulnerability reaching an accepting state (possibly with other events taking place between the events in the matching sequence) is called a *violation trace* or *violation*.

3.1.2 Proof by Games Approach.

The Proof by Games approach to improving the usefulness of automated vulnerability detection tools for software (such as MOdelchecking Programs for Security properties, also known as MOPS) is to provide an infrastructure for eliminating in a scalable way detected violations that happen to be false alarms (the original MOPS approach relied on manual software engineer inspection to eliminate false alarms). In the particular case of MOPS, violation traces are

generated based on FSAs that describe sequences of events that may represent a vulnerability. For example, in Figure 2 we illustrate CWE-250: Execution with Unnecessary Privileges, a violation on UNIX-like systems that involves performing an operation at a privilege level that his higher than the minimum level required for that operation. This can be represented using a pair of FSAs (the state space of the overall combined FSA is the Cartesian product of the state spaces of the two component FSAs). A violation state occurs when the first FSA (representing the level of user privilege) is in the state `priv` and the second FSA (representing whether a system call to `execl()` has been made) is in the state `exec`.

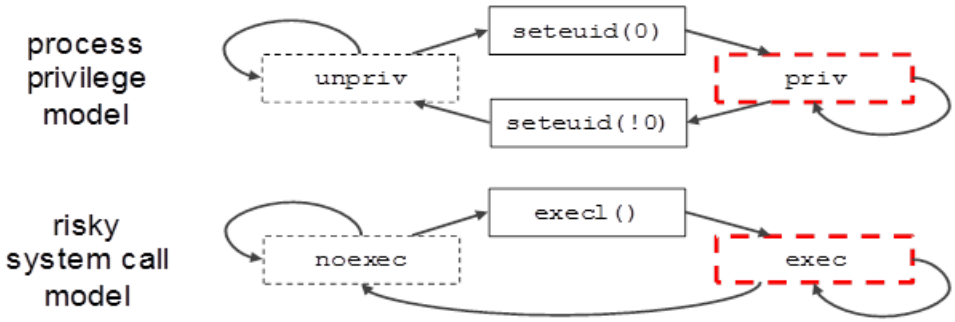


Figure 2. FSAs for CWE-250, a violation on UNIX-like systems.

For each security property represented as an FSA, MOPS outputs a set of violation traces, each of which represents the (or a) shortest from the set of execution traces that all share similar properties: they all end at the same node in the CFG, and they all end with the same transition to error state in the FSA. MOPS is control flow sensitive but data flow insensitive: any data, variable, and value information that may make certain paths through the CFG *unrealizable* (i.e., impossible) is ignored, which is the reason that in practice many traces are false alarms.

3.1.2.1 Converting Source Code to a CFG and Matching an FSA to a CFG

The concrete syntax of a program in an imperative language such as C (such as the program presented in Figure 3) can be parsed into an AST (illustrated in Figure 4), and this abstract syntax tree can then be converted into a CFG (illustrated in Figure 5). The initial transformation to an AST can be accomplished with a standard parser for the source language, and tools such as MOPS produce both the AST and CFG as part of their normal operation.

```
c = 0;
while (c < 1) {
  if (c > 1) {
    turn_on_microwave();
  } else {
    put_fork_in_microwave();
  }
  c = c + 1;
}
```

Figure 3. Source code of a simple program in the C programming language.

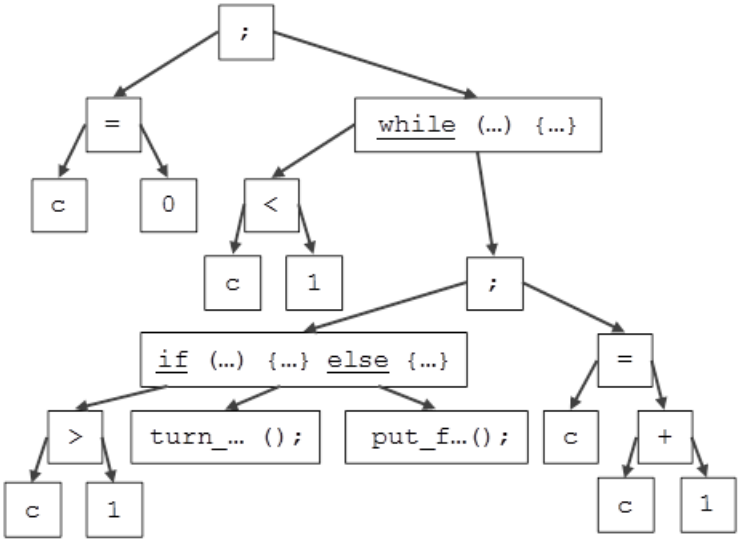


Figure 4. Example of an abstract syntax tree (AST) for the program in Figure 3.

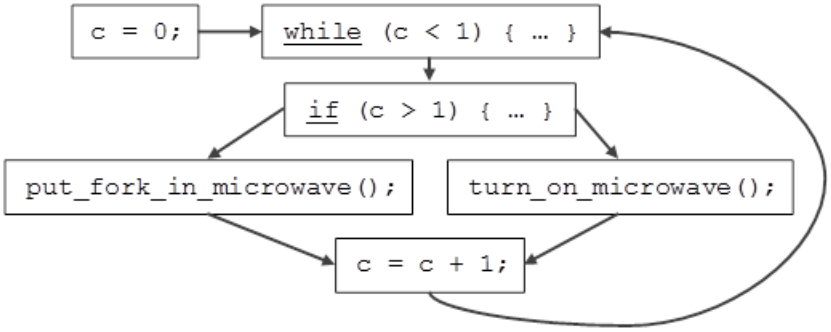


Figure 5. Example of a control flow graph (CFG) for the program in Figure 3.

Tools like MOPS can locate potential matches within the CFG to an FSA representing a vulnerability by performing state space exploration of the FSA transitions in the CFG and determining whether terminal states are ever reached using efficient algorithms. These algorithms scale to CFGs obtained from large programs that have millions of lines of code, and are guaranteed to detect all potential violations in such CFGs. Figure 6 illustrates one possible violation trace within the CFG in Figure 5 that matches a simple FSA describing an undesirable behavior for a program.

3.1.2.2 Eliminating False Alarms

If a violation trace is a false alarm, it may be possible to identify it as such by analyzing the particular data, variable, and value information for that trace and showing that the path is logically unrealizable. This can be done by examining all the AST nodes for the program that correspond to the statements represented by nodes in the CFG that fall on the trace; any expressions that occur in these AST nodes that govern variables or data (e.g., branching and loop conditions, variable assignments, and so on) can be assembled into a logical formula that governs the relationships between the variables that occur on that particular trace region. This formula must be solvable in order for that particular trace to be possible. If the logical formula happens to

be provably unsolvable (i.e., logically false), then that particular trace is unrealizable. If it is solvable, then that violation trace is not a false alarm.

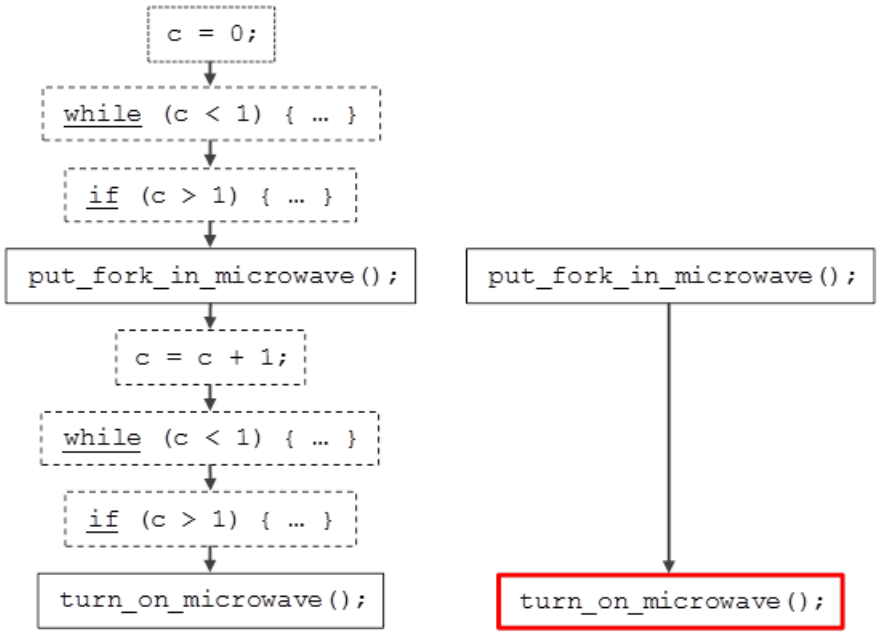


Figure 6. Trace (left) matching an FSA describing an undesirable behavior (right).

For example, given the trace in Figure 6 through the CFG in Figure 5, we can derive the following logical formula by traversing the expression nodes within the AST in Figure 4:

$$c = 0 \ \wedge \ c < 1 \ \wedge \ c \leq 1 \ \wedge \ c' = c + 1 \ \wedge \ c' < 1 \ \wedge \ c' > 1$$

We can observe that the above formula must be logically false because the integer variable *c'* cannot be both greater and less than one simultaneously. Thus, the particular trace through the CFG cannot be realizable as an actual execution through the program corresponding to the CFG.

Determining whether a logical formula is provably false is a difficult task in general. Fortunately, satisfiability modulo theories (SMT) solvers exist that can automatically answer this question for a large class of logical formulas, though the time it takes to do so for any particular formula can be significant. The time required can also depend on the properties of the formula being considered: the size of the formula, the complexity of the subformulas within it, the particular operations that occur in it, and so on. By choosing particular trace regions intelligently, it may be possible to speed this process up.

Making intelligent choices about what trace regions to consider is a problem that is potentially amenable to crowdsourcing via a game in which eliminating false alarms is the goal (potentially obfuscated from the user). The PBG infrastructure uses the concept of violation traces to delineate and define levels for human players in an interactive game: each game level is built around a single violation trace, and completing that level requires proving that the particular trace is unrealizable.

It must be possible to eliminate some traces while still retaining others that go through the same CFG. One way to keep track of traces that have been eliminated and those that remain is to expand the CFG into an ART.

In this way, once one trace is eliminated, the other remaining traces can still exist as reachable paths within the ART. Figure 7 illustrates an ART derived from the CFG in Figure 5. Notice that the trace in Figure 6 is a path within this ART that starts at the root.

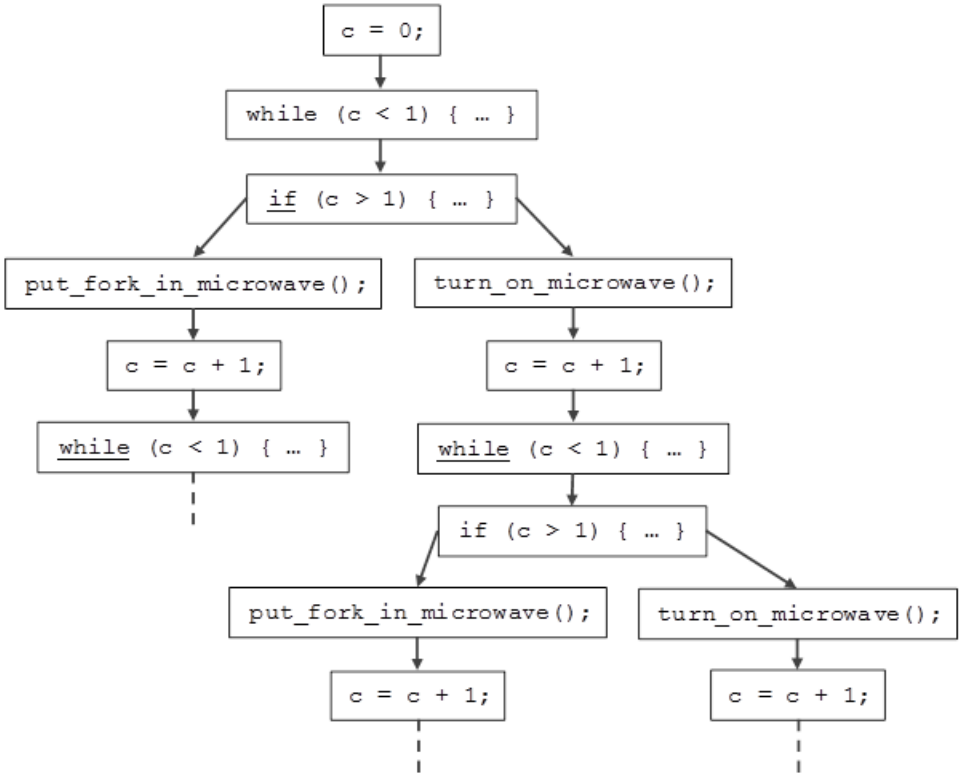


Figure 7. Portion of the abstract reachability tree (ART) for the CFG in Figure 4.

For non-trivial programs that contain loops, the ART is necessarily infinite in size. However, it is possible to partially expand a CFG so that it approaches the ART for all paths of some finite, bounded size. Sometimes it is effective to expand trace segments within the loop for just a few iterations to find a contradiction without expanding the entrance to the loop (this is discussed at greater length in the future directions section). This process can potentially be expensive, and it may be difficult to determine which portions of a CFG are helpful to expand.

3.1.2.3 Expansion of CFG into ART, and Trace Elimination

Human input can provide insightful guidance that may exceed the brute force search abilities of computers. The example in Figure 8 presents a small C function involving locks and unlocks (commonly used as a model checking example program) and an FSA that codifies the software flaw of calling lock twice without an intervening unlock (and vice versa). The program is actually correct as written, but when building the CFG for this function and analyzing it against the FSA presented, violation traces can be found (as shown in Figure 9). One approach to proving that the software in fact does not have this particular vulnerability is to convert the CFG

into an equivalent graph that has no violations. There are two operations on the CFG that can be used to achieve this eventual goal: (1) *cleaving* and (2) *edge removal*.

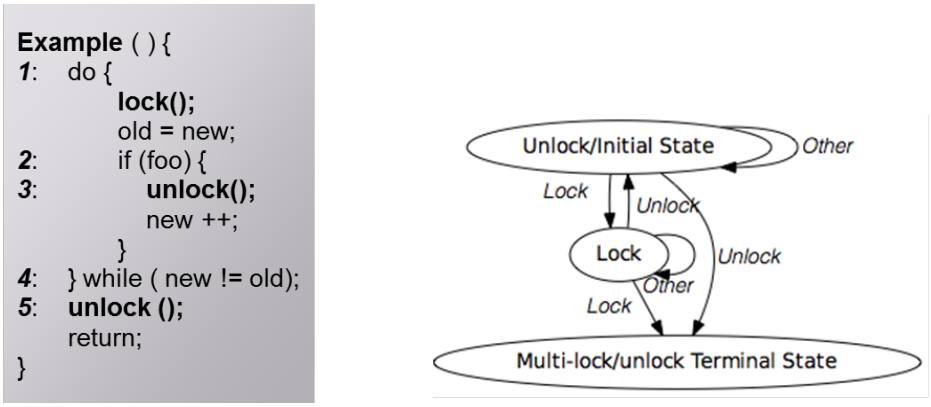


Figure 8. Example of a C program and FSA for multiple lock/unlock software flaw.

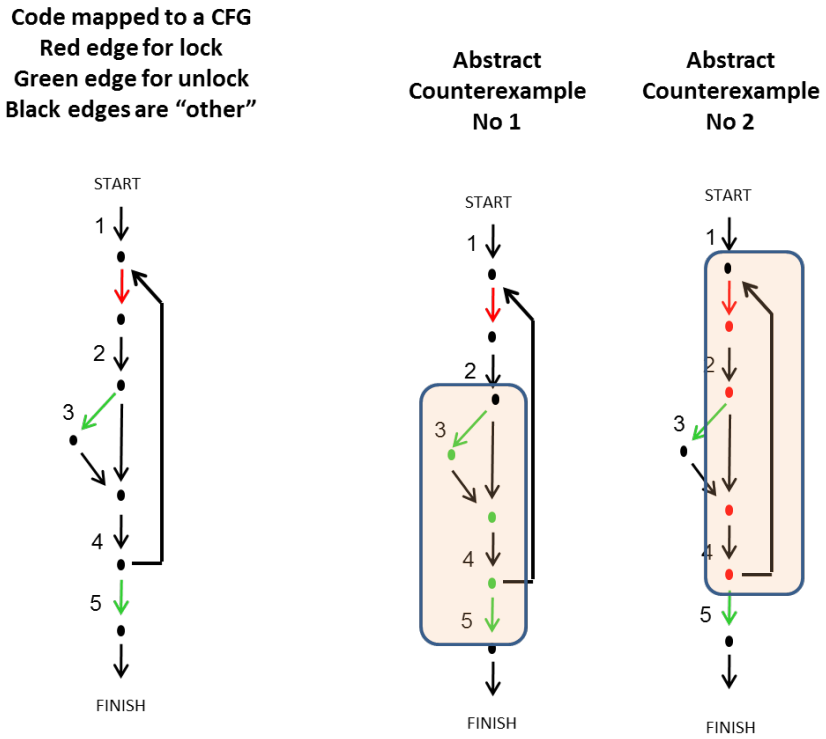


Figure 9. Violation traces in the example program in Figure 7.

Cleaving takes a node of in-degree at least 2 and splits it into 2 or more nodes. The in-bound edges into the original node are allocated to one of the new nodes and the outbound edges are duplicated for each of the new nodes. In terms of control flow, cleaving simply expands the CFG 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 10 illustrates a cleave operation.

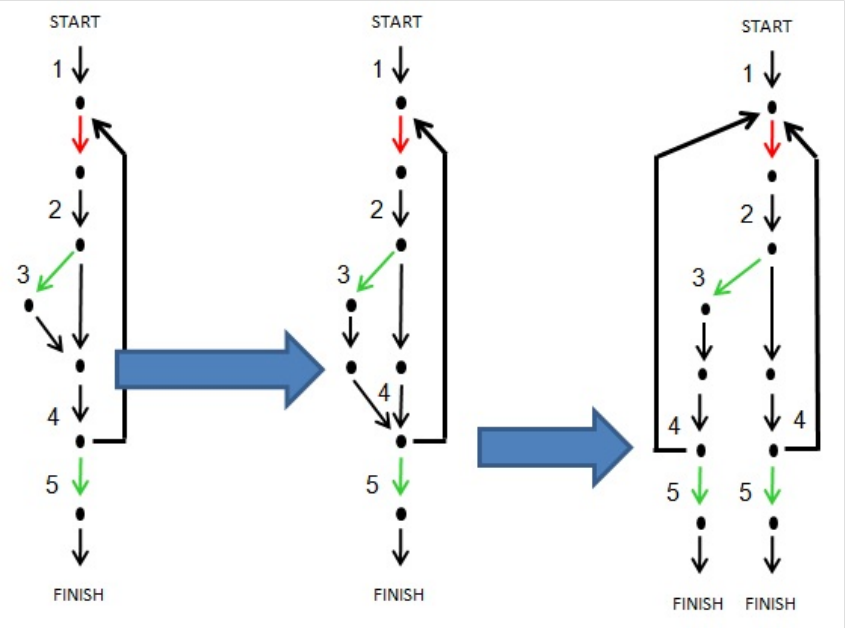


Figure 10. Example of a cleaving operation.

There are two cleave steps involved for this example. We cleave first at the node before edge 4. Then, we cleave next at the node after edge 4, as it becomes a node satisfying the two constraints mentioned above. Note now that one can't cleave any more on this path. In a sense, such cleaves perform an “unzip” function on the CFG.

How does a human know where to cleave? While this problem can be addressed in general by supplying other kinds of information (discussed in Section 3.3.3.4), in this simple example the initial cleave point might be suggested by the fact that it is the only cleavable point in both of the traces.

Removing an edge in the ART which is logically unreachable proves a violation trace traversing that edge is impossible. For example, the left hand edge 5 in the cleaved graph in Figure 10 is a candidate for removal. Any time this left hand edge 5 is reached in the cleaved graph, an FSA match will be created. Similarly, the right hand loop back is also the immediate cause of an FSA match.

A human might propose that these edges should be removed, and the PBG infrastructure can verify that removal is legal by using the data constraints in the software to build a logical formula that is then submitted to an SMT solver. In the example case, the predicate (`old = new`) is the key piece of information that helps prove that the targeted edges are indeed never reachable by an actual execution of the function. The final graph is shown in Figure 11.

One can view the final graph above 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. The fact that this final graph represents all the execution traces of the software may be useful to a compiler for other purposes as well (perhaps data storage issues).

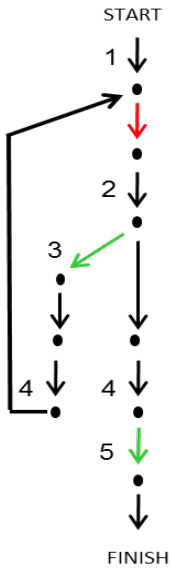


Figure 11. Final CFG showing no violation traces.

3.1.2.4 Handling Special Loop Expansion Cases Using Cleaving

The presence of infinite loops in a program presents a particular challenge to any automated analysis whose purpose is to eliminate violation traces. It may be possible to use CFG manipulations based on cleaving operations that maintain certain graph properties; these techniques could be applied automatically, or augmented with human input. Investigating the potential usefulness of these techniques is the subject of potential future work.

3.1.3 Mathematical Formulation.

In this section we provide a more precise mathematical formalism for the ideas introduced in the more informal presentation found in Section 3.1.1. The main advantages of developing a formal definition of these concepts and algorithms is two-fold: (1) a mathematical definition provides an implementation-independent specification of the correct behavior for an application that accomplishes these tasks, and (2) it enables the use of formal reasoning to prove statements about the properties of the algorithms involved in the application. More generally, a formal treatment may lead to new insights that are obfuscated within an implementation.

3.1.3.1 Graph Homomorphisms and Simulation

Let $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ be graphs, where a graph consists of a set V of vertices and a set E of edges between those vertices. We write $\rightarrow_i$ for the edges of graph G_i, so that $v \rightarrow_i u$ means that there is an edge from vertex v to vertex u in G_i.

A ***graph homomorphism*** is a map H: $V_0 \rightarrow V_1$ such that

$$v_0 \rightarrow_0 u_0 \text{ implies } H(v_0) \rightarrow_1 H(u_0).$$

H doesn't have to be injective; it can agglomerate vertices of G_0 by sending them to the same target in G_1. If edges in G_0 and G_1 also have labels l drawn form a set L, then we may also require, for all l in L, that:

$$v_0 \rightarrow_{l,0} u_0 \text{ implies } H(v_0) \rightarrow_{l,1} H(u_0). \quad (1)$$

In this case, we say that H is an L-preserving homomorphism.

Suppose that we are given graphs with some set of vertices distinguished as the ***initial states***. If Equation 1 holds at least for all v accessible from initial states, then we say that G_1 ***can simulate*** G_0 modulo H. This matches the normal notion of simulation, meaning that every execution of G_0 has as its image some execution in G_1.

3.1.4 Programs, Their Control Flow Graphs, and Their Phase Spaces.

Suppose that P is a program. We keep P fixed throughout. Let $G_C = (V_C, E_C)$ be the control-flow graph for P. Each node in V_C represents a program counter location in the code of P, and each edge in E_C represents a transfer of control, either by a statement, or as the effect of a control operator. For the sake of connecting later with security goal FSAs, we assume that some edges e in E_C may have labels taken from a set L. We assume that L has a silent label, and we write $u \rightarrow v$ to mean $u \rightarrow_\varepsilon v$ where ε is such a label. In MOPS, the labels L are certain significant statements for the purposes of vulnerability analysis, often system calls or calls to library procedures.

We will also let S be the set of (data) states of P. By this we mean the ***stack*** of arguments and local variables; the ***store*** that associates values with the global variables; and the ***heap***. We can also regard the list of not-yet-consumed characters of future input as a component of the states. Some states are ***initial***, for instance in the C language when the store contains ***argc*** and a pointer ***argv*** into the heap, where an array of strings is to be found.

We define:

$V_P = V \times S$, which represents the program's phase space;

$E_P = \{(v, s) \rightarrow (u, t) : v \rightarrow u \text{ in } E_C \text{ and the instruction in the PC at } v \text{ transforms } s \text{ to } t\}$

$G_P = (V_P, E_P)$

We regard edge $(v, s) \rightarrow (u, t)$ as having a label l if edge $v \rightarrow u$ has label l in E_C

G_P represents the ***trajectories through the phase state space*** of P. G_P is a monstrously big infinite graph.

A path through V_P following edges in E_P is an ***execution*** if it starts with (v_0, s_0) where v_0 in V_C is an initial program counter location and s_0 is an initial state.

3.1.4.1 Representation of MOPS

MOPS analyzes programs P for correctness with respect to an FSA. We assume here that the labels in the FSA come from a set L. As its initial processing step, MOPS computes a compacted graph G_{mops} that eliminates vertices and edges that are not relevant for L. From the compaction pseudo code in [10], one can see that there is homomorphism H_{mops}: $G_P \rightarrow G_{mops}$. (Actually, to make this correct, one needs to view G_{mops} as containing edges with modified labels) Apparently, for correctness, the only essential condition on MOPS is that H_{mops} should be an L-preserving homomorphism.

Naturally, MOPS endeavors to compact G_P as much as possible; this means that for a large set of L-preserving homomorphisms H: $G_P \rightarrow G_i$, H_{mops} should do at least as much as H. Here "do at least as much" means destroy at least as many differences. Thus, it means that H_{mops} should ***factor through*** H, i.e., for some J, $H_{mops} = J * H$.

In particular, H_{mops} identifies any two node (v, s_0) and (v, s_1) in V_P which differ only in their state component.

3.1.4.2 Refinement

Refinement means the converse of ***factors through***. To refine another homomorphism means to identify fewer G_P nodes than the latter. In particular, a refinement of H: $G_0 \rightarrow G_2$ means a graph G_1 together with a pair of homomorphisms J and K such that

$$J: G_1 \rightarrow G_2 \text{ , } K: G_0 \rightarrow G_1 \text{ , and } H = J * K$$

Now we can understand cleaving and edge removal as L-preserving refinements. In particular, they refine H_{mops}: $G_P \rightarrow G_{mops}$.

To cleave, one simply introduces a sibling v' for a given node v in G_{mops} with in-degree >1. One partitions the in-arrows u → v among v and v', and one duplicates the out-arrows $v \rightarrow u$ as arrows $v' \rightarrow u$.

To see that this is a refinement, consider that if the pre-image $H^{-1}(v)$ has cardinality > 1, then one can partition it, sending some nodes to v and the others to v'. The collapsing map that sends v' to v shows that this is a refinement. If $H^{-1}(v)$ has cardinality 1, then we let v' be outside range(K).

As for edge elimination, when there is no execution P of G_P such that $H(p)$ traverses an edge $v \rightarrow u$ or $v' \rightarrow u$, then we can eliminate this edge. That is, the graphs with and without this edge are bisimilar under the identity map.

3.1.4.3 Abstract Interpretation

Abstract interpretation is a topic that may be useful to future projects, to help create and represent new forms of clues for game play. We include here a very short summary of this topic. Abstract interpretation can be developed in much greater depth, as for example by Cousot and Cousot [11]. For our current purposes, the much more condensed approach below suffices.

We regard abstract interpretation as leaving the control-flow graph unchanged, and operating only on the data states S. Thus, let H_C: $G_C \rightarrow G'$ be some compaction of the control-flow graph, with vertices V'. We are interested in abstract interpretations that act compatibly on the data state space S, mapping it to some (reduced) state space S'.

Thus, an ***abstract interpretation*** is a map A: $S \rightarrow S'$ that acts on the state components. We regard $[H_C, A]$ as mapping to a graph G_T with vertices $V_T = V' \times S'$. We equip G_T with the set of edges $E_T = \{ ((v', s'), (u', t')) : (v, s) \rightarrow_P (u, t\}$ for some

$$v \text{ in } H_C^{-1}(v'),\ u \text{ in } H_C^{-1}(u'),\ s \text{ in } A^{-1}(s'), \text{ and } t \text{ in } A^{-1}(t') \}.$$

Suppose that we want to use an abstract interpretation to verify some property φ of executions in G_P. We regard φ as a property of sequences of nodes v_0, v_1, … where v_i is in V_P (note that we treat φ as the set of all sequences that satisfy this property). An abstract interpretation is too coarse if it allows executions in G_T whose inverse image does not respect φ. The inverse image of a sequence of G_T vertices $[H_C, A]^{-1}(u_0,u_1,\ldots)$ is the set of all sequences $(v_0,v_1,\ldots)$ such that $[H_C, A](v_i) = u_i$.

Thus, $[H_C,A]$ ***respects*** φ if, for every p in Executions(G_T),

$$[H_C, A]^{-1}(p) < \varphi.$$

Since $[H_C, A]$ (Executions(G_P)) $<$ Executions(G_T), for a useful abstraction $[H_C, A]$ we have

$$[H_C, A]^{-1}([H_C, A] (\text{Executions}\})(G_P))) < \varphi$$

That is, the abstraction/concretization round trip must stay within φ.

3.2 High-Level System Architecture

The following three sections provide the actual implementation details of the PBG system. In this section we introduce all the basic components of the system and briefly discuss their interactions, in order to provide the reader with an architectural roadmap.

As Figure 12 illustrates, PBG implements a client/server architecture: game clients run on players' devices, and the game server routes client requests to – and delivers results from – the math systems, which themselves run on one or more machines (private or cloud-based) depending on availability and load. Figure 12 shows pre-processing on the left, and real-time gameplay on the right. Note that nearly half of the components span the divide, since they support both preprocessing and real-time gameplay.

Starting from the top-left: a web-based **Admin GUI** allows for external visibility into and control of the system, and among other things allows for the uploading of **Source Code** and **Security Properties** (encoded as finite state automata files) into the system. These are routed to an **Input Processor** that runs preprocessing on the raw input and then routes it to the **Model Checking and Verification Controller** which sends it through the **Model Checker**, in order to obtain the initial abstract model of the software per property, any violations of that property, and the clues that will inform player actions. The **Game Level Generator** then uses all this to construct game levels, which are persisted in **Level Storage**, for gameplay. In Phase One, the Game Level Generator also provided level metadata describing each level to the **Resource Allocator** – a mechanism designed to match players (with their varying skill sets) to suitable levels – provided by co-contractor Charles River Associates.

During gameplay sessions, each player runs an instance of the **Game Client** (which, in Phase One, could also be controlled by a scripted **Robot**, in order to explore automated strategies). The Game Client receives level data from the **Game Server** and implements the game as the player sees it. In the course of gameplay the player submits trace region selection moves, and the Game Server conveys these moves to the Model Checking and Verification Controller system. The latter refines the model to the point where the move can be submitted to the **Constraint Solver**. If the Solver determines that the move is valid, the refined model is re-run through the Model

Checker – if any violations remain, another is provided to the player and the level continues; if not, the level has been solved, the player is rewarded, and the result is stored in the **Game Results** database.

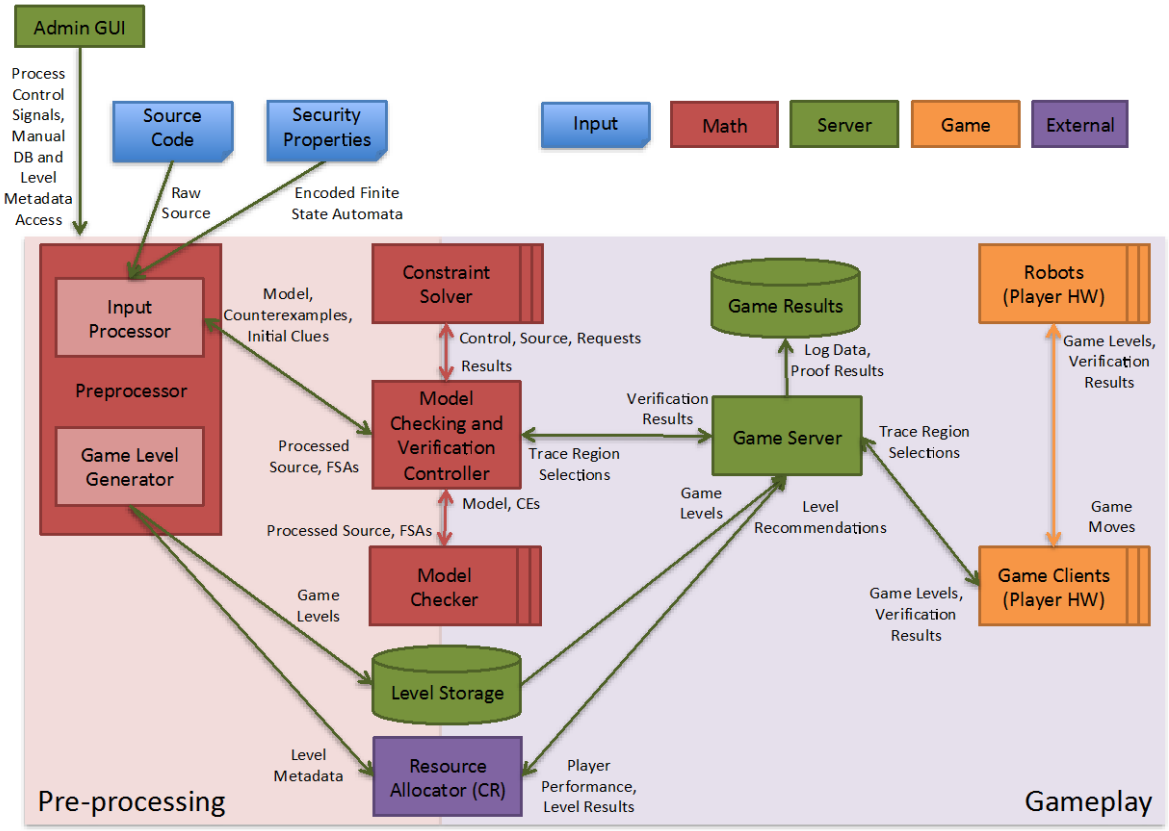


Figure 12. Proof by Games High-Level Architecture

In the following three sections we delve into the implementation details of the math, game client, and game server systems, respectively.

3.3 Math System

Here we describe the architectural and implementation details of the PBG math system.

3.3.1 Architectural Overview of Math System.

A high level view of the math system and its interactions with other system components is shown in Figure 13. The two major components of the math system are shown in blue. The first is the game and level generation component and the second is the runtime component.

The math functionality runs within one or more dedicated virtual machines (VMs) under the control of the game server, shown in pink, which is responsible for storing and restoring the state of the math VMs as well as passing player selections to the math system and math results back to the game client.

The game and level generation pre-processing operates on two types of input:

- C language source code to be examined for security property violations
- An FSA file containing a state machine specifying the security property to be checked in the source code

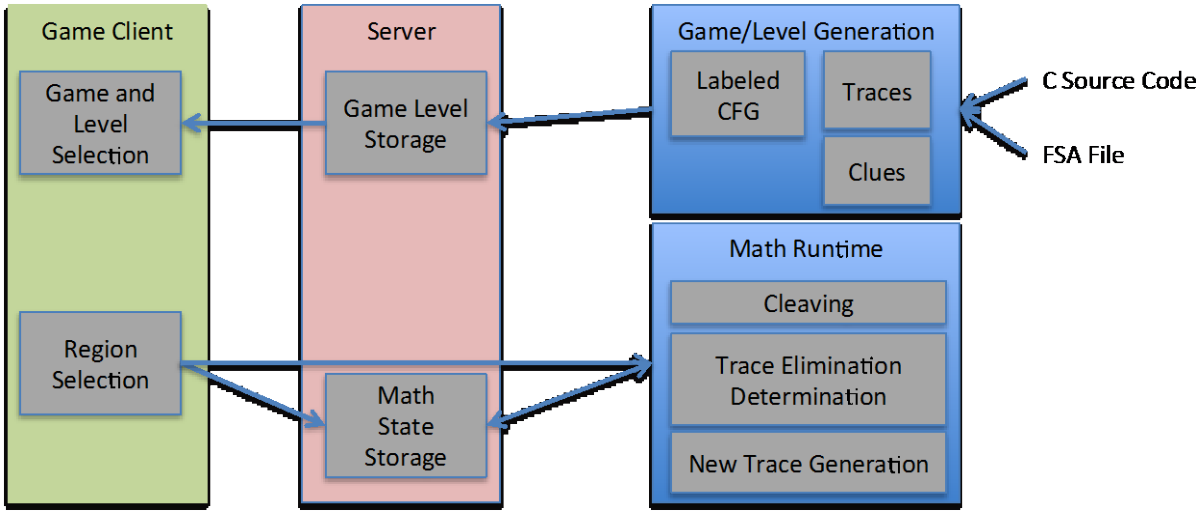


Figure 13. Math System Architecture

Three products are generated which constitute a "game" as shown in the "Game/Level Generation" rectangle in Figure 13. They are:

- *Labeled CFG* – The control flow graph is produced from the input source code. In its most expanded form, it includes nodes for all executable statements and edges corresponding to flow of control including branching. In the PBG context, we refer to this as the *complete CFG*. Since the size and complexity of the complete CFG is typically too large and complex for a player, a compacted CFG is prepared and passed to the game system. Further, a labeling process is performed on the CFG where any executed statement that would cause a transition in the FSA is marked with the associated transition.
- *List of Traces* – A list of traces is produced. Each trace is a shortest path through the code that causes the security violation FSA to be driven to an accepting state. Each trace constitutes a level in the game. The task performed by the player's moves amount to proving that these traces are unrealizable. This is done by finding regions of code in the trace that are unrealizable for any set of values of the variables.
- *Clue Data* – Ultimately, the task of the player is to find regions of the trace through the CFG for which there is no set of variable values that would allow execution of that sequence of code statements. In order to recognize that region, the user is given information about the declaration, modification and value testing of variables. This information is coded in a pre-calculated dependency graph, which is used by the game logic to populate the game interface with various "clues".

The math runtime module is shown in blue in the lower right hand corner of Figure 13. This module interacts with the player's real-time gameplay choices. Its major goals involve modifying the CFG depending on the region chosen by the player through cleaving operations, determining if a trace can be eliminated based on analysis of variables used within the chosen region, and generation of new traces in the modified CFG.

3.3.2 Interaction of Player with Math Processing.

The sequence of player actions and math responses proceeds as follows, and is illustrated in Figure 14:

1. The player is presented with information for a single trace, including clue information.
2. The player chooses a region within that trace as shown in the "Player" column on the upper left of Figure 14.
3. The player's choice is transmitted to the math system.
4. In the math system, shown in the right-hand column in Figure 14, the CFG is altered so that all branch points within the selected region are cleaved producing a modified CFG (this is the "region is cleaved" rectangle in Figure 14)
5. Now, the math system derives a logical formula from the region selected by the player. This formula is evaluated for satisfiability.
6. If the formula in (5) is realizable, a failure message is sent to the player who must choose a new region on the same trace, returning to (2).
7. If the formula in (5) is not realizable, the last edge produced by a cleave in the region is removed producing a modified CFG.
8. The new CFG is now processed and analyzed to see if a new trace (not seen before) occurs.
9. If no new trace is found in (8), the program is free from the defect specified by the FSA and a message that the player wins is sent to the game side.
10. If a new trace is found in (8), the player must now choose a region to eliminate it and returns to (2).

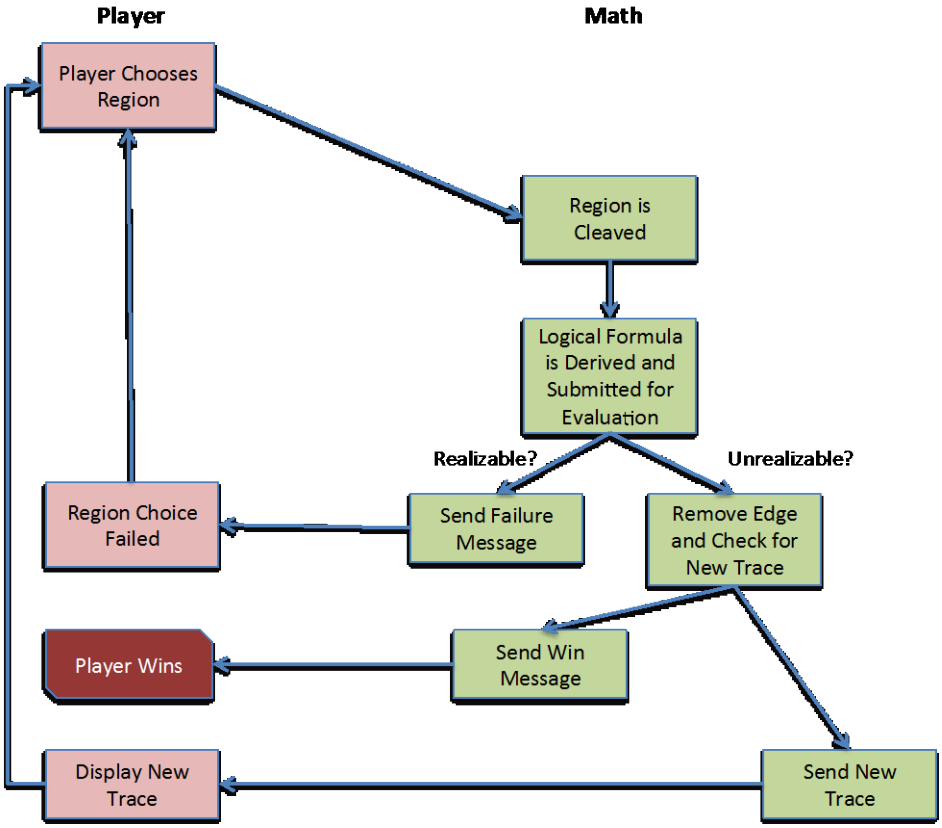


Figure 14. Player region choice and math response sequence

3.3.3 Game Generation.

Games are initially generated offline and stored in the server for later player access. The stages of game generation are shown in Figure 15.

The input to game generation is shown in the upper left hand corner of Figure 15. It consists of:

- *C source code* – The source code of the program to be analyzed.
- *make output* – It is assumed that the executable whose code will be examined is built using the make utility. The output of a trial run of *make* is captured for use during game generation.
- *FSA file* – This is a file that contains the safety violation FSA in the format specified by MOPS.

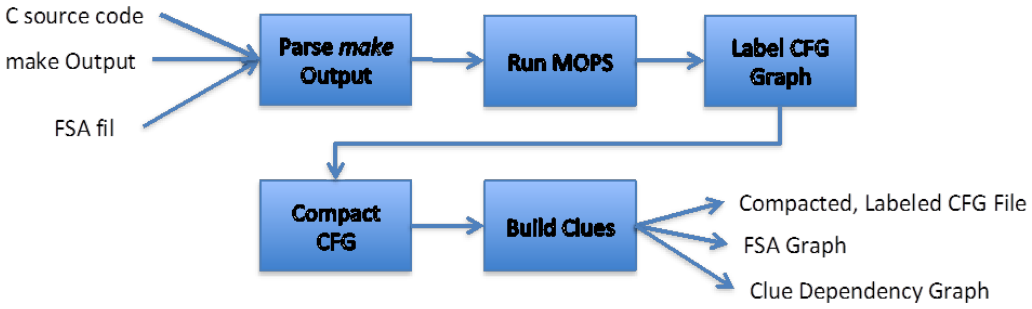


Figure 15. Stages of game generation

The following operations are performed in game generation, as shown in Figure 15:

- *Parsing of make output* – MOPS works by inserting its own *cc1* portion of the *gcc* compiler. The pre-processor and cc1 must be run for each C file using the same switches used in the compilation run by *make*, providing such values as the definition of pre-processor macro values. To extract these flags, the verbose output of *make* is analyzed.
- *Running MOPS* – MOPS is run and produces the CFGs for the various functions in the program as well as a set of traces. This is discussed at greater length in subsection 3.3.3.1.
- *Labeling the CFG Graph* – As one form of clue, locations in the CFG where the FSA can make a transition are shown to the player. This involves labeling the CFG with this information.
- *Compacting the CFG* – MOPS produces two versions of the CFG, the first being complete with nodes corresponding to all executable statements and the second containing only nodes that lead to transitions in the FSA. The first version produces a CFG that is too large and complex for game play while the second eliminates nodes that may contain information about the use and modification of variables. The math processing produces a CFG that preserves all branching information as well as nodes involving function calls but eliminates long non-branching regions.
- *Building Clues* – The AST and CFG are used in conjunction to produce a data structure that represents the data flows within the program. This data structure includes the CFG in its entirety, but also extends the CFG with data flow edges between statements when there is a data flow relationship between them (e.g., between a statement assigning a value to a variable and a statement in which that variable is used in a conditional expression). The data structure also includes an "internal" data flow graph for each node in the CFG (i.e., for each statement in the program). Each graph encodes which variables in its corresponding statement affect the

variable (or variables) being modified in that statement. For example, the variable in a right-hand side expression of an assignment statement affects the variable being defined on the left-hand side. In both cases, the data structure incorporates effects due to conditional branching (e.g., if a variable is used in a conditional expression within a conditional statement, then it necessarily affects any variables being assigned within the body of that conditional statement).

Game generation produces three products, which are forwarded to the server for eventual use by game clients. These are shown on the lower right hand side of Figure 15. They are:

- *Compacted, labeled CFG file* – The CFG file with the appropriate level of detail as well as information about edges, which cause potential transitions in the FSA.
- *FSA graph* – An appropriately labeled representation of the FSA corresponding to the security threat.
- *Clue dependency graph* – A graph providing locations in the CFG where various variables are accessed or modified.

3.3.3.1 MOPS Code Analysis

MOPS consists of multiple processing stages as shown in Figure 16. These stages are:

- *Parsing C source code* – In this stage, the standard C pre-processor is run followed by the MOPS specially modified *cc1* run which produces a binary format CFG for each function. These CFGs are saved. They contain full information about all executable statements and branching. The MOPS suite also includes tools to transform the binary CFGs into text format which are more easily understood and manipulated.
- *Merging multiple CFGs* – The binary CFGs from multiple compilation units produced in the C parsing stage are merged into a single CFG for analysis.
- *Compacting CFG* – MOPS analyzes the CFG and determines which nodes and edges refer to items associated with FSA transitions. Only those items found in the FSA are retained leading to a much smaller and more easily analyzed, compacted FSA. It should be noted that the compacted CFG is missing many of the nodes and edges that would be needed to determine if a trace is realizable based on possible values of variables accessed along the trace.
- *Model Checking* – This is where the real work of finding possible violations takes place as well as finding shortest path traces that correspond to these violations. The traces that are produced are expanded to be full traces, which are paths in the complete CFGs rather than in the compacted CFGs.

Figure 16. MOPS processing steps.

3.3.3.2 CFG and FSA Preparation

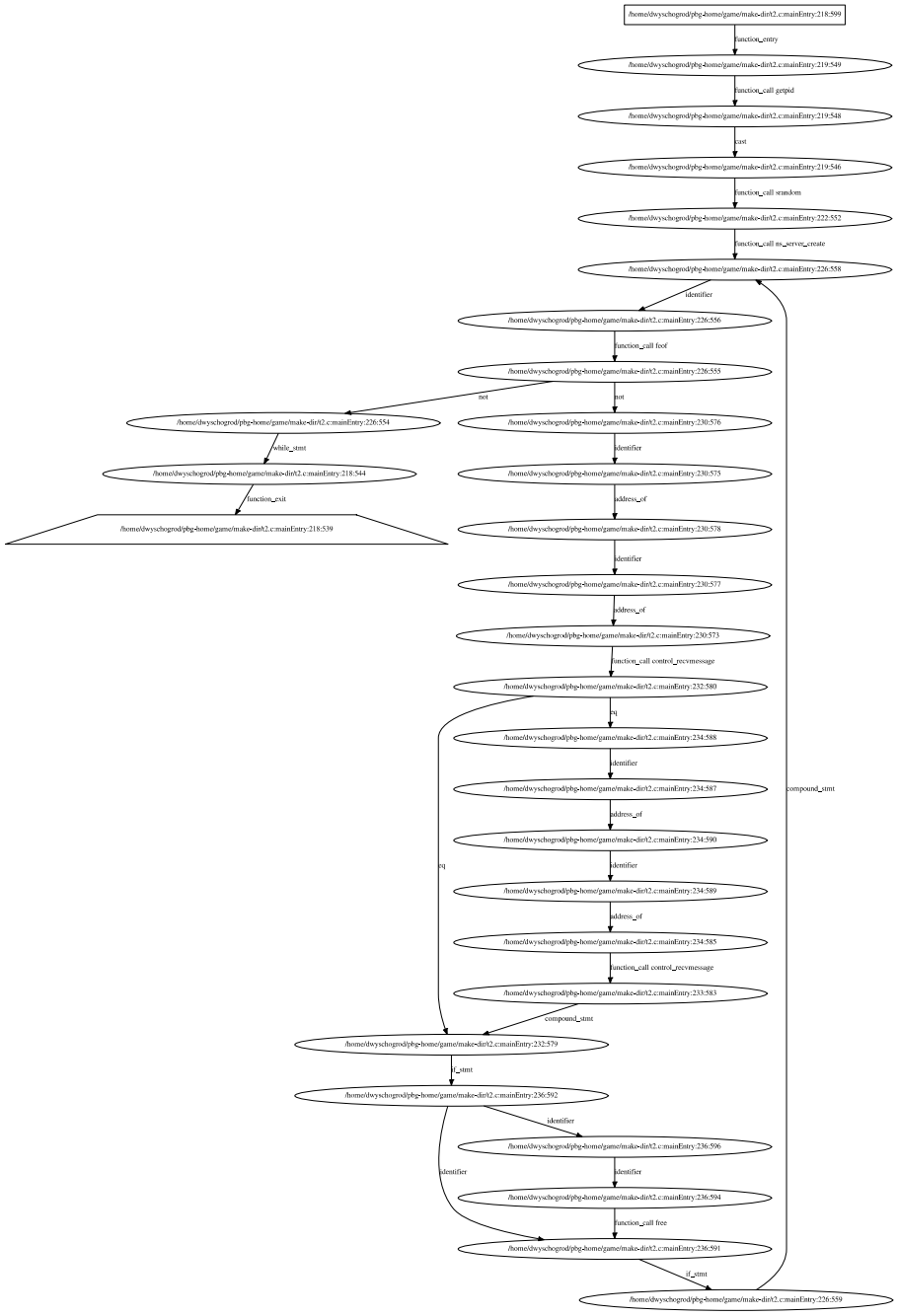
Both the CFG and FSA are forwarded to the server for storage as DOT files, a common format for presenting annotated graphs. In the case of the FSA, MOPS contains code that transforms the MOPS FSA text format to an annotated DOT format. In the case of the CFG, MOPS generates two different versions and the game generation produces a third, which is the one used in game

play. The corresponding C code is shown in Figure 17. Examples of the various versions for are shown in Figure 18

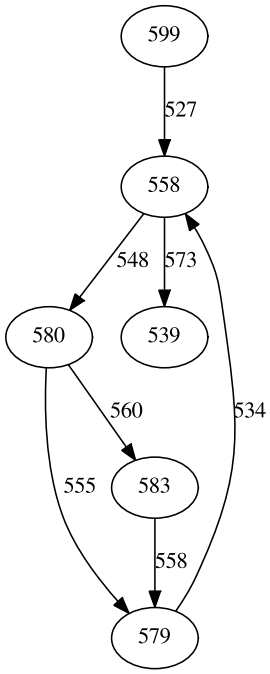
Figure 18(a) shows the complete CFG for the source code shown in Figure 17. As well as the full branching structure, long regions of straight-line execution are shown. The form of this directed graph was deemed too complicated to show to players. Figure 18(b) shows the compacted version produced by MOPS. This version has only those nodes that involve transitions of the FSA shown in Figure 18 (d) where the call to *control_recvmessage* triggers the *receive_request* transition. Notice that the branches at node 555 and 592 associated with the initial *while* and the conditional for the *free* function are missing since they don't involve a *control_recvmessage* call. Clearly, this level of compaction entails the loss of branches that may involve relevant accesses and modifications of variables that can affect later conditionals. Figure 18 (d) shows the branch-preserving compaction which retains all branching as well as all nodes associated with function calls. This is provided to the server for use in game play.

```
void mainEntry(int i) {
  srandom((unsigned) getpid());

  /* Take us out of "setup" */
  ns_server_create();
  //ns_server_create();

  /* read commands & dispatch them */
  while(!feof(stdin)) {
      char* bufptr = NULL;
      size_t bufsize = sizeof(bufptr);

    control_recvmessage(&bufptr, &bufsize);

    if(i == 7)
      {
      control_recvmessage(&bufptr, &bufsize);
      }
    if(bufptr) free(bufptr);
  }
}
```

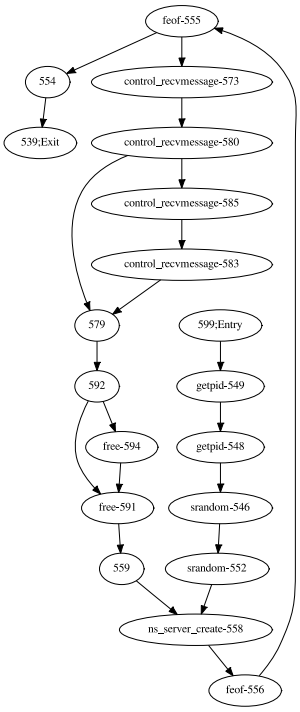
Figure 17. Example C Source Code



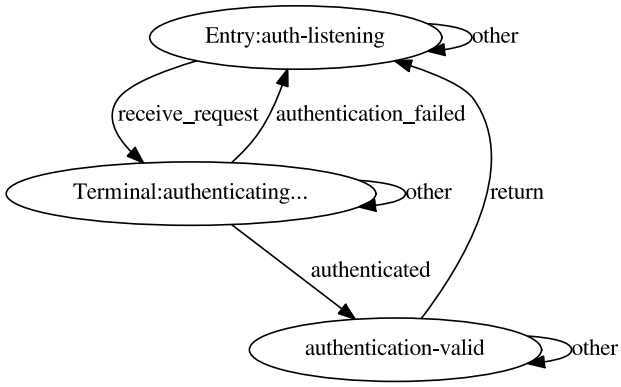
18a. Complete CFG



18b. MOPS compacted CFG



18c. branch maintaining CFG



18d. FSA

Figure 18. (a) Complete CFG, (b) MOPS compacted CFG (c) branch maintaining CFG, (d) shows FSA

3.3.3.3 Trace Preparation

MOPS produces traces based on the complete CFG. In order for this to be represented to the player, the trace must be mapped to the nodes and edges found on the branch-preserving CFG. This is performed to produce a set of traces that will form the levels presented to the player.

3.3.3.4 Clues

The AST and CFG for the program are obtained from MOPS. The CFG already contains information that ties individual CFG nodes to the corresponding nodes in the AST. This makes it possible to determine for each node in the CFG what variables it modifies and what variables it uses, and it makes it possible to determine which CFG node corresponds to a statement node in the AST.

A recursive algorithm traverses the AST and uses a *context* data structure to keep track of variables, including (1) the statements in which they are modified and (2) the statement in which they are utilized (i.e., they appear or are read). This context data structure makes it possible to determine at any point the effect variables that have already been defined have on any variable that appears in a given statement that is currently being analyzed. As the algorithm traverses the tree and builds up this context, it inserts new edges into the CFG corresponding to any data flow relationships it discovers. It also inserts any “internal” relationships between variables for each statement.

This process also keeps track of indirect relationships between variables through conditional branching or looping constructs. For every branching or looping construct statement node encountered in the AST, any variables used in the condition expression for that construct are added to the context and marked as such. Then, any variables modified within the body of that construct are affected by the variables in the condition.

The data flow graphs built using this process are *transitively closed*, in that indirect relationships between two variables for which intermediate variables exist are also encoded in the resulting data structure.

3.3.4 Handling Player Moves.

3.3.4.1 Math-side Interactive Game Playing Architecture

The following sequence takes place after a player chooses a region of a trace for an edge removal:

1. The choice of region is translated by the game client into a sequence of cleaves. A cleave operation is recorded for every location within the region where a cleave can be performed. The end of the region is immediately after the least cleave (this is enforced on the game playing side). A request containing the sequence of cleaves and the removal of the last edge in the region is prepared and forwarded to the server.
2. The server starts up a VM (or obtains an available idle VM) from the Amazon EC2 cloud, in which the math processing runs. The request from the game client is encoded in a file provided by the server along with a per-player, per-level persistent database containing a copy of the current state of the CFG after previous cleaves and edge removals.
3. The math VM applies the newly requested cleaves and determines whether the requested edge can be removed. If the edge can be removed, it is determined whether the resulting modified CFG produce any new traces.
4. The results of the analysis are stored in a series of files and the instance of the math VM suspends itself.
5. The server retrieves the result file from the appropriate location and passes the extracted result to the game client. In addition, the server stores the updated persistent database to be presented to the math VM along with the next player request.

A more detailed representation of the processing performed by the math VM is shown in Figure 19. The details of the processing are described in the following sections. In addition, the various operations will be illustrated with a simple example. The code for this example, along with a corresponding FSA are shown in Figure 20. One of the traces corresponding to a possible violation is shown on the CFG in red.

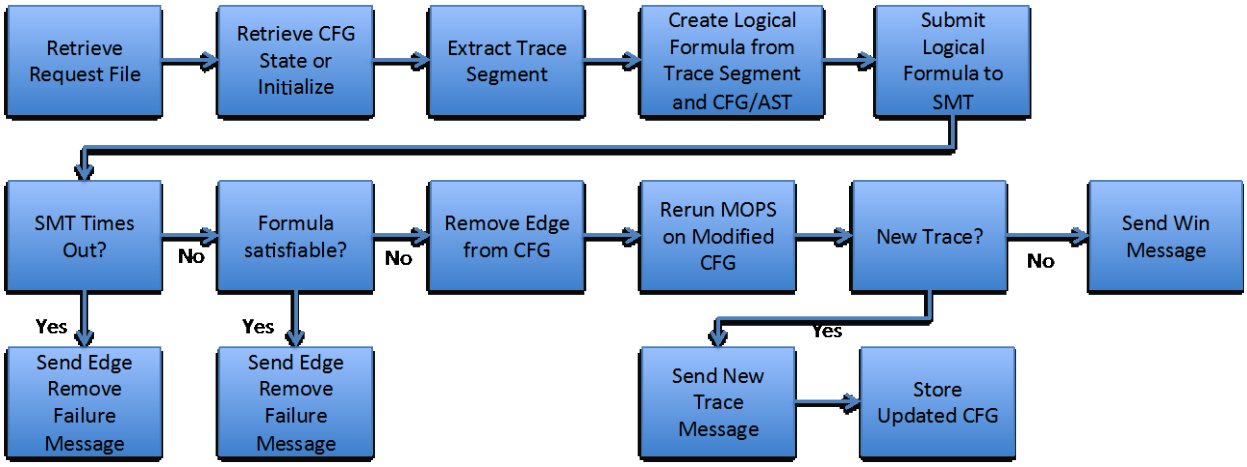
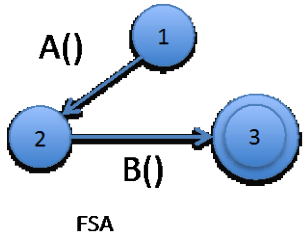


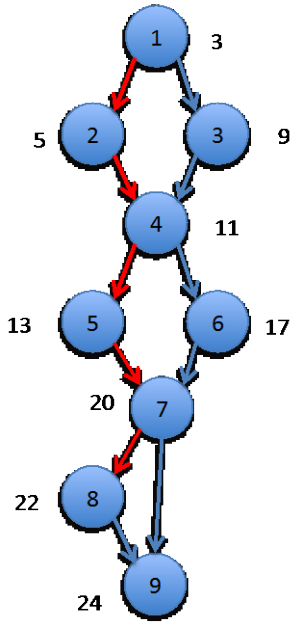
Figure 19. Math processing stages in interactive game playing.

```
1   int f(x)
2   {
3     if(x > 0)
4     {
5           x=3;
6     }
7     else
8     {
9           x=4;
10    }
11    if(x == 5)
12    {
13          A();
14    }
15    else
16    {
17          A();
18          x += 5;
19    }
20    if(x<0)
21    {
22          B();
23    }
24  }
```

C Source Code



FSA



CFG with Trace

Figure 20. Simple example of source code, FSA and CFG with trace.

3.3.4.2 Initialization of Math VM

When the math VM starts up, it first retrieves the request file. This file consists of a sequence of cleaves and a single suggested edge removal. This step is shown as the first box in Figure 19.

The math VM then checks to see if a persistent database exists. This database contains the current state of the previously cleaved and edge removed CFG. If this persistent database exists, the current state of the CFG is loaded from it. If it does not exist, the initial CFG state is retrieved from the game generation data.

3.3.4.3 Trace Segment Extraction

Once the latest modified CFG has been retrieved, the cleaves from the most recent request file are applied. An example is shown in Figure 21. Node 4 is cleaved into nodes 4' and 4". Node 5 is cleaved into 5' and 5". The resulting graph is shown in Figure 21.

The next step involves the extraction of the trace segment. Starting at the edge whose removal had been requested, a path is traced backwards along each and continues until either a node of in-degree greater than one is encountered or the entry point of the function is reached. The partial path, or trace segment, is shown in red in Figure 21.

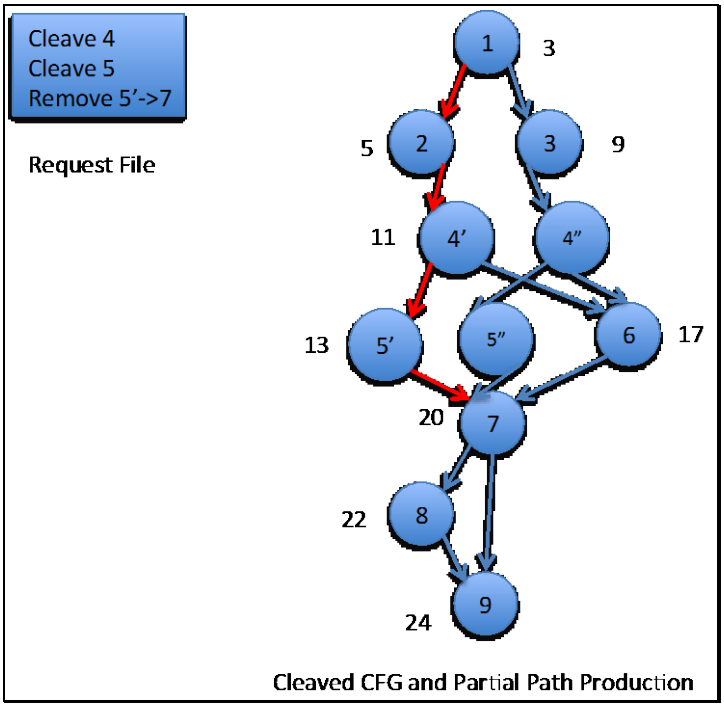


Figure 21. Cleaving and partial path production.

3.3.4.4 Preparing and Submitting Logical Formula to SMT Solver

Once the trace segment has been extracted, the next step involves forming a logical formula from the expressions found in that region of the code, as shown in Figure 20. To continue our example from Figure 21, Figure 22 shows the logical formula extracted from the partial path. In the actual system, MOPS provides files that link the CFG nodes to the corresponding AST segments, which

can be used to extract statements and expressions that employ variables. The logical formula is then translated into SMT-LIB syntax for submission to the SMT solver.

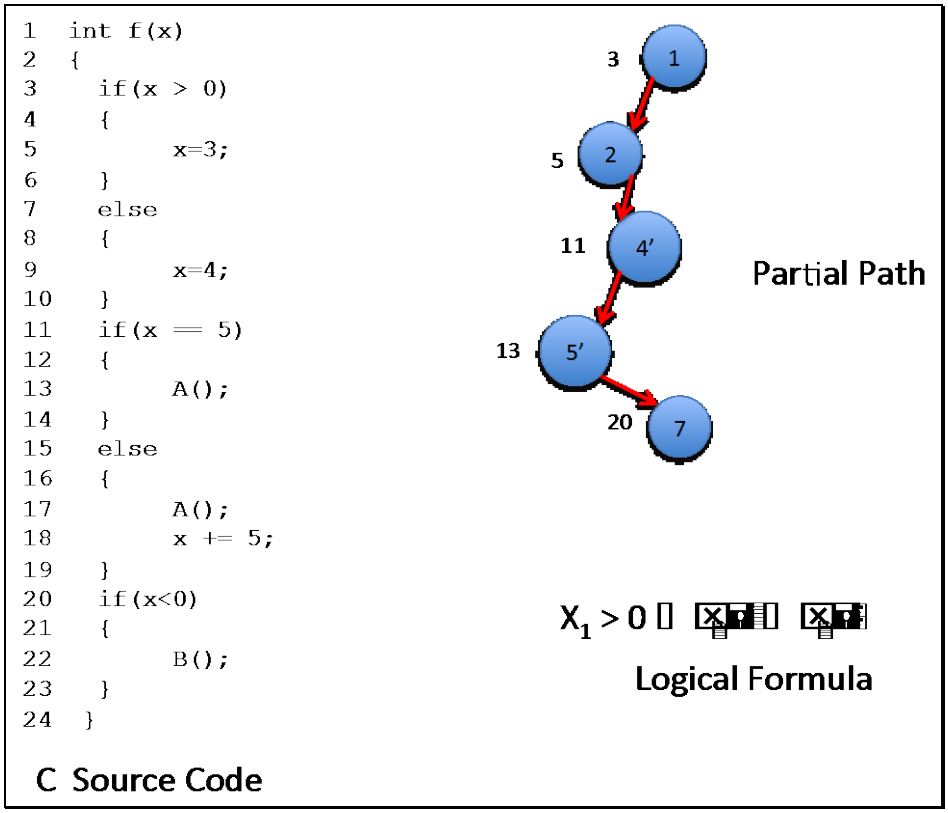


Figure 22. Logical formula extraction.

3.3.4.5 Interpreting SMT Results

The SMT solver can return with one of three possible results:

1. The formula is realizable. In that case, some set of values of the variables can be found which satisfies the logical formula and the edge cannot be removed.
2. The formula is not realizable. This means that no set of variables exist which will allow this trace segment to be executed. Since it can never be reached, the edge can be removed.
3. The SMT solver timed out. In this case, in keeping with the conservative assumption that we never declare code to be free of security violations when it is not, we do not allow the removal of the edge.

In the case where the formula is realizable, the edge cannot be removed and a failure message is sent to the server to be forwarded to the game client. The same takes place if the SMT solver times out.

In case the formula is not realizable, the process proceeds to the MOPS rerun phase, described in the next section.

3.3.4.6 MOPS Rerun

In the case where the trace segment is unrealizable, the requested edge is removed from the CFG. Our running example instance with the requested edge removed is illustrated in Figure 23. The text version of the CFG is then modified accordingly, is then converted back to a standard MOPS binary CFG, and then run with the same FSA. If the new run produces no new traces for the given CFG, the violation has been eliminated and the player wins. An appropriate report file indicating the win is put into place. Only traces that lead to the same violation node lead to further game play since traces leading to other violation nodes constitute different levels.

If a new trace is found, it is compacted and returned as part of the result set of files. The player then attempts to cleave and remove an edge from this new trace.

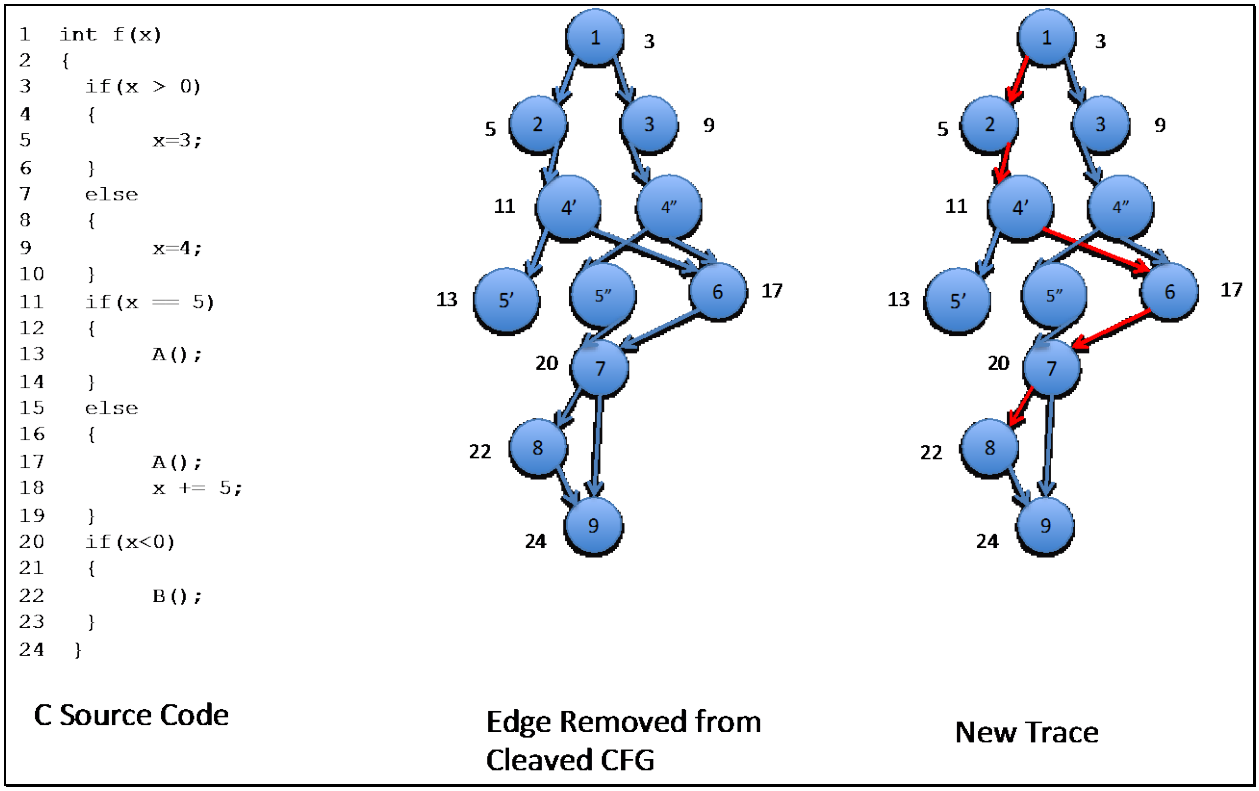


Figure 23. Results of edge removal and MOPS rerun.

3.3.4.7 Preparing Results File

Result files are placed in particular directories where the server will extract them when the math VM completes its current run. The result typically indicates success or failure but may include a new trace.

3.3.4.8 Cleanup and Modified CFG Storage

Before terminating, the math VM stores the history of cleaves and edge removals as well as the latest state of the modified CFG in preparation for its next invocation for the same player, same

game and same level. By terminating, it indicates to the server that its state should be stored and that it relinquishes all dynamic resources.

3.3.5 Supporting Tools.

3.3.5.1 MOPS

MOPS consists of a suite of tools written mostly in Java but also containing a replacement for the *gcc cc1* pass written in C. It also contains useful utilities for outputting CFG in text format as well as output that associates nodes in the CFG with constructs in the AST. Unfortunately, since the *cc1* was derived from an older version of *gcc*, it does not support a small number of newer *gcc* only extensions to C, which must be modified by hand if found in code to be analyzed.

3.3.5.2 SMT Solver

The system produces logical SMT formulas using the standard SMT-LIB format, which is supported by many mainstream SMT solvers. Thus, many different SMT solvers could be plugged into the application. However, there are specific idiosyncrasies (in terms of both performance and functional support) associated with each solver when it comes to special operations, such as bit vector operations. The translator that builds SMT formulas is geared particularly to target two SMT solvers: Alt-Ergo [8] and CVC4 [9]. Alt-Ergo is a simple SMT solver that allows for easier debugging, but lacks certain features. CVC4 has much broader support for different operations, and would be the best choice of solver to use in a production system.

3.3.6 Design Changes from Phase 1 to Phase 2.

The Phase 1 PBG Math System was based on the Cloud9/KLEE symbolic execution engine [12]. The primary advantage of applying Cloud 9 was that it significantly reduced the amount of new code that had to be written. It was one of the CSFV program goals to reuse existing formal verification tools as much as possible, so that the bulk of the program research effort could be focused on the deployment of games and the testing of crowd sourcing. One of the disadvantages of the symbolic execution approach is that it is exclusively a top-down reasoning process. Our early plan was to start with the top-down Cloud9 implementation and then later to augment it with a bottom-up approach. The early deployment of the Cloud9 system showed promise, but system testing in early 2013 exposed some problems with the Cloud9 approach. We designed and deployed modifications to correctly deal with loops in the CFG. Using our modification of the Cloud9/KLEE test case tool, we were able to search the reachability space in the loop-free portion of a CFG. This reachability search is exhaustive in a loop free region provided that there are no time-outs from the calls to the internal SAT/SMT checkers, and this provides the proof that an edge can be removed. Given the nature of the top-down edge elimination algorithm, the game player is encouraged to unroll loops to the extent necessary to create removable edges.

Near the end of Phase 1, additional problems were discovered with the Cloud9 approach and it became clear that a new approach was needed. In Phase 2, the Math System was redesigned using a custom-built C language analysis tool with direct submission of the results to the Alt-Ergo SMT checker. The advantage of the new system is that the design is much more robust than in Phase 1. One drawback is that because we wrote new code to process C syntax, we were not able to cover the entire C language. The tests of the Phase 2 system showed that the math engine

was fundamental correct in its design. When problems were detected, they usually dealt with (1) the interaction between the math system and the game system, or (2) the submission of C code that was outside the scope of the custom-built processor.

3.3.7 System Limitations.

3.3.7.1 C Language Limitations

The PBG math system only handles a subset of the full C language. The limitations relate primarily to powerful constructs such as function pointers and complex expressions, such as nested array references. These limitations did not pose an obstacle for the proof-of-concept aspect of the project, which addressed whether game play can support formal verification. The C language limitations would be an issue if PBG were to be deployed and run on arbitrary C code. Many static analysis tools have similar limitations [13]. The source of many of these limitations is that a logical formula with simple variables must be presented to the SMT solver. Simple array references can frequently be represented as variables, but complex indices or pointer arithmetic cannot be handled by these methods.

3.3.7.2 Error Analysis across Function Bodies

PBG only handles error traces inside a single function at one time. The player can choose which function to play, but their cleaves or edge removals cannot cross function boundaries. Much of this limitation is due to difficulties in parameter passage and associating values, particularly with C pointer passing semantics. This is a significant limitation.

3.3.7.3 Model Checking Limitations

The nature of MOPS model checking also places limits on the verifications that can be addressed. Most significantly, we can only test for undesirable behaviors that can be expressed as FSAs. Further, the transitions in the FSAs that can be tested for involve rudimentary matches to function calls or simple expressions and are not checking for “deeper” properties such as array bounds checking.

3.4 Game Client

This section describes the game client design and implementation, and the process and factors that contributed to its design.

3.4.1 Game Design.

In the Phase Two game, *Ghost Map: Hyperspace*, the player assumes the role of a space mercenary preventing aliens from invading through rifts depicted on hyperspace maps. These maps are direct representations of the control flow graphs produced by the math system, and the “rifts” through which the aliens invade are direct representations of the violation traces output from the math system.

The player’s goal, ultimately, is to find a section of each rift on which they can successfully deploy their “rift sealer” drones in order to seal the rift and prevent the alien invasion. A successful rift seal corresponds, naturally, to a proof that the violation trace it represents is in fact a false alarm.

Gameplay consists of these three discrete phases:

Layout Arrangement – At the start of each level, the player is presented with a "hyperspace map" that represents the control flow graph output from the math system. In the Layout Arrangement activity, players are awarded points for organizing the control flow graph by reducing the number of crossed edges while sorting the nodes. Sound effects and animated graphics highlight the player's progress. Figure 23 shows a simple example of this phase in action. By dragging one of the hyperspace map nodes, the player has uncrossed several crossed edges, scoring several hundred points in the process. Activity in this phase allows the player to get a sense of the shape and flow of the current hyperspace map, and to take ownership of its form in preparation for the next phase of gameplay, Actuator Manipulation. The game requires the player to score a certain minimum number of points required in order to proceed to the Actuator Manipulation phase.

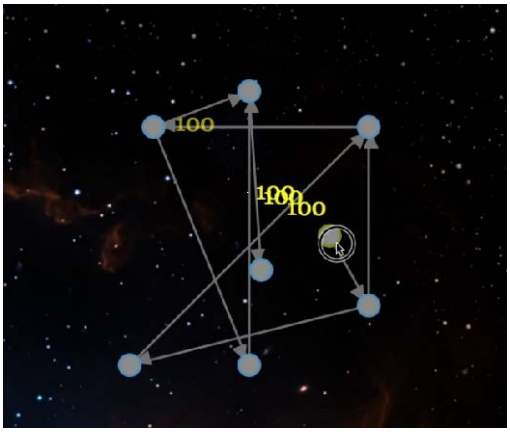


Figure 24. Layout Arrangement

Actuator Manipulation – This phase involves the actual math work that represents the system's output: players attempt to seal rifts in hyperspace, and each success yields a proof that a particular violation report from the math system is in fact a false alarm. In order to guide their actions, the system provides a graphical representation of the variable dependency data described in Section 3.3. In the game narrative, the "energy signature" table (See Figure 25) that delivers this information is presented as data collected about the aliens' use of different kinds of energy as they attempt to break through the rift. The player must use this information along with the graph structure (e.g. looking for places where the cone of influence of a single variable spans multiple branch points in the CFG) in order to find suitable selections on the violation trace for submission to the math system (and, ultimately, the SMT solver).

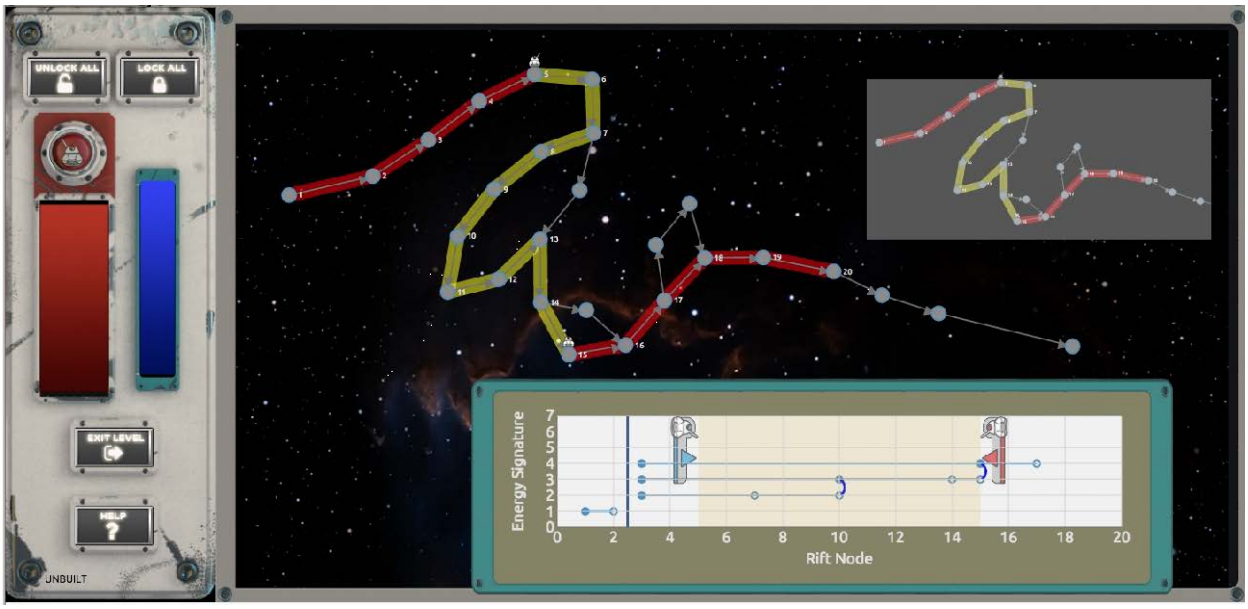


Figure 25. Selecting a Region on the Violation Trace

Attack Resolution – The third phase of the game involves attack resolution. In this activity, the player engages in a real-time combat "mini-game" where attacking alien ships must be repelled. This game is a pure engagement activity that operates while the SMT checker is working in the background (which may take several minutes). Figure 26 and Figure 27 show the basics of the mini-game: the player deploys sensors (on nodes) to detect and reveal hidden aliens, and zappers (on edges) to destroy them.

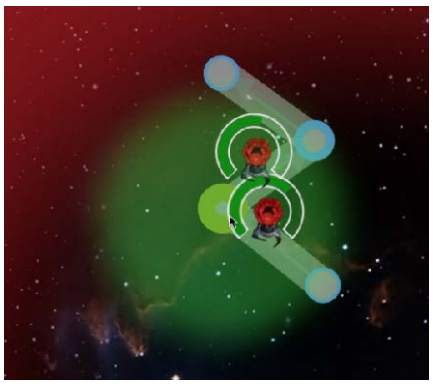


Figure 26. Using Sensors

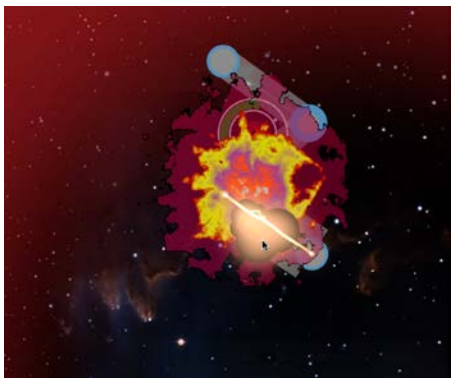


Figure 27. Using Zappers

3.4.2 Game Design Process.

Given the nature of PBG – making a compelling game out of deeply technical mathematical problems – the design of a suitable gameplay model was naturally one of the team's greatest challenges. This section describes some of the aspects of that challenge, and the process that led to the final design.

3.4.2.1 Motivation from Math System

As described in Section 3.3, the essential math system affordance involves the selection of a region on the program execution trace under consideration, that will be sent to an SMT solver to determine whether it is in fact realizable or not. This fundamental feature of the Math system provided a clear target for analogy for the game design process.

While considering a space-opera-themed narrative and game design framework, the game design team saw and cultivated analogies first between the control flow graph and a hyperspace map (with the graph edges representing hyperspace links between star systems), and second between a program execution trace and a "rift" in a hyperspace map: a tear in the fabric of hyperspace, created and utilized by an invading alien force to overwhelm and take control of star systems including those controlled by the player.

In this context, the game design could easily describe and motivate the player's challenge: given a limited supply of rift-sealing drones, find sections of each rift small enough that the drones wouldn't run out of power, yet large enough that the rift seal would be effective.

3.4.2.2 Guru-Based Crowdsourcing Model

Initially the Proof by Games team sought to design and develop a game that might appeal even to casual players seeking to play a quick game level while, say, waiting for the bus. Ultimately, though, the design team came to recognize that the only practical approach appealed to those players – "Gurus" – willing to embrace the "citizen science" theme of the CSFV program and – motivated largely by their desire to contribute to the project in the name of science, and the good of humanity – dive deeply enough to understand the real essence of the game, and thereby develop an ability to submit game moves with a maximal chance of success. We discuss this more deeply in Section 5.4.

3.4.2.3 Changes from Phase One

The PBG game system underwent significant changes in both design from Phase One to Phase Two. Here we review these changes and their motivation.

As discussed in Appendix B, the output of a crowd-sourcing system can be characterized by this simple formula:

$$totalOutput = playerTime \times playerEfficiency$$

Here, *playerTime* represents the product of the number of players (largely a function of marketing, including word-of-mouth) and the average time commitment per player (a function of engagement, as well as quantity of content) and *playerEfficiency* represents the degree to which player effort is put directly to generating the results that the system ultimately seeks.

The Phase One game – *Ghost Map* – sought to maximize player efficiency, and therefore included nothing more than the essential player activity required to provide output relevant to the math system. (In the Phase Two design, this corresponds to the Actuator Manipulation phase.)

In Phase Two, responding to a clear and explicit challenge from the DARPA team, the PBG team considered the possibility that adding "pure engagement" features solely for the purpose of

increasing player engagement (at the cost of player efficiency) might actually increase overall system output. This led to the introduction of the Layout Arrangement and Attack Resolution gameplay elements. In addition to enhancing player engagement, these elements serve to galvanize the player's mental model of the CFG, and to cover dead time during back-end processing.

The Phase Two game also saw the introduction of data dependency clues (see "energy signatures" above), which replaced a presentation of the FSAs that was included in the Phase One game. We discuss this further in Section 3.4.2.2.

3.4.3 Implementation.

Here we review some of the implementation details of the game client, from both Phase One and Phase Two. The core functionality required from the game client was broadly similar across both phases: in addition to providing the usual sundry user interfaces for level selection and flow, the system needed to present the control flow graph provided by the math system, afford the player mechanisms to interact with it, and of course transact with the math system via the game server.

3.4.3.1 Phase One: Unity3D

The Phase One game, *Ghost Map*, was deployed using the Unity game engine, which functions as a browser plug-in. The primary benefit of Unity is that many common game functions are already implemented in the engine. This allows a game to be developed more efficiently and with a more professional look than would be possible with manual coding. On the other hand, the drawbacks to browser plug-ins include security risks, typically frequent update requirements (although Unity seemed better than most in this regard), and general increased reluctance by users to installation of plug-ins.

Moreover, the advantage of Unity's providing many game-related functions out of the box only went so far. In the course of design and development, it became clear to us during Phase One that really what we were creating was something in between a game and an information visualization platform. Unity provided a good deal of functionality that we did not need (animation of 3D articulated models, for example) and was lacking functionality that was critical to our design such as a force-directed graph layout mechanism, and strong 2D interface tools.

All these factors together drew us to reconsider our choice, and ultimately make the tough decision to move to an entirely new platform for the Phase Two implementation.

3.4.3.2 Phase Two: HTML5

In Phase 2, the PBG team built *Ghost Map: Hyperspace* in HTML5, making heavy use of the D3 data visualization suite to render the CFG with force-directed layout. This brought several advantages:

Although the graph layout mechanism designed from scratch in Unity for Phase One worked well for small levels, it was not heavily optimized and suffered from decreased frame rates when displaying larger levels. On the other hand, the D3 package for HTML5 is a well-supported open source set of components with a large and thriving community that has contributed significantly to its optimization and general robustness.

Ghost Map: Hyperspace, unlike *Ghost Map*, required the introduction of an entirely new and fairly complex 2D user interface element to convey the variable dependency clue data provided by the math system, and to allow the player to explore and interact with this information. Without a strong 2D user interface mechanism, this element would have taken considerable effort to write from scratch in Unity. In HTML5, on the other hand, the entire focus of the platform is on 2D interface design and presentation. We were able to prototype and finally implement this mechanism in a fraction of the time it would have taken in Unity.

Finally, because Unity began its life as an engine for use with standalone "thick client" game applications (as opposed to browser-based games) its support for the kind of parallel asynchronous network transactions required by PBG was awkward, and required excessively complex code that led to more bugs and debugging time than we would have liked. HTML5, on the other hand, was designed from its beginning with this kind of parallel asynchronous network activity in mind, which led to much simpler, cleaner code, that was easier to read and therefore less prone to bugs and less expensive to extend and maintain.

3.5 Game Server Implementation

The game server is the primary point of contact for game clients. It keeps track of all game-related information starting with the data of the game levels themselves as well as the long-term status of every game level that is being played or has been played. The server communicates with the *Topcoder Services* on behalf of the player to keep the player's performance and scores up-to-date as well as giving him credit for awards he has earned.

The game server implements a set of RESTful web services written in Java using the *Jersey* framework and deployed in a *Tomcat* servlet container. The web services are primarily intended for use by game clients, but there are also services to support administrative activity. A *Mongo* object database provides a persistent store for player, game level, and administrative information.

The server creates *AWS EC2* virtual machine instances[1] as needed to run the math tools backend and interacts with those instances to utilize those tools. The server also supplies web pages that allow a privileged user (administrator) to examine the state of the server and perform housekeeping chores. This section describes the game server and its interactions with external and internal entities.

3.5.1 Architecture.

Figure 28 shows the interactions between the game server and other entities. *Topcoder Services* provide player authentication and visibility of a players' performance through web pages called the *minisite*. A player using a web browser interacts directly with the *minisite to log in and view his past performance and ranking with respect to other players*.

[1] The term instance is used in two ways here. It is used to refer to an instance of a player playing a game level and it is also used to refer to an EC2 virtual machine instance. Usually, the context can be used to determine which meaning applies. EC2 instance is used for the former and the latter is referred to as a level instance.

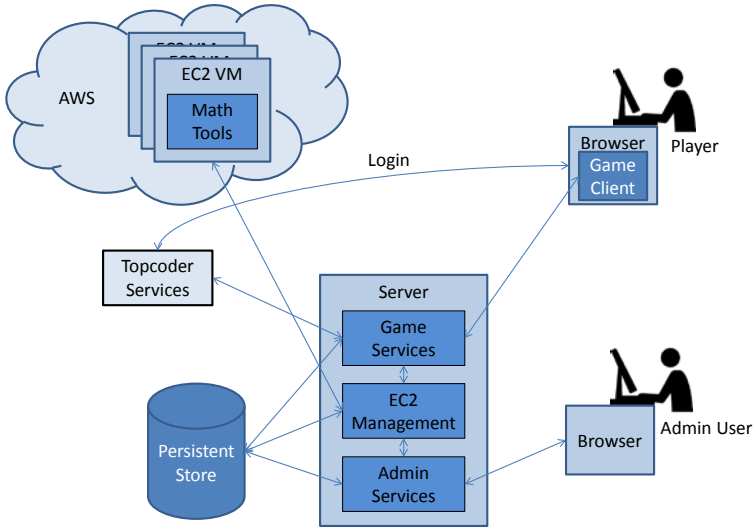


Figure 28. Game Server Architecture

The player chooses to play a game using controls on the *minisite*, which direct the browser to a web page housed in the game server causing the browser to load the game client. The game client interacts with the game server to select and play game levels or tutorials. As game play proceeds, the game server sends performance information to *Topcoder Services* so it can be later displayed by the *minisite*.

Administrative actions can be taken by a privileged user. Such actions include installing new games, diagnosing issues with game play, and gathering statistics.

A persistent store keeps track of all the long-term state and is updated by the game server as administrative tasks are performed and gameplay occurs. The persistent store is the definitive repository of gameplay state. The information in the *minisite* is synchronized to the state in the persistent store. More detail about the roles of the entities in this diagram is provided below.

3.5.2 Persistent Store.

While the persistent store is nearly invisible outside of the server, we describe it first because it is central to all activity performed by the game server in that it provides persistence for all aspects of server state. Persistent state is maintained in a *Mongo* object database as a set of “collections”. Each type of data is kept in its own collection so the data within a collection is homogeneous.

Examples of collections include, *PlayerInfo* and *LevelInfo* which correspond to a specific player and a specific game level, respectively. Another collection is *LevelInstance*, which corresponds to a particular player playing a specific level. Other collections record the progress of game play and details about events during game play. Other information is stored for the purpose of doing statistical analysis of player behavior to assist in making improvements and informing future efforts in this area. Other specific data collections will be discussed in the context of the part of the server that uses them.

Files are stored in the database supported by two collections. The *FileMeta* collection stores directory-like information. This information includes a name and a namespace as well as a chunk count. The *FileData* collection stores the actual bytes of the files in a series of chunks. Each chunk is limited in size to conform to *Mongo* restrictions on the size of objects in a collection. Files with a size that exceeds that limit require multiple chunks. Each chunk has a length and index within the file as well as the id of the *FileMeta* object it is part of.

The use of namespaces avoids name collisions between different aspects of the system as well as avoiding collisions between files of the same name in different instances of players playing the same game level. These files naturally have the same name and it would be burdensome and error-prone to require unique names for every such file. A namespace is identified by the id of an object in the database. Every object could have a corresponding namespace, but in practice, only certain types of objects are used as namespaces. These will be described as they are encountered in the following discussions.

Code generation is used to implement the object classes for items stored in the persistent store. The *Mongo* API makes such an extension feasible. The fields of each type of object are declared in a text file from which code generation is performed. This provides a type-safe interface to the persistent store and improves code readability.

3.5.2.1 Persistent Store Collections

Table 2 lists the persistent store collections.

Table 2. Persistent Store Collections

CreditTransaction	Transactions that change a players credits including type of transaction, amount, date, and reason.
FileData	The byte data of files in chunks. Each chunk identifies the *FileMeta* it belongs to and the index of the chunk within the file.
FileMeta	The metadata of files including namespace, name, and chunk count.
GameInfo	Information about games, which consists chiefly of the name.
LevelInfo	Information about levels including the *game* it is part of, the bounty (credits for a win), the name, and the *CWE*.
LevelInstance	Information about levels played by a player including current state of play and completion date (if completed).
LevelInstanceSegment	Information about segments of play of *LevelInstances*. Segments are delineated by reports of actions taken by players from the game client. Segments serve as checkpoints to permit a player to resume play without loss and also serve as points where the math tools are invoked to check edge removals.
LevelMetaData	Information about the complexity of levels.
LevelMetaGlobals	Information for interpreting the complexity of levels.
MouseClicks	Information about the mouse clicks of a player for analysis of how a player plays the game.
MouseMovement	Information about mouse movement also for analysis of how a player plays the game.
PlayerInfo	Information about players.
TutorialCompletion	Information about when a player completed tutorials.
VMInstanceEntry	Information about *EC2* instances.

3.5.3 EC2 Interface.

Amazon Elastic Compute Coud (EC2) virtual machine instances are spawned to execute math tools. The math tools are installed on an *EC2* virtual machine and saved as an Amazon Machine Image (AMI). Math tool installation is performed outside the server and is not documented here. The id or name of the AMI for the server to use is specified as part of the configuration information.

EC2 instances are created from the configured AMI as needed using web services supplied by Amazon. When an *EC2* instance is created, its IP address is recorded and the initialization process is monitored to determine when it has completed. Once initialization has completed, the *EC2* instance is added to an *EC2* instance pool for use as needed.

A pool of instances is maintained to service the math tool jobs. This pool has a maximum limit on the total number of instances and minimum and maximum limits on the number of idle instances. Math tool jobs are queued until they can be serviced without exceeding the maximum limit on the total number of instances. As math tool jobs complete, the *EC2* instance can be reused for another job or it can become idle.

The number of *EC2* instances varies due to the conflicting goals of always having an idle *EC2* instance to quickly execute a math tool job while not having (and paying for) too many *EC2* instances. The procedure is as follows: If the number of idle instances exceeds the maximum idle instances limit, the instance is terminated instead of being returned to the pool. Periodically, if the number of idle instances exceeds the minimum number of idle instances limit, then one idle instance is selected and terminated. In the absence of activity, the number of instances gradually decreases down to the minimum.

When there is a malfunction in running a math tool (error status returned), the *EC2* instance is always terminated. This insures that any damage causing or caused by the malfunction cannot propagate to any subsequent math tool jobs.

Each *EC2* instance created is recorded in the *VMInstanceEntry* collection of the persistent store. This permits such instances to be recovered in the event of a server restart. This information is also reconciled with the list of instances reported through the *EC2* API.

3.5.4 Math Tools Support.

Math tool jobs are created as needed according to the game logic or administrative requests. An example of the former is the *EdgRmvChk* tool, which is invoked when the player attempts an edge removal. An example of the latter is the installation of a new game by an administrator. Math tool jobs are recorded in the *RemoteActivity* collection of the persistent store. This is largely for debugging and statistics collection purposes. *RemoteActivity* objects are retained indefinitely.

As noted above, the math tools are run on *EC2* instances. A given *EC2* instance runs only one math tool job at a time. This simplifies the design of the tools since they do not need to be concerned with colliding with other jobs when using resources such as files.

By design, a math tool is guaranteed to have a particular initial state dependent only on the task it must undertake. Also by design, this state is available to the math tools in the filesystem of the virtual machine; there are no external web sites or servers or processes that need initialization. This means there must be no residual effects of earlier jobs executed on the same *EC2* instance. Such residual effects might be the presence of temporary or result files. Rather than require the math tools to clean up this residue, the cleanup is performed by the server. When an *EC2* instance is first initialized, the contents of the home directory are recorded as the pristine state of the math tools. Before a subsequent reuse, the home directory is restored to that pristine state by deleting everything that was not originally present. This is not a perfect restoration, but a compromise that does not restore files that were deleted or overwritten. In principle, all files could be restored exactly, but doing so would consume resources and introduce delay, hence the decision to just delete files left behind.

Communication with the *EC2* instance is via *ssh*. Before a job is run, the filesystem is prepared by uploading files using *ssh* and tar. After a job is run, the final state is captured by downloading files using *tar* and *ssh*. The exact files depend on the job, but are separated into five types and two directions. The types are: *Game*, *Level*, *Levels*, *LevelInstance*, *Requests*, and *Results*. The directions are *To Math* and *From Math*.

Game files vary only from game to game. Ordinarily, they are written when a game is installed and are otherwise read-only. *Level* files vary from level to level. *Levels* is similar to *Level*, but is used when a tool manipulates multiple levels in one job. Individual level information is stored in subdirectories named by the level. *LevelInstance* files vary from *LevelInstance* to *LevelInstance* and generally evolve as the math tools are applied.

To Math refers to files that are uploaded (server to math tools). *From Math* refers to files that are downloaded.

A *MathProfile* specifies the mapping of file type and direction to filesystem location. Each type of job has a different *MathProfile.* Each element of a profile specifies the movement of one type of file and generally corresponds to one directory on the math tools *EC2* instance and one object (or namespace) in the persistent store. For upload, all the files in the namespace corresponding to the current object of the given type are transferred to the directory specified by the element. For download, all the files in the directory specified by the element are transferred to files in the persistent store under the namespace corresponding to the current object of the given type.

After the files have been uploaded, but before the job is started, a dump of the working directory of the math tool is taken and stored to a *dump-before.tgz* file in the namespace of the *RemoteActivity* corresponding to the job. While a job is executing, output from *stdout* and *stderr* of the math tool is logged for debugging purposes. When the job completes, a second dump of the working directory is taken and stored to a *dump-after.tgz* file in the namespace of the *RemoteActivity*. These dump file are for debugging purposes. They allow the circumstances giving rise to an issue in the math tools to be reproduced.

If a job malfunctions (returns an error status), the *EC2* instance is terminated a never used again. This is insurance that the malfunction has not somehow clobbered the *EC2* instance. The dump files recorded permit the malfunction to be reproduced.

3.5.5 Game Services.

The server provides RESTful web services for the game clients to use. Service arguments are encoded in the URL of the service. Results are JSON-encoded with minor exceptions. Each service method includes an access token provided to the game client by the game server when the game client is initialized.

The game services are divided into four sections according to the nature of the service. This division is somewhat arbitrary, but reflects the principal phases of gameplay: navigation to a specific game level to play, initializing the game client with information about the selected game level, and actual gameplay, with miscellaneous other services coming last.

3.5.5.1 Level Selection Services

To begin, a player needs to view what is available to play. There may be existing levels he has started, but not finished or levels he hasn't played yet, and so on. Table 3 lists the level selection services.

Table 3. Level Selection Services

findLevels(playerId)	Finds and returns existing paused levels that a player has. These represent game levels that the player has temporarily set aside that may be resumed to continue play.
findCompletedLevels(playerId)	Finds and returns existing levels that a player has completed.
findNewLevels(playerId)	Finds and returns levels that the player has not yet played.
findBlockedLevels(playerId)	Finds and returns levels that are blocked. That is, levels that are waiting for a math tools job to complete.
findChangedLevels(playerId)	Finds and returns levels that have changed status since the last interrogation of changed levels. This removes the cost of polling every level to detect changes. Note that polling cannot be entirely eliminated because web services use http(s) transport protocol.
requestMatch(playerId)	Selects and returns a nice set of levels for a player to play.

3.5.5.2 File and Information Services

Once the player has selected a level to play, the game client must retrieve information about that level to set up the game level. Table 4 lists the methods which provide that capability.

Table 4. File and Information Services

listInstanceFiles(playerId, levelId)	Finds and returns the names of all files in the namespaces of the *LevelInstance*, *LevelInfo*, and its *GameInfo*.
getInstanceFile(playerId, levelId, filename)	Retrieves the contents of a file from the persistent store. Three namespaces are searched starting with the *LevelInstance* followed by the *LevelInfo*, and finally the *GameInfo*. The first such file is returned. An override mechanism allows the persistent store to be bypassed substituting a file from the server filesystem to be substituted.
listLevelFiles(levelId)	Finds and returns the names of all files in the namespace of the specified *LevelInfo*.
getLevelFile(playerId, levelId)	Retrieves the contents of a file from the persistent store. Only the *LevelInfo* namespace is searched.
getLevelDetails(playerId, levelId)	Retrieves detailed information about a level.
getLevelDetailsForPlayer(playerId)	Retrieves detailed information about all levels. More details are included for levels that a player has played or is playing.

3.5.5.3 Game Play Services

Once the game client has loaded the information about the game level, play commences and continues until the player stops playing. Table 5 lists the methods that are used to keep the server informed about progress.

Table 5. Game Play Services

levelStarted(playerId, levelId)	Starts a new level for a player. There should be no *LevelInstance* for this player and level. For debugging convenience, this rule is not enforced. Instead, the existing *LevelInstance*, and all it entails, is discarded.
levelPaused(playerId, levelId, actionList)	Changes the *LevelInstance* state to *PAUSED*. If *actionList* contains an edge removal action, a math tools *EdgRmvChk* job to check the edge removal request is initiated and the state becomes *BLOCKED*. The state remains *BLOCKED* until the edge removal check completes when it changes to *PAUSED*.
levelResumed(playerId, levelId)	Attempts to resume the level. If the level is not currently *PAUSED*, a failure response is returned otherwise, the state is changed to *ACTIVE* and a success response is returned. The response includes the outcome of the edge removal check, if any.
reportLevelStatus(playerId, levelId, actions)	A keep-alive method. Used by the game client to report that it is still working on the given level, but needs no services from the server at this time. The actions are recorded to allow a restart if the player abandons his session.
recordTutorialStarted(…)	Records that a tutorial has been started.
recordTutorialCompletion(…)	Records that a tutorial has been completed.

3.5.5.4 Other Services

Finally, there are miscellaneous methods that do not fall neatly into the preceding categories. Please refer to Table 6.

Table 6. Other Services

errorReport(playerId,levelId,msgs)	Allows the game client to initiate a diagnostic dump. This assists in finding bugs in the interaction between the game client and the math tools.
reportLog(playerId, logRequest)	Allows the game client to log information in the server's log file.
getCredits(playerId)	Get the player's current credits.
getLevelCredits(playerId, levelId)	Get credit information about a level.
recordNonMovement(…)	Records player activity (or lack thereof).
recordMouseClick(…)	Records player activity.
recordHardward(…)	Record information about the game client computer.

3.5.6 Topcoder Services.

Topcoder Services include player authentication, player score reporting, player achievement reporting, and overall problem-solving progress. Each of these is described below.

3.5.6.1 Player Authentication

Player authentication is provided by the *Topcoder Services* and does not involve the server. However, the server must validate incoming requests from the game client. By design, the first request from the game client is a request for a particular URL. For the legacy game, this was the html file that launched the game. For the current game, the request is for a (JavaScript) file defining constants needed for game play. In either case, the server invokes the *Topcoder Services verifySession* function passing the cookies from the request as arguments. If the player has logged in, these cookies serve to identify the player and the *Topcoder Services* respond with a *playerId. If the* player *has not logged in, no playerId is returned. If a valid playerId is obtained, the requested file is generated and returned. A random access token is included within the file returned that is used for all subsequent services.*

A configuration option may allow anonymous play. If a valid playerId is not returned by the verifySession function, but anonymous play is allowed, a new, valid player is created in the persistent store and its playerId is used for game play.

3.5.6.2 Player Score Reporting

Player performance is measured in terms of scores for completed levels. The *Topcoder Services* maintain a database of these values. The server maintains the master copy of this information in the persistent store. The *Topcoder Services* database should contain the same information. The *Topcoder Services* scores are reconciled against the server scores as needed.

Typically, when a player completes a level, a new score is reported to the *Topcoder Services* identifying the player, level, and score. But, communication with the *Topcoder Services* can fail. When this occurs, a flag is set indicating that the two versions of scores are out-of-sync and a periodic attempt to reconcile is initiated. Reconciliation attempts are repeated until successful. Reconciliation is also performed when server restarts because it is assumed that there may have been a failure earlier.

Reconciliation consists of computing all the scores that should be present on the *Topcoder Services* by scanning all the completed levels of all players and computing the score for each one and then interrogating the *Topcoder Services* for the scores it has stored. Extraneous scores are removed from the *Topcoder Services*. Ordinarily, extraneous scores only arise when some sort of administrative action such as removing a game or resetting a player has been taken. Missing scores are added and incorrect scores are corrected.

3.5.6.3 Player Achievement Reporting

Certain events during game play are considered worthy of an achievement award. For example, the completion of a tutorial and the first level won are rewarded. Each achievement has a corresponding “badge” that identifies the achievement and can be shown on the *minisite*. Achievement badge icons are created and painted by the server as needed.

The achievements awarded can always be determined from the set of levels that have been completed. Again, the *Topcoder Services* maintains a database of achievements that have been awarded that might get out-of-sync due to communication. Reconciliation of such a discrepancy is identical to that performed when scores are out-of-sync except that achievements are removed, added, or corrected rather than scores.

3.5.6.4 Overall Problem-Solving Progress

Topcoder Services maintain a database of which CWEs have been proven absent from a particular computer program. The projects report that they have proven on a particular date that a particular CWE is absent from a particular program.

This mechanism does not permit reading the completed CWEs nor removing already reported completions. Reconciliation is not possible due to the absence of these capabilities. This is not a particular concern since the code coverage is not complete at this time. If it were, this deficiency would need to be corrected.

3.5.7 Configuration.

A limited number of configurations of the server are statically predefined by the code. One such configuration is defined to be the default, with provision for selecting an alternative configuration at runtime. Each predefined configuration establishes values for a number of different constants. For example, *VM_IMAGE_NAME* specifies the name of the AMI to be used for running math tools.

3.5.7.1 Configuration Values

Table 7 lists the configuration constants that may be set.

Table 7. Configuration Constants

MINISITE_URI	URI of the *minisite*
RA_URI	URI of the *RA* (legacy resource allocator)
API_URI	URI of the API (*Topcoder Services*)
CWE_URI	URI of the CWE (*Topcoder Services*)
VM_IMAGE_NAME	Name of the *EC2* VM *AMI* to use for *EC2* instances. The *AMI* can be specified in either of two ways: by name and by id. See *AMI_ID* below.
AWS_CREDENTIALS_RESOURCE	Identifies which *AWS* credentials to use
AWS_KEY_PAIR_ID	Identifies which key pair to use for encryption
AWS_KEY_FILE_NAME	Identifies the key pair to use
AWS_ACCOUNT	The *AWS* account to use for *EC2* instances
AMI_ID	The id of the *AMI* to use for *EC2* instances. The *AMI* can be specified in either of two ways: by name and by id. See *AMI_NAME* above.
VPC_ID	The id of the virtual private cloud to be used, if any.
USE_PRIVATE_IP_ADDRESS	Specifies if the *AWS* private IP address of the server should be used or if the *AWS* public IP address should be used.
SUBNET_ID	Specifies the *AWS* private subnet to use, if any.
EC2_MAX_INSTANCES	Maximum number of *EC2* instances to create.
EC2_MAX_IDLE_INSTANCES	Maximum number of idle *EC2* instances to retain.
EC2_MIN_IDLE_INSTANCES	Minimum number of idle *EC2* instances to retain.

3.5.8 Deployment.

The server is deployed as a war file. It presumes the presence of services and applications such as *tomcat* and *mongod. These services were already installed in several cases and manually installed where needed.*

3.5.8.1 Deployment Modes

The following configuration modes are pre-defined:

- ***DEVELOPMENT*** mode is intended for local execution during debugging by a developer. A developer can adjust these as needed without disturbing other deployment configurations such as *STAGING* or *PRODUCTION*.
- ***STAGING*** mode is used for deployment on the staging site.
- ***PRODUCTION*** mode is used for deployment on the production site.
- ***BARE*** mode disables all external services such as the *Topcoder Services*.
- ***DVD*** mode is used for deployment on a DVD. All external services are assumed to be on the same host (localhost).
- ***NOJAIL*** mode is similar to DVD mode except the math tools are executed directly without the protection of a *chroot* jail.

3.5.8.2 Deployment Housekeeping

While running, the server continually writes log files that can become quite voluminous. To preclude eventually running out of (virtual) disk space, a daily *cron* job downloads such log files to a large local disk and deletes the log files from the server.

3.5.9 Administrative and Debug Services.

The server provides a number of web pages through which a privileged user (administrator) can view and alter the data in the persistent store, view statistics, view log files on the server and other information. The admin page also displays the version number of the server software being run. Each of these pages or sections of a page is described below.

3.5.9.1 Persistent Store Collections

Every collection used by the server in the *Mongo* database can be presented in tabular form as a web page. The presentation for the *FileMeta* collection displays files as links that yield the content of the file. This allows such files to be viewed or downloaded by clicking on the link. Text files are displayed directly, non-text files are downloaded. For example the *dump-before.tgz* files can be downloaded for analysis by simply clicking on the link. As it stands, this feature does not scale well because the collections can become quite large if there are a large number of game plays. But, in a development or staging environment, easy access to this information proved very useful.

3.5.9.2 Log File Listings

Log files generated as the server runs are accessible through this page and may be viewed directly or downloaded for perusal with a text editor.

3.5.9.3 Upload Game Archive

A set of files to be used by the *LevelCreation* math tool are packaged as an archive (.tgz or .zip) and uploaded through this interface. The files in the archive are copied into the persistent store as *Game* files and the *LevelCreation* math tool is executed producing a set of *Level* files for each level in the game. These files are written to the persistent store ready to be played by players.

3.5.9.4 Diagnostic Dump

This debugging tool collects together all the information relevant to a specific execution of the *EdgRmvChk* math tool. Using this tool requires first identifying the *LevelInstanceSegment* of interest. This can be done by viewing the *RemoteActivity* collection if the *playerId* and approximate time are known. The server log (*pbgserver.log*) may also be used. The diagnostic dump includes the *dump-before.tgz* and *dump-after.tgz* files as well as an excerpt from server log covering the time during which the *EdgRmvChk* math tool was running.

3.5.9.5 Set Game Matcher Weights

The Game Matcher is used to select appropriate game levels for a player to play based on various complexity aspects of the game. This interface allows an administrator to adjust the weights applied to each aspect of the game levels to distinguish difficult from easy levels.

3.5.9.6 Recalculate Game Matcher Weights

The derived weights and factors of the Game Matcher are recalculated.

3.5.9.7 Clear All

The persistent store is completely cleared. This is a drastic measure suitable only for development, testing or staging environments.

3.5.9.8 Clear Game

All information related to a particular game is removed from the persistent store. This is also pretty drastic and used only when it is discovered that a particular game is buggy.

3.5.9.9 Reset Player

Information about a particular player can be removed from the persistent store. Also, information about all anonymous players can be removed.

4. RESULTS AND DISCUSSION

Here we describe and discuss some of the gameplay results and usability test results we saw in the course of deploying Proof by Games.

4.1 System Usage Statistics and General Evaluation

The Phase 1 Game Play period began with a soft opening to the public in November 2013. Game play of Ghost Map classic peaked shortly after the public opening (when the program press and marketing push was at its strongest) and then continued at a lower level through 2014 and the first few months of 2015.

The CSFV Phase 2 Game Play period began with a soft opening in early May 2015. Initially, the games were available to the CSFV performers only. Around 15 May, the games became available to the program friends and family group, and DARPA announced the games to the public on Wednesday 27 May.

Figure 29 and Figure 30, respectively, show the number of unique players and the number of game levels completed over time.

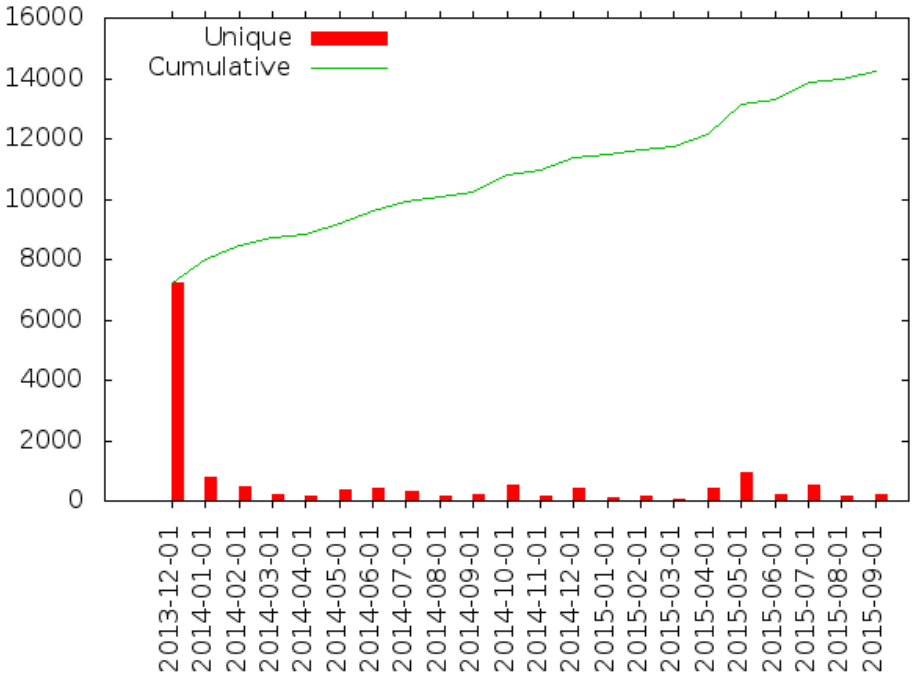


Figure 29. Proof by Games Users Over Time

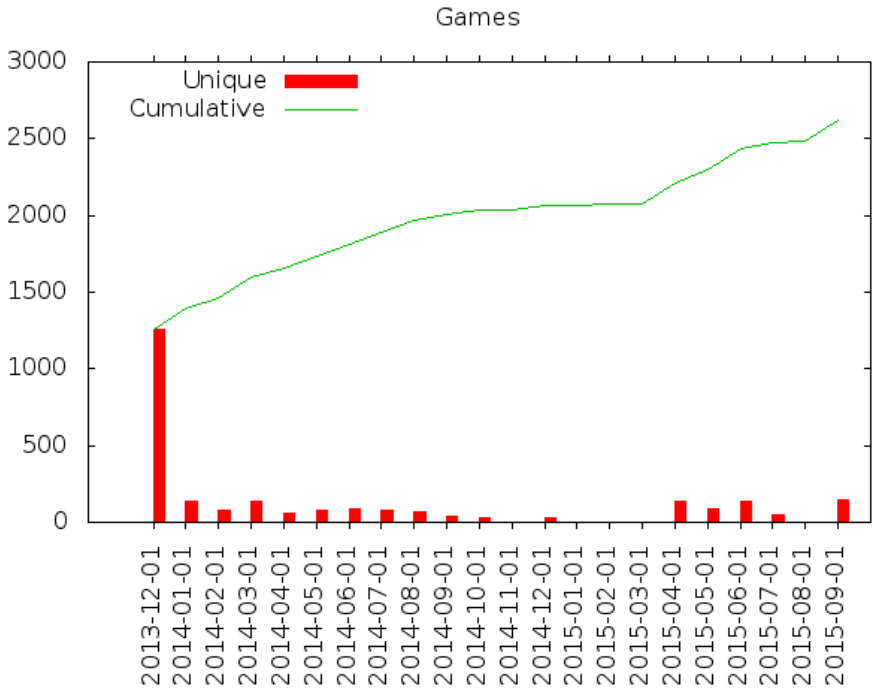


Figure 30. Game Levels Completed Over Time

4.2 Comparison with State of the Art

Proof by Games faced general limitations imposed by automated model checking and SMT solvers, and specific constraints within the PBG system itself (see 3.3.7). Despite the known issues with traditional brute-force approaches to refining model checking results, such approaches will inarguably excel at certain classes of problems – especially given increasingly powerful computer systems and increasingly clever heuristics.

On the other hand – while model checking is largely an automated process, it does commonly address algorithmically unsolvable questions, and hence there is need for human guidance. During the project we compared the deployment of the Ghost Map security properties to the best current freely available model checking tool, which is CPAchecker from the University of Passau [7]. There are Ghost Map security challenges involving loops where it is possible to produce an infinite sequence of traces, such that each time an edge is removed, a new longer trace which goes around the loop at least one more time is formed. For example, consider the following code:

```
int main( )
{
  int LOCK = 0;
  int i;
  int j;
  for (i = 0; i <= j; i++)
    { if ( i % 2  == 0 )
      { if (LOCK == 0) {LOCK = 1;}  else {goto ERROR;}
      }
        else
      { if (LOCK == 1) {LOCK = 0;} else {goto ERROR;}
      }
    }
  return (0);
  ERROR:
  return (-1);
}
```

The model checking problem is expressed above using the "GOTO error" construct which is the default approach in CPAChecker. It is straightforward to take MOPS-like specifications using FSAs and express them in this CPA syntax. The loop in this code has an unspecified end value, so abstract counterexamples can be arbitrarily long. CPAchecker cannot verify this code using its defaults setting, but correctness can be easily established using the Ghost Map game approach, provided the player choses the correct refinements.

4.3 Mechanical Turk

Near the end of Phase Two development, the PBG team conducted a test involving the Amazon Mechanical Turk (AMT) system, in which we offered "Turkers" (unskilled workers receiving a small payment for the completion of nominal tasks) the opportunity to play a customized, utilitarian version of Ghost Map: Hyperspace – with all game-related elements removed – for pay.

To our surprise, every game level that we posted to AMT was completed in less than a day, at a cost (set by us) of $1.00 each. Moreover, Turkers were required to complete the (substantial) game tutorial before they were allowed to receive any real game levels – and the fee for tutorial completion (again set by us) was a mere $0.01.

This was a remarkable result compared with the main Internet release (despite heavy activity at initial launch – see 4.1) where, during periods of relatively little activity, sometimes weeks would go by with no proofs generated.

4.4 User Testing and Interaction

This section reviews the forums in which the team engaged naïve players, whether they were recruited internally by the PBG team, provided by co-contractors across the CSFV program, or pulled from the public at large.

4.4.1 UCF Usability Tests.

BBN coordinated with subcontractor University of Central Florida (UCF) to execute two informal and two formal usability tests in the course of the PBG project. These tests spanned both project phases, and involved the recruitment of graduate students in psychology who experienced the game with no prior exposure.

The results suggested that while a small fraction of the subject pool was able to comprehend and succeed at the game, in general the participants experienced significant frustration just in comprehending the core concepts presented in gameplay. Although users generally liked the premise of the game and that it presented challenges to their critical thinking skills, this summary quote from the second study captures the ultimate sentiment unfortunately well: "Many participants that did not describe the game as fun or engaging noted that they felt the game had the potential to be engaging and fun, but they could not enjoy it because they were too focused on trying to understand the goals and objectives of the game."

One of the significant challenges that the game design team faced was creating a tutorial that would convey everything users needed to play the game successfully, above and beyond the basic challenge of simply conveying a great deal of information in a small amount of tutorial content. It was critical to make it clear that playing real levels would involve a high degree of uncertainty, and that players would presumably be applying heuristics, but never knowing without a doubt that a given move was guaranteed to succeed. The team did improve the tutorial content to make more clear what heuristics might be applied (and that they were nonetheless never guaranteed to work), as a result of the final usability test and the initial guru event. Although this updated content itself has not undergone further usability testing, it was presented at the final guru event, where it received a positive and encouraging response.

4.4.2 YouEye Usability Tests.

CSFV co-contractor GameDocs executed two remote testing sessions with usability testing service YouEye. In these sessions, a pool of test subjects designed to represent the general population was assembled and introduced to the game, with no prior exposure.

The results of these tests generally reflect what we saw in the UCF usability tests: a small fraction of the subjects understood the core concepts in the game and managed to succeed, but the majority of testers expressed frustration with the complexity of the concepts the game presented and the effort required to learn how to play.

4.4.3 Guru Events.

On two occasions near the project's end, the CSFV program organized "Guru Events" where those players from the general public who had performed most effectively at the CSFV games were invited to attend a special event designed around them. These events included presentations from the government and game teams, gameplay observation, and general discussion. Oddly enough (as a result of the vagaries of bureaucracy and scheduling), each guru event involved only a single guru, and in both cases the guru was a player of Ghost Map: Hyperspace. In any case, the essential result of these events was to make clear this critical fact, that we had already begun to suspect: guru players approach Proof by Games much differently than casual players – and casual players simply don't stand a chance.

During incredibly valuable in-person gameplay observation and subsequent interviews and discussion, these key points arose repeatedly:

- The game needed more clear and explicit instructions for how to interpret and utilize the provided variable dependency clues in order to guide player actions. We added this information after the first guru event and validated it successfully at the second.
- Guru players tend to be drawn to the "Citizen Scientist" aspect of the system, taking an interest in the underlying problem, and taking a kind of ownership of the system so far that even flaws become a source of engagement, since gurus can imagine – and will happily convey – ideas for how to make it better.

The combat gameplay element – while providing nothing of direct value toward solving the real-world verification challenges – proved a key source of engagement even for guru players.

5. CONCLUSIONS

In this section we outline and discuss what we learned in the course of Proof by Games.

5.1 Public Release

Although we were ultimately unable to apply the Proof by Games system to the sort of large-scale software artifacts that our original targets (BIND, Linux kernel) represented (see discussion regarding the constraints on scalability in Section 3) we were nonetheless heartened that the smaller-scale programs we did deploy as PBG content drew thousands of players to engage with the system and ultimately generate 2,592 proof artifacts.

5.2 Mechanical Turk and Tools for Experts

The strong Amazon Mechanical Turk (AMT) test results suggest that AMT represents a viable alternative to game-based crowd-sourcing, in some contexts. Here, as in the case of the main gameplay results, the key remaining question is to what degree the content could be scaled up while retaining the strong performance and remarkably low cost of the system.

The AMT test also afforded the PBG team the opportunity to consider what Ghost Map: Hyperspace would look like as an engineer's utility, as opposed to a crowd-sourced game. In addition to supporting the AMT test, this was an interesting potential first step toward mutating the PBG system into a tool usable by experts or programmers to assist in checking their code and in which they have complete access to the underlying code base. We imagine a future in which analysts receive training in "edge removal" for such complex situations as endless loop unrolling to provide intuition and prioritization for expensive machine operations.

5.3 General Applicability of PBG Graph Manipulations

As discussed in Section 3.3, the PBG design required the development of mathematical mechanisms to create homomorphic but more precisely manipulable variations of the original control flow graphs output from the MOPS system. These graph manipulations can produce variants of the CFG that preserve the set of execution paths found in the original CFG, but are more amenable to abstract analysis and automated formal reasoning. For example, by unwinding loops within a CFG, it is possible to isolate individual execution paths and construct corresponding logical formulas that can be proven false automatically using an SMT solver, while the logical formulas derived from the original CFG cannot. In some cases, the formulas derived from transformed graphs may also be simpler, and thus can be proven false more quickly in practice.

5.4 Guru-Based Crowd-Sourcing Model

Ultimately, our experience interacting directly with users chronicles a significant shift over time: initially, we aimed to appeal to casual gamers, but we struggled to appeal to that demographic due to the inherent mathematical complexity of the system. Eventually, we came to embrace a "guru"-based model, marketing the "Citizen Scientist" aspect of the game, with the understanding that a small but highly motivated fraction of the crowd would provide the vast majority of useful output.

User testing and guru events together told a story that we had partially anticipated ("Casual players will struggle with the complexity of Proof by Games; focus on the Guru players") but not fully appreciated ("On the other hand, Guru players are so much better than casual players that you should take full advantage of their skills; do not underestimate them.")

The very nature of the Proof by Games system involves a unit of gameplay (a MOPS witness trace of execution through a complete piece of software) that is conveniently well defined but relatively large. For this reason, we ultimately had no practical choice but to adopt a guru model, in which we present the game to a large population and then seek out that small fraction of users who can comprehend the full picture the game presents.

However, our UCF and YouEye usability test results skewed our perspective and caused us to hold out hope for the casual player nonetheless. As a result, Ghost Map: Hyperspace abandoned the graphical presentation of the FSAs used to encode the vulnerabilities whose absence we seek to prove (since usability tests in phase one suggested that few players understood what the presentation of those FSAs meant), even though in the later Guru Events we learned that for a guru such presentations are not in fact problematic.

If we were to continue Proof by Games development, we would maintain the present focus on the guru model, and present all the information available in both the Phase One and Phase Two games, without worrying about those non-guru players incapable of comprehending all of that information.

5.5 Problem Transformation

Consider two dimensions of problem specification related to transformation.

5.6 Dimension 1: Degree of Partitioning

Even within the CSFV program, we see significantly different degrees of partitioning. Games like Xylem and Monster Proof, seeking loop invariants, presented relatively small pieces of work to users; games like Paradox and Ghost Map required users to consider an entire code base (albeit compressed in various ways) at once. If (as in the case of CSFV) one sets an explicit goal of transforming the problem at hand – and to the degree that one considers partitioning a problem in itself transforming it – this arguably puts greater burden on approaches that perform less partitioning, since they must provide some other transformation mechanism.

5.7 Dimension 2: Problem Definition Precision

We can readily define some problems (traveling salesman) mathematically, while others (language translation) are not so readily encoded. Those that are amenable to mathematical definition – as in CSFV – are naturally good candidates for automated solvers; in this case generally only NP-hard problems are good candidates for the kinds of approaches in which problem transformation is required. Fuzzier problems – those that require knowledge about the world and/or more open-ended creativity to solve – typically demand a similar sort of open-ended creativity to transform (and the transformation itself may be fuzzy), even while they may be good candidates for un-transformed crowdsourcing. Consider the DARPA Network Challenge as an example of a transformed fuzzy problem: DARPA wished to explore the problem of rapid

social network deployment, so presented the motivating problem of finding physically distributed red balloons – a problem whose solution entailed solving the real problem.

5.8 Problem Transformation in CSFV and PBG

It is still possible that crowd-sourced gameplay may help with software verification, but identifying software verification challenges that are amenable to a game representation is challenging. It is necessary to identify problems that must be solved by existing validation, verification, and analysis tools for which the best-known algorithms (including the best known heuristic optimization algorithms) are impractical (e.g., exponential time in the average case). Furthermore, the problems must either be compact themselves, or it must be possible to divide them into compact problem parts (for conversion to levels that humans can handle). Finally, there should ideally be some information available about each instance of a problem that existing algorithms and tools cannot currently incorporate, but which gives some information about the problems (e.g., data flow relationships between variables). Humans may be able to incorporate such information in novel ways to find solutions to existing problems.

PBG successfully incorporated a collection of software verification artifacts to create an interactive game that presented to non-expert players portions of software verification problems as levels. The particular static analysis technique that we chose had some qualities that made it amenable to conversion to a game (control flow graphs and data flows on those graphs are straightforward to visualize and manipulate without knowledge of their underlying meaning). Unfortunately, the technique itself (and the tools associated with the technique) had some limitations in terms of coverage of language features and scalability to large software. It is possible that these issues could be overcome using the same overall design if the software development effort were scaled up substantially.

One inherent difficulty with the technique employed (as well as any other potential technique for converting verification problems to game levels) is that hiding the true meaning of the problem being solved while obtaining useful player input is difficult: if enough information to solve the problem is supplied, then the player may be able to infer the true meaning of the problem. The opacity constraint is particularly problematic because it is not possible to supply a variety of different kinds of information to users and let them decide (or, perhaps, collectively rank) what information may actually be useful. In PBG we faced a low-degree-of-partitioning problem with a mathematically precise definition, and we attempted to construct a lossy transformation to present the problem to the crowd. That lossiness was a fundamental problem for us.

Another inherent difficulty is that the vulnerability in a particular piece of software may be caused by interaction between many disparate components. Dividing the problems into small chunks may not be a useful exercise if the problems are generated from information about a large portion of a software artifact, or from mutually interdependent analyses that span many different components. This means that either only local vulnerabilities can be detected or dismissed by players of a game (which can usually be solved automatically, anyway), or that games must present a large and cumbersome amount of information to players.

6. RECOMMENDATIONS

Based on the conclusions drawn in the previous section, we end this report with the following summary recommendations regarding possible future work based on Proof by Games.

System Scalability

Perhaps the most significant question still to pursue in the PBG system as it stands is to what extent it can be scaled. If limitations on program size and complexity – and perhaps most significantly, problem transformation – can be overcome (as model checkers and SMT solvers become more robust over time and/or if the constraints on problem transformation can be eased or otherwise managed), it would be interesting to evaluate the performance of the crowd on larger levels representing potential security violations in real-world industrial software. This applies both in the Amazon Mechanical Turk world and in the world of gamers on the public Internet.

Cognitive Study of Clue Data

One possible follow-on activity addressing the problem transformation challenge might study how players use the clue information that informs their decisions. This would involve taking complex tasks and portraying the information players need for solutions in many ways, and then testing which are most effective and for which types of players – especially with larger-scale content.

Social Elements for Player Retention

Given the challenges PBG faced in retaining players after the marketing campaign for the initial public launch, it would be interesting to investigate the effects of additional social elements in the game design and surrounding infrastructure: for example, mechanisms to allow for shared problem-solving (in real time or asynchronously) online; division of the math work into component parts that could be performed and shared independently; and the establishment of a strongly moderated, Stack Overflow-style forum.

Visual Tools for Navigating Code and Static Analysis

The Amazon Mechanical Turk experience showed that the de-gameified PBG client application can be used effectively by non-experts without any game-based motivation, without knowledge of the nature of the application they are analyzing, and without access to its original source code or other supporting material (specifications, commentary, or other documentation). Especially given the growing awareness among software developers generally of the power and importance of static analysis combined with their persistent reluctance to invest in its use, it would be very interesting to provide software developers with static analysis tools augmented by the visual interface PBG provides. This “Gordian Knot” solution to the challenges posed by the problem transformation constraints opens an entirely new and exciting avenue of pursuit.

References

[1] R. Watro, K. Moffitt, T. Hussain, D. Wyschogrod, J. Ostwald, D. Kong, C. Bowers, E. Church, J. Guttman, and Q. Wang, “Ghost Map: Proving Software Correctness using Games,” Lisbon, Portugal, 2014, pp. 212–219.

[2] K. Moffitt, J. Ostwald, R. Watro, and E. Church, “Making Hard Fun in Crowdsourced Model Checking: Balancing Crowd Engagement and Efficiency to Maximize Output in Proof by Games,” in 2nd International Workshop on CrowdSourcing in Software Engineering (CSI-SE 2015), Florence, Italy, 2015.

[3] D. Dean, S. Gaurino, L. Eusebi, A. Keplinger, T. Pavlik, R. Watro, A. Cammarata, J. Murray, K. McLaughlin, J. Cheng, and T. Maddern, “Lessons Learned in Game Development for Crowdsourced Software Formal Verification,” in 2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE’15), Washington, DC, 2015.

[4] D. Wyschogrod, A. Lapets, and R. Watro, “The Loop Paper,” Jul-2015.

[5] R. Watro, L. Kennard, and S. Watro, “Playing the Subset Coloring Game,” Jun-2015.

[6] Beyer, Dirk, et al. "The software model checker Blast." International Journal on Software Tools for Technology Transfer 9.5-6 (2007): 505-525.

[7] D. Beyer, Dirk and M. E. Keremoglu, “CPACHECKER: A tool for configurable software verification,” in Computer Aided Verification, Springer Berlin Heidelberg 2011, pp. 184–190.

[8] Bobot, François, et al. "The Alt-Ergo automated theorem prover, 2008." (2013).

[9] Barrett, Clark, et al. "Cvc4." Computer aided verification. Springer Berlin Heidelberg, 2011.

[10] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, “Efficient algorithms for model checking pushdown systems,” in Computer Aided Verification, 2000, vol. 1855, pp. 232–247.

[11] P Cousot & R Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”, In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238—252, Los Angeles, California, 1977. ACM Press, New York.

[12] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in OSDI, 2008, vol. 8, pp. 209–224.

[13] Anderson, P. The use and limitations of static-analysis tools to improve software quality. CrossTalk: The Journal of Defense Software Engineering, 18-21 (2008). https://buildsecurityin.us-cert.gov/sites/default/files/0806Anderson_0.pdf

Appendix A. Ghost Map: Proving Software Correctness using Games

DISTAR Case #22556

Approved 3/18/2014

Ghost Map: Proving Software Correctness using Games

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.ed

***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 counter-example 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.

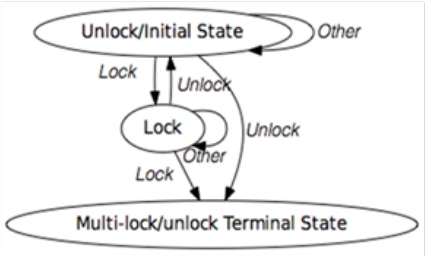


Figure 1. Finite State Automaton (FSA) for lock/unlock software errors.

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.

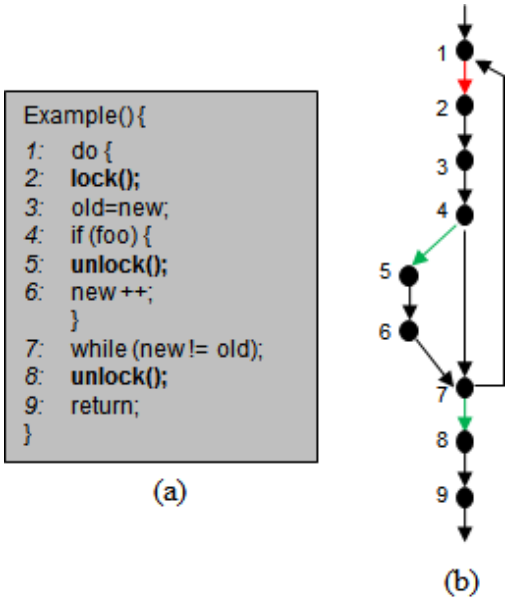


Figure 2. Test program (a) for lock-unlock analysis and corresponding CFG (b).

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.

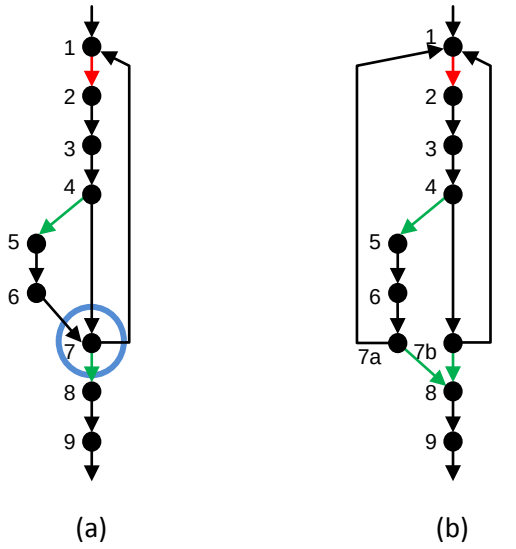


Figure 3. Illustration of cleaving operation.

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.

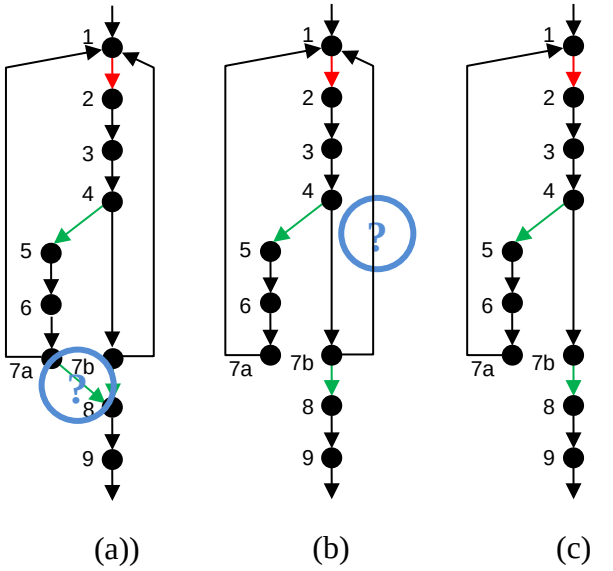


Figure 4. Illustration of edge removal to produce a CFG containing no counter-examples.

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 counter-example 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 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.

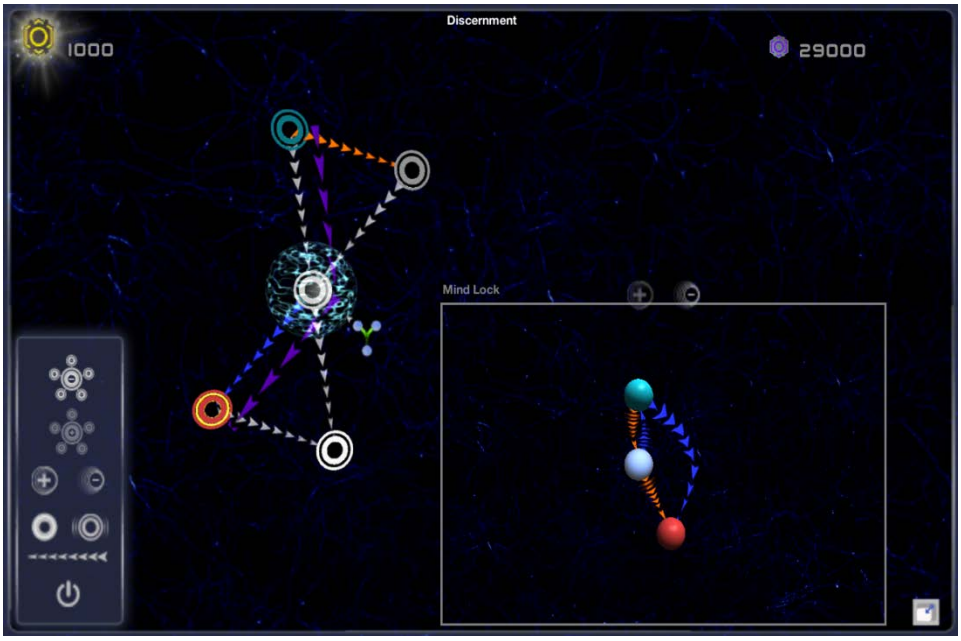


Figure 5. The primary game screen for Ghost Map.

- 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 penalizes 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.

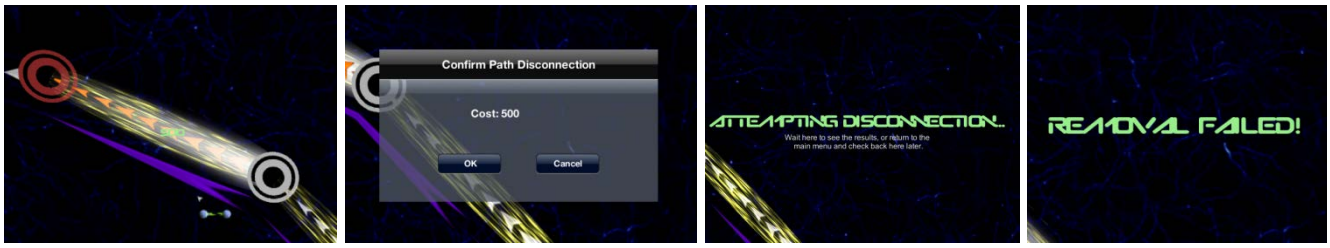


Figure 6. Action scenes from the Ghost Map game (figures 6a through 6d).

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.

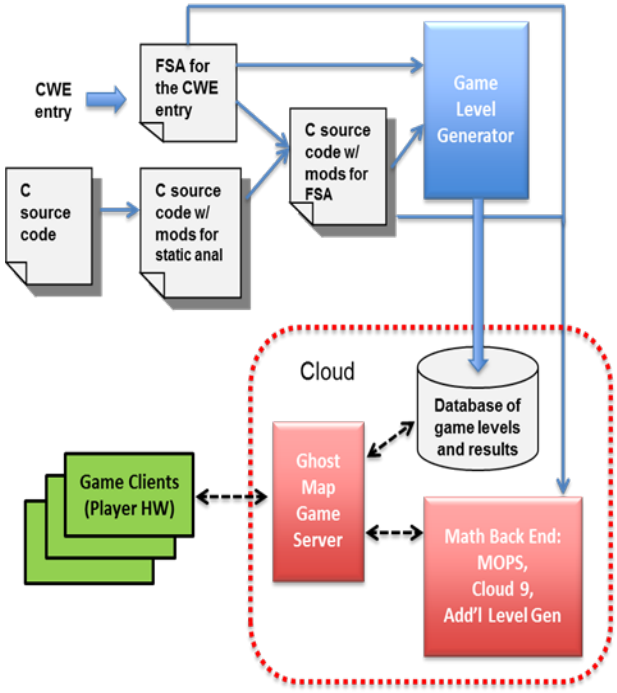


Figure 7. The Ghost Map game system architecture.

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.

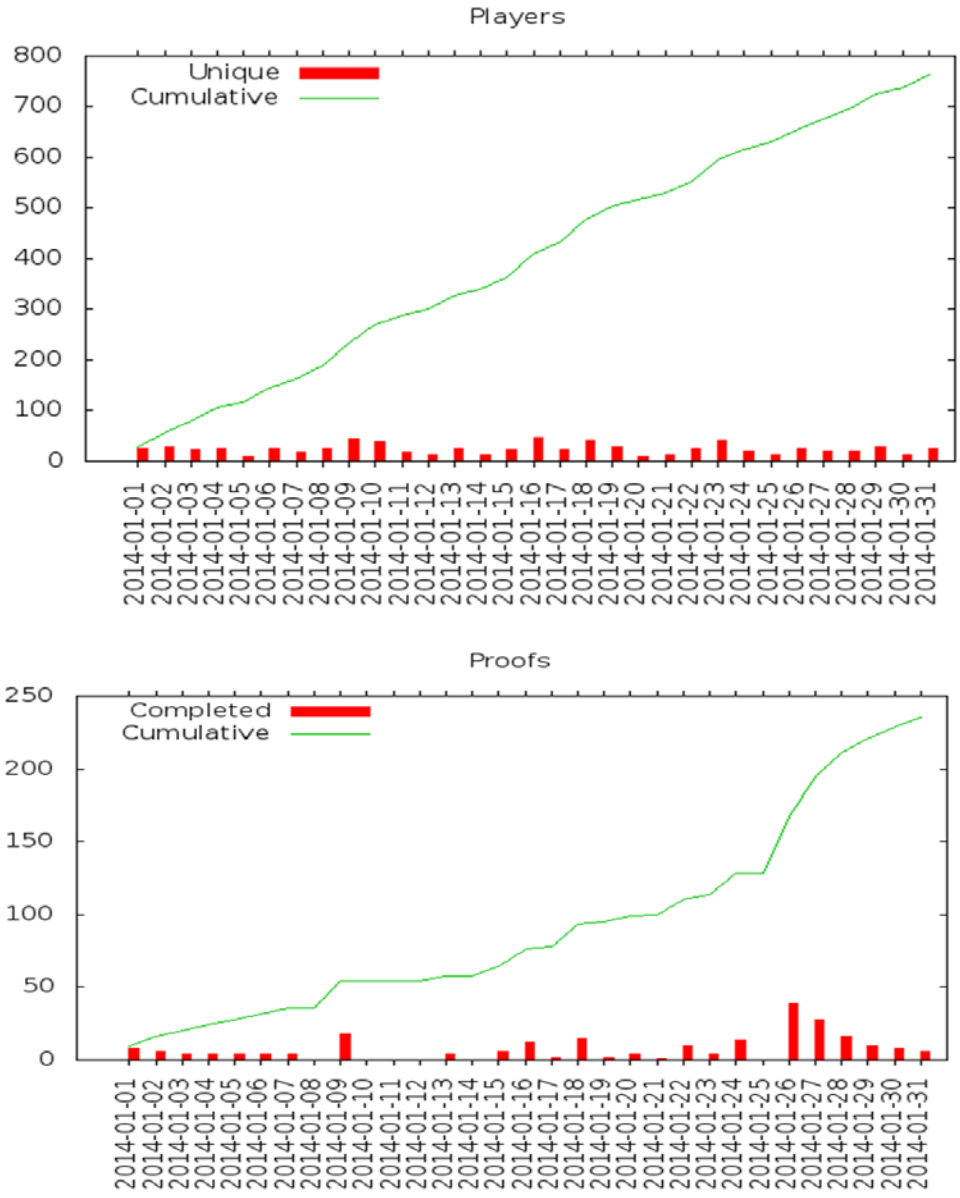


Figure 8. Ghost Map player and proof data from January 2014.

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.

Appendix B. Making Hard Fun in Crowdsourced Model Checking

DISTAR Case #24096

Approved 1/28/2015

Making Hard Fun in Crowdsourced Model Checking

Balancing Crowd Engagement and Efficiency to Maximize Output in *Proof by Games*

Kerry Moffitt, John Ostwald, Ron Watro
Raytheon BBN Technologies
Cambridge, MA, USA
kmoffitt, jostwald, rwatro@bbn.com

Eric Church
BreakAway, Ltd.
Baltimore, MD, USA
echurch@breakawayltd.com

Abstract—We describe *Proof by Games*, a system that mathematically proves software to be free from certain defects by transforming the refinement of formal model checking results into a public, browser-based, crowdsourced game. We then introduce *Ghost Map: Hyperspace*, the second-generation game in the *Proof by Games* system, and describe how we are introducing game elements whose sole purpose is to increase the level of engagement. This may reduce player efficiency but should increase crowd size and engagement – and ultimately increase the total level of crowd contribution to the real-world task.

Index Terms—browser-based, CEGAR, crowdsourcing, formal methods, model checking, software, video games

I. Introduction

Faced with the significant cost of vulnerabilities introduced by flaws in computer software, developers of mission-critical software applications have for decades employed *formal methods* capable of proving the absence of such flaws mathematically. The cost of applying such methods is also significant, however – and for this reason, only a small fraction of the world's software is privileged with the assurances that formal methods provide. We built – and continue to develop – the *Proof by Games* system (see the first-generation game at ghostmap.verigames.com) in order to tap into the significant mental energy expended every day by online gamers solving challenging puzzles in entertainment games. By presenting in an entertainment game context the logical puzzles that naturally emerge from the application of formal methods, we transform gamers' desire and ability to solve such puzzles into real-world work proving software correctness. In this paper, we describe specific design elements introduced in the second-generation *Proof by Games* release (*Ghost Map: Hyperspace*, in development) aimed at leveraging narrative and gameplay to trade player efficiency for crowd size and engagement, thereby ultimately increasing overall crowd output.

II. System Overview

In *Ghost Map: Hyperspace*, the player assumes the role of a space mercenary preventing aliens from invading through rifts depicted on hyperspace maps. Each time a player secures a rift, the player scores points in the game and, invisibly, creates a partial proof about the correctness of a piece of software.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited.)

The game is built on the MOdelchecking Programs for Security properties (MOPS) tool [1]. MOPS checks C software for known software flaws, such as those found on the SANS/MITRE Common Weakness Enumeration (CWE) Top 25 list [2]. Each level in the game is a puzzle that represents a potential counterexample to correctness (i.e., a potential flaw) found by MOPS. Through the gameplay, players investigate and manipulate the data constraints associated with the potential counterexample in order to establish that it is spurious. In this way, *Ghost Map: Hyperspace* extends MOPS with a counterexample-guided abstraction refinement (CEGAR) model [3], 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 initial (high) level of abstraction.

When playing *Ghost Map: Hyperspace*, the player is presented with a representation of the Control Flow Graph (CFG) of the target software, annotated with a counterexample trace from MOPS and information about the scopes and interactions of the variables in the original program (in anonymized form). The player's job is to decide, based on all this information, what segment of the trace should be more closely examined for contradictions in the variable values that would be necessary for its formation. This segment is then sent to a Satisfiability Modulo Theories (SMT) solver, which attempts to verify a contradiction. For this to succeed, the segment selected must be long enough to include all relevant variable interactions, but short enough that the solver can complete its processing in a reasonable amount of time.

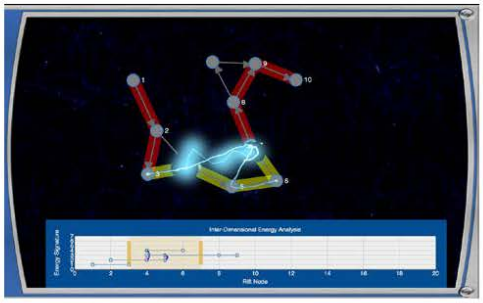


Figure 1: A tutorial level in *Ghost Map: Hyperspace*, showing the annotated CFG (top) and variable information (bottom)

III. Engagement and Efficiency

Consider the following simplistic characterization of the magnitude of output of a crowdsourced system:

$$totalOutput = playerTime \times playerEfficiency$$

We can think of *playerTime* as the product of the number of players (largely a function of marketing, including word-of-mouth) and the average time commitment per player (a function of engagement, as well as quantity of content).

The concept of *playerEfficiency* may be taken to include a broad range of complex dynamics, but let us again adopt a simplistic characterization for the current purpose: consider *playerEfficiency* the degree to which player effort is put directly to generating the results that the system ultimately seeks. A value of 0 would yield a "pure entertainment" game, while a value of 1.0 would yield something more like a graphical tool designed for the experts who actually generate the desired results for a living.

Based on the success of systems like *FoldIt* [4], the first generation of *Proof by Games* aimed for a very high *playerEfficiency* value, exposing the underlying mathematics to players with only a thin layer of narrative surrounding it. Although we were pleased to see over 1,000 users and nearly 1,000 small proofs completed in the system's first three months online, one might nevertheless suppose, given the level of success enjoyed by *FoldIt*, that folding proteins is inherently more compelling to average players than generating formal proofs about computer software.

One important difference between *FoldIt* and *Proof by Games*, however, is that *Proof by Games* is not at liberty to share with the player all the information at its disposal. In particular: because we allow for the possibility that *Proof by Games* will be used with proprietary software, we do not assume that we can distribute the target software source code.

In the second generation of the *Proof by Games* system, we aim to engage players more fully despite these challenges. We have identified the following problems and solutions:

Problem: The game tutorials take the player as quickly as possible to the real math work, providing scant opportunity to build up any sense of investment in the game for players who are not inherently attracted to and skilled at the core work.

Solution: Allow and encourage the player to explore the full range of affordances that the game provides – both by bringing them out more fully in their own tutorials (e.g. focused on manual graph layout to augment the automated layout, which many players claim to enjoy in itself) and by allowing players some agency in how the tutorials are structured (e.g. by spending extra time on one topic, whether because it's particularly challenging or simply because it's particularly engaging). Also, make better use of the tutorials to introduce and integrate the game narrative, which provides a source of engagement in itself, as well as the framework of a mental model on which players can hang the key gameplay concepts required for success in the game. We hope and anticipate that this will provide players with a maximal sense of investment in the game by the time they play real levels.

Problem: The submission of careless selections comes with only an abstract cost (potentially reduced score), and therefore limited commitment to doing otherwise – and ultimately limited satisfaction with gameplay.

Solution: Require players to build up credit before they can afford to submit selections for back-end processing – not by forcing them to engage in arbitrary, unrelated gameplay, but by rewarding them with credits and graphical flairs for performing the kinds of actions (graph organization, variable analysis) that they need to perform anyway in the course of successful play.

Problem: Back-end processing takes time – sometimes minutes for a single submission. While we were happy in the first-generation system to allow players to jump into another level while processing proceeded on the first one, this can break flow and damage the mental model that the player has built of the first level (and they may well still need that model, if their submission fails to complete the level outright).

Solution: Offer players a "pure entertainment" (*playerEfficiency* = 0) mini-game that engages a cognitive process similar to the central game, to maintain interest and interaction – right on the graph from which they just submitted their selection. Although this solution carries the most risk of reducing player efficiency, we expect that it will ultimately increase overall player engagement and output – especially in cases where back-end processing completes relatively quickly.

IV. Conclusion

We have introduced *Ghost Map: Hyperspace* and some design enhancements that should increase crowd size, engagement, and contribution to the task of proving computer software free from certain vulnerabilities. We look forward to reporting on the results of this work after the public release of *Ghost Map: Hyperspace* in the summer of 2015.

Acknowledgment

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.

The authors also acknowledge the broader Crowd Sourced Formal Verification team for their contributions to this effort.

References

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

[2] Christey, S., Brown, M., Kirby, D., Martin, B., & Paller, A. (2011). CWE/SANS Top 25 most dangerous software errors. *2011-06-29]. http..//cwe, mitre, org/top25/: CWE-79.*

[3] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50(5)*: 752-794.

[4] Khatib, Firas, et al. "Algorithm discovery by protein folding game players."*Proceedings of the National Academy of Sciences* 108.47 (2011): 18949-18953.

Appendix C. Lessons learned in game development for crowdsourced formal verification

DISTAR Case #25125

Approved 8/10/2015

Lessons learned in game development for crowdsourced formal verification

DREW DEAN[1], SEAN GUARINO[2], LEONARD EUSEBI[2], ANDREW KEPLINGER[3], TIM PAVLIK[4], RONALD WATRO[5], AARON CAMMARATA[6], JOHN MURRAY[1] JOHN CHENG[7], THOMAS MADDERN[7],

[1]*Computer Sciences Laboratory*
SRI International
{ddean, jxm}@csl.sri.com

[2]*Human Effectiveness*
Charles River Analytics
{sguarino, leusebi@cra.com}

[3]*Left Brain Games*
andrew@circuitbot.net

[4]*Center for Game Science*
University of Washington
pavlik@cs.washington.edu

[5]*Raytheon BBN*
rwatro@bbn.com

[6]*VoidALPHA*
aaron.cammarata@voidalpha.com

[7]*Information Innovation Office*
Defense Advanced Research Projects Agency
{john.cheng.ctr, thomas.maddern.ctr@darpa.mil}

1. 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[2].

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[3] 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[4], we derive a direct cost of $1 billion for the verification effort. In those intervening 3 years, Linux

[2] http://www.h-online.com/open/features/What-s-new-in-Linux-3-10-1902270.html

[3] http://royal.pingdom.com/2012/04/16/linux-kernel-development-numbers/

[4] http://money.usnews.com/careers/best-jobs/salary

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 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.

2. Circuitbot/Dynamakr

Authors: Andrew Keplinger[1], Mathew Barry[2], J. Nelson Rushton[3], Greg Izzo[1] & Qianji Zheng[3]

[1]*Left Brain Games*

[2]*Kestrel Technology*

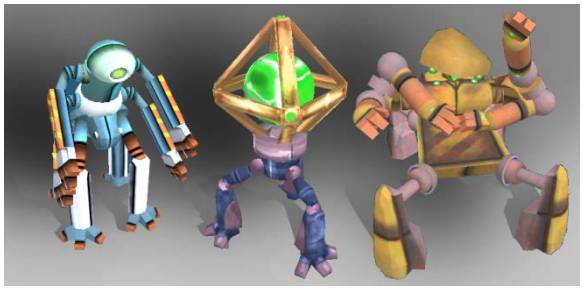
[3]*Department of Computer Science, Texas Tech University*

2.1 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.

2.2 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.

2.3 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.

2.4 Circuitbot

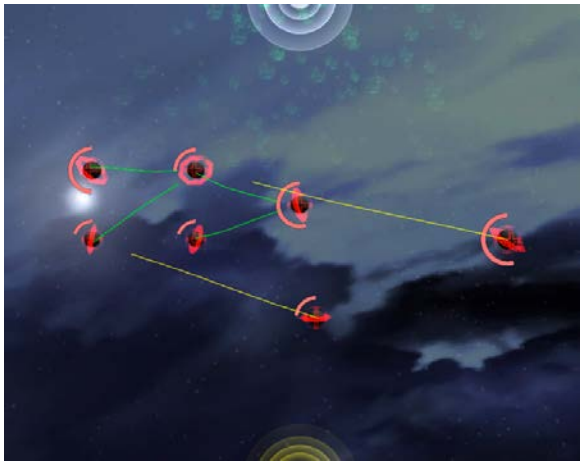
The Circuitbot game employs a turn-based strategy in which the motivational system drives the player back to the "work" part we want accomplished for verification. The universe of Circuitbot is the near future exploration and exploitation of near Earth asteroids, along with the development of a space 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

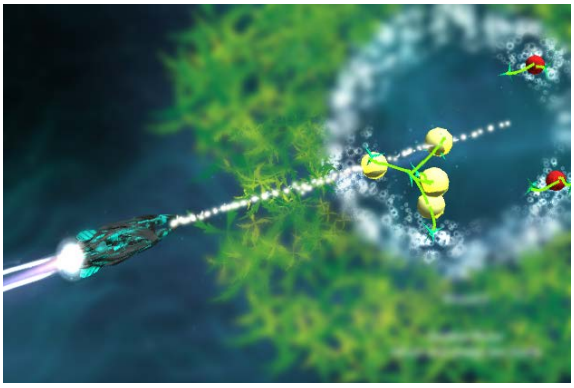
analysis of player-generated results back into the verification backend. After much analysis and some reworking on the software analysis side, we realized that we were looking at the information too narrowly. We would receive a game level that represents too small a portion of the software; and focusing 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.

2.5 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.

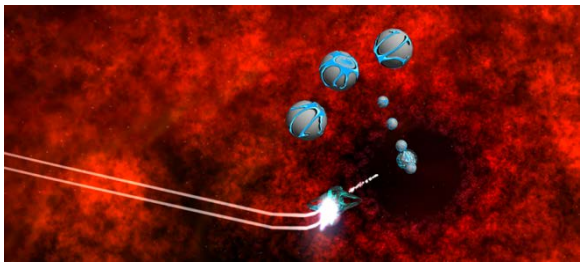


The reward game became Dyna-makr. Conceptually, Dynamakr is a quantum level 3D printer. Inside the Dynamakr the player examines patterns and feeds them into the Dynamo. The player first takes on the challenge of finding patterns that will produce the most energy in the Dynamo, as sometimes patterns will 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 autosolver. 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.

3. Flow Jam and Paradox

Authors: Tim Pavlik[1], Craig Conner[1], Jonathan Burke[1], Matthew Burns[1], Werner Dietl[2], Seth Cooper[3], Michael Ernst[1], Zoran Popović[1]

[1]*Center for Game Science, Computer Science & Engineering, University of Washington*

[2]*Electrical and Computer Engineering, University of Waterloo*

[3]*College of Computer and Information Science, Northeastern University*

3.1 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.

3.2 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.

3.3 Paradox Game Play

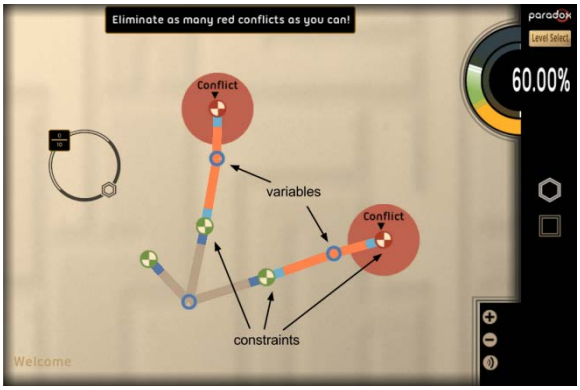


Figure 2-31: Paradox variables, constraints, and conflicts.

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.

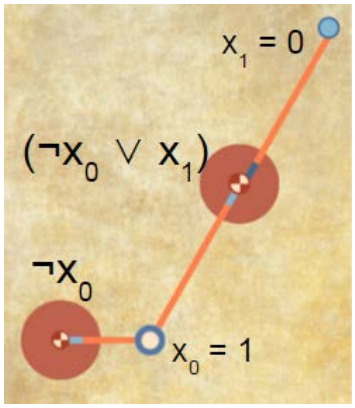


Figure 2-32: A Paradox level representing the formula: ¬x0 ∧ (¬x0∨x1). The red circles represent conflicts are shown for the unsatisfied constraints involving variables x0 and x1.

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.

3.4 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.

Figure 2-33: A previous version of the game, Flow Jam, required players to adjust variables individually.

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.

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.

3.5 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.

3.6 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.

3.7 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.

3.8 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. 2 developers annotated a program, one starting from unannotated source code and one startingfrom 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).

3.9 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.

4. Ghost Map/Hyperspace

Authors: Ronald Watro[1], Kerry Moffit[1], John Ostwald[1], Eric Church[2], Dan Wyschogrod[1], Andrei Lapets[1], Linsey Kennard[1]

[1]*Raytheon BBN*

[2]*BreakAway Games*

4.1 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 [Martin, 2011 #21514] 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 [Chen, 2002 #21512][Chen, 2004 #21513]. 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.

4.2 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.

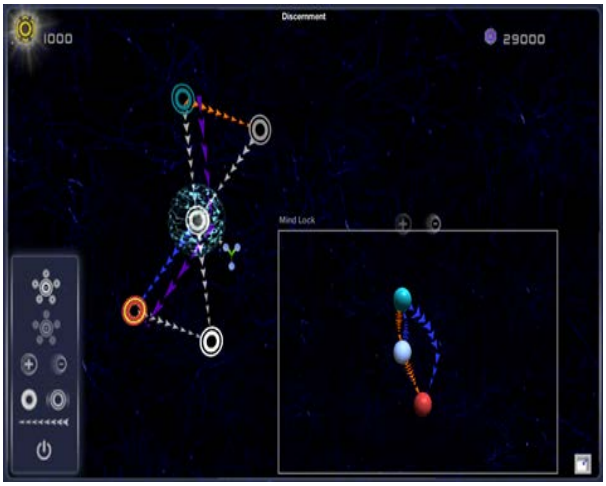


Figure 3-34: Simple Example of Ghost Map Level

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 [Watro, #21515] and at the Verigames web site.

4.3 Ghost Map Hyperspace

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.

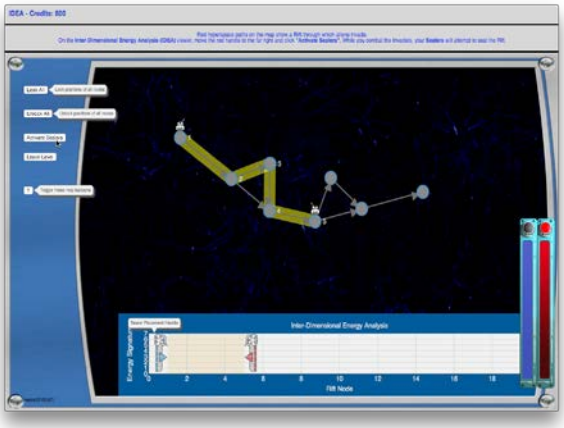


Figure 3-35: Example of a Ghost Map Hyperspace Level

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 back-end) 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 [Moffitt, 2015 #21511].

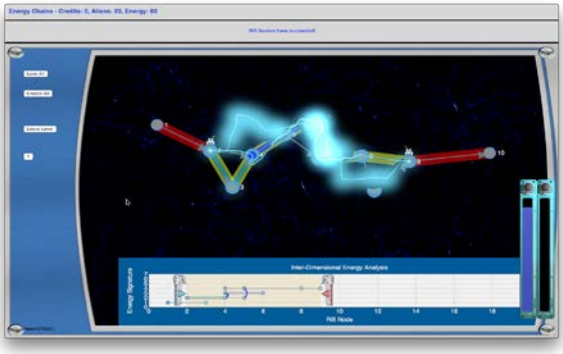


Figure 3-36: Using energy clues to seal rifts

4.4 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.

5. StormBound/Monster Proof

Authors: Aaron Cammarata[1] & Aaron Tomb[2]

[1]*VoidALPHA*

[2]*Galois, Inc.*

5.1 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.

5.2 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.

5.3 Game Descriptions

Figure 4-37:StormBound play screen

- Story-driven engagement
- "Magepunk" universe, blend of brass/steam and glowing magical runes
- Goal was to "completely hide the math": allow players to make assertions without any math or numbers in-game
- Targeted a broader, casual audience
- Used Unity Webplayer, embedded in a MeteorJS web page

Figure 4-38: Monster Proof Game Screen

- Resource-gathering and collection
- Cute cartoon monsters, emphasis on tongue-in-cheek humor
- Goal was to "completely show the math": give players tons of context, and focus on efficiency and comprehension
- Targets a focused puzzle-game audience
- Used Famo.us for HTML/CSS Sprites, and MeteorJS for web page / server

5.4 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.

5.5 Game 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.

5.6 Conclusions/Future Plans

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.

6. Xylem/Binary Fission

Authors: John Murray[1], Heather Logas[2], & Jim Whitehead[2]

[1]*Computer Science Laboratory, SRI International*

[2]*Department of Computational Media, University of California, Santa Cruz*

6.1 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.

6.2 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.

6.3 Game Descriptions

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

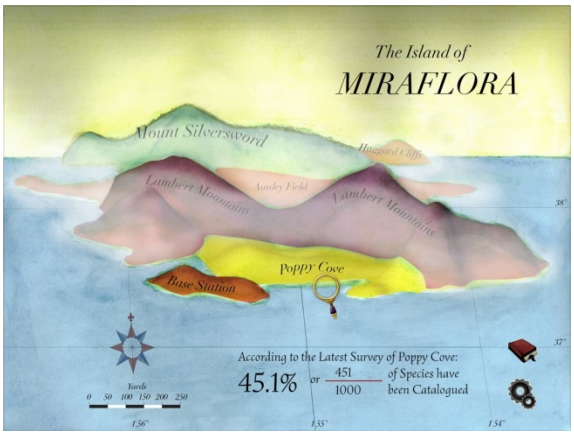


Figure 6-39:Xylem: Miraflora Island

Figure 6-40:Xylem: Floraphase Comparator

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.

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).

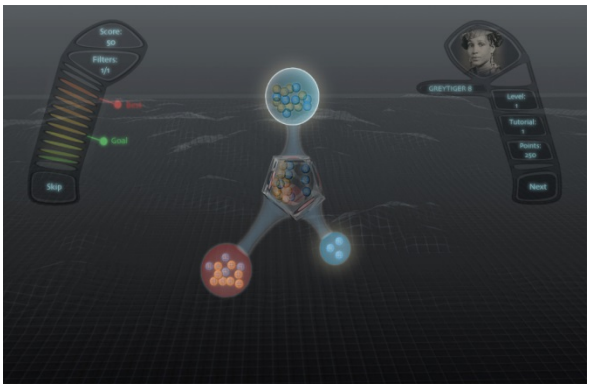


Figure 6-41:Binary Fission: Tree Structure

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.

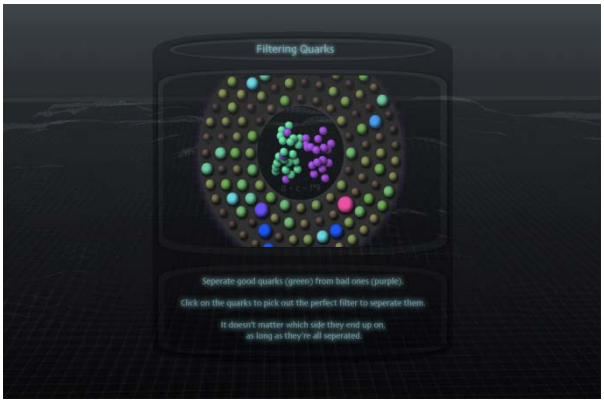


Figure 6-4:Binary Fission: Fliter Selection

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.

6.4 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 crowd-sourced 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.

6.5 Conclusions/Future Plans

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.

7. 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.

8. 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] Latham, D. C. (1986). Department of Defense trusted computer system evaluation criteria. Department of Defense.

[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.

Appendix D. Exploiting Information Flows in Model Checking for Software Validation

Exploiting Information Flows in Model Checking for Software Validation

Andrei Lapets Daniel Wyschogrod Ron Watro
Raytheon BBN Technologies, 10 Moulton Street, Cambridge, MA 02138, USA
{alapets, dwyschogrod, rwatro}@bbn.com

Abstract

One method for formally analyzing the safety and correctness of software programs written using imperative languages involves the detection of potentially dangerous paths through the program's control flow graph (CFG). Each path is found by matching the CFG against a finite-state automaton describing a policy violation. However, because this matching process does not take into account information flows within the program, it produces many false positives. One way to eliminate each path that is a false positive is to generate a logical formula for that path reflecting the information flows within it; if the formula is false, then the path does not represent a possible path through the program. Satisfiability modulo theories (SMT) solvers can be used to determine whether a formula is false, but running a solver on a formula is a costly operation. We describe a technique for reducing this cost by minimizing the complexity of the logical formulas submitted to the SMT solver; this minimization process is guided by knowledge of the information flows within the original program.

1. Introduction

Model checking is a tool that can help formally and automatically verify that software applications do not violate policies that specify their correct behavior. Model checking approaches [? ? ?] typically involve the construction of a simplified ("abstract") model of the software application, and then generating proofs that these abstract constructions satisfy specific properties associated with the correctness and safety of the original software application. However, analyzing even the simplified abstract models of the software applications often involves searching a large state space; this can make the validation process costly and possibly even intractable. Scaling validation techniques to real-world software is often impractical as a result.

Background and Motivation. [Dan/Ron] Some exposition about background. As successful exploits against critical cyber systems become ubiquitous, it becomes exceedingly important that vulnerabilities in software be eliminated at program design and implementation time. One tool for supporting software correctbess is the MOdelchecking Programs for Security properties (MOPS) [?]. MOPS analyzes software applications written in the C programming language for common software flaws, such as the SANS/MITRE Common Weakness Enumeration (CWE) Top 25 list [?]. The CWE list categorizes questionable coding practices that can lead to exploitable vulnerabilities. MOPS represents CWE entires (and other possible vulnerabilities) as abstract control flow patterns in Finite State Automata (FSA). Potential matches to a vulnerability are located by computing the product of the FSA and the control flow graph (CFG) of the sofwtare and determining whether it is non-empty using efficient algorithms [?]. This algorithm scales to CFGs obtained from large programs that have millions of lines of code, and can detect all potential violations in such CFGs.

A key techncial challenge in effectively applying MOPS-like tools is separating the list of potential security violations into actual issues and false alarms. The original MOPS approach relied on manual software engineer inspection to resolve the violation set. Other model checking tools such as BLAST and CPAchecker use an automated process for performing this analysis. In this paper we provide a new approach to violation analysis.

Organization. The rest of the paper is organized as follows. Section ?? provides a high-level context for the work presented in this paper and discusses related research efforts. Section ?? describes how violations can be detected in CFGs. Section ?? describes how false positives produced by the detection process can be eliminated, and how knowledge of information flows within the original source program can be used to make restrict the complexity of the resulting logical formulas, making the process of eliminating false positives more efficient. Section ?? describes how this approach is implemented within the context of a software validation tool, and provides some examples of how this technique was evaluated on actual software applications. We conclude and outline future work in Section ??.

2. Related Work

[Everyone] Relevant references about other software validation using these techniques, as well as the efficiency of SMT solvers on particular classes of formula.

The use of model checking to support software correctness has been vigorously investigated over the last fifteen years. The MOPS approach that models software vulnerabilities as FSAs is a convenient specification scheme but model checking can be applied to more general specifications as well. In particular, model checking often uses specificational assertions at each point in the control flow graph. These assertions are usually Boolean combinations of *predicates*, where a predicate is a Boolean expression built from program and specification variables.

There are a variety of techniques for analyzing the abstract counterexamples produced by model checking. One well-known approach is CounterExample Guided Abstraction Refinement (CE-

GAR), developed by Edmund Clarke and his students [?]. BLAST [?] and its successor CPAchecker [?] apply model checking and CEGAR to the software correctness challenge. Under CEGAR, the abstraction used in the model checking is refined based on properties of the counterexample in order to develop a more accurate model that may eliminate the abstract counterexample. CEGAR is applied together with attempts to show that the abstract counterexample is genuine. For example, in applications to software, test case generation tools such as Cloud9 [?] can be applied to confirm abstract counterexamples.

The CEGAR approach for software correctness is efficiently deployed using a lazy evaluation technique [?]. In this approach, refinement of the model is done by adding additional assertions to the control flow graph, but only at the nodes where these assertions are required, based on the analsyis of the abstract counterexample. The selection of the refinement assertions is part of the art of model checking, as will be discussed below.

The scalability of software verification techniques is always a challening issue. Henzinger and others [?] have proposed a technique where large blocks of software are encapsulated and reasoned about as single entities in the model checking. They show that this leads to performance gains.

Model checking is just one of many techniques for software verification during design and development. A recent new effort in this area is MergePoint [?], a software validation system that combines static and dynamic execution to support automated analysis of binaries on a large scale. This software in part relies on a backend SMT solver to eliminate infeasible execution paths.

One of the driving forces in the evolution of static analysis for software verification has been the advances made in the efficiency of automated theorem proving technology. Say more here ...

3. Background and Definitions

Let S be the set of all possible statement constructs within a programming language, and let $\mathcal{L}$ be a set of labels. We define a *graph* that represents (some or all) possible execution paths within a program to be a tuple $G = (V, E)$ where V represents statements (or basic blocks) that can appear on execution paths in the program and $E \subset V \times V$ is a set of directed edges between basic blocks. When a program is executed, control can only flow from node to node along directed edges, starting at a unique source node in the graph that we denote $v_0 \in V$. Let $\ell : V \to \mathcal{L}$ be a map that specifies a label for each node, and let $\sigma : V \to S$ be a map that specifies the particular statement construct that appears at a node. Define a *path* in G to be an ordered sequence of labeled nodes $\langle (v_{i_1}, \ell(v_{i_1})), (v_{i_2}, \ell(v_{i_2})), ..., (v_{i_k}, \ell(v_{i_k})) \rangle$ where $v_{i_j} \in V$ and where $(v_{i_j}, v_{i_{j+1}}) \in E$.

Definition 3.1. Given two CFGs G and G', we say that $G' \prec G$ if for every path p in G', p is also a path in G. If $G \prec G'$ and $G' \prec G$, then we say $G \simeq G'$.

Given a tuple $P = (G, v_0, \ell, \sigma)$, if no two nodes in G have the same label under ℓ and P represents a program, then we call P a *control-flow graph* (CFG) for that program. If there exists a $Q = (G', v_0', \ell', \sigma')$ where $G \simeq G'$ and G' contains no cycles, then we say Q is the *abstract reachability tree* (ART) corresponding to P (note that G' may have infinitely many nodes and edges).

4. Detection of Violations in Programs

[Dan] Description of MOPS approach and introduction of CFGs and CWEs. Discussion of cleaving and edge removal.

As a simple introduction to model checking of potential security vulnerabilities, we consider CWE-250, which as a violation on UNIX-like systems is represented in Figure ??. This property is

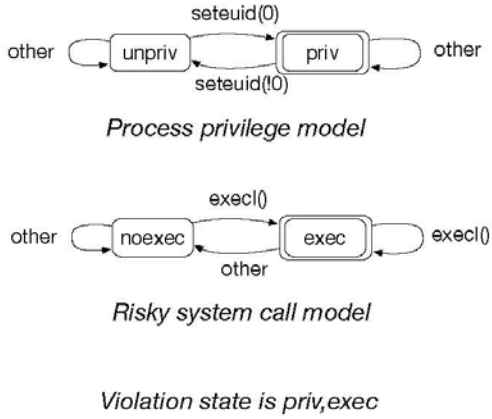


Figure 1. A pair of FSAs where the first represents the level of user privilige and the second involves a call to the system execl call.

an attempt to execute a system call at elevated (root) privilege. This is most easily represented as two separate FSAs as shown in the figure. The first FSA represents the transition from any effective user ID (euid) to an euid of zero which indicates root. The accepting state, indicated by the double border, corresponds to execution in a privileged state. The transition from the unprivileged state to the privileged state can only take place through the seteuid(0) call while a transition to an unprivileged state takes place through a call to *seteuid* with any non-zero argument. The second FSA corresponds to the execution of a command using the *execl* call. The requirement that system calls are not permitted at root privilege is potentially violated if both FSAs are in their respective accepting state simultaneously.

As can be seen from the CWE-250 example, code sequences can drive security violation FSAs to an accepting state. Yet, while the DISTP *(what is this ?)* approach is sound in that it does not overlook any false violations, it may produce false alarms. This limitation is due to that fact that CFG model checking only takes the CFG of a program and does not take data values into account. Thus, a particular hypothetical path through a CFG may lead to a terminal state of the security property FSA, but no actual values of the program variables involved would allow that path to be taken. That trace through the code constitutes a false alarm.

A simple, if contrived, example can be used to illustrate a CFG model checking produced false alarm trace. Consider the following program:

```
1:  int example (int x )
2:  {
3:    char * args[] = {"/bin/sh"};
4:
5:    while(x < 2)
6:     {
7:       if(x == 2)
8:       {
9:         seteuid(0);
10:      }
11:      else
12:      {
```

```
13:            seteuid(getuid());
14:        }
15:
16:        if(x == 3)
17:        {
18:            execv("/bin/sh",args);
19:        }
20:        x = x + 1;
21:    }
22: }
```

If we apply the FSA shown in Figure ?? to the above code, an application of CFG model checking produces the trace shown in Figure ??. From the trace in ??, it can be seen that we enter the loop at line 5, take the conditional branch at line 7, and then take the conditional branch at line 15. The resulting logical formula from this trace is:

$$x < 2 \wedge x \neq 2 \wedge x = 3 \tag{1}$$

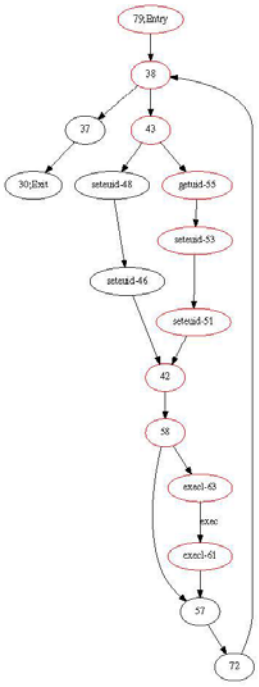


Figure 2. This false alarm trace is emitted by MOPS when the FSA in the previous figure is applied to the code in the text.

Obviously, no value of x can satisfy this logical formula. Since this logical formula is unsatisfiable, the sequence of branch choices taken in this trace do not correspond to any real values of x. In general, any sequence of expressions involving variables used in the trace that lead to a contradiction invalidates the entire trace. Thus, we do not have to submit a logical formula which accumulates all

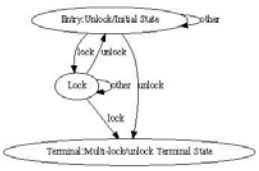


Figure 3. This FSA corresponds to the requirement that two locks in a row or two unlocks in a row can never be executed.

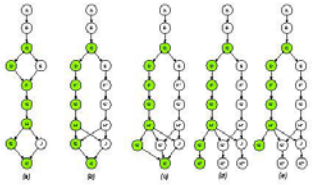


Figure 4. Starting with the trace in green, we perform a "cleave" operation to locally transform the CFG to an ART.

variables along the path to find a contradiction. Since the logical formula consists of the conjunction of expressions, any set of expression from any segment of the trace that leads to a contradiction will invalidate the trace. Thus, by carefully selecting the region to evaluate, smaller logical formulae can be submitted to the SMT solver. More importantly, in certain cases, attempting to evaluate each expression in order for a trace can lead to the creation of an infinite sequence of alternate traces, all of them constituting false alarms. We will examine how this can happen.

At this stage, we have seen how a particular trace leading to a potential security violation can be eliminated as a false alarm. However, while techniques that use model checking to analyze the CFG for possible security violations can find all candidate locations within the CFG, they cannot find all possible traces to these locations since there may be an infinite number of these, particularly when loop structures are involved.

The CFG representation of a program can be shown to be equivalent to a representation known as an Abstract Reachability Tree (ART). The CFG is a more compact representation of a program and has a finite number of nodes and edges. The ART expands every branch point to form a tree [?]. In particular, as loops are unwound, the ART can be infinite in extent. In addition, a particular node in the CFG may correspond to multiple nodes, possibly an infinite number, in the ART. A trace in the CFG corresponds to a single path from the root in the ART. The traces found for possible security violations found by model checkers are the result of an FSA reaching a terminal state at a CFG node. For a violation occurring at a particular location in a CFG, there may be more than one trace that describes it.

If, as shown in the previous example, a set of expressions found on a trace leads to a contradiction, we can view this as the removal of the final edge in the portion of the ART trace from which the expressions of the logical formula have been extracted. We can then produce a hybrid graph which is the standard CFG except in the region where branches have been expanded to produce a local portion of an ART. In Figure ?? (a), a CFG with a trace segment highlighted in green is shown. In order to partially transform the

CFG to an ART, we start at the first node of the trace segment, C, and find the first node beneath it with in-degree greater than one. This is node F. We then split F into F' and F". Now node G has two predecessors, F' and F" and we can split it and continue to split H into H' and H" as well. This leads to diagram (b). Note that in diagram (b), both H' and H" both have edges to G and J. But only nodes H' and G are in green because they are both direct descendants of D which is on the trace. Diagram (d) corresponds to the complete cleaving and the trace C-D-F'-G'-H'-G'-K' constitutes an local ART in that every node has at most one ancestor. We further modify this graph by removing the edge at the end of the trace, leading to diagram (e). We can then re-submit the modified graph (e) to the model checker to see if there are any new traces that correspond to alternate paths to the location of a violation. We can keep repeating this until all traces are removed or until a trace remains that cannot be removed.

There are instances involving loops where we end up with an infinite sequence of traces such that each time an edge is removed, a new longer trace which goes around the loop at least one more time is formed. We now present an instance where this can happen. Assume a security violation involving locking where we require that we never have either two locks or two unlocks in a row. The FSA for this violation is shown in Figure ??. Consider the following code:

```
1:  void example ()
2:  {
3:    int x = 0;
4:
5:    while(TRUE)
6:     {
7:       if(x == 0)
8:       {
9:         lock();
10:        x=1;
11:      }
12:      else
13:      {
14:        unlock();
15:        x=0;
16:      }
17:    }
18:  }
```

Clearly, from inspection of the code, every lock is followed immediately by an unlock and the FSA cannot be driven into an accepting state. Yet, when model checking is applied to the CFG for this code, we find a violation shown in Figure ?? where the false alarm violation is shown in red. It consists of passing through the lock condition on both the first and second times around the loop. Obviously, the values of x would not allow this.

Figure 5. This is the CFG corresponding to the lock/unlock *while* loop code.

We now set about to demonstrate that the trace in Figure ?? is a false alarm. using the cleaving technique demonstrated in Figure ??. We begin cleaving at the top of the loop with Node 2 and continue all the way to Node 5 where nodes 2,3 and 4 are shown twice in Figure ??. Finally, we attempt to remove the 3"-4" edge by checking the logical formula for the entire ART trace to see if it contains a contradiction. This leads to the formula:

$$x_0 = 0 \wedge x_0 = 0 \wedge x_1 = 1 \wedge x_1 = 0 \tag{2}$$

which, in fact, contains a contradiction for x_1 resulting in the removal of the 3"-4" edge.

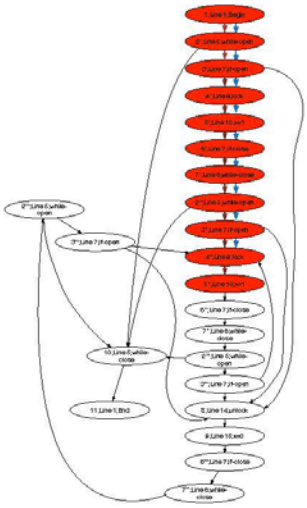


Figure 6. This is a partially unwound CFG corresponding to the lock/unlock *while* loop code.

We now submit the graph in ?? with the 3"-4" edge removed to the model checker. We now find that a new trace to the same violation is found as shown in Figure ??. This trace goes through the unlock call and then goes through two locks in a row leading again to a another violation at the 4"-5" edge. We can cleave through the small 4" to 3"' loop again, but we will simply produce a new violation.

However, if we begin cleaving at node 6 (within the loop), around the loop and until node 4, we end up with the trace shown in Figure ??. Collecting the expressions along the blue arrows, we have:

$$x_n = 1 \wedge x_n = 0 \wedge x_{n+1} = 1 \tag{3}$$

We can now remove the 4"-5' edge. As can be seen from Figure ??, once the 4"-5' edge is removed, no new trace is produced.

We see that the choice of the trace segment to evaluate has two implications. The first is that the size of the logical formula sent to the SMT checker for evaluation can be greatly reduced by the clever choice of the trace subsection, leading to faster evaluation. The second is that certain choices of trace segments lead to the failure of a false alarm to be removed where the correct choice would lead to a correct removal of the false alarm.

[Dan] Cleaving and edge removal leading to other traces

[Dan] Surgical handling of loop in proposal case

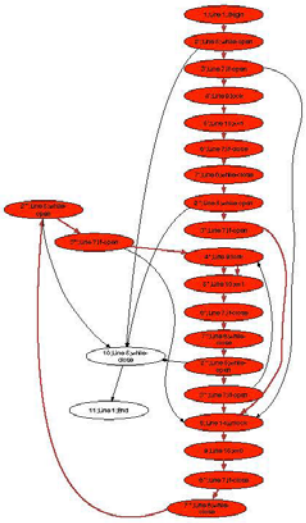


Figure 7. After unwinding and removal of edge based on its being unreachable, a new trace is found.

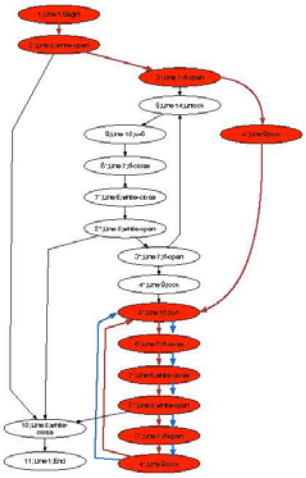


Figure 8. After unwinding and removal of edge based on its being unreachable, the violation is removed.

5. Elimination of False Positives

[Andrei] Definition of relevant information flows as annotations on the CFG.

Information flows and CFG regions. [Andrei] Description of a CFG region and its relationship to information flows (i.e., how it can be determined using information flow annotations).

CFG regions and formulas. [Andrei] Conversion of CFG regions to formulas, and relationships between the region size and formula complexity.
[Andrei/anyone] Find reference(s) of analytical or empirical results showing the efficiency of SMT solver(s) as a function of formula complexity.

6. Implementation and Performance

[???] Presentation of some examples of the technique being applied to actual code.

Implementation. [???] Relevant details about the implementation.

7. Conclusion and Future Work

[???] More conclusion/future work text...

There are plans to deploy the techniques described within this paper in the context of a crowd-sourced software validation game Ghost Map [?]. The game interface will allow human users to specify regions within the CFG based on visualizations supplied to them about the information flows within the program. Users will then be able to exploit information flows to limit the size and complexity of the logical formulas submitted to the backend SMT solver when false positive detections of violations need to be eliminated.

Acknowledgments

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.

Appendix E. Playing the Subset Coloring Game

Playing the Subset Coloring Game

June 2015

In this short paper, we discuss a classic combinatorial problem and its representation as a puzzle game. While no new theorems are proved in this paper, we hope it will be interesting to see how game play can help support teaching and general awareness of mathematics.

In the last few years, a number of efforts have attempted to use crowd sourced game play to support research activities. Perhaps the best know example is the *fold.it* game [1], which lets players fold proteins in search for molecules with specific properties. There is also a game from UK cancer research [2] and several papers and web sites that discuss formal verification of software using games [3,4]. The current research grew out of the Verigames web site [5] and the Ghost Map series of games [6,7].

Introduction

Consider a set $\mathcal{S}$ containing subsets drawn from a set of size $N > 3$. We want to "color" each subset in $\mathcal{S}$ so that overlapping sets always get different colors. Thus, we are looking for a function f: $\mathcal{S} \rightarrow M$ so that:

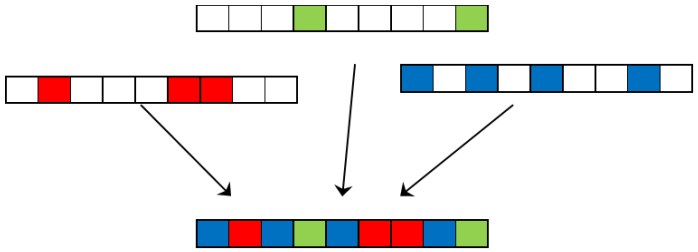
$$\text{For all } s_1 \text{ and } s_2 \text{ in } \mathcal{S},\ f(s_1) = f(s_2) \rightarrow [\, s_1 = s_2 \text{ or } s_1 \cap s_2 = \{\,\}\,] \qquad (1)$$

The primary interest is, for a given $\mathcal{S}$, finding the minimum M for which a coloring is possible, and hopefully finding a simple description of a minimal coloring.

There are a several motivations for this discussion. The elements of the set of size N may represent resources that are required to complete some task. Each subset can be viewed as a task that requires the contained resources. The coloring function finds tasks that can be completed in parallel, as they require disjoint resources. The problem also has a representation in graph theory, where the elements of $\mathcal{S}$ become nodes in a graph, and there is an edge between s_1 and s_2 iff they have non-empty intersection. A coloring function f as above provides a coloring of the graph in the usual graph-theoretic sense (i.e., no neighboring nodes have the same color) and the minimal M is the chromatic number of the graph. There is also a connection to the unsolved Erdős-Faber-Lovász conjecture [8] and follow-on work by Hindman [9] on coloring families of sets with small pairwise intersections.

As in [9], one can consider a subset of N as a piece in an abstract puzzle game, where the pieces are rows of length N, with darkened entries that correspond to the elements in the subset. So, for example, the set {0,2,4,7} for $N = 9$ (counting from zero) is represented as follows:

The puzzle aspect is that multiple pieces must fit together in order to be the same pre-image of f, as shown next.



Above and in the sequel, the figures use different colors for subset elements as they are assembled into a single row, so the individual subsets will be visible inside each row.

Playing with the Doubletons

In this section we cover playing the subset game with the set of all 2-element subsets. This is a particular easy case and serves as a good example. As discussed above, represent each of the $N(N-1)/2$ doubletons as a bit vector of length N with exactly two bits set. We will discuss "loading" the doubletons into an NxN square, where the square is represented as a bit array. By loading a doubleton (i,j), $0 <= i < j <= N-1$, into row r, $0 <= r <= N-1$, of the square, we mean that the bit values in positions (r,i) and (r,j) of the square are changed from 0 to 1. Loading fails if the bit array value is already set to 1 for one of the positions being addressed. We are interested in successfully loading all the doubletons into the square. In other words, we are looking for a function f: $[N]^2 \rightarrow N$ which has the following property:

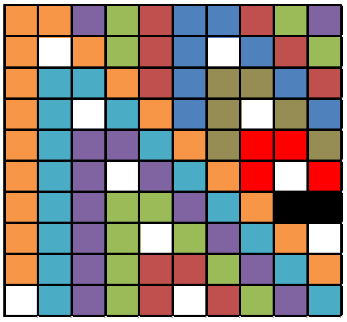
For all i, j, $0 <= i < j <= N-1$, $[\, f(i,j) = f(i',j')$ and $(i = i'$ or $j = j')\,]$ implies $(i = i'$ and $j = j')$ (2)

Given such an f, loading the doubleton $\{i,j\}$ into row $f(i,j)$ in the square succeeds in loading all doubletons into the square. This f is a coloring in the sense defined above, as equation (2) is just (1) specialized to the case where S is the set of all doubletons.

Hindman [9] notes that lexicographic order works to load doubletons into the square, and thus

$$f(i,j) = i+j-1 \bmod N \qquad (3)$$

is a suitable selection for f. The proof of equation (2) is obvious for this f. A graphical representation of f for N=10 is shown below. The orange boxes are doubletons starting with 0, continuing through the two black boxes, which represent the doubleton {8,9}.



The loading function $i+j-1$ mod N places some doubletons in every row of the square. When N is even, we can look to fit the doubletons into just $N-1$ rows of the square rather than all N. Some experimentation leads to the following improved doubleton loading function for even N:

$$g(i,j) = i+j\text{-}1 \bmod N\text{-}1 \text{ if } i = 0 \text{ or } j < N\text{-}1 \;;\; 2i - 1 \bmod N\text{-}1 \text{ otherwise} \qquad (4)$$

It is clear that *g* leaves the bottom row of the *N*x*N* square empty since it produces values modulo *N*-1. We show now that *g* satisfies equation (2) in two steps, first assuming that *i*=*i*' holds and next assuming that *j*=*j'* holds.

Assume *i*=*i*' with the goal to show that *j*=*j'*. There are three cases:

Case (1) Assume that *i*=*i'*=0. Then *g*(*i,j*) = *g*(*i',j'*) is expanded using the first clause of *g*'s definition on both sides of the equation, so we have that *j*-1=*j'*-1 mod N-1. Since both *j* and *j'* are non-zero, we have *j*=*j'* as desired.

Case (2) Assume that *j* < N-1 and *i*=*i'*=\0. There are two subcases.

Case (2a) Assume that j' < N-1. Then the first clause of *g* applies twice so we get *j*=*j'* as in Case 1.

Case (2b) Assume that *j'*=*N*-1. Then by expanding *g* twice, we get *i*+*j*-1 mod *N*-1 = 2*i'* -1 mod *N*-1, so *j*-1 = *i'* -1 mod *N*-1. Since *j* and *i*=*i'* are all nonzero, we have *i*=*j*, which is a contradiction, so this case is empty.

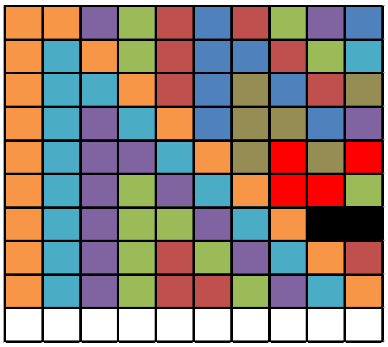
Case (3) Assume that *j* = *N*-1 and *i*=*i'*=\0. If *j'* < *N*-1, then we get a contradiction as in Case (2b), so *j*=*j'*.

Now assume that *j*=*j'* and prove that *i*=*i'*. There are two additional cases:

Case (4) Assume j=j'<N-1. Then as in Case 1 above, the first clause of *g*'s definition applies twice, so we have that *i*-1 = *i'*-1 mod *N*-1. Since neither *i* nor *i'* can be N-1, we have *i*=*i'*.

Case (5) Assume *j*=*j'*=*N*-1. If *i* and *i'* are both zero, then we are done. If exactly one of them is zero, then we get a contraction as in Case 2b. Finally, if both *i* and *i'* are nonzero, then the second clause of *g*'s definition applies twice, which yields 2*i*-1 = 2*i'*-1 mod *N*-1. Because *N* is even, the function 2*i*-1 mod *N*-1 is one-to-one for *i*=1 to *N*-2, running sequentially through the odd numbers from 1 to *N*-3 and then the even numbers from 0 to *N*-2. It follows that *i*=*i'*.

The figure below shows *g* for *N*=10 using the same color pattern as the previous figure.



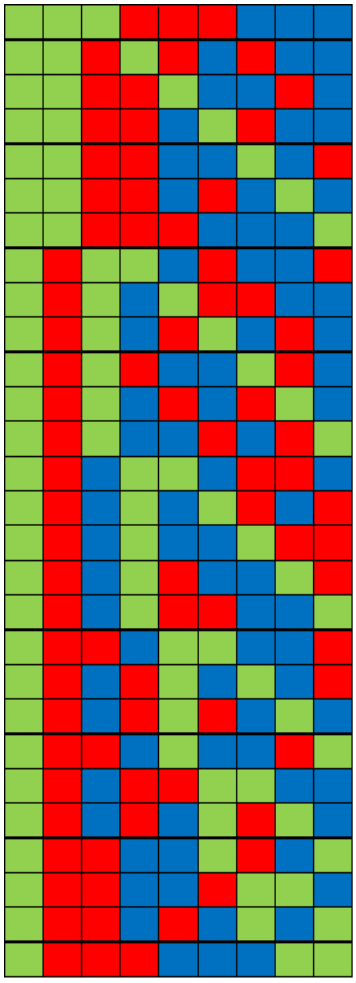
It is of course critical that the proof for *g* meeting equation (2) uses the premise that *N* is even.

What about triples?

Given the success of loading doubletons into the square, one might try the same game with other size subsets. For example, for an $N = 3k$ for $k > 2$, we can try to the three-element subsets into a rectangle that has N columns and $N(N-1)(N-2)/2$ rows. Taking $N = 9$, there are 84 3-element subsets of 9 and we'd like to fit them into 28 rows. Hand experimentation in this case didn't find any simple pattern, but it's straight forward to find a solution with a computer program, as shown below.

The 3-element subsets of 9, colored with 28 colors, the minimum number possible (counting from 1)

1.	[1, 65, 84]	1,2,3 – 4,5,6 – 7,8,9
2.	[2, 56, 83]	1,2,4 – 3,5,7 – 6,8,9
3.	[3, 53, 82]	1,2,5 – 3,4,8 – 6,7,9
4.	[4, 52, 80]	1,2,6 – 3,4,7 – 5,8,9
5.	[5, 54, 76]	1,2,7 – 3,4,9 – 5,6,8
6.	[6, 51, 79]	1,2,8 – 3,4,6 – 5,7,9
7.	[7, 50, 81]	1,2,9 – 3,4,5 – 6,7,8
8.	[8, 46, 78]	1,3,4 – 2,6,9 – 5,7,8
9.	[9, 44, 74]	1,3,5 – 2,6,7 – 4,8,9
10.	[10, 42, 73]	1,3,6 – 2,5,8 – 4,7,9
11.	[11, 38, 77]	1,3,7 – 2,4,8 – 5,6,9
12.	[12, 41, 71]	1,3,8 – 2,5,7 – 4,6,9
13.	[13, 45, 66]	1,3,9 – 2,6,8 – 4,5,7
14.	[14, 47, 61]	1,4,5 – 2,7,8 – 3,6,9
15.	[15, 48, 57]	1,4,6 – 2,7,9 – 3,5,8
16.	[16, 49, 55]	1,4,7 – 2,8,9 – 3,5,6
17.	[17, 43, 59]	1,4,8 – 2,5,9 – 3,6,7
18.	[18, 40, 62]	1,4,9 – 2,5,6 – 3,7,8
19.	[19, 34, 72]	1,5,6 – 2,3,9 – 4,7,8
20.	[20, 39, 60]	1,5,7 – 2,4,9 – 3,6,8
21.	[21, 36, 63]	1,5,8 – 2,4,6 – 3,7,9
22.	[22, 33, 69]	1,5,9 – 2,3,8 – 4,6,7
23.	[23, 35, 64]	1,6,7 – 2,4,5 – 3,8,9
24.	[24, 37, 58]	1,6,8 – 2,4,7 – 3,5,9
25.	[25, 32, 67]	1,6,9 – 2,3,7 – 4,5,8
26.	[26, 31, 68]	1,7,8 – 2,3,6 – 4,5,9
27.	[27, 30, 70]	1,7,9 – 2,3,5 – 4,6,8
28.	[28, 29, 75]	1,8,9 – 2,3,4 – 5,6,7



In the table above on the left, the three element subsets of 9 are numbered 1 to 84 in lexicographical order, and the triples in square brackets are triples of subsets. On the right we show the visualization.

The General Solution

It was proven in 1975 by Zsolt Baranyai [10] that one can always color the set of k-tuples drawn from a set of size kN with the minimum number of colors, which is $C(n\text{-}1, k\text{-}1)$. The proof was achieved by finding a powerful generalization of this result and then proving the generalization by induction. No direct proof is known and actual examples of colorings are still best found by experimentation and game

play, either manual or computer-aided. Expositions of Baranyai's work can be found in good texts on combinatorics, such as van Lint and Wilson [11]

References

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

[2] Cancer Research UK, http://www.cancerresearchuk.org/-support-us/play-to-cure-genes-in-space, retrieved: May, 2015.

[3] 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.

[4] 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.

[5] Verigames, www.verigames.com, retrieved: May, 2015.

[6] Ronald Watro, et al., "Ghost Map: Proving software correctness using games," SECURWARE 2014: Eighth International Conference on Emerging Security Information, Systems and Technologies (Lisbon, Portugal), November 2014.

[7] Kerry Moffitt, John Ostwald, Ron Watro, and Eric Church, "Making Hard Fun in Crowdsourced Model Checking: Balancing Crowd Engagement and Efficiency to Maximize Output in Proof by Games," 2nd International Workshop on CrowdSourcing in Software Engineering (CSI-SE 2015, Florence, Italy), May 2015.

[8] P. Erdős, "Problems and results in graph theory and combinatorial analysis," Proceedings of the Fifth British Combinatorial Conference, 169-92, Congressus Numerantium, No. XV, Utilitas Math. (1976).

[9] Neil Hindman, "On a conjecture of Erdős, Faber, and Lovász about n-colorings," Canad. J. Math., 33 (1981), pp. 563–570.

[10] Z. Baranyai, "On the factorization of the complete uniform hypergraph, in Infinite and Finite Sets," Vol I, Colloq. Math. Soc. J. Bolyai 10, North Holland, (A Hajnal, Vera T. Sos, eds.), 1975, pp 91-108.

[11] J. H. van Lint and Richard Michael Wilson, *A Course in Combinatorics*, Cambridge University Press, 2001.

Appendix F. PBG Human Subject Experimentation Protocol

The game play portion of the CSFV program fell under the regulations for Human Subject Experimentation. Accordingly, a PBG protocol for the experiment was developed and submitted for approval, both to BBN's Internal Review Board (IRB), which is the New England IRB

Proof by Games Experimentation

Version 1.2

July 11, 2013

Title: Proof by Games Experimentation – Option Phase

Protocol Number: PBG-1.2

Approval Date: PBG-1.2 was last approved by NEIRB on 9/15/2014

Principal Investigator:
Dr. Ronald J Watro
Lead Engineer
Raytheon BBN Technologies
rwatro@bbn.com
(617) 873-2551

1. BACKGROUND AND INTRODUCTION

In the option phase of the Proof by Games (PBG) project, part of the Defense Advanced Research Projects Agency (DARPA) Crowd Sourced Formal Verification (CSFV) program, Raytheon BBN Technologies (henceforth BBN) will develop a set of arcade-style computer games that are to be made available to the public over the Internet. BBN is one of five contractors developing games for this DARPA program. Games from all five are to be made available on a single web site maintained by TopCoder Inc., also funded by DARPA. The CSFV games offered by the TopCoder web site are intended to be fun games that also will accomplish tasks that assist in the computer security analysis of software.

BBN and TopCoder, while independent contractors, will share responsibility for the execution for our portion of the CSFV Internet-based game play research. For example, TopCoder will be responsible for the selection of players for their web site and the execution of consent agreements. BBN will be responsible to protect the confidentiality of any game player personally identifiable information that TopCoder sends to BBN.

The TopCoder research protocol for CSFV is included below as Appendix 3. The TopCoder CSFV protocol was approved by the Air Force Office of the Surgeon General on 24 June 2013, number FWR20120332X.

1.1 CSFV

Unreliable software places huge costs on the economy. The current state of practice is that released software typically contains about one to five bugs (errors) per thousand lines of code. Errors can cause software programs to stop working or to work incorrectly. Errors can be costly to reproduce, find, and fix. Some errors are security flaws that make the software programs, and the computers that they run on, vulnerable to attack and compromise.

Formal program verification is the only way to be certain that a given piece of software is free of errors. Formal program verification is a time-consuming technical analysis of a software program intended to demonstrate using mathematical proofs that the software program under analysis has a specific feature or characteristic, such as the absence of a specific type of bug or security flaw. Formal program verification is currently performed manually by specially-trained engineers using formal program verification software tools. Consequently, due to the requirement of large amounts of skilled labor, formal program verification has been too costly to apply beyond certain small, critical software components.

The CSFV program seeks to make formal program verification more cost-effective by enlarging the population that can participate in verification. The approach is to transform verification into a more accessible task by creating games that are intuitively understandable and are fun to play. Completion of a game effectively allows a game player to provide the information that a specially-trained engineer would provide to a formal program verification tool in order to complete a formal verification proof.

The primary technical challenge faced by CSFV is construction of automated game-level builders capable of transforming formal verification models into compelling games. A particular game level is a function of the program verification tool, the property to be verified and the program being verified. Each game level is provided to the "crowd," to people who play the games. Game solutions are used to populate a database, and then are mapped back into program annotations sufficient to allow the program verification tool to make progress toward formal verification of a specific program property.

TopCoder will routinely provide de-identified game play results to the PBG game developers at BBN. When deemed useful, TopCoder will provide e-mail addresses for players of special interest to BBN (and other game developers) to facilitate follow-on discussion related to the games.

1.2 Background Information on BBN

Raytheon BBN Technologies (BBN) is a research and development organization headquartered in Cambridge Massachusetts. BBN works in a variety of technical areas, including cyber security; communications and networking technologies; speech and natural language tools; and planning and logistics systems. Much of BBN's research is funded by DARPA and other US government agencies.

The PBG project is led by the BBN Cyber Security business unit, with support from the BBN staff that deal regularly with serious games used for training purposes. The PI (Dr. Watro), the co-PI (Mr. Kerry Moffitt) and the gaming lead (Dr. Taleb Hussain) have all completed CITI Human Subject Research training.

1.3 PBG Project

In the base phase of the CSFV program, which is currently underway (and is not the subject of this research protocol), BBN is designing and building the first PBG game, called GhostMap. Overview information on Ghost Map is included in Appendix 1. In summary, GhostMap will be similar in style to the famous "PacMan" arcade game, where the player directs a token that travels over a course while being chased by adversaries. When the token reaches specific points on the course, it becomes energized and can now turn the tables and give chase to its pursuers. Full details are in the PBG proposal, attached as Appendix 2.

In option phase of PBG, which will start in August or September, and which is the subject of this research protocol, BBN will provide its GhostMap game and possible subsequent games to TopCoder for deployment on an online web site that is open to the public. Individuals who register for accounts on the TopCoder web site will be able to play the games. The nature and content of the BBN games will be appropriate for all adults.

1.4 PBG UCF Study

In addition to the Internet-based game play assessment research conducted by BBN in coordination with TopCoder, the PBG team also includes the University of Central Florida (UCF) as a subcontractor to BBN. UCF will perform play testing of the PBG games in an in-person manner on their campus in Orlando FL, using a protocol that has been defined and

approved by the UCF IRB. UCF will report to BBN only de-identified game play information. No BBN personnel are considered engaged researchers for the UCF study. Any game players from the UCF study that desire to interact with BBN will be directed to register through the TopCoder web site. The UCF protocol was approved by the Air Force Office of the Surgeon General on 17 April 2013, number FWR20130109X.

1.5 PBG Research (Option Phase)

The technical research that will be conducted under this DARPA program has the objective of developing games that when played produce data that is useful in the computer security analysis of software programs. To meet this objective, PBG will develop games and software tools that will create game levels based on the software programs under analysis. PBG will base its games on the model checking verification process. PBG will develop a system that provides game levels that the TopCoder web site will in turn deliver to the game players.

The community web site that TopCoder is building will host the BBN games and software tools (as well as games and tools from other performers). TopCoder is responsible for recruiting and registering players, storing the data from the games, and generate reports from the data. TopCoder will provide game data to BBN and the other game developers. The TopCoder research protocol is attached as Appendix 3.

All BBN interaction with game players will take place over the Internet, predominantly via the web site, but possibly also through direct e-mail between a player and the BBN developers. The web site will be a gaming community web site that fosters an online community of people who are interested in playing these games. The members of this online community will play the games and will be able to communicate with each other in public forums on the web site.

1.6 Stand-alone Operation

BBN will also deliver a stand-alone version of the gaming software and tools to DARPA. Use of the standalone gaming software and tools after it has been delivered to DARPA is outside the scope of this research protocol.

2. STUDY RATIONALE

The PBG project is focused on the development of compelling games that when played also create data that is useful in the computer security analysis of a software program. The key unknown factor is whether crowds of human players using game can be more successful at proving software is correct than highly paid verification experts using conventional verification tools.

The UCF study being performed under a separate IRB approval will provide initial impression data on the PBG games, collected from college student populations with varied backgrounds. This data will help BBN address whether the game elements are easy to understand and if the graphics are appropriate.

The reports from TopCoder on play at their website will allow BBN to observe game play results over an extended period of time. This will help determine whether the game is fun

for players and whether game players can successfully accomplish the formal verification tasks. In addition, we are interested to see if human users can learn to be more effective at software verification as they grow more experienced with the game.

3. OBJECTIVES

The objectives of BBN's work on the Option Phase of PBG are:

- Provide a PBG system that reads C language software files and potential vulnerability specifications and outputs game level data that when played helps determine whether the software suffers from the listed vulnerabilities.
- Revise and improve the PBG system based on de-identified data received from UCF and TopCoder and from interactions with actual game players.

The objective of the human subject experimentation is to judge the effectiveness of human players at performing verification through gaming.

4. STUDY DESIGN

BBN began work on PBG in July 2012, with the goal of having a game that is ready for "beta" testing (i.e., the first use by outside game players) by August or September 2013.

All participation in the game studies will be entirely voluntary. Game players can come to the TopCoder web site at their discretion and participate as little or as much as they want. Visitors to the site at first will be able to find information about the site and the objectives of the site. They will be able to try the games, but their personal performance data will not be associated with them (i.e., will be recorded as anonymous) if they have not registered. Playing games will require completion of an informed consent with TopCoder, as described in Appendix 3.

BBN will receive regular reports of de-identified player activity from TopCoder. Based on these reports, either BBN or TopCoder may identify a player with a unique skill (or a common problem) that we should investigate more closely. In these cases, TopCoder will transmit to BBN, in encrypted form, the e-mail address that is associated with the user name. The consent form signed by the player explicitly allows the transmission of this data to game developers. BBN may then initiate contact with the player to discuss the details of the game design, for the purposes of improving the game. Participation by the player in these direct discussions is entirely option. Players may also volunteer for additional discussions with game developers by using collaboration tools on the web site.

5. STUDY POPULATION

TopCoder's protocol states that the gaming web site will be directed to adults (age 18 and above). They expect that the initial interest in the CSFV games will come from news articles about the program and word of mouth. Interest in the BBN game may also come from BBN employees, their families and friends, and the participants in the UCF study.

6. PATICIPANT ELIGIBILITY

TopCoder controls the criteria for access to web site play:

Inclusion Criteria: The game players will be self-selecting and entirely voluntary. Participants have the option to join or leave at any time, with no repercussions. Players have the option to contact game developers for additional discussion through collaboration tools on the web site.

Exclusion Criteria: The first exclusion criterion is that only adults 18 years or older are allowed to participate in the study. Another criterion for exclusion would be demonstrated anti-social or other problematic behavior on the web site, on a case-by-case basis. Our goal is to provide an online web site experience that fosters interest in the games. Behavior on the web site that is detrimental to that goal (e.g., offensive comments, language, etc.) will not be allowed, and will result in suspension or exclusion from the site. TopCoder will provide a variety of channels for participants to communicate with site administrators and report offensive behavior (e.g., telephone, email, web site form).

BBN controls the criteria for selecting users for additional interaction based on their play characteristics. The inclusion criterion is the presence of unique and/or unexplained high or low scores in some aspect of the game. There are no additional exclusion criteria.

7. STUDY ASSESSMENTS – PLAN AND METHODS

See Study Design (#4) above. BBN will be reviewing the data to identify game levels that teach the game players how to effectively solve the games. We also will be determining how to better use the game results in the security analysis of software.

8. STUDY CONDUCT

See Study Design (#4) above.

9. STUDY TREATMENT

N/A

10. EVALUATION OF ADVERSE EVENTS

N/A

11. ETHICAL CONSIDERATIONS

Risk/benefit assessment:

Risks: There will be no discomfort or health risk to the participants. The primary risk is that we will be collecting certain data about the participants, including the information that they provide during registration, their activity on the web site, and the results of their game play. This is "minimal risk," in that it is no different than other gaming web sites on the Internet.

Benefits: By participating in this research, participants will have the opportunity to play the games provided on the site, providing them with free games to play and gaming community involvement.

Informed consent process:

BBN relies on the informed consent process conducted by TopCoder (and approved by their IRB). This is described in Appendix 3 below. Any players that contact BBN directly will be asked to register with TopCoder before discussion can begin.

Participant confidentiality:

TopCoder assigns to each user a numeric identifier that is different from their user name. BBN will use this numeric identifier as the key in a database table that stores game play data. The associate of e-mail addresses with user IDs and numeric user codes is maintained separately from game data and is always encrypted. BBN holds only a portion of this data, for those users identified as appropriate for additional discussion. Thus, BBN's storage of personally identifiable information is minimize and kept separate from game data.

12. STUDY MONITORING AND OVERSIGHT

N/A

13. INVESTIGATIONAL PRODUCT MANAGEMENT

N/A

14. DATA ANALYSIS

The data collected will be analyzed in to determine whether the games can successfully be used as part of a computer security analysis, and whether the games are interesting enough to attract a large audience of participants.

15. INVESTIGATOR STATEMENT

I have reviewed the above protocol and agree that it contains all the information needed to conduct the study. We will comply with the protocol and applicable regulatory requirements, and will not begin the study until all necessary IRB and other regulatory approvals have been obtained.

_____

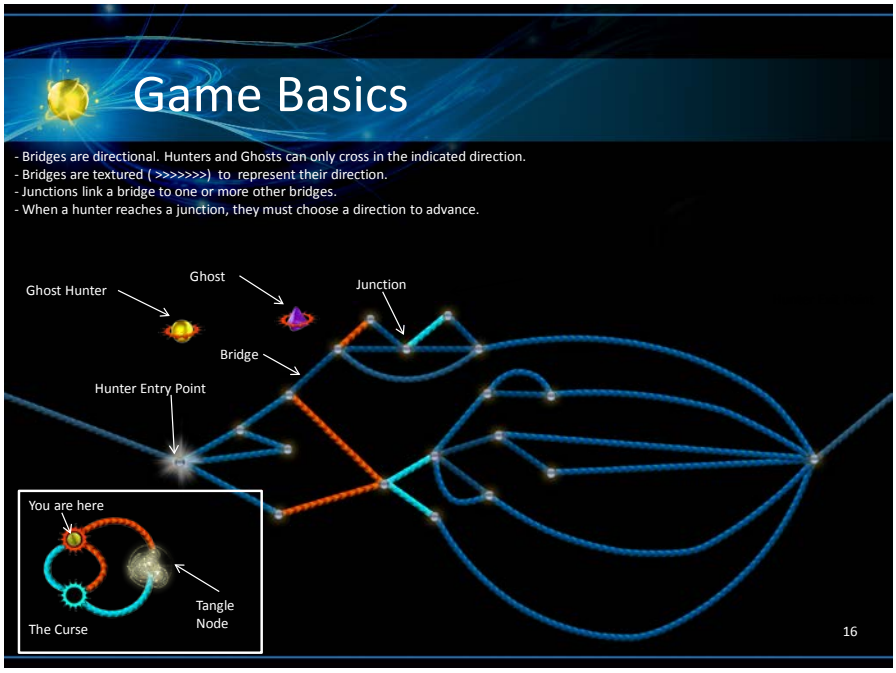
Dr. Ronald Watro, Principal Investigator

Lead Engineer, BBN Technologies

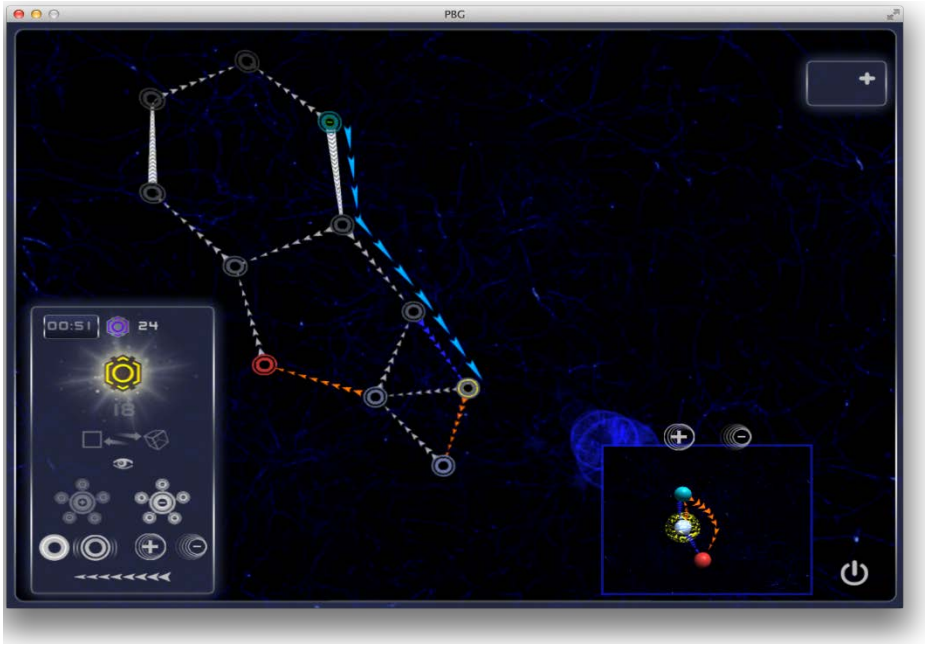
APPENDICIES (to the PBG Experimentation Plan)

Appendix 1 Ghost Map Game Overview

Original game concept from the PBG proposal:



Current game screen shot:



List of Symbols, Abbreviations, and Acronyms

AFRL …… Air Force Research Laboratory

AMT …….. Amazon Mechanical Turk

ART .....Abstract Reachability Tree

BAG .....BreakAway Games

BLAST .....Berkeley Lazy Abstraction Software Verification Tool

CEGAR .....CounterExample-Guided Abstraction Refinement

CFG .....Control Flow Graph

CMU .....Carnegie Mellon University

CSFV .....Crowd Sourced Formal Verification

CWE .....Common Weakness Enumeration

DARPA …. Defense Advanced Research Projects Agency

FSA .....Finite State Automaton

IRB .....Institutional Review Board

MOPS .....MOdelchecking Programs for Security properties

PBG …….. Proof by Games

SMT .....Satisfiability Modulo Theories

UCF .....University of Central Florida