AFRL-RI-RS-TR-2016-089

CIRCUITBOT

KESTREL TECHNOLOGY, LLC.

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

AFRL-RI-RS-TR-2016-089 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /	**/ S /**
CARL THOMAS	RICHARD MICHALAK
Work Unit Manager	Acting Technical Advisor, Computing & Communications Division Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

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

4. TITLE AND SUBTITLE CIRCUITBOT	**5a. CONTRACT NUMBER** FA8750-12-C-0202
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62303E
6. AUTHOR(S) Matthew Barry, Nelson Rushton, Andrew Keplinger, Gregory Izzo, Qianji Zheng, Arnaud Venet, Henny Sipma	**5d. PROJECT NUMBER** CSFV
	5e. TASK NUMBER KE
	5f. WORK UNIT NUMBER ST
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Kestrel Technology LLC 3260 Hillview Avenue Palo Alto, CA 94304-1225	**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 St Arlington, VA 22203-2114	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2016-089

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 25938
Date Cleared: 15 MAR 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The CircuitBot project developed a distributed algorithm for performing analysis of C programs. A constraint generator first analyzed a target program's C source files to prepare a collection of constraints describing the use of pointers and offsets. There is no known closed-form solution to this problem, but human experts can help auto-solvers break free when they become stuck. The project distributed these constraints to game players on the Internet, using a crowd of game players to invoke human intuition schemes to solve the constraints problem. Game rules described valid moves allowing player to generate a memory graph performing improved C program verification.

15. SUBJECT TERMS

Formal Verification, Static Analysis, Abstract Interpretation, Pointer Analysis, Fixpoint Iteration

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** **CARL THOMAS**
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	330	**19b. TELEPHONE NUMBER** (*Include area code*) **N/A**

TABLE OF CONTENTS

List of Figures

List of Tables

1.0 SUMMARY

The CircuitBot project developed an innovative distributed algorithm supporting the static analysis of C-language programs. We created the distributed algorithm in such a way to lend itself to crowd-sourced contributions via game play. In particular, the player-provided solutions of puzzle games contribute to the construction of *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 known classically 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.

To evaluate the effectiveness of this approach, the project developed two conventional free online games, named *CircuitBot* and *Dynamakr*, and one paid online game named *VIPER* (Verification Enhanced by PCS (Paid Crowd Source) Enhanced Results). These games performed essentially the same work but explored different approaches to attracting players. The project developed a fourth version of the game, called *Arcweaver*, that ran as an auto-solver without human players. We collected and preset herein the player and analysis data for each of these.

The project's approach to program verification using this new distributed algorithm involved three steps. A constraint generator first analyzed a target program's C source files to prepare a collection of constraints describing the use of pointers in the program. A backend server distributed these constraints to players on the Internet as a collection of game levels, aiming to use the crowd of game players to invoke human intuition as an iteration scheme to build the points-to graph. Inside each game engine was a set of game rules that described valid moves for the game level. Game players solved these game levels by making moves in a judicious order, where each move consisted of adding arcs to the graph that result in satisfying a single constraint. Eventually, as the crowd of players completed the collection of game levels and satisfied all of the constraints, the game play yielded a fixpoint solution – but the time required to reach this solution, and whether the process halted, depended on triggering constraints in a wise order, as well as performing operations that lost information but sped up the solution process or allowed it to halt. When the game play sessions achieved a solution, we saved the resulting graph. The final step passed this graph to a custom static analyzer we developed to use this points-to graph to support its analysis of the target program. We compared the before-and-after verification results to find that our pointer analysis approach substantially increased the quantity of automatically-proved verification results. In particular, our analyzer performed checks of memory safety properties that describe over 50 weaknesses that comprise the primary cyber attack surface in C programs.

The first half of this report explains the analysis techniques we developed that ultimately were embodied in three Internet games and one robotic auto-solver system. The second half of the report provides the data we gathered describing both the game player performance results and the program verification results.

2.0 INTRODUCTION

As an introduction to the problem we set out to solve we explain why the problem we chose was both interesting and difficult. Why did we decide to take up this particular challenge and apply a crowd-sourced approach as a solution? Section 2.1 introduces the key elements of program verification and in particular provides the details of what we mean by *proof obligations*. This section steps through an example C program, applying the steps of generating proof obligations, generating inductive invariants at all locations, developing function preconditions, and discharging proof obligations. Section 2.2 on page 12 then describes our keen interest in the BIND source code as the challenge target, and describes our initial insights into why BIND in particular was an interesting challenge. Our CodeHawk analyzer already performed very well on this target with assertions and counter-examples analysis. There was however a significant residual of unresolved warnings. Using the insights we gleaned from this evaluation, we decided early in the program that in order to improve static analysis of programs like BIND we needed to do something about pointer analysis.

2.1 Proof Obligations

The following is a brief explanation of some major concepts in program verification. An in-depth explanation of the terminology used here can be found in [4].

Program A program is a tuple $\mathcal{P} : \langle \mathcal{V}, \mathcal{L}, I(V), \mathcal{S} \rangle$ that consists of a set of program variables V that may include distinct variables for the initial values of the function parameters, a set of program locations $\mathcal{L}$ (with a distinguished initial location ℓ_0), an initial condition $I(V)$ that is an assertion over the program variables at location ℓ_0, and a set of program statements $\mathcal{S}$ that map program states to program states, where a program state is a valuation of all variables in V.

Assertion Map An assertion map is map $\mathcal{A} : \mathcal{L} \mapsto A(V)$ from program locations $\mathcal{L}$ to assertions A over the program variables V.

Invariant Map An invariant map is an assertion map $\mathcal{I} : \mathcal{L} \mapsto A(V)$ such that at each location ℓ the assertion $\mathcal{I}(\ell)$ holds for all possible runs of the program (all possible valuations of the program variables that are consistent with the program statements).

Inductive Assertion Map An assertion map is inductive if it is implied by the initial condition and preserved by all program statements. It can be shown that all inductive assertion maps are invariant maps. Not all invariant maps, however, are inductive assertion maps. The only way, however, for non-trivial programs, to prove that an assertion map is an invariant map is to show that it is implied by an inductive assertion map. Hence the only useful assertion maps to us are inductive assertion maps. All invariants generated by CodeHawk are inductive invariants.

Strongest Program Invariant Map The strongest inductive assertion map, $\mathcal{I}_{\mathcal{P}}$ is the assertion map that exactly describes the reachable states of the program for all possible inputs. All inductive assertion maps for $\mathcal{P}$ are implied by $\mathcal{I}_{\mathcal{P}}$. We can strengthen a program invariant map by restricting its inputs; we will denote the strengthened invariant map by $\mathcal{I}_{\mathcal{P}[C]}$, where C are the constraints imposed on input variables.

In general, we do not know $\mathcal{I}_{\mathcal{P}}$ and we do not have to. The CodeHawk abstract interpretation engine generates an assertion map that is an over-approximation of the reachable state space, an inductive assertion map that is implied by $\mathcal{I}_{\mathcal{P}}$. We will refer to the assertion map generated by the CodeHawk engine as $\mathcal{I}_{generated}$.

Proof Obligation A proof obligation $p(V)$ is an assertion at a particular location $\ell \in \mathcal{L}$ of the program over the program variables; a proof obligation is valid if it is implied by the strongest possible invariant at that location, that is, if $\mathcal{I}_{\mathcal{P}}(\ell)(V) \Rightarrow p(V)$. A proof obligation is *proven valid* ("yes" or green in CodeHawk terms) if it is implied by the invariant generated at that location, that is, if $\mathcal{I}_{generated}(\ell)(V) \Rightarrow p(V)$. A proof obligation is *inconsistent* if its negation is implied by the strongest possible invariant at that location, that is, if $\mathcal{I}_{\mathcal{P}}(\ell)(V) \Rightarrow \neg p(V)$ and *proven inconsistent* ("no" or red in CodeHawk terms) if its negation is implied by the generated invariant at that location, that is, if $\mathcal{I}_{generated}(\ell)(V) \Rightarrow \neg p(V)$. A proof obligation is *indeterminate* if neither the proof obligation itself nor its negation is implied by the strongest possible invariant (this happens, for example, when the value of a variable depends on user input). A proof obligation is open ("maybe" or orange in CodeHawk terms) if neither the proof obligation itself nor its negation is implied by invariant generated at that location.

Remaining Proof Obligation The remaining proof obligation $r(V)$ of a proof obligation $p(V)$ at location ℓ is the simplification of $p(V)$ against the generated location invariant $\mathcal{I}_{generated}(\ell)$, with variables rewritten to initial variables as much as possible. For example the remaining proof obligation of $p(x, y) : x > 0 \wedge y > 0$ against the generated location invariant $x = x_0 + 2 \wedge y_0 \geq 0 \wedge y = y_0 + 3$ would be $r(x_0) : x_0 > -2$, where x_0 and y_0 denote the values of the arguments passed to the parameters x and y, and x and y in p denote the values of x and y at location ℓ.

Trace A trace is a sequence of location and program state pairs,

$$\mathcal{T} : (\ell_0, s_0), (\ell_1, s_1), (\ell_2, s_2), \ldots$$

such that (1) ℓ_0 is an initial location and the variable assignment s_0 is consistent with the initial condition $I(V)$, and (2) for each pair of successive pairs $(\ell_i, s_i), (\ell_j, s_j)$, the program states (s_i, s_j) are consistent with the statement(s) $\sigma \in \mathcal{S}$ that lead from ℓ_i to ℓ_j.

Counterexample A counterexample to a proof obligation p at location ℓ_p is a trace $\mathcal{T}$

$$\mathcal{T} : (\ell_0, s_0), (\ell_1, s_1), (\ell_2, s_2), \ldots, (\ell_p, s_p)$$

such that the program state s_p does not satisfy the assertion of p.

2.1.1 Example

Consider the following program

```c
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

void f(char *a, int x, int y) {
char *p;
int i;

a[0] = '*';
a[1] = '*';

p = &a[2];

for (i=x; i<y; i++) {
p[i] = p[i+1];
}
}

void g1() {
char s[] = "Hello" ;
printf ("g1:␣%s\n", s);
f(s,0,0);
printf ("g1:␣%s\n", s);
}

void g2() {
char s[] = "A␣much␣longer␣sentence" ;
printf ("g2:␣%s\n", s) ;
f(&s[10],2, 7);
printf ("g2:␣%s\n", s) ;
}

void g3() {
char s[] = "Going␣negative" ;
printf ("g3:␣%s\n", s);
f(&s[10],-8,-3);
printf ("g3:␣%s\n", s);
}

void g4() {
char v[] = "Next␣string" ;
char s[] = "012345678901234" ;
printf ("g4:␣%s\n",s) ;
f(s,13,14);
printf ("g4:␣%s\n",s) ;
}

void g5() {
char s[] = "Hello" ;
printf ("g5:␣%s\n",s) ;
f(s,42,14);
printf ("g5:␣%s\n",s) ;
}

```

```
void g6(int n, char *t) {
char *s = (char *) malloc (n) ;
strncpy(s,t,n-1);
printf ("g6:␣%s\n",s);
f (s, n-10, n-3);
printf ("g6:␣%s\n",s);
}

int main(int argc, char** argv) {
if (argc > 1) {
char *command = argv[1];
if (!strcmp(command,"g1")) { g1() ; }
if (!strcmp(command,"g2")) { g2() ; }
if (!strcmp(command,"g3")) { g3() ; }
if (!strcmp(command,"g4")) { g4() ; }
if (!strcmp(command,"g5")) { g5() ; }
if (!strcmp(command,"g6")) {
if (argc == 2) { g6(25,"012345678901234567890123"); }
if (argc > 2) { g6(strlen(argv[2])+1,argv[2]); }
}
}
}
```

When compiled and run with some input, the output of this program for various inputs is

```
$ ./example g1
g1: Hello
g1: **llo
$
$ ./example g2
g2: A much longer sentence
g2: A much lon**r entennce
$
$ ./example g3
g3: Going negative
g3: Goin negaa**ve
$
$ ./example g4
g4: 012345678901234
g4: **2345678901234NNext string
$
$ ./example g5
g5: Hello
g5: **llo
$
$ ./example g6
g6: 012345678901234567890123
g6: **2345678901234567890123
$
```

```
$ ./example g6 user-input-string
g6: user-input-string
g6: **er-inputstring
```

2.1.2 Proof Obligations

The first step in the analysis is to generate proof obligations. We will focus on function `f` in this example. Function `f` contains four memory accesses, which give rise to the following eight proof obligations (a lower-bound and upper-bound condition for each access).

At line 7:

$$\begin{aligned} lb_7 &: a_{\mathit{offset}} \geq 0 \\ ub_7 &: a_{\mathit{offset}} < a_{\mathit{size}} \end{aligned}$$

At line 8:

$$\begin{aligned} lb_8 &: a_{\mathit{offset}} + 1 \geq 0 \\ ub_8 &: a_{\mathit{offset}} + 1 < a_{\mathit{size}} \end{aligned}$$

At line 13:

$$\begin{aligned} lb_{13a} &: p_{\mathit{offset}} + i \geq 0 \\ ub_{13a} &: p_{\mathit{offset}} + i < p_{\mathit{size}} \\ lb_{13b} &: p_{\mathit{offset}} + i + 1 \geq 0 \\ ub_{13b} &: p_{\mathit{offset}} + i + 1 < p_{\mathit{size}} \end{aligned}$$

For pointer variable a the "associate" variable a_{offset} denotes the distance (in bytes) between the value of a and the start of the memory block that a points at. The associate constant a_{size} denotes the size of the memory block that a points at. We assume that all pointer variables contain valid addresses; we do not consider the possibility of null dereference in this example.

2.1.3 Invariants and Remaining Proof Obligations

The next step is to generate (inductive) invariants for all locations in the function. The relevant location invariants generated for lines 7, 8 and 13 are given below. In these invariant assertions a 0-subscript denotes the initial value (at function entry) of the variable. We omit the 0-subscript on the size-variables, since we assume the size of a buffer does not change, that is, we always have $a_{\mathit{size}} = a_{\mathit{size},0}$. At lines 7 and 8 we have the invariant assertion:

$$a_{\mathit{offset}} = a_{\mathit{offset},0}$$

producing the remaining proof obligations:

$$\begin{aligned} lb_7 &: a_{\mathit{offset},0} \geq 0 \\ ub_7 &: a_{\mathit{offset},0} < a_{\mathit{size}} \\ lb_8 &: a_{\mathit{offset},0} \geq -1 \\ ub_8 &: a_{\mathit{offset},0} < a_{\mathit{size}} - 1 \end{aligned}$$

At line 13 we have the invariant assertions:

$$\begin{aligned}
a_{\mathit{offset}} &= a_{\mathit{offset}}, 0 \\
p_{\mathit{offset}} &= a_{\mathit{offset}} + 2 \\
p_{\mathit{size}} &= a_{\mathit{size}} \\
x &= x_0 \\
y &= y_0 \\
i &\geq x \\
i &< y
\end{aligned}$$

producing the remaining proof obligations

$$\begin{aligned}
lb_{13a} &: a_{\mathit{offset},0} + x_0 \geq -2 \\
ub_{13a} &: a_{\mathit{offset},0} + y_0 < a_{\mathit{size}} - 1 \\
lb_{13b} &: a_{\mathit{offset},0} + x_0 \geq -3 \\
ub_{13b} &: a_{\mathit{offset},0} + y_0 < a_{\mathit{size}} - 2
\end{aligned}$$

Thus, without any function preconditions, that is, without any constraints on $a_{\mathit{offset},0}$, a_{size}, x_0, and y_0, we have eight open proof obligations for function `f`.

2.1.4 Function Preconditions

The next step is to impose constraints on the arguments passed to the function by declaring a function precondition. The price for a function precondition is paid at the call sites of the function: new proof obligations are created for each function precondition for each call site. For example, we could start with declaring the simple precondition

$$c_1 : a_{\mathit{offset},0} \geq 0.$$

Taking this precondition as our initial condition for the function discharges proof obligations lb_7 and lb_8, so we only have six open proof obligations left in function `f`. Declaring this precondition, however, causes the following six new proof obligations to be created:

$$\begin{aligned}
p_{20,c1} &: s_{\mathit{offset}} && \geq 0 \\
p_{27,c1} &: (\&s[10])_{\mathit{offset}} && \geq 0 \\
p_{34,c1} &: (\&s[10])_{\mathit{offset}} && \geq 0 \\
p_{42,c1} &: s_{\mathit{offset}} && \geq 0 \\
p_{49,c1} &: s_{\mathit{offset}} && \geq 0 \\
p_{57,c1} &: s_{\mathit{offset}} && \geq 0
\end{aligned}$$

Fortunately all of these proof obligations are readily discharged with the invariants generated for these functions, so the precondition c_1 results in a net reduction of two open proof obligations.

Alternatively, we could impose a more aggressive precondition, say

$$\begin{aligned}
c_2 &: a_{\mathit{offset},0} && = 0\\
c_3 &: a_{\mathit{size}} && \geq 2\\
c_4 &: x_0 && = 0\\
c_5 &: y_0 && = 0
\end{aligned}$$

Taking this precondition as our initial condition for the function discharges all proof obligations in function `f`. We now, however, have 24 new proof obligations, shown below with their simplifications against the invariants generated at those locations:

$$\begin{aligned}
p_{20,c2} &: s_{\mathit{offset}} = 0 && \mathit{true}\\
p_{20,c3} &: s_{\mathit{size}} \geq 2 && \mathit{true}\\
p_{20,c4} &: 0 = 0 && \mathit{true}\\
p_{20,c5} &: 0 = 0 && \mathit{true}
\end{aligned}$$

$$\begin{aligned}
p_{27,c2} &: (\&s[10])_{\mathit{offset}} = 0 && \mathit{false}\\
p_{27,c3} &: s_{\mathit{size}} \geq 2 && \mathit{true}\\
p_{27,c4} &: 2 = 0 && \mathit{false}\\
p_{27,c5} &: 7 = 0 && \mathit{false}
\end{aligned}$$

$$\begin{aligned}
p_{34,c2} &: (\&s[10])_{\mathit{offset}} = 0 && \mathit{false}\\
p_{34,c3} &: s_{\mathit{size}} \geq 2 && \mathit{true}\\
p_{34,c4} &: -8 = 0 && \mathit{false}\\
p_{34,c5} &: -3 = 0 && \mathit{false}\\
p_{42,c2} &: s_{\mathit{offset}} = 0 && \mathit{true}\\
p_{42,c3} &: s_{\mathit{size}} \geq 2 && \mathit{true}\\
p_{42,c4} &: 13 = 0 && \mathit{false}\\
p_{42,c5} &: 14 = 0 && \mathit{false}
\end{aligned}$$

$$\begin{aligned}
p_{49,c2} &: s_{\mathit{offset}} = 0 && \mathit{true}\\
p_{49,c3} &: s_{\mathit{size}} \geq 2 && \mathit{true}\\
p_{49,c4} &: 42 = 0 && \mathit{false}\\
p_{49,c5} &: 14 = 0 && \mathit{false}
\end{aligned}$$

$$\begin{aligned}
p_{57,c2} &: s_{\mathit{offset}} = 0 && \mathit{true}\\
p_{57,c3} &: s_{\mathit{size}} \geq 2 && n_0 \geq 2\\
p_{57,c4} &: n - 10 = 0 && n_0 = -10\\
p_{57,c5} &: n - 3 = 0 && n_0 = -3
\end{aligned}$$

Thus, this precondition results in ten proof obligations that are inconsistent and three new open proof obligations, a net reduction of five open proof obligations. The high number of proof obligations that are inconsistent, however, suggests that the precondition is too strong, and that we should look for a weaker precondition.

Many more preconditions can be proposed. The most general precondition to function `f` that implies the validity of all proof obligations in this case can be derived to be:

$$a_{\mathit{offset},0} \geq 0 \wedge a_{\mathit{offset},0} + 2 \leq a_{\mathit{size}} \wedge (x_0 < y_0 \rightarrow (a_{\mathit{offset},0} + x_0 \geq -2 \wedge a_{\mathit{offset},0} + y_0 < a_{\mathit{size}} - 2))$$

which we could represent as follows:

$$\begin{array}{lrr}
c_1 : & a_{\mathit{offset},0} \geq & 0 \\
c_6 : & a_{\mathit{offset},0} \leq & a_{\mathit{size}} - 2 \\
c_7 : x_0 < y_0 \rightarrow & a_{\mathit{offset},0} + x_0 \geq & -2 \\
c_8 : x_0 < y_0 \rightarrow & a_{\mathit{offset},0} + y_0 < & a_{\mathit{size}} - 2
\end{array}$$

Taking this precondition as our initial condition for the function, all proof obligations are discharged in function `f`. It generates 24 new proof obligations as follows:

$$\begin{array}{lrl}
p_{20,c1} : s_{\mathit{offset}} & \geq 0 \rightsquigarrow \mathit{true} & \\
p_{20,c6} : s_{\mathit{offset}} & \leq s_{\mathit{size}} - 2 \rightsquigarrow \mathit{true} & \\
p_{20,c7} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + 0 & \geq -2 \rightsquigarrow \mathit{true} & \\
p_{20,c8} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + 0 & < s_{\mathit{size}} - 2 \rightsquigarrow \mathit{true} &
\end{array}$$

$$\begin{array}{lllr}
p_{27,c1} : (\&s[10])_{\mathit{offset}} & \geq 0 & \rightsquigarrow 10 \geq 0 & \mathit{true} \\
p_{27,c6} : (\&s[10])_{\mathit{offset}} & \leq s_{\mathit{size}} - 2 & \rightsquigarrow 10 < 21 & \mathit{true} \\
p_{27,c7} : x_0 < y_0 \rightarrow (\&s[10])_{\mathit{offset}} + 2 & \geq -2 & \rightsquigarrow 10 + 2 \geq -2 & \mathit{true} \\
p_{27,c8} : x_0 < y_0 \rightarrow (\&s[10])_{\mathit{offset}} + 7 & < s_{\mathit{size}} - 2 & \rightsquigarrow 10 + 7 < 21 & \mathit{true}
\end{array}$$

$$\begin{array}{lllr}
p_{34,c1} : (\&s[10])_{\mathit{offset}} & \geq 0 & \rightsquigarrow 10 \geq 0 & \mathit{true} \\
p_{34,c6} : (\&s[10])_{\mathit{offset}} & \leq s_{\mathit{size}} - 2 & \rightsquigarrow 10 + 2 < 15 & \mathit{true} \\
p_{34,c7} : x_0 < y_0 \rightarrow (\&s[10])_{\mathit{offset}} - 8 & \geq -2 & \rightsquigarrow 10 - 8 \geq -2 & \mathit{true} \\
p_{34,c8} : x_0 < y_0 \rightarrow (\&s[10])_{\mathit{offset}} - 3 & < s_{\mathit{size}} - 2 & \rightsquigarrow 10 - 3 < 13 & \mathit{true}
\end{array}$$

$$\begin{array}{lllr}
p_{42,c1} : s_{\mathit{offset}} & \geq 0 & \rightsquigarrow 0 \geq 0 & \mathit{true} \\
p_{42,c6} : s_{\mathit{offset}} & \leq s_{\mathit{size}} - 2 & \rightsquigarrow 0 < 14 & \mathit{true} \\
p_{42,c7} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + 13 & \geq -2 & \rightsquigarrow 13 \geq -2 & \mathit{true} \\
p_{42,c8} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + 14 & < s_{\mathit{size}} - 2 & \rightsquigarrow 14 < 14 & \mathit{false}
\end{array}$$

$$\begin{array}{lllr}
p_{49,c1} : s_{\mathit{offset}} & \geq 0 & \rightsquigarrow 0 \geq 0 & \mathit{true} \\
p_{49,c6} : s_{\mathit{offset}} & \leq s_{\mathit{size}} - 2 & \rightsquigarrow 0 < 6 & \mathit{true} \\
p_{49,c7} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + 42 & \geq -2 & \rightsquigarrow \mathit{false} \rightarrow 42 \geq -2 & \mathit{true} \\
p_{49,c8} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + 14 & < s_{\mathit{size}} - 2 & \rightsquigarrow \mathit{false} \rightarrow 14 < 4 & \mathit{true}
\end{array}$$

$$\begin{array}{lllr}
p_{57,c1} : s_{\mathit{offset}} & \geq 0 & \rightsquigarrow 0 \geq 0 & \mathit{true} \\
p_{57,c6} : s_{\mathit{offset}} & \leq s_{\mathit{size}} - 2 & \rightsquigarrow 0 < n_0 - 2 & \\
p_{57,c7} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + (n - 10) & \geq -2 & \rightsquigarrow n - 10 \geq -2 & n_0 \geq 8 \\
p_{57,c8} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + (n - 3) & < s_{\mathit{size}} - 2 & \rightsquigarrow n - 3 < n - 2 & \mathit{true}
\end{array}$$

Thus, this precondition identifies one inconsistent proof obligation ($p_{42,c8}$) and creates one new open proof obligation ($p_{57,c7}$).

2.1.5 Counterexample

The presence of an inconsistent proof obligation for a function call can be handled in two ways: (1) weakening the precondition of the function called or (2) exhibiting a counterexample and (to make the program safe) creating new preconditions for the calling function to ensure that the call is not reachable (which is the only way to satisfy an inconsistent proof obligation).

A counterexample can indeed by constructed for the call at line 42,

$$\begin{aligned}
\ell_{42} &: s_{\mathit{offset}} = 0, s_{\mathit{size}} = 16, \mathit{arg}_x = 13, \mathit{arg}_y = 14 \\
\ell_{3} &: a_{\mathit{offset}} = 0, a_{\mathit{size}} = 16, x = 13, y = 14 \\
\ell_{10} &: p_{\mathit{offset}} = 2, p_{\mathit{size}} = 16, x = 13, y = 14 \\
\ell_{12} &: i = 13, p_{\mathit{offset}} = 2, p_{\mathit{size}} = 16
\end{aligned}$$

The trace demonstrates that if function `f` is called with the given arguments the proof obligation

$$\mathit{ub}_{13b} : p_{\mathit{offset}} + i + 1 < p_{\mathit{size}}$$

is violated. The only way to make the call to `f` on line 42 unreachable is to declare the precondition for `g4` to be false, which will discharge the proof obligation $p_{42,c8}$, and creates a new inconsistent proof obligation

$$p_{67-\mathit{then},c9} : \mathit{false}$$

2.1.6 Constraints on User Input

The open proof obligation,

$$p_{57,c7} : x_0 < y_0 \rightarrow s_{\mathit{offset}} + (n - 10) \geq -2 \rightsquigarrow n_0 \geq 8$$

can be handled in the same way as before. We declare the function precondition

$$c_{10} : n_0 \geq 8$$

for function `g6`, which discharges the proof obligation $p_{57,c7}$ and creates two new proof obligations:

$$\begin{aligned}
p_{70-\mathit{then},c10} &: 25 \geq 8 \rightsquigarrow \mathit{true} \\
p_{71-\mathit{then},c10} &: (\mathit{strlen}(\mathit{argv}[2]) \geq 8
\end{aligned}$$

The second proof obligation imposes a constraint on user input: the length of the string passed as the first argument to the program must be at least 8.

2.1.7 Summary

Based on the analysis performed the program can be declared safe (with respect to the primary proof obligations in `f` that were considered here) if the inputs to the program satisfy the following minimum constraints on the first and and second command-line arguments, denoted by arg_1 and arg_2:

$$\begin{aligned}
U_1 &: \mathit{arg}_1 \neq g4 \\
U_2 &: (\mathit{arg}_1 = g6) \rightarrow \text{strlen}(\mathit{arg}_2) \geq 8
\end{aligned}$$

Notice that there are several alternative ways to “make” the program safe, all of which, however, would involve more restrictions on the user input. For example a trivial way to make the program safe is to impose the constraints

$$\begin{aligned}
U_1 &: arg_1 \neq g1\\
U_2 &: arg_1 \neq g2\\
U_3 &: arg_1 \neq g3\\
U_4 &: arg_1 \neq g4\\
U_5 &: arg_1 \neq g5\\
U_6 &: arg_1 \neq g6
\end{aligned}$$

which would allow the function preconditions to be declared *false* for all functions except `main`. Obviously this is not a very useful outcome of the analysis. The notion of the severity of the restrictions to be imposed on user input to allow the analysis to go through, however, could be a basis for scoring the game.

2.2 Challenges with BIND

The CSFV Program offered BIND as an example of an interesting candidate for analysis. Our team found interest in this source code because of its maturity (it had been well verified before we arrived), its code base size (there are many load modules including the tests), and most of all its *use of pointers*. Section 2.2.1 describes our findings upon an initial manual inspection of the code. This manual inspection led to a significant and exciting change in direction for our team and our static analysis technology. Section 2.2.2 then describes the initial approach we developed to tackle the new challenge. Section 2.2.3 describes some of the lingering technical challenges we thought we would encounter from the outset of the project, many of which we encountered and solved later in the project as explained herein.

2.2.1 Initial Manual Review

BIND comes as a set of distinct programs and tools (`named`, `nsupdate`, `dig`, etc.) that perform the various tasks involved in domain name resolution. The source code for each utility can be found under the `bin` directory of the standard distribution. All of those programs are based on a common API located under the `lib` directory of the distribution. BIND implements the nodes of a complex distributed system and it seems unlikely that we can perform a full verification of any property without some model of the environment. A manual review of the code, although incomplete, revealed that *most pending verification conditions are due to insufficient information on the pointer structure among objects in the memory.* This finding is what impelled our change in analysis technique for the project.

The review revealed some encouraging observations:

- Although extremely pointer-intensive, the BIND programs manipulate relatively flat data structures, without unions. Dynamic collections of objects seem to be mainly implemented using simply-linked lists, whereas arrays seem to be mostly used to store character strings and are usually statically allocated. The common points-to relation looks like: (malloc@c) + offset(f) -> (malloc@c') + 0. In other words, (1) there are not many pointers to a position somewhere inside a memory block, and (2) pointers usually sit in fields within a compound structure. Arrays of pointers occur and they are usually dynamically allocated. The position of an object in the array doesn't appear to carry any context information.

- There are function pointers, which seem to be essentially stored in fields of data structures. The dispatch procedures consist of cascading conditional statements, where the function pointers are invoked by explicitly accessing a field in some compound data structure. There appeared to be no use of arrays of function pointers.

The upshot is that a field-sensitive points-to model of the memory should provide good enough precision to discharge a significant number of verification conditions. Constant offsets should work fine and we see no need for a complex array-sensitive analysis, as objects in collections are uniformly manipulated.

This leaves us with the review's observations presenting challenges:

- Memory allocation is buried inside helper functions and this requires some level of context-sensitivity or inlining.
- Information about the size of a dynamically allocated memory block is often stored in a field of a larger structure that also contains a pointer to the block. After close inspection of pieces of code manipulating such structures, it seems highly unlikely that intervals would be sufficient to track the size information precisely. Some kind of relational information would have to be preserved.
- There are a few instances where complex pointer arithmetic is used, mainly to decode messages received or sent by the BIND utilities. There is little hope we can resolve the corresponding buffer-overflow checks in any case. These situations appear to be isolated and should have little impact on the overall verification process.

2.2.2 Overview of Approach

The process of determining a conservative approximation of the pointer structure in memory is called *pointer analysis*. We propose the crowd players to perform the pointer analysis. The game dynamics mimics that of the pointer analysis, whereas the players build the approximate memory graph by moving addresses (payloads of some sort) along the pointer constraints (the circuits). Each game level corresponds to the pointers constraints associated to a function. These constraints we generate in advance using CodeHawk and they are substantially smaller than the code itself. In particular, CodeHawk abstracts away all control structures (loops, conditional, jumps). Game play yields the memory graph in a distributed fashion. The game ends when the memory graph can no longer be changed by the players' actions.

For example, consider the following program:

```
/* example with two game levels main and set */
int *P1, *P2;
int G1, G2;
```

```
void set(int **p, int *a) {
  *p = a;
}
main() {
  set(&P1, &G1);
  set(&P2, &G2);
}
```

The function `set` can be modeled as a game level that takes two payloads (the arguments) and creates a pointer link between them (a connection in the pointer graph). A payload is a pair `(&x, off)` where the first component is the symbolic address of a variable (or dynamically allocated memory block) and `off` is an offset in bytes, which can be either a constant or a range of offsets. The offset information is automatically computed by CodeHawk and it does not have to appear as is in the game (it can be masqueraded into something more appealing to the player). The two function calls in the game level corresponding to the main function act by propagating a payload to another level (the one for `set`). Using this form of interaction, the players can construct the following memory graph:

`(&P1, 0) -> (&G1, 0)`, `(&P2, 0) -> (&G2, 0)`

The game controller detects stabilization when the memory graph can no longer be modified and there is no pending payload at the entry points of each level (i.e., the parameters of each function). This corresponds to a fixpoint computation and the game must make sure that all functions have been processed at least once between two such checks. A common strategy in pointer analysis is to process each function in some topological ordering of the call graph, starting from main. The game must therefore be organized in "waves" that go through the call graph without ignoring any function (level). This also imposes constraints on the way players are distributed and can move across levels.

2.2.3 Technical Challenges

There are additional difficulties that are not exposed in the simple example described in the previous section:

- Function pointers are heavily used in BIND and must be taken into account. There is no theoretical difficulty. It just means that new pathways between levels may be added as the game unfolds. When checking for stabilization, the game shall make sure that the call graph hasn't changed since the last check.

- Functions may also return pointers, which means there may be a two-way dependency across levels with entry points (the function parameters) and exit points (the return statements). This does not require any change in the algorithm, but it may lead to very long games if the call graph is explored in a top-down fashion (return values are propagated one level at a time at each pass). A good strategy that has proven effective in practice is to alternate top-down passes (from the main functions to the leaves) and bottom-up passes (from the leaves to the main function).

- The analysis of structures that contain an array of elements in one field and the size of the array in another would require carrying scalar values in the payloads. These values would also have to be related with the size of memory blocks. The review of BIND hinted at little or no gain in precision, would this feature be implemented.

- Functions that return a pointer to a dynamically allocated memory block pose a challenge. Our approach must be able to distinguish between a block that is created at two different calls of the function, otherwise it would confuse completely unrelated data structures. This happens in BIND and the return value is tunneled back across a number of function calls. This can be done at the CodeHawk level by inlining, i.e. expanding the pointer constraints of the called function into those of the calling function. This can be fine-tuned and does not change the model whatsoever. There is an additional difficulty in the memory management API of BIND, where one of the helper functions, which ultimately calls `malloc`, is a function pointer stored in a global variable. Again, this can be dealt with at the CodeHawk level and should not have any impact on the game model.

- The greatest challenge in our approach is hidden in the description of the example, where we say that players propagate payloads to the entry points of the next level (the called function). We must make sure that these values are somewhat related, otherwise by picking `&P1` for the first parameter and `&G2` for the second, a player could generate the spurious link `(&P1, 0) -> (&G2, 0)`. This is a problem known as context-sensitivity, which could cause a combinatorial explosion when pointer values are tunneled across long call chains. This issue is critical for precision.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section we present the main technical explanation of our approach, several examples to illustrate the techniques we programmed into the analyzers and games, and the three games we developed the embody the main iteration devices. To support the deployment and operation of the games we first developed a distributed software architecture and implemented a number of document stores and backend services shared by all of our games.

First, we construct the technical development behind the game model and iteration. Section 3.1 on the following page explains our approach to the C program pointer constraints problem, with Section 3.2 on page 20 providing an example of the procedure. Sections 3.3 on page 24 and 3.4 on page 30 then describe how we turned this approach into the design of a game model that served as the core of our distributed game play system.

Second, we explain implementation details supporting the verification. Section 3.5 on page 45 describes the implementation of the game model, which we embedded into the games and embedded into auto-solvers. Section 3.6 on page 75 describes the implementation of a static analyzer using the pointer analysis results to improve verification results.

Third, we explain the supporting architecture and services not specifically related to verification. Section 3.7 on page 98 describes our client-server architecture using the Amazon Web Services (AWS) cloud, an unstructured document store server, and a suite of stateless backend services. Section 3.7 on page 98 describes the architecture, Section 3.8 on page 100 the backend services, and Section 3.9 on page 106 the related web services between the game, backend, and supporting sign-on and social media services.

Finally, we describe the details of each of the three games. Section 3.10 on page 106 describes the CircuitBot game. Section 3.11 on page 130 describes the Dynamakr game. Section 3.12 on page 146 describes the VIPER game.

3.1 Semantics of Pointer Constraints

Section 3.1.1 describes our development of the constraint generator's syntax specification. Sections 3.1.2 on the following page, 3.1.3 on page 18, and 3.1.4 on page 20 then describe the semantics of domains, constraints, and fixpoint iteration to support the game play and subsequent verification.

3.1.1 Syntax of Constraints

We assume that we are provided with a set $\mathcal{F}$ of symbols denoting the names of functions in the program and two disjoint sets of variables $\mathcal{V}_a$ and $\mathcal{V}_n$ used to represent address and offset information respectively in points-to constraints. We denote by $\mathcal{L}$ a set of symbols denoting the addresses of global data (global variables, dynamically allocated memory blocks, etc.). We assume that there is a special type of location $\mathbf{fun}(f)$, which represents the location of a function $f \in \mathcal{F}$ and is used to model function pointers. We denote by $\mathbf{I}$ the set of intervals over $\mathbb{Z} \cup \{-\infty, +\infty\}$ defined in the standard way. Intervals can be endowed with the structure of a lattice $(\mathbf{I}, \subseteq, \sqcup, \bot, \sqcap, \top)$ and the extension of the usual arithmetic over $\mathbb{Z} \cup \{-\infty, +\infty\}$. Note that this lattice is complete for both $\sqcup$ and $\sqcap$.

Each function f in $\mathcal{F}$ is assigned a set $\mathcal{V}_a(f)$ of symbolic variables and a set $\mathcal{V}_n(f)$ of numerical variables. These sets are disjoint for all functions in $\mathcal{F}$. Each function f has a set $\{f_1 \dots f_n\}$ of formal parameters and a return value f_r. We assume that each function has a return value for the sake of uniformity.

Constraints are split up among **Read** constraints and **Write** constraints. We first define a general notion of address as follows:

$$\mathbf{Address} ::= \mathbf{var}(a) \mid \mathbf{loc}(\ell)$$

where $a \in \mathcal{V}_a$ and $\ell \in \mathcal{L}$. A **Read** constraint has the following syntax:

$$\mathbf{Read} ::= (a, o) \supseteq rhs_1 \cup \dots \cup rhs_n$$

where $a \in \mathcal{V}_a$, $o \in \mathcal{V}_n$ and the rhs_i are right-hand side expressions **ReadRhs** defined as follows:

$$\mathbf{ReadRhs} ::= (a, e) \mid \mathbf{deref}(a, e) \mid f_r \mid f_i \mid \mathbf{retfp}(a, e)$$

where $a \in \mathbf{Address}$, $f \in \mathcal{F}$, f_i is a formal parameter of function f, f_r is the return value of function f and e is an offset expression **OffsetExpr** defined as follows:

$$\mathbf{OffsetExpr} ::= i \mid c_0 + c_1 o_1 + \dots + c_n o_n$$

where $i \in \mathbf{I}$ is an interval, $c_0, \dots, c_n \in \mathbb{Z}$ are integer coefficients and $o_1, \dots, o_n \in \mathcal{V}_n$ are numerical variables. An offset expression is either an interval or the linear combination of numerical offset variables.

A **Write** constraint is defined as follows:

$$\begin{array}{rcl}\textbf{Write} & ::= & *(a,e) \leftarrow (a',e') \\ & | & f_i \leftarrow (a,e) \\ & | & \textbf{fp}_i(a,e) \leftarrow (a',e') \\ & | & f_r \leftarrow (a,e) \\ & | & \textbf{memcpy}(a,e,a',e',l)\end{array}$$

where $a, a' \in$ **Address**, e, e' are offset expressions defined by **OffsetExpr**, f_i is a formal parameter of function $f \in \mathcal{F}$, f_r is the return value of function f, $i \in \mathbb{N}$ and $l \in \mathbf{I}$.

A system of constraints S is given by a collection of **Read** constraints $\mathbf{Read}_S(f)$ and **Write** constraints $\mathbf{Write}_S(f)$ for each function $f \in \mathcal{F}$, such that each variable in a constraint associated to f belongs to either $\mathcal{V}_a(f)$ or $\mathcal{V}_n(f)$. In other words, there are no free variables in a constraint system.

3.1.2 Semantic Domains

In the rest of this document, we will use the notations $\sqsubset$, $\sqcup$ and $\sqcap$ to denote the operations of a lattice without specifying which lattice they refer to, as this should be obvious from the context. The power set $\wp(\mathcal{L})$ ordered by set inclusion can be endowed with the structure of a complete lattice. We denote by $\mathbf{M}$ the product lattice $\wp(\mathcal{L}) \times \mathbf{I}$. An element of $\mathbf{M}$ is a pair (a, i), where a is a set of addresses and i is an interval of byte offsets. Let $\mathbf{V}(\mathcal{F})$ be the set of all formal parameters f_i and return values f_r for all functions $f \in \mathcal{F}$. We denote by $\mathbf{F}$ the product lattice defined as follows

$$\mathbf{F} = \prod_{v \in \mathbf{V}(\mathcal{F})} \mathbf{M}$$

and ordered by the pointwise extension of the order on $\mathbf{M}$. If $\mathbf{f} \in \mathbf{F}$, we denote by $\mathbf{f}(v)$ the v-th component of $\mathbf{f}$. If $\mathbf{f} \in \mathbf{F}$, $v \in \mathbf{V}(\mathcal{F})$ and $m \in \mathbf{M}$, we denote by $\mathbf{f} \oplus (v, m)$ the element $\mathbf{f}' \in \mathbf{F}$, such that $\mathbf{f}'(v) = \mathbf{f}(v) \cup m$ and $\mathbf{f}'(v') = \mathbf{f}(v')$ if $v' \neq v$. If $A = \{(v_1, m_1), \ldots, (v_n, m_n)\}$, we denote by $\mathbf{f} \oplus A$ the element $(\ldots(\mathbf{f} \oplus (v_1, m_1)) \cdots \oplus (v_n, m_n))$.

If $f \in \mathcal{F}$, we denote by $\mathbf{E}_f$ the product lattice defined as follows

$$\mathbf{E}_f = \left(\prod_{v \in \mathcal{V}_a(f)} \wp(\mathcal{L}) \right) \times \left(\prod_{v \in \mathcal{V}_n(f)} \mathbf{I} \right)$$

We denote by $\mathbf{E}$ the product lattice given by

$$\mathbf{E} = \prod_{f \in \mathcal{F}} \mathbf{E}_f$$

If $\mathbf{e} \in \mathbf{E}$, we denote by $\mathbf{e}_f$ the f-th component of $\mathbf{e}$ and by $\mathbf{e}_f(v)$ the v-th component of $\mathbf{e}_f$ for $v \in \mathcal{V}_a(f) \cup \mathcal{V}_n(f)$. If $\mathbf{e} \in \mathbf{E}$, $f \in \mathcal{F}$, $v \in \mathcal{V}_a(f) \cup \mathcal{V}_n(f)$ and x is an element of $\wp(\mathcal{L})$ or $\mathbf{I}$ depending on the type of v, we denote by $\mathbf{e} \oplus (f, v, x)$ the element of $\mathbf{E}$ that coincides with $\mathbf{e}$ everywhere except on the f, v-th component, where it is equal to $\mathbf{e}_f(v) \sqcup x$. If A is a finite set of a triples (f, v, x), we define $\mathbf{e} \oplus A$ as previously.

We denote by **I2** the set $\wp(\mathbf{I}\times\mathbf{I})$ of sets of boxes of $\mathbb{Z}^2$. If $O \in \mathbf{I2}$, we define the denotation $\delta(O) \in \wp(\mathbb{Z}^2)$ of O as follows

$$\delta(O) = \bigcup_{(i,j)\in O} (i \times j)$$

δ defines a preorder $\preceq$ on **I2** given by

$$O \preceq O' \text{ iff } \delta(O) \subseteq \delta(O')$$

We define the equivalence relation $\equiv$ on **I2** as

$$O \equiv O' \text{ iff } \delta(O) = \delta(O')$$

We denote by **O** the quotient set $\mathbf{I2}/_{\equiv}$ ordered by the projection of $\preceq$. This endows **O** with the structure of a complete lattice.

We define the domain **G** of points-to graphs as

$$\mathbf{G} = \prod_{(s,t)\in\mathcal{L}^2} \mathbf{O}$$

endowed with the structure of the product lattice. If $\mathbf{g} \in \mathbf{G}$ and $(s,t) \in \mathcal{L}^2$, we denote by $\mathbf{g}(s,t)$ the (s,t)-th component of $\mathbf{g}$. If $s,t \in \mathcal{L}$ and $i,j \in \mathbf{I}$, we denote by $\mathbf{g} \oplus (s,i,j,t)$ the points-to graph that coincides with $\mathbf{g}$ everywhere, except on (s,t) where it is equal to $\mathbf{g}(s,t) \sqcup \{(i,j)\}$. If A is a finite set of tuples (s,i,j,t), we define $\mathbf{g} \oplus A$ as previously.

Finally, the domain **C** of game configurations is defined as

$$\mathbf{C} = \mathbf{F} \times \mathbf{E} \times \mathbf{G}$$

and is endowed with the structure of the product lattice. The solution of the game will be defined in the next section as the least fixpoint of a monotonic operator on **C**. Note that the previous definitions of operation $\oplus$ can be readily extended so as to have an infinite set as the right-hand side operand.

3.1.3 Semantics of Constraints

Let $\mathbf{c} = (\mathbf{f}, \mathbf{e}, \mathbf{g})$ be a game configuration in **C**. Let $f \in \mathcal{F}$ be a function. We want to define the semantics of points-to constraints for f. The semantics of an **Address** a is a function $[\![a]\!]_f : \mathbf{C} \to \wp(\mathcal{L})$ defined as follows

$$\begin{array}{rcl} [\![\mathbf{var}(a)]\!]_f\mathbf{c} & = & \mathbf{e}(a) \\ [\![\mathbf{loc}(\ell)]\!]_f\mathbf{c} & = & \{\ell\} \end{array}$$

The semantics of an **OffsetExpr** e is a function $[\![e]\!]_f : \mathbf{C} \to \mathbf{I}$ defined as follows

$$\begin{array}{rcl} [\![i]\!]_f\mathbf{c} & = & i \\ [\![c_o + c_1 o_1 + \cdots + c_n o_n]\!]_f\mathbf{c} & = & [c_o, c_0] +_{\mathbf{I}} ([c_1, c_1] \times_{\mathbf{I}} \mathbf{e}_f(o_1)) +_{\mathbf{I}} \cdots \\ & & +_{\mathbf{I}} ([c_n, c_n] \times_{\mathbf{I}} \mathbf{e}_f(o_n)) \end{array}$$

The semantics of an **ReadRhs** expression rhs is a function $\llbracket rhs \rrbracket_f : \mathbf{C} \to \wp(\mathbf{M})$ defined as follows

$$\begin{array}{rcl}
\llbracket (a,e) \rrbracket_f \mathbf{c} & = & \{(\llbracket a \rrbracket_f \mathbf{c}, \llbracket e \rrbracket_f \mathbf{c})\} \\
\llbracket g_r \rrbracket_f \mathbf{c} & = & \{\mathbf{f}(g_r)\} \\
\llbracket g_i \rrbracket_f \mathbf{c} & = & \{\mathbf{f}(g_i)\} \\
\llbracket \mathbf{deref}(a,e) \rrbracket_f \mathbf{c} & = & \{(\{t\}, i) \mid \exists s \in \llbracket a \rrbracket_f \mathbf{c} : \exists i' \in \mathbf{I} : (i', i) \in \mathbf{g}(s,t) \\
 & & \quad \wedge \llbracket e \rrbracket_f \mathbf{c} \sqcap i' \neq \bot\} \\
\llbracket \mathbf{retfp}(a,e) \rrbracket_f \mathbf{c} & = & \{\mathbf{f}(g_r) \mid \exists s \in \llbracket a \rrbracket_f \mathbf{c} : \exists i, i' \in \mathbf{I} : (i', i) \in \mathbf{g}(s, \mathbf{fun}(g)) \\
 & & \quad \wedge \llbracket e \rrbracket_f \mathbf{c} \sqcap i' \neq \bot \wedge i \neq \bot\}
\end{array}$$

If cst is a **Read** constraint $(a, o) \supseteq rhs_1 \cup \cdots \cup rhs_n$, we denote by R the set

$$R = \llbracket rhs_1 \rrbracket_f \mathbf{c} \cup \cdots \cup \llbracket rhs_n \rrbracket_f \mathbf{c}$$

Then, the semantics of cst is a function

$$\llbracket cst \rrbracket_f \mathbf{c} : \mathbf{C} \to \wp\left((\mathcal{V}_a(f) \times \wp(\mathcal{L})) \cup (\mathcal{V}_n(f) \times \mathbf{I})\right)$$

defined as follows

$$\begin{array}{l}
\llbracket cst \rrbracket_f \mathbf{c} = \{(a, \ell) \mid \exists A \in \wp(\mathcal{L}) : \exists i \in \mathbf{I} : (A, i) \in R \wedge \ell \in A\} \\
\qquad\qquad \cup \{(o, i) \mid \exists A \in \wp(\mathcal{L}) : (A, i) \in R\}
\end{array}$$

Now, if S is a system of constraints, the semantics of the **Read** constraints in S is the function

$$\llbracket S \rrbracket_{\mathbf{Read}} : \mathbf{C} \to \wp\left((\mathcal{F} \times \mathcal{V}_a \times \mathcal{L}) \cup (\mathcal{F} \times \mathcal{V}_n \times \mathbf{I})\right)$$

defined as follows

$$\llbracket S \rrbracket_{\mathbf{Read}} \mathbf{c} = \{(f, v, x) \mid f \in \mathcal{F} \wedge \exists cst \in \mathbf{Read}_f(S) : (v, x) \in \llbracket cst \rrbracket_f \mathbf{c}\}$$

The semantics $\llbracket cst \rrbracket_f$ of a **Write** constraint cst is the function

$$\llbracket cst \rrbracket_f : \mathbf{C} \to \wp(\mathbf{V}(\mathcal{F}) \times \mathbf{M}) \times \wp(\mathcal{L} \times \mathbf{I}^2 \times \mathcal{L})$$

defined as follows

$$\begin{array}{rcl}
\llbracket *(a,e) \leftarrow (a', e') \rrbracket_f \mathbf{c} & = & (\emptyset, \{(s, \llbracket e \rrbracket_f \mathbf{c}, \llbracket e' \rrbracket_f \mathbf{c}, t) \mid s \in \llbracket a \rrbracket_f \mathbf{c} \wedge t \in \llbracket a' \rrbracket_f \mathbf{c}\}) \\
\llbracket f_i \leftarrow (a,e) \rrbracket_f \mathbf{c} & = & (\{(f_i, (\ell, \llbracket e \rrbracket_f \mathbf{c})) \mid \ell \in \llbracket a \rrbracket_f \mathbf{c}\}, \emptyset) \\
\llbracket \mathbf{fp}_i(a,e) \leftarrow (a', e') \rrbracket_f \mathbf{c} & = & (\{(f_i, (\ell, \llbracket e' \rrbracket_f \mathbf{c})) \mid \ell \in \llbracket a' \rrbracket_f \mathbf{c} \wedge \exists s \in \llbracket a \rrbracket_f \mathbf{c} : \exists i, i' \in \mathbf{I} : \\
 & & \quad (i, i') \in \mathbf{g}(s, \mathbf{fun}(f)) \wedge \llbracket e \rrbracket_f \mathbf{c} \sqcap i \neq \bot \wedge i' \neq \bot\}, \emptyset) \\
\llbracket f_r \leftarrow (a,e) \rrbracket_f \mathbf{c} & = & (\{(f_r, (\ell, \llbracket e \rrbracket_f \mathbf{c})) \mid \ell \in \llbracket a \rrbracket_f \mathbf{c}\}, \emptyset) \\
\llbracket \mathbf{memcpy}(a, e, a', e', l) \rrbracket_f \mathbf{c} & = & (\emptyset, \{(s, i -_{\mathbf{I}} \llbracket e' \rrbracket_f \mathbf{c} +_{\mathbf{I}} \llbracket e \rrbracket_f \mathbf{c}, i', t) \mid s \in \llbracket a \rrbracket_f \mathbf{c} \\
 & & \qquad \wedge \exists s' \in \llbracket a' \rrbracket_f \mathbf{c} : (i, i') \in \mathbf{g}(s', t) \\
 & & \qquad \wedge i \sqcap (\llbracket e' \rrbracket_f \mathbf{c} +_{\mathbf{I}} l) \neq \bot \wedge i' \neq \bot\})
\end{array}$$

Now, if S is a system of constraints, the semantics of the **Write** constraints in S is the function

$$\llbracket S \rrbracket_{\mathbf{Write}} : \mathbf{C} \to \wp(\mathbf{V}(\mathcal{F}) \times \mathbf{M}) \times \wp(\mathcal{L} \times \mathbf{I}^2 \times \mathcal{L})$$

defined as follows

$$[\![S]\!]_{\mathbf{Write}} : \mathbf{C} \to [\![S]\!]_{\mathbf{Write}}\mathbf{c} = \dot{\bigcup_{f \in \mathcal{F}}} \{[\![cst]\!]_f \mathbf{c} \mid cst \in \mathbf{Write}_f(S)\}$$

where $\dot{\cup}$ denotes the pointwise extension of the set union operator to the components of a pair.

3.1.4 Fixpoint Semantics of the Game

The semantics of a game given by a system of constraints S is expressed as the least fixpoint of an extensive function $\mathbb{F} : \mathbf{C} \to \mathbf{C}$. Let $\mathbf{c} = (\mathbf{f}, \mathbf{e}, \mathbf{g})$ be a game configuration. We denote by (f, g) the semantics $[\![S]\!]_{\mathbf{Write}}\mathbf{c}$ of **Write** constraints in S. Then $\mathbb{F}$ is defined as follows

$$\mathbb{F}(\mathbf{c}) = (\mathbf{f} \oplus f, \mathbf{e} \oplus [\![S]\!]_{\mathbf{Read}}\mathbf{c}, \mathbf{g} \oplus g)$$

Following Tarski's theorem, the least fixpoint of $\mathbb{F}$ is the limit of the transfinite sequence $(\mathbb{F}_n)_{n \geq 0}$ defined inductively as follows:

$$\begin{cases} \mathbb{F}_0 & = & \bot \\ \mathbb{F}_{n+1} & = & \mathbb{F}(\mathbb{F}_n) \\ \mathbb{F}_\lambda & = & \bigsqcup\{\mathbb{F}_n \mid n < \lambda\} \text{ for a limit ordinal } \lambda \end{cases}$$

Playing the game amounts to applying a certain iteration strategy to get to the least fixpoint. Approximations have to be inserted during the game play as the least fixpoint may no be computable in finite time. Therefore, we generally end up with a post-fixpoint. This is sufficient to carry out buffer overflow analysis in a sound manner though.

3.2 Procedure Example

We will consider the following C program as our running example:

```
struct S {
  int *p1;
  int *p2;
};

int A[10];
int B[10];

void init1(struct S *p) {
  p->p1 = &A[0];
}

void init2(struct S *p) {
  volatile int random;
  p->p2 = random ? &B[5] : &B[0];
}

main() {
  struct S *pS;

  pS = malloc(sizeof(struct S)); // Label M

  init1(pS);
  init2(pS);

  pS->p1[5] = pS->p2[5];
}
```

3.2.1 Memory Model

We assume that pointers and integers are four bytes long on the platform considered. An abstract memory location L is a triple (a, o, s):

- a is either the address of a program variable $\&v$, a memory block `block@l` dynamically allocated at program location l, or the address $\&f$ of a function (function pointers are supported by the analysis)
- o is an offset inside the memory block expressed as an interval of bytes
- s is the size of the memory block expressed as an interval of bytes

The nodes of the points-to graph are pairs (a, s), where a is the address of a memory block and s is the size of the block, as described above. An edge $(a, s) \to (o, o') \to (a', s')$ is labeled by a pair of interval (o, o') of byte offsets. The interval o denotes the position in the source block while o' denotes the position within the target block. For clarity, in the rest of the document a singleton interval $[n, n]$ will simply be denoted by n.

3.2.2 Generation of Points-To Constraints

After running an analysis with CodeHawk, we obtain the following system of constraints:

```
init1 {
  param1 + 0 → (&A, 0, 4 0)
}

init2 {
  param1 + 4 → (&B, 20, 40) param1 + 4 → (&B, 0, 40)
}

main {
  pS → (block@M, 0, 8)
  init1(param1 = pS)
  init2(param1 = pS)
  let tmp = get_points_to_targets(pS, 0) in
    check(0 ≤ offset(tmp) + 20 ≤ size(tmp) - 4)
  let tmp = get_points_to_targets(pS, 4) in
    check(0 ≤ offset(tmp) + 20 ≤ size(tmp) - 4)
}
```

The function `size` (resp. `offset`) returns the size of (resp. position inside) the memory block denoted by the memory location given in argument.

The statement `check` denotes an array-bound safety condition that can be discharged (or not) at the end of the analysis. For clarity we have omitted the safety conditions corresponding to the dereference of `pS` (in this case, they can be solved statically by CodeHawk).

All sizes and offsets have been expressed in bytes. We assume that all memory objects are aligned on 32-bit word boundaries.

3.2.3 Resolution of Points-To Constraints

We need to define an environment for each function in the system that keeps track of the memory locations assigned to the parameters and local variables. The types of the objects manipulated by the analysis are defined as follows:

$$\begin{aligned}
\mathit{SymbolicAddresses} &= \{\mathit{block@l}, \&v, \&\mathit{function}, \ldots\} \\
\mathit{Function} &= \{f, g, \ldots\} \\
\mathit{Parameters} &= \{\mathit{param1}, \mathit{param2}, \ldots\} \\
\mathit{PointsToGraphs} &= \wp(\mathit{SymbolicAddresses} \times \mathit{ZInt} \times \mathit{ZInt} \times \mathit{SymbolicAddresses}) \\
\mathit{MemoryLocations} &= \mathit{SymbolicAddresses} \times \mathit{ZInt} \times \mathit{ZInt} \\
\mathit{PointsToSets} &= \wp(\mathit{MemoryLocations}) \\
\mathit{Environments} &= \mathit{PointsToSets}^{\mathit{Parameters}} \\
\mathit{Configurations} &= \mathit{PointsToGraphs} \times \mathit{Environments}^{\mathit{Functions}}
\end{aligned}$$

Here $\wp$ denotes the powerset operator and *ZInt* denotes the set of integer intervals (with possibly one or two open bounds $\pm\infty$. The main object manipulated by the analysis and produced by the game is an element of *Configuration*.

Initially, the environments in the configuration are empty and so is the points-to graph:

```
Points-to graph: empty
Environment init1:param1 → {}
Environment init2: param1 → {}
Environment main: pS → {}
```

Functions can be processed in any order as long as no function is ever left over (in Abstract Interpretation terms, this is called a chaotic iteration strategy with a fairness assumption). We stop processing functions when the points-to graph and the environments can no longer be changed.

We choose an arbitrary processing order (which coincidentally is the optimal one). A key aspect of the game approach is to leave it to the brains of players to choose a good iteration strategy. This is notoriously difficult to do automatically and in practice the iteration strategy often has to be manually tweaked in order to get decent computation times. That can be seen as a benefit of encoding the pointer analysis problem as a game.

We process the points-to constraints linearly in the order they appear. Again, this is arbitrary (and also optimal in this case) and should be left to the player?s discretion.

We start by processing function `main`. Variable `pS` gets assigned a memory location, which in turn is transmitted to functions `init1`and `init2`. Once we?re done processing function `main` we obtain the configuration:

```
Points-to graph: empty
Environment init1:param1 → { ( block@M, 0, 8) }
Environment init2: param1 → { (block@M, 0, 8) }
Environment main: pS → { (block@M, 0, 8) }
```

We process function `init1`. The single constraint in the function creates an edge in the points-to graph, originating from offset 4 in the dynamically allocated memory block and pointing to the first element of array `A`. At the end we obtain the configuration:

```
Points-to graph: {(block@M,8) → (0,0) → (&A,40)}
Environment init1:param1 → { ( block@M, 0, 8) }
Environment init2: param1 → { (block@M, 0, 8) }
Environment main: pS → { (block@M, 0, 8) }
```

We process function `init2`. Similarly, the function creates two new edges in the graph:

```
{(block@M,8) → (4,20) → (&B,40)}
```

and

```
{(block@M,8) → (4,0) → (&B,40)}
```

We have several options here. We could just decide to leave these two edges in the graph. However, this approach might result in a combinatorial explosion for larger and more complex programs. Another option is to allow only one edge between two nodes in the graph. In this case, we simply perform the convex union on the offsets labeling the edges, so as to obtain a sound approximation. Hence, at the end of this process we get the configuration:

```
Points-to graph: {(block@M,8) → (0,0) → (&A,40),
                           (block@M,8) → (4, [0,20]) → (&B,40)}
Environment init1:param1 → { ( block@M, 0, 8) }
Environment init2: param1 → { (block@M, 0, 8) }
Environment main: pS → { (block@M, 0, 8) }
```

If we process all functions one more time, the points-to graph and function environments can no longer be changed. We have thus reached a fixpoint, and we can now proceed with the verification of the safety conditions.

3.2.4 Verifying the Array-Bound Checks

For the first safety condition, we need to evaluate `get_points_to_targets(pS, 0)`, which returns after examination of the points-to graph the unique memory location `(&A, 0, 40)`. The safety condition can then be rewritten to $0 \leq 0 + 20 \leq 36$, which is true. This array-bound condition is therefore discharged by the analysis.

Evaluating `get_points_to_targets(pS, 4)` in the second safety condition returns the memory location `(&B, [0, 20], 40)`. Using interval arithmetic, we need to check that $0 \leq [0, 20] + 20 \leq 36$. The left-hand side of the inequality is fine, but the right-hand side cannot be ascertained (we manipulate over-approximations and we cannot say for sure that all values in the interval do occur in reality). In this case, we issue a warning, as there is a potential off-by-one array access error.

Approved for Public Release; Distribution Unlimited.

3.3 Formalization of the Game Model

This section describes the game model's conceptual realization in terms of states, visualization, and game rules. We describe how the rules affect game play and how the game results become products for subsequent program verification.

3.3.1 Representation

There are just four atomic elements in a system of pointer constraints:

- A function identifier $\mathbf{f}id$
- A symbolic variable $\mathbf{a}id$
- A numerical variable $\mathbf{o}id$
- The location of an object in memory $\mathbf{loc}_{id}$

These four elements are uniquely determined by an integer identifier id. Symbolic and numerical variables always come as a pair $(\mathbf{a}id, \mathbf{o}id)$. Inside the game, those components shall always be referred to using their unique identifier. The connection between the unique identifiers and the name of the actual program components (functions, variable, arguments, etc.) is recorded in the dictionary and used by CodeHawk to produce the verification reports. The only use of the verification dictionary in the game is for determining whether a memory object $\mathbf{loc}_{id}$ represents a function (for function pointer resolution).

On top of these four atomic components come function parameters $\mathbf{f}id_n$, where n is the rank of the argument, and function return values $\mathbf{f}id_r$.

3.3.2 Internal State

The internal state of the game is represented by

1. A collection of symbolic assignments:

$$\begin{aligned} \mathbf{a}id &\mapsto \{\mathbf{loc}_{n1}, \mathbf{loc}_{n2}, \dots\} \\ \mathbf{f}id_n &\mapsto \{\mathbf{loc}_{m1}, \mathbf{loc}_{m2}, \dots\} \\ \mathbf{f}id_r &\mapsto \{\mathbf{loc}_{k1}, \mathbf{loc}_{k2}, \dots\} \end{aligned}$$

2. A collection of internal assignments:

$$\begin{aligned} \mathbf{o}id &\rightsquigarrow [a, b] \\ \mathbf{f}id_n &\rightsquigarrow [a', b'] \\ \mathbf{f}id_r &\quad\;\; [a'', b''] \end{aligned}$$

3. And a collection of points-to edges:

$$(\mathbf{loc}_1, [a, b]) \rightarrow (\mathbf{loc}_2, [a', b'])$$

There may be more than one assignment to a game element. We assume that the game maintains a canonical representation of the internal state (keeping a single points-to sets for each symbolic variable for example).

3.3.3 Visualization

The nodes of the game consist of:

- Symbolic variables $\mathbf{a}id$
- Function arguments $\mathbf{f}id_n$
- Function return values $\mathbf{f}id_r$
- Memory locations $\mathbf{loc}_{id}$

Given an internal state:

$$\begin{aligned}
\mathbf{a}id &\mapsto \{\mathbf{loc}_{n1}, \mathbf{loc}_{n2}, \dots\} \\
\mathbf{f}id_n &\mapsto \{\mathbf{loc}_{m1}, \mathbf{loc}_{m2}, \dots\} \\
\mathbf{f}id_r &\mapsto \{\mathbf{loc}_{k1}, \mathbf{loc}_{k2}, \dots\} \\
\mathbf{o}id &\rightsquigarrow [a, b] \\
\mathbf{f}id_n &\rightsquigarrow [a', b'] \\
\mathbf{f}id_r &\quad\ [a'', b''] \\
(\mathbf{loc}_1, [a, b]) &\rightarrow (\mathbf{loc}_2, [a', b'])
\end{aligned}$$

edges consist of

$$\begin{aligned}
\mathbf{a}id &\Rightarrow \mathbf{loc}_{n1} \\
\mathbf{a}id &\Rightarrow \mathbf{loc}_{n2} \\
\mathbf{f}id_n &\Rightarrow \mathbf{loc}_{m1} \\
\mathbf{f}id_n &\Rightarrow \mathbf{loc}_{m2} \\
\mathbf{f}id_r &\Rightarrow \mathbf{loc}_{k1} \\
\mathbf{f}id_r &\Rightarrow \mathbf{loc}_{k2} \\
\mathbf{loc}_1 &\Rightarrow \mathbf{loc}_2
\end{aligned}$$

The numerical information can be attached to edges in an abstract way that is easily understandable by the players (e.g., energy level).

Notation. *If exp is an offset expression, as described in the semantic specification Section 3.1, we denote by* **eval**(*exp*) *its evaluation as an interval in the current internal state.*

An alpha constraint is satisfied if every target of its right node is a target of its left node.

3.3.4 Constraints in the Game Play

1. An $\alpha(l, r)$ constraint is satisfied if every target of its right node is a target of its left node.

2. A $\beta(l, r)$ constraint is satisfied if every indirect target of its right node is a target of its left node.

3. A $\gamma(l, r)$ constraint is satisfied if every target of its right node is a target of every target of its left node.

4. A $\delta(l, r)$ constraint is satisfied if every indirect target of its right node is a target of every target of its left node.

5. An $\alpha_0(l, r)$ constraint is satisfied if its right node is a target of its left node.

3.3.5 Game Rules

We denote by $\mathbf{p}$ the relation used in the game play constraints. For example, $\beta(l, r)$ is equivalent to $\forall x \forall y. \mathbf{p}(\mathbf{r}, x) \wedge \mathbf{p}(\mathbf{x}, y) \Rightarrow \mathbf{p}(\mathbf{l}, y)$.

1. **Constraints**: $(\mathbf{a}id, \mathbf{o}id) \supseteq (\mathbf{a}id', e), aid \supseteq \mathbf{a}id', \mathbf{o}id' \supseteq e$

   (a) **Constraint**: $(\mathbf{a}id, \mathbf{o}id) \supseteq (\mathbf{a}id', e)$

   if $\mathbf{a}id' \mapsto S$ and $\mathbf{eval}(e) = [a, b]$
   then $\mathbf{a}id \mapsto S$ and $\mathbf{o}id \mapsto [a, b]$

   $\forall x. \mathbf{p}(\mathbf{a}id', x) \Rightarrow \mathbf{p}(\mathbf{a}id, x)$

   $\mathbf{alpha}(\mathbf{a}id, \mathbf{a}id')$

   if $\mathbf{a}id' \mapsto \{\mathbf{loc}\}$ and $\mathbf{eval}(e) = [a, b]$
   then $\mathbf{a}id \mapsto \{\mathbf{loc}\}$ and $\mathbf{o}id \mapsto [a, b]$

   $\mathbf{alpha0}(\mathbf{a}id, \mathbf{a}id')$

   If you have **loc** on the right-hand side of an expression, then it is equivalent to an $\mathbf{a}id \mapsto \{\mathbf{loc}\}$ for a fresh id. Similarly, if you have an expression on the right-hand side, it can be interpreted as $\mathbf{o}id = e$ for a fresh id.

   (b) **Constraint**: $\mathbf{a}id \supseteq \mathbf{a}id'$ or $\mathbf{a}id \supseteq \mathbf{loc}$

   if $\mathbf{a}id' \mapsto S$
   then $\mathbf{a}id \mapsto S$

   $\forall x. \mathbf{p}(\mathbf{a}id', x) \Rightarrow \mathbf{p}(\mathbf{a}id, x)$

   $\mathbf{alpha}(\mathbf{a}id, \mathbf{a}id')$

   If the right-hand side of the constraint is a location, then $\mathbf{a}id \mapsto \{\mathbf{loc}\}$.

   (c) **Constraint**: $\mathbf{o}id \supseteq e$

   if $\mathbf{eval}(e) = [a, b]$
   then $\mathbf{o}id \rightsquigarrow [a, b]$

2. **Constraint**: $(\mathbf{a}id, \mathbf{o}id) \supseteq \mathbf{deref}(\mathbf{a}id', e)$ or $\mathbf{deref}(\mathbf{loc}_1, e)$

   if rhs is $\mathbf{a}id'$ and $\mathbf{a}id'' \mapsto \{\mathbf{loc}_1, \dots\}$ or rhs is $\mathbf{loc}_1$, then

   if $\mathbf{loc}_1$ is a function and $\mathbf{eval}(e) \neq \emptyset$ then $\mathbf{a}id \mapsto \{\mathbf{loc}_1\}$ and $\mathbf{o}id \rightsquigarrow [-\infty, +\infty]$
    else
    if $\mathbf{eval}(e) = [a, b]$ and $(\mathbf{loc}_1, [a', b']) \rightarrow (\mathbf{loc}_2, [a'', b''])$
    and $[a, b] \bigcap [a', b'] \neq \emptyset$
    then $\mathbf{a}id \mapsto \{\mathbf{loc}_2\}$ and $\mathbf{o}id \rightsquigarrow [a'', b'']$

   $\forall x \forall y . \mathbf{p}(\mathbf{a}id', x) \wedge \mathbf{p}(x, y) \Rightarrow \mathbf{p}(\mathbf{a}id, y)$

   $\mathbf{beta}(\mathbf{a}id, \mathbf{a}id')$

3. **Constraint**: $(\mathbf{a}id, \mathbf{o}id) \supseteq \mathbf{f}id'_r$

   if $\mathbf{f}id'_r \mapsto S$ and $\mathbf{f}id'_r \rightsquigarrow [a, b]$
   then $\mathbf{a}id \mapsto S$ and $\mathbf{o}id \rightsquigarrow [a, b]$

   $\forall x . \mathbf{p}(\mathbf{f}id'_r, x) \Rightarrow \mathbf{p}(\mathbf{a}id, x)$

   $\mathbf{alpha}(\mathbf{a}id, \mathbf{f}id'_r)$

4. **Constraint**: $(\mathbf{a}id, \mathbf{o}id) \supseteq \mathbf{f}id'_n$

   if $\mathbf{f}id'_n \mapsto S$ and $\mathbf{f}id'_n \rightsquigarrow [a, b]$
   then $\mathbf{a}id \mapsto S$ and $\mathbf{o}id \rightsquigarrow [a, b]$

   $\forall x . \mathbf{p}(\mathbf{f}id'_n, x) \Rightarrow \mathbf{p}(\mathbf{a}id, x)$

   $\mathbf{alpha}(\mathbf{a}id, \mathbf{f}id'_n x)$

5. **Constraint**: $(\mathbf{a}id, \mathbf{o}id) \supseteq \mathbf{retfp}(\mathbf{a}id', e)$ or $\mathbf{retfp}(\mathbf{loc}_1, e)$

   if (rhs is $\mathbf{a}id' \mapsto \{\mathbf{loc}_1, \dots\}$ or rhs is $\mathbf{loc}_1$) and $\mathbf{eval}(e) = [a, b]$ and
   $(\mathbf{loc}_1, [a', b']) \rightarrow (\mathbf{loc}_2, [a'', b''])$
   and $\mathbf{loc}_2$ is a function $\mathbf{f}id''$ and $[a, b] \bigcap [a', b'] \neq \emptyset$ and $\mathbf{f}id''_r \mapsto S$ and $fid''_r \rightsquigarrow [a''', b''']$

   then
    $\mathbf{a}id \mapsto S$ and $\mathbf{o}id \rightsquigarrow [a''', b''']$

   Game Rule. $\forall x \forall y \forall z . \mathbf{p}(\mathbf{a}id', x) \wedge \mathbf{p}(x, y) \wedge \mathbf{p}(y, z) \Rightarrow \mathbf{p}(\mathbf{a}id, z)$ *(if we identify the location of a function with the function itself and all its parameters and return value as well).*

else if (rhs is $\mathbf{a}id' \mapsto \{\mathbf{loc}_1, \dots\}$ or rhs is $\mathbf{loc}_1$) and $\mathbf{loc}_1$ is a function $\mathbf{f}id''$ and $\mathbf{f}id''_r \mapsto S$ and $\mathbf{f}id''_r \rightsquigarrow [a''', b''']$

then
$\quad\mathbf{a}id \mapsto S$ and $\mathbf{o}id \rightsquigarrow [a''', b''']$

Game Rule. $\forall x \forall y. \mathbf{p}(\mathbf{a}id', x) \wedge \mathbf{p}(x, y) \Rightarrow \mathbf{p}(\mathbf{a}id, y)$ *(if we identify the location of a function with the function itself and all its parameters and return value as well)*

The first case is a new sort of rule, and unique in that it involves three jumps from $\mathbf{a}id'$. The second case is just $\mathbf{beta}(\mathbf{a}id, \mathbf{a}id')$.

6. **Constraint**: $*(\mathbf{a}id, e)$ or $*(\mathbf{loc}_1, e) \leftarrow (\mathbf{a}id', e')$ or $(\mathbf{loc}_2, e')$

   if (rhs is $\mathbf{a}id \mapsto \{\mathbf{loc}_1, \dots\}$ or rhs is $\mathbf{loc}_1$) and $\mathbf{eval}(e) = [a, b]$ and (rhs is $\mathbf{a}id' \mapsto \{\mathbf{loc}_2, \dots\}$ or rhs is $\mathbf{loc}_2$) and $\mathbf{eval}(e') = [a', b']$ then
   $(\mathbf{loc}_1, [a, b]) \rightarrow (\mathbf{loc}_2, [a', b'])$

   $\forall x \forall y. \mathbf{p}(\mathbf{a}id, x) \wedge \mathbf{p}(\mathbf{a}id', y) \Rightarrow \mathbf{p}(x, y)$

   $\mathbf{gamma}(\mathbf{a}id, \mathbf{a}id')$

7. **Constraint**: $\mathbf{f}id'_n \leftarrow (\mathbf{a}id, e)$ or $(\mathbf{loc}, e)$

   if (rhs is $\mathbf{a}id \mapsto S$ or rhs is $S = \{\mathbf{loc}\}$) and $\mathbf{eval}(e) = [a, b]$ then $\mathbf{f}id'_n \mapsto S$ and $\mathbf{f}id'_n \rightsquigarrow [a, b]$

   $\forall x. \mathbf{p}(\mathbf{a}id, x) \Rightarrow \mathbf{p}(\mathbf{f}id'_n, x)$

   $\mathbf{alpha}(\mathbf{f}id'_n, \mathbf{a}id)$

8. **Constraint**: $\mathbf{f}id'_r \leftarrow (\mathbf{a}id, e)$ or $(\mathbf{loc}, e)$

   if (rhs is $\mathbf{a}id \mapsto S$ or rhs is $S = \{\mathbf{loc}\}$) and $\mathbf{eval}(e) = [a, b]$ then $\mathbf{f}id'_r \mapsto S$ and $\mathbf{f}id'_r \rightsquigarrow [a, b]$

   $\forall x. \mathbf{p}(\mathbf{a}id, x) \Rightarrow \mathbf{p}(\mathbf{f}id'_r, x)$

   $\mathbf{alpha}(\mathbf{f}id'_r, \mathbf{a}id)$

9. **Constraint**: $\mathbf{fp}_n(\mathbf{a}id, e)$ or $\mathbf{fp}_n(\mathbf{loc}_1, e) \leftarrow (\mathbf{a}id', e')$ or $(\mathbf{loc}, e')$

   if (lhs is $\mathbf{a}id \mapsto \{\mathbf{loc}_1, \dots\}$ or lhs is $\mathbf{loc}_1$) and $\mathbf{eval}(e) = [a, b]$

then

if $(\mathbf{loc}_1, [a', b']) \to (\mathbf{loc}2, [a'', b''])$
and $\mathbf{loc}_2$ is a function $\mathbf{f}id''$ and $[a, b] \bigcap [a', b'] \neq \emptyset$ and (rhs is $\mathbf{a}id' \mapsto S$ or rhs is $S = \{\mathbf{loc}\}$) and $\mathbf{eval}(e') = [a''', b''']$
then $\mathbf{f}id'''_n \mapsto S$ and $\mathbf{f}id''_n \rightsquigarrow [a'', b'']$

Game Rule. $\forall x \forall y \forall z . \mathbf{p}(\mathbf{a}id', z) \wedge \mathbf{p}(\mathbf{a}id, x) \wedge \mathbf{p}(x, y) \Rightarrow \mathbf{p}(y, z)$ *(as in rule 5, we assume that we identify the location of a function with the function itself and all its parameters and return value as well).*

else if $\mathbf{loc}_1$ is a function $\mathbf{f}id''$ and (rhs is $\mathbf{a}id' \mapsto S$ or rhs is $S = \{\mathbf{loc}\}$) and $\mathbf{eval}(e') = [a''', b''']$
then $\mathbf{f}id''_n \mapsto S$ and $\mathbf{f}id''_n \mapsto [a''', b''']$

Game Rule. $\forall x \forall z . \mathbf{p}(\mathbf{a}id', z) \wedge \mathbf{p}(\mathbf{a}id, x) \Rightarrow \mathbf{p}(x, z)$ *(as in rule 5, we assume that we identify the location of a function with the function itself and all its parameters and return value as well).*

The first case is a *reverse delta* like a delta but the added arc goes the opposite direction. Let's call it $\mathbf{epsilon}(\mathbf{a}id', \mathbf{a}id)$. The second is $\mathbf{gamma}(\mathbf{a}id, \mathbf{a}id')$.

10. **Constraint**: $\mathbf{memcpy}(\mathbf{a}id$ or $\mathbf{loc}_3, e, \mathbf{a}id'$ or $\mathbf{loc}_1, e', l)$

    if (rhs is $\mathbf{a}id' \mapsto \{\mathbf{loc}_1, \ldots\}$ or rhs is $\mathbf{loc}_1$) and $\mathbf{eval}(e') = [a, b]$ and
    $(\mathbf{loc}_1, [a', b']) \to (\mathbf{loc}_2, [a'', b''])$
    and $([a, b] +_l ([0, 0] \cup \mathbf{eval}(/))) \bigcap [a', b'] \neq \emptyset$
    and (rhs is $\mathbf{a}id \mapsto \{\mathbf{loc}_3, \ldots\}$ or rhs is $\mathbf{loc}_3$) and $\mathbf{eval}(e) = [a''', b''']$
    then $(\mathbf{loc}_3, [a', b'] -_l [a, b] +_l [a''', b''']) \to (\mathbf{loc}_2, [a'', b''])$

    $\forall x \forall y \forall z . \mathbf{p}(\mathbf{a}id', x) \wedge \mathbf{p}(x, y) \wedge \mathbf{p}(\mathbf{a}id, z) \Rightarrow \mathbf{p}(z, y)$

    $\mathbf{delta}(\mathbf{a}id, \mathbf{a}id')$

3.3.6 Production of Results

Consider a symbolic variable $\mathbf{a}id$ and its companion offset variable $\mathbf{o}id$. If looking it up in the dictionary brings up the following entry

```
<memory-element uid=id>
  <anchor context="..." function="..." name="..."/>
</memory-element>
```

then the values of $\mathbf{a}id$ and $\mathbf{o}id$ shall be reported in the results file as follows:

if $\mathbf{a}id \mapsto \{\mathbf{loc}_1, \ldots, \mathbf{loc}_n\}$ and $\mathbf{o}id \mapsto [a, b]$ and $\mathbf{size}(\mathbf{loc}_1) \cup \cdots \cup \mathbf{size}(\mathbf{loc}_n) = [a', b']$
then add the following entry to the XML results file, substituting for the attribute values:

```
<anchor-value uid=id offset-lb=a offset-ub=b size-lb=a’ size-ub=b’ />
```

If some locations in $\{\mathbf{loc}_1, \ldots, \mathbf{loc}_n\}$ do not have a size in the dictionary, then they have to be removed. If after removal of those locations the resulting set is empty then the size is $[0, 0]$.

If we have the following entries in the dictionary:

```
<memory-element uid="2">
  <malloc context="" function="allocate" location="1">
    <interval-value lower-bound="0" upper-bound="256"/>
  </malloc>
</memory-element>
<memory-element uid="6">
  <global-variable name="P" size="32"/>
</memory-element>
```

Then $\mathbf{size}(\mathbf{loc}_2) = [0, 256]$ and $\mathbf{size}(\mathbf{loc}_6) = [32, 32]$.

3.4 Design of the Game Model

We now present a design of a game model to accompany its formalization from Section 3.3. The pointer analysis the game players will enable is based on the flow-insensitive, context-insensitive pointer analysis proposed by Andersen [1], extended with offsets. The introduction of offsets allows more precise handling of structs containing pointers, which are prevalent in BIND, the target application to be analyzed.

The objective of the game is to have the players construct a points-to graph and points-to sets that satisfy all constraints extracted from the program. These points-to sets and points-to graph are then used to discharge memory access conditions. At the end of the game the result of the checks, and the points-to graph and points-to sets are sent back to CodeHawk.

In principle all actions performed by the players could be performed automatically. The hard part in an automatic computation, however, is to choose the order in which to process the constraints. The order in which the constraints are processed does not affect the final result, but it can greatly affect the number of operations, and thereby the time needed, to reach the final result. Thus, the "creative contribution" of the players would be to try to come up with an optimal order to finish the game fast or gain points.

3.4.1 Definitions

A game instance consists of the following components:

- $\mathcal{M} = \{m_1, m_2, \ldots\}$: a set of memory regions. A memory region can be a stack-allocated variable or array or a dynamically allocated memory region on the heap, or a global variable or array;
- $\mathcal{A} \subseteq \mathcal{M} \times (\mathcal{N} \cup \{\top\})$: a set of addresses, where each address (m, o) consists of a memory region $m \in \mathcal{M}$ and a non-negative offset $o = 0, 1, 2, 3, \ldots$. The value $\top$ (top) denotes an unknown offset. Given a set $A = \{(m_1, o_1), (m_2, o_2), \ldots\}$, we write $A + k$ to denote $\{(m_1, o_1 + k), (m_2, o_2 + k), \ldots\}$;
- $\mathcal{S} : \mathcal{M} \mapsto (\mathcal{N} \cup \{\top\})$, a size function that maps each memory region to its size in bytes;

- $\mathcal{V} = \{v_1, v_2, \ldots\}$: a set of variables;
- $\mathcal{F} = \{f_1, f_2, \ldots\}$: a set of functions;
- $\mathcal{L} : \; \mathcal{F} \mapsto 2^{\mathcal{V}}$, a mapping that identifies the local variables of a function; We require that each variable only belong to one function, that is, $\forall f_1, f_2 \in \mathcal{F} \; . \; f_1 \neq f_2 \rightarrow \mathcal{L}(f_1) \cap \mathcal{L}(f_2) = \emptyset$. Variables that do not belong to any function are global;
- $\mathcal{C} \subseteq \mathcal{F} \times \mathcal{F}$, the call graph;
- Con: a set of constraints, described below;
- $p : \; \mathcal{V} \mapsto 2^{\mathcal{A}}$, a points-to function that maps variables to a set of addresses
- $\mathcal{G} \subseteq \mathcal{A} \times \mathcal{A}$, the points-to graph, a directed graph where an edge $(m_1, o_1), (m_2, o_2)$ denotes the fact that the address in memory region m_1 at offset o_1 may point at memory region m_2 at offset o_2. Note that the data type of (m_1, o_1) must be a pointer. For a set of addresses $a \subseteq \mathcal{A}$ we write $\mathcal{G}(a)$ to denote the maximal set of addresses $b \subseteq \mathcal{A}$ such that $\forall x \in b \; . \; \exists y \in a \; . \; (y, x) \in \mathcal{G}$, that is, b is the target set of a.
- Chk: a set of verification conditions, described below.

3.4.2 Objective

At the start of the game CodeHawk provides elements $\mathcal{M}$, $\mathcal{S}$, $\mathcal{V}$, $\mathcal{F}$, $\mathcal{L}$, $\mathcal{C}$, and Con. These are static and do not change during the game[1]. The objective of the game is to construct the points-to function p and the points-to graph $\mathcal{G}$ such that they satisfy the constraints expressed in Con. The result is used to attempt to discharge the verification conditions in Chk. Finally, at the end of the game, the result of the checks and p and $\mathcal{G}$ is sent back to CodeHawk.

3.4.3 Representation

The game does not need names. In fact, to hide the identity of the program being analyzed, it is preferred not to reveal names. Therefore each of $\mathcal{M}$, $\mathcal{V}$, and $\mathcal{F}$ can be represented by a single number, namely their size. The other components can simply refer to index numbers on each kind to represent relationships. CodeHawk will retain a mapping between index numbers and actual program entities to interpret the results.

3.4.4 Example: Definitions

Continuing the running example we illustrate the definitions and operations described below.

```
struct S {
 int *p1;
 int *p2;
};
```

[1] In a future version of the game the call graph $\mathcal{C}$ may change during the course of the game, when we incorporate the analysis of function pointers.

```
5  int A[10];
6  int B[10];
7
8  void init1(struct S *u) {
9    u->p1 = &A[0];
10 }
11
12 void init2(struct S *u) {
13   volatile int random;
14   u->p2 = random ? &B[5] : &B[0];
15 }
16
17 main() {
18   struct S *pS;
19   pS = malloc(sizeof(struct S));
20   init1(pS);
21   init2(pS);
22   pS->p1[5] = pS->p2[5];
23 }
```

At the start of the game CodeHawk would provide the following data:

- $\mathcal{M} = \{m_1, m_2, m_3\}$, denoting the two global arrays `A` and `B` and the memory region allocated at line 19, respectively; (communicated simply as $|\mathcal{M}| = 3$);
- $\mathcal{S}$: $\mathcal{S}(m_1) = \mathcal{S}(m_2) = 40$, $\mathcal{S}(m_3) = 8$
- $\mathcal{V} = \{v_1, v_2, v_3\}$, denoting `pS`, `u` in `init1`, and `u` in init2 (that is, $|\mathcal{V}| = 3$);
- $\mathcal{F} = \{f_1, f_2, f_3\}$, denoting `main`, `init1`, `init2` (that is, $|\mathcal{F}| = 3$);
- $\mathcal{L}$: $\mathcal{L}(f_1) = \{v_1\}$, $\mathcal{L}(f_2) = \{v_2\}$, $\mathcal{L}(f_3) = \{v_3\}$
- $\mathcal{C} = \{(f_1, f_2), (f_1, f_3)\}$

3.4.5 Basic Assignments

The points-to-graph is the solution to a constraint system constructed based on pointer assignments in the program. Below we describe the different kinds of basic assignments with their associated constraints. In the description v and w are program variables.

1. `v = &w`
   constraint: $(m, 0) \in p(v)$, where m is the memory region that holds variable w;

2. `v = &w[i]`
   constraint: $(m, k) \in p(v)$, where m is the memory region that holds the array $w[]$, and $k = i\ s$, where s is the size in bytes of the element type of w;

3. `v = w`
   constraint: $p(w) \subseteq p(v)$

4. `v = *w`
   constraint: $\forall x \in p(w)\ .\ \mathcal{G}(x) \subseteq p(v)$

5. `*v = w`
   constraint: $\forall x \in p(v)\ .\ p(w) \subseteq \mathcal{G}(x)$

6. `*v = *w`
   constraint: $\forall x \in p(v)\ \forall y \in p(w)\ .\ \mathcal{G}(y) \subseteq \mathcal{G}(x)$

7. `v = w->f` [also written as `v = (*w).f`]
   constraint: $\forall x \in p(w)\ .\ \mathcal{G}(x + k) \subseteq p(v)$, where k is the offset of field `f`

8. `v->f = w` [also written as `(*v).f = w`]
   constraint: $\forall x \in p(v)\ .\ p(w) \subseteq \mathcal{G}(x + k)$

9. `v->f1 = w->f2` [also written as `(*v).f1 = (*w).f2`]
   constraint: $\forall x \in p(v)\ \forall y \in p(w)\ .\ \mathcal{G}(y + k_2) \subseteq \mathcal{G}(x + k_1)$, where k_1 and k_2 are the offsets of fields `f1` and `f2`, respectively.

Assignment types (1), (3), (4), and (5) are illustrated in Figures 1,2,3, and 4.

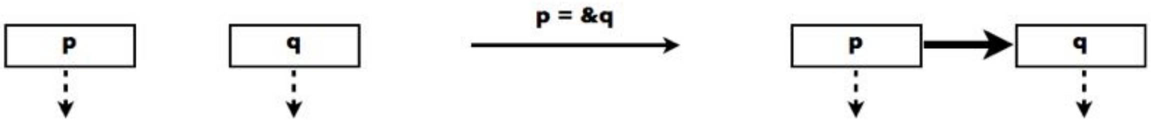


Figure 1: Assignment of address

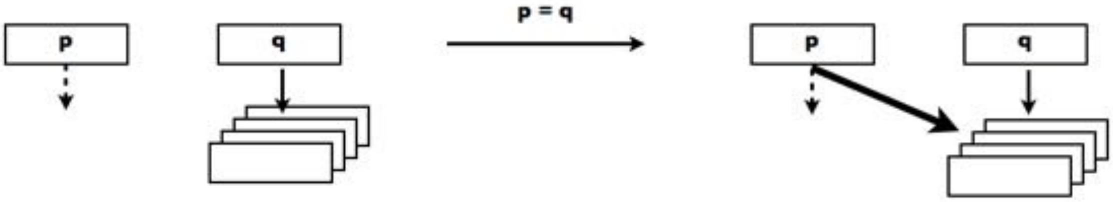


Figure 2: Assignment of pointer variable

Assignment types (4-6) are special cases of (7-9), with $k = 0$ so in the remainder we will only consider the general case.

An array access is treated in one of two ways: if the address of the array is taken, the memory region that holds the array is in $\mathcal{M}$ and the access is treated as a pointer dereference. If the address is not taken, the element accessed in the array can be treated simply as a variable (most often an aggregate variable that represents every element in the array).

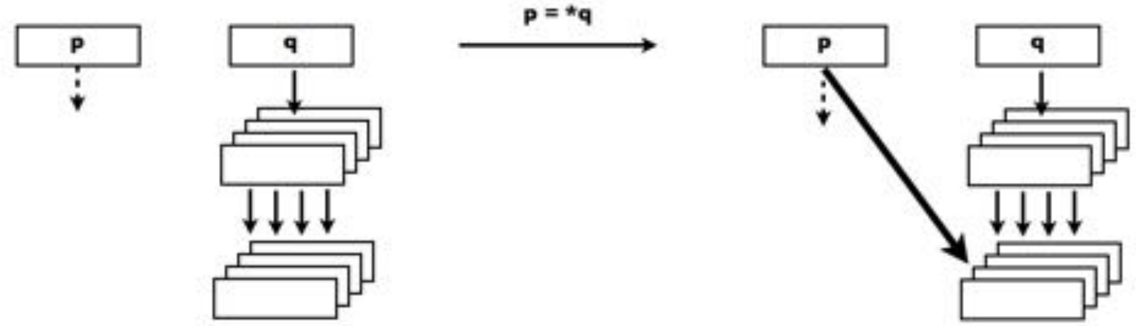


Figure 3: Assignment of derefenced pointer

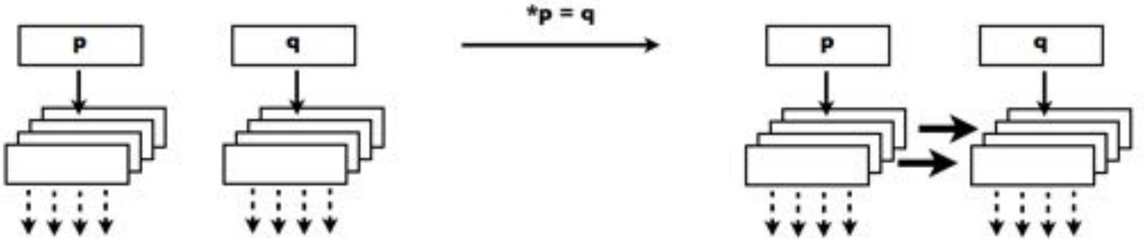


Figure 4: Assignment to dereferenced pointer

3.4.6 Representation

The different types of constraints can be communicated as follows:

1. $((m, 0), var(v))$
2. $((m, k), var(v))$
3. $(var(w), var(v))$
4. $(\text{deref}(var(w), 0), var(v))$
5. $(var(w), \text{deref}(var(v), 0))$
6. $(\text{deref}(var(w), 0), \text{deref}(var(v), 0))$
7. $(\text{deref}(var(w), k), var(v))$
8. $(var(w), \text{deref}(var(v), k))$
9. $(\text{deref}(var(w), k_2), \text{deref}(var(v), k_1))$

3.4.7 Example: Basic Assignments

The basic assignments in our example program are

```
 9 u->p1 = &A[0]
```

form: `v->f = &w[i]`
constraint: $\forall x \in p(\text{init1\_u}) \;.\; (m_1, 0) \in \mathcal{G}(x)$

```
14 u->p2 = random ? &B[5] : &B[0]
```

form: `v->f = &w[i]`
constraint: $\forall x \in p(\text{init2\_u}) \;.\{(m_2, 0), (m_2, 20)\} \subseteq \mathcal{G}(x + 4)$

Note that no constraint is included for the assignment at line 22, because this assignment is an integer assignment, not a pointer assignment.

3.4.8 Function Calls

Function calls give rise to two types of assignment. Given a function f with parameters of pointer type $u_1, u_2, \ldots, u_n$ (we ignore all other, non-pointer parameters for now), a function call $e = f(a_1, a_2, \ldots, a_n)$ produces two types of assignments:

Parameters $u_i = a_i$, for $i = 1, \ldots, n$, where u_i can be treated as a local variable of the callee and a_i can be any pointer expression (a variable, or a dereference). The assignments and their corresponding constraints, as given above, are added to the set of assignments of the caller, but they directly affect the constraint satisfaction state of the callee.

Return value If the function returns a pointer (address), each return statement `return r` in the callee gives rise to the assignment $e = r$ in the caller (calling function).

A special type of function call is a call to a function that allocates memory on the heap, such as `malloc` or `strdup`:

Dynamic memory allocation `v = allocatingFunction (size)`
constraint: $(m, 0) \in p(v)$, and $\mathcal{M}(m) = \mathit{size}$

3.4.9 Example: Function Calls

The function calls in our example are

```
 19   pS = malloc(sizeof(struct S));
```

constraint: $(m_3, 0) \in p(\text{main\_pS})$

```
 20   init1(pS);
```

assignment: $\text{init1\_u} = \text{main\_pS}$
constraint: $p(\text{main\_pS}) \subseteq p(\text{init1\_u})$

```
 21   init2(pS);
```

assignment: $\text{init2\_u} = \text{main\_pS}$
constraint: $p(\text{main\_pS}) \subseteq p(\text{init2\_u})$

3.4.10 Operational Description

The objective of the game is to construct a points-to-graph that satisfies all constraints of the program. This can be accomplished by having the players perform a fixed point computation in which edges are added to the graph and addresses are added to the points-to sets of variables according to the following rules, derived from the constraints above, until no more edges and addresses need to be added to satisfy the constraints, or, with these rules, until none of the actions has any effect any more.

1. `v = &w`
   action: $p(v) := p(v) \cup \{(m, 0)\}$

2. `v = &w[i]`
   action: $p(v) := p(v) \cup \{(m, k)\}$

3. `v = w`
   action: $p(v) := p(v) \cup p(w)$

4. `v = w->f` [also written as `v = (*w).f`]
   action: $p(v) := p(v) \cup \bigcup_{x \in p(w)} \mathcal{G}(x + k)$, where k is the offset of field `f`

5. `v->f = w` [also written as `(*v).f = w`]
   action: $\mathcal{G} := \mathcal{G} \cup \{(a_1, a_2) \mid a_1 \in p(v) + k, a_2 \in p(w)\}$

6. `v->f1 = w->f2` [also written as `(*v).f1 = (*w).f2`]
   action: $\mathcal{G} := \mathcal{G} \cup \{(a_1, a_2) \mid a_1 \in p(v) + k_1, a_2 \in \bigcup_{x \in p(w)} \mathcal{G}(x + k_2)\}$, where k_1 and k_2 are the offsets of fields `f1` and `f2`, respectively.

3.4.11 Organization of the Game

The game consists of rooms, one room for each function. On the "world map" the players can see all rooms. Two rooms are connected if there exists an edge from one to the other in the call graph. Each room has a light, which is on if the constraints in that room are not satisfied (that is, there is work to be performed in that room), and off otherwise (nothing to do). Players can go to any room to perform actions to satisfy the constraints. Their actions may have "remote consequences," for example, the addition of an edge to the points-to graph, as in rule (5) and (6) above, may cause the constraints in another room to change from satisfied to not satisfied, thus remotely turning on the light in that room (the remote room does not have to be connected to the room in which the action was performed). The effect of a function call can be viewed as dropping new addresses on the doorstep of the room representing the called function, thus potentially changing its constraint satisfaction status. Once the callee is processed the change in return value may change the constraint satisfaction status of the caller (and potentially of other callers).

Every action with potentially remote consequences requires, in principle, a re-computation of the constraints in every room that could be affected. It is, however, not imperative that this re-computation be performed immediately after the action is taken. Re-computation can be performed at certain time intervals. Re-computation must be performed, however, when all rooms are dark, to ascertain whether indeed all constraints are satisfied.

3.4.12 Example: Game Play

The game instance associated with our example program has three rooms, described below with their associated constraints.

room 1: init1
constraint: $\forall x \in p(\text{init1\_u})\ .\ (m_1, 0) \in \mathcal{G}(x)$
room 2: init2
constraint: $\forall x \in p(\text{init2\_u})\ .\{(m_2, 0), (m_2, 20)\} \subseteq \mathcal{G}(x + 4)$
room 3: main
constraint: $(m_3, 0) \in p(\text{main\_pS})$
constraint: $p(\text{main\_pS}) \subseteq p(\text{init1\_u})$
constraint: $p(\text{main\_pS}) \subseteq p(\text{init2\_u})$

Initially the points-to sets for all variables and the points-to graph are the empty set. Thus initially the constraints in room 1 and room 2 are satisfied, since $p(\text{init1\_u}) = p(\text{init2\_u}) = \emptyset$, and therefore the light is off. The constraints in room 3 are not satisfied. In particular, the first constraint is not satisfied, and thus the light is on. A player can satisfy the constraint by adding the requested address $(m_3, 0)$ to the points-to set of main_pS. This action causes the second and third constraint to be not satisfied anymore. The player can now drop the address in $p(\text{main\_ps})$ on the doorstep of room 1 and room 2, by adding the address to $p(\text{init1\_u})$ and $p(\text{init2\_u})$, causing the light to switch on in both rooms. At this point the light in room 3 will switch off, because all constraints are satisfied.

The state of the game at this point is:

$$\begin{aligned} p(\text{main\_pS}) &= \{(m_3, 0)\} \\ p(\text{init1\_u}) &= \{(m_3, 0)\} \\ p(\text{init1\_u}) &= \{(m_3, 0)\} \\ \mathcal{G} &= \emptyset \end{aligned}$$

The player can now go to room 1 (or room 2) and add an edge to $\mathcal{G}$ between $(m_3, 0)$ and $(m_1, 0)$, thus satisfying the constraint, causing the light to turn off. In room 2 the player needs to add two edges to $\mathcal{G}$: $((m_3, 4), (m_2, 0))$ and $((m_3, 4), (m_2, 20))$ to satisfy the constraint. These actions do not have other consequences, and this stage of the game is over, with state

$$\begin{aligned} p(\text{main\_pS}) &= \{(m_3, 0)\} \\ p(\text{init1\_u}) &= \{(m_3, 0)\} \\ p(\text{init1\_u}) &= \{(m_3, 0)\} \\ \mathcal{G} &= \{((m_3, 0), (m_1, 0)),\ ((m_3, 4), (m_2, 0)),\ ((m_3, 4), (m_2, 20))\} \end{aligned}$$

reflecting the actual relationships between memory region, shown in Figure 5.

3.4.13 Verification Conditions

To prove that a program is not at risk for buffer overflow the safety of every memory access must be proven. This is done by generating verification conditions for each memory access that express that the access is within the bounds of the allocated memory region being

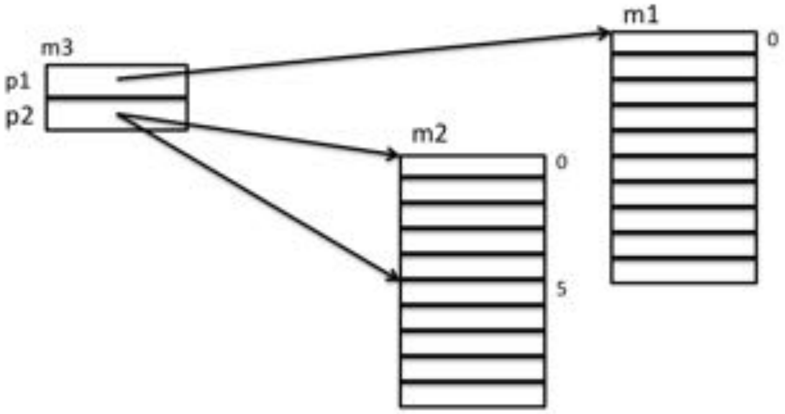


Figure 5: Memory layout

accessed, and proving that these verification conditions hold for every possible execution. This proof is performed by proving that the condition is implied by the invariants generated for that location. The points-to sets and points-to graph provide the basis for this implication by supplying the information which memory regions could be accessed.

A majority of the memory accesses (60-70% at the start of the CSFV Program, up to 80% at the end of the Program)) is already proven safe by CodeHawk; these conditions will not be included in the game instance. For example, the memory accesses in functions `init1` and `init2` would be proven automatically. The memory access in function `main`, however, cannot be proven without knowing the points-to set of `pS`, `(*pS).f1`, and `(*pS).f2`.

In general, for any memory access `(*v).k` with type size t bytes, the following check needs to be performed:

$$\forall x \in p(v)\ .\ \text{offset}(x) + k + t \leq \mathcal{S}(\text{base}(x))$$

where base(x) is the first element of the address and offset(x) is the second element.

A two-level memory access `(*(*v).k1).k2` with type size t generates the following check (in addition to the first-level check described before):

$$\forall x \in p(v)\ .\ \forall y \in \mathcal{G}(x + k_1)\ .\ \text{offset}(y) + k_2 + t \leq \mathcal{S}(\text{base}(y))$$

We can in principle continue to increase the level, but CodeHawk will usually generate additional variables to represent the intermediate sets, and so we will not encounter more than two levels.

A special case is the situation where a points-to set is found empty upon dereference. This is considered an error (a potential null-dereference error) and should be reported back to CodeHawk.

The IDs of the verification conditions that hold are also reported back to CodeHawk.

If the size of a memory region or the offset of an address is not known, the check cannot be performed.

3.4.14 Representation

A verification condition for memory access `(*v).k` is represented by the tuple (id, v, k, t) where id is an identification number, and t is an integer, representing the size of the type accessed, in bytes.

A verification condition for memory access `(*(*v).k1).k2` is represented by the tuple (id, v, k_1, k_2, t)

3.4.15 Example: Verification Conditions

The example program has six memory accesses:

9	$u \rightarrow p1$	$(*u).0$
12	$u \rightarrow p2$	$(*u).4$
22	$pS \rightarrow p1$	$(*pS).0$
22	$pS \rightarrow p2$	$(*pS).4$
22	$pS \rightarrow p1[5]$	$(*(*pS).0).20$
22	$pS \rightarrow p2[5]$	$(*(*pS).4).20$

The verification conditions associated with the first four memory accesses are discharged by CodeHawk automatically, and so are not made part of the game instance. Checks for the last two verification conditions are represented by the tuples $(1, 1, 0, 20, 4)$ (id, index of pS, first offset, second offset, size of type (int) accessed) and $(2, 1, 4, 20, 4)$.

With the results for the points-to sets and points-to graph computed, we can perform the checks: For check 1:

$$\begin{aligned} p(\text{main\_pS}) &= \{(m_3, 0)\} \\ \mathcal{G}((m_3, 0)) &= \{(m_1, 0)\} \end{aligned}$$

and thus we need to check

$$0 + 20 + 4 \leq 40$$

which holds, and thus the memory access is proven safe.

For check 2 we have

$$\begin{aligned} p(\text{main\_pS}) &= \{(m_3, 0)\} \\ \mathcal{G}((m_3, 0) + 4) &= \{(m_2, 0), (m_2, 20)\} \end{aligned}$$

giving rise to the conditions

$$\begin{aligned} 0 + 20 + 4 &\leq 40 \\ 20 + 20 + 4 &\leq 40 \end{aligned}$$

of which the first one holds, but the second does not. Thus this memory access cannot be proven safe. Note that it is not necessarily an error, because all points-to sets are over-approximations, so they may contain addresses that cannot occur at the location of access in any execution.

3.4.16 Example: Communication with CodeHawk

For the example program, the communication with CodeHawk would include the following data

From CodeHawk to Game

```
M = 3
V = 3
F = 3
S = [ (1,40), (2,40), (3,8) ]
L = [ (1,1), (2,2), (3,3) ]
C = [ (1,2), (1,3) ]
Con:
  fun 1: [ ( (3,0), var(1) ) ; ( var(1), var(2) ) ; ( var(1), var(3) ) ]
  fun 2: [ ( (1,0), deref (var(2),0) ) ]
  fun 3: [ ( (2,0), deref (var(3),4) ) ; ( (2,20), deref (var(3),4) ) ]
Chk:
  fun 1: [ (1,1,0,20,4) ; (2,1,4,20,4) ]
```

From Game to CodeHawk

```
1: [ (3,0) ]
2: [ (3,0) ]
3: [ (3,0) ]
G: [ ((3,0),(1,0)), ((3,4)(2,0)), ((3,4),(2,20)) ]
safe: [ 1 ]
error: [ ]
```

3.4.17 A Larger Example

Consider the following program

```
struct S {
  int *a ;
  struct S *next ;
} s1, s2, s3, s4 ;

int *p;

int initialize (struct S *v, int *a) {
  v->a = a;
}

int redirect (struct S *v1, struct S *v2) {
  v1->next = v2->next;
}

int reassign (struct S *v) {
  p = v->next ;
}


int main (int argc, char ** argv) {

  int p1,p2,p3 ;

  initialize(&s1,&p1) ;
  s1.next = &s2 ;

  initialize(&s2,&p2) ;
  s2.next = &s3 ;

  initialize(&s3,&p3) ;
  s3.next = &s1 ;

  redirect (&s1, &s2) ;
  redirect (&s2, &s3) ;

  reassign (&s1) ;
  reassign (&s2) ;

}
```

We will process the constraints generated by this program in order. The first set of constraints, generated by the main function, consists entirely of constraints generated by statements of the type `p = &q`, and simply require that the given variable points to the given address. For example the first constraint states that pointer variable 190 should point to the address of variable 206, which will result in an edge from 190 to 206. The number after the colon indicates the number of edges added by that constraint (the score for taking the action).

```
(address(206) in var(190)): 1
(address(205) in var(190)): 1
(address(204) in var(190)): 1
(address(183) in var(195)): 1
(address(183) in var(189)): 1
(address(183) in var(182)): 1
(address(182) in var(198)): 1
(address(182) in var(195)): 1
(address(182) in var(194)): 1
(address(182) in var(189)): 1
(address(182) in var(181)): 1
(address(181) in var(198)): 1
(address(181) in var(194)): 1
(address(181) in var(189)): 1
(address(181) in var(183)): 1
```

The result of processing these constraints is given in Figure 6.

The next constraint is

```
(deref(var(198)) in var(185)): 2
```

which is generated by a statement of the type `p = *q`, in function `reassign`. It states that all variables that are pointed to by variables pointed to by variable 198 (that is all nodes

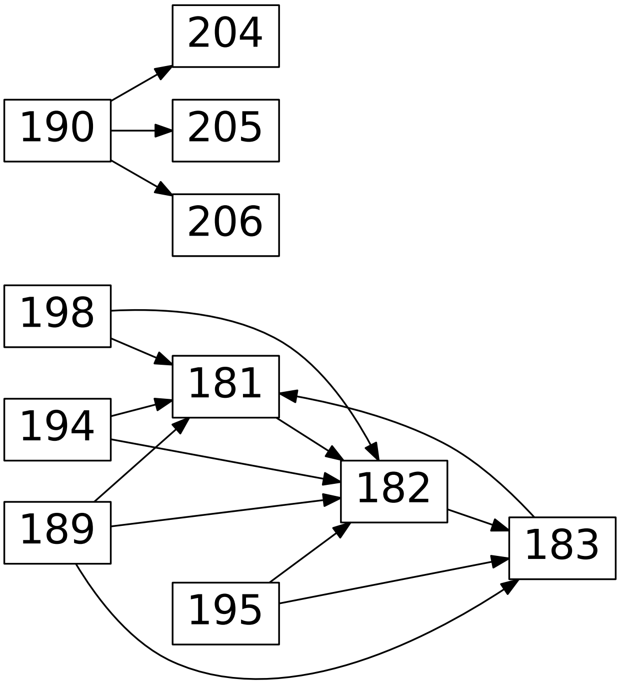


Figure 6: Points-to graph after the first set of constraints has been processed

that are two hops away from 198) should be included in the points-to set of variable 185. Variable 198 points to variables 182 and 181; variable 182 points to 183, variable 181 points to 182, and thus we add the edges (185,183) and (185,182), as shown in Figure 7.

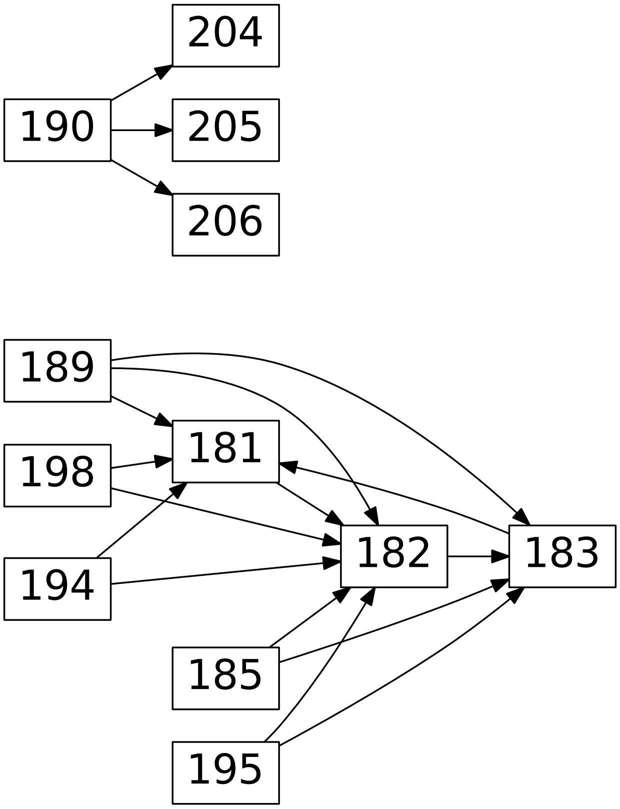


Figure 7: Points-to graph after `(deref(var(198)) in var(185))` is processed

The next constraint is

```
(deref(var(195)) in deref(var(194))): 3
```

which is generated by a statement of the type `*p = *q` in function `redirect`. It states that all nodes that are two hops away from variable 195 should be included in the variables pointed to by variable 194. The variables that are two hops away from 195 are 183 and 181, the variables pointed to by variable 194 are 181 and 182, and so we create the edges (181,183), (181,181), (182,183), and (182,181). Although we have four edges, we only get three points, because the edge (182,183) already existed. The result is shown in Figure 8.

The next constraint is

```
(var(190) in deref(var(189)))
```

which is generated by a statement of the type `*p = q`, in function `initialize`. It states that all variables pointed to by variable 190 should be included in the points-to sets of all variables

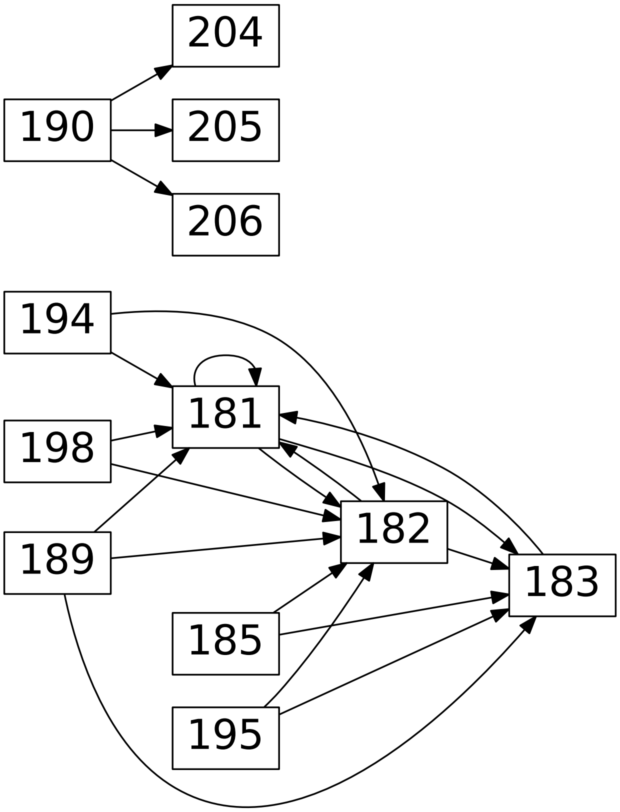


Figure 8: Points-to graph after `(deref(var(195)) in deref(var(194))` is processed

pointed to by variable 190. Three variables are pointed to by variable 190: $\{204, 205, 206\}$, and three variables are pointed to by variable 189: $\{181, 182, 183\}$, and thus we add an edge from each of $\{181, 182, 183\}$ to each of $\{204, 205, 206\}$, resulting in 9 new edges. The result is shown in Figure 9.

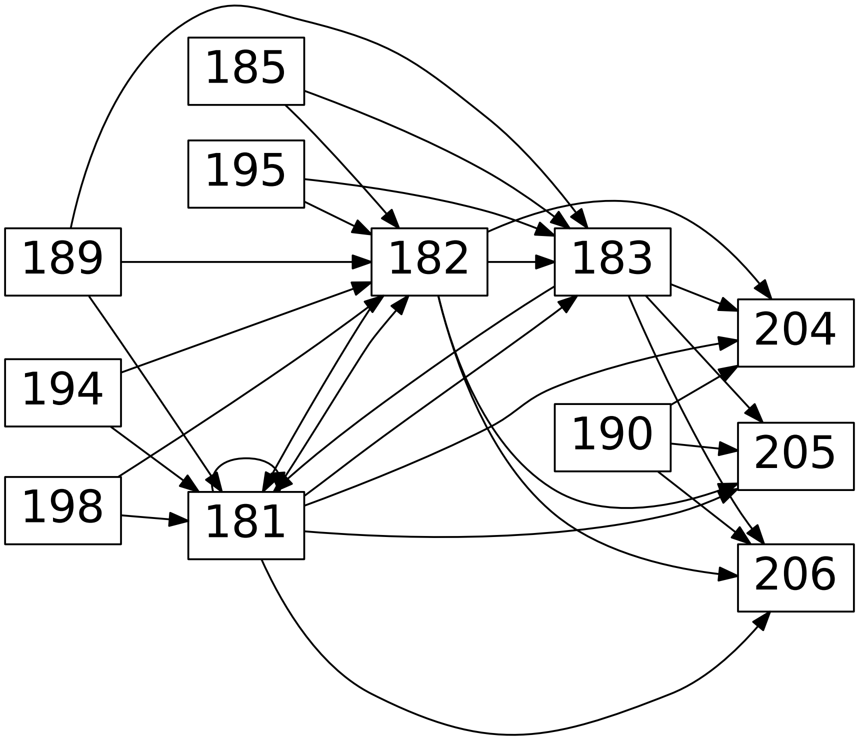


Figure 9: Points-to graph after `(var(190) in deref(var(189)))` is processed

Finally we return to the constraint

```
(deref(var(198)) in var(185)): 4
```

that we processed before, but because of other edges that were added more work needs to be done on this constraint. Finding the new edges is left as an exercise for the reader. After processing this constraint the system is stable and we have reached the final result for the points-to graph, as shown in Figure 10.

3.5 Implementation of the Game Model

We implemented the game model in the C# language using classes to represent intervals, nodes, arcs, constraints, and game instances. The game model was bundled with the game code to run in the player's browser, but we could also run it standalone with an auto-solver. Inheritance was useful here since there are several different subtypes of the node type. The

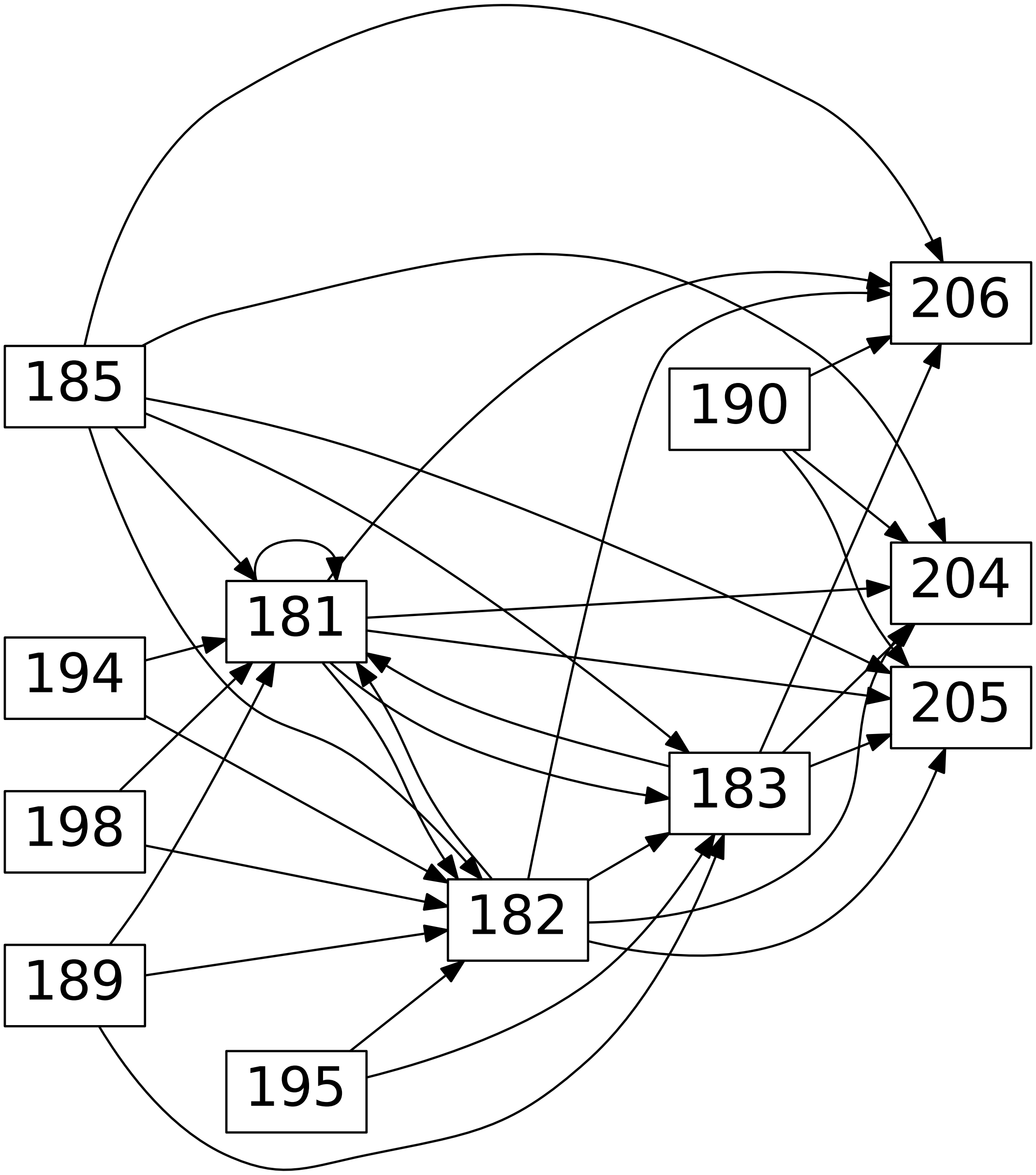


Figure 10: Final result for the points-to graph

nontrivial aspects of the game model implementation were (1) devising an exhaustive set of test cases, covering every salient case for each game rule and simple enough to analyze by hand, and (2) creating a strategy for ordering the game instances generated by a program, to achieve faster solutions by either an auto-solver or a human player or players.

3.5.1 Test Suite

The test suite may be of general interest since any modification or extension of this rule set would also have to pass most or all of these test cases. The test cases are listed below.

Rule 1

Example 1 Constraint: $(a_1, o_1) \supseteq (a_2, o_2)$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$.
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$.

Example 2 Constraint: $(a_1, o_1) \supseteq (a_2, o_3)$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 \rightarrow loc_1$;
$o_3 \rightarrow [1, 3]$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 \rightarrow loc_1$;
$o_3 \rightarrow [1, 3]$.

Example 3 Constraint: $(a_1, o_1) \supseteq (a_2, o_3)$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1, 3]$;
$a_3 \rightarrow \emptyset$;
$o_3 \rightarrow \emptyset$;
.
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;

$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1, 3]$;
$a_3 \rightarrow \emptyset$;
$o_3 \rightarrow \emptyset$;
.

Example 4 Constraint: $(a_1, o_1) \supseteq (a_2, [2, 4])$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1, 3]$;
.
Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [2, 4]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1, 3]$;
.

Example 5 Constraint: $(a_1, o_1) \supseteq (loc_2, o_2)$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
.
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;

Example 6 Constraint: $(a_1, o_1) \supseteq (loc_2, [2, 4])$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
.
Game state after triggering the constraint: $a_1 \rightarrow loc_2$;
$o_1 \rightarrow [2, 4]$;

Example 7 Constraint: $(a_1, o_1) \supseteq (a_2, o_3)$.
Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 2]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [1, 3]$;
$a_3 \rightarrow loc_3$;
$o_3 \rightarrow [2, 4]$;
.

Game state after triggering the constraint: $a_1 \to loc_1, loc_2$;
$o_1 \to [0,3]$;
$a_2 \to loc_2$;
$o_2 \to [1,3]$;
$a_3 \to loc_3$;
$o_3 \to [2,4]$;
.

Example 8 Constraint: $(a_1, o_1) \supseteq (loc_2, o_3)$.
Game state before triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [0,5]$;
$a_3 \to loc_3$;
$o_3 \to [2,4]$;
Game state after triggering the constraint: $a_1 \to loc_1, loc_2$;
$o_1 \to [0,5]$;
$a_3 \to loc_2$;
$o_3 \to [2,4]$;

Example 9 Constraint: $(a_1, o_1) \supseteq (a_2, [1,3])$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Rule 2

Example 1 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_2)$.
Functions in the dictionary: $\emptyset$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 2 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.

Functions in the dictionary: loc_1.

Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;

Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;

Example 3 Constraint: $(a_1, o_1) \supseteq deref(a_2, [1, 4])$.

Functions in the dictionary: $\emptyset$.

Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
$(loc_1, [2, 3]) \to (loc_2, [5, 6])$.

Game state after triggering the constraint: $a_1 \to loc_2$;
$o_1 \to [5, 6]$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
$(loc_1, [2, 3]) \to (loc_2, [5, 6])$.

Example 4 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.

Functions in the dictionary: $\emptyset$.

Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
$(loc_1, [2, 3]) \to (loc_2, [5, 6])$.

Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;

$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
$(loc_1, [2, 3]) \to (loc_2, [5, 6])$.

Example 5 Constraint: $(a_1, o_1) \supseteq deref(a_2, [1, 4])$.
Functions in the dictionary: $\emptyset$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;

Example 6 Constraint: $(a_1, o_1) \supseteq \mathit{deref}(a_2, [1, 4])$.
Functions in the dictionary: $\emptyset$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
$(loc_1, [7, 8]) \to (loc_2, [5, 6])$.
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
$(loc_1, [7, 8]) \to (loc_2, [5, 6])$.

Example 7 Constraint: $(a_1, o_1) \supseteq \mathit{deref}(a_2, o_3)$.
Functions in the dictionary: $\emptyset$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;

Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to \emptyset$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;

Example 8 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.
Functions in the dictionary: $\emptyset$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
$a_3 \to loc_1$;
$o_3 \to [1, 3]$;
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
$a_3 \to loc_1$;
$o_3 \to [1, 3]$;

Example 9 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.
Functions in the dictionary: loc_1.
Game state before triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [2, 4]$;
$a_2 \to loc_1$;
$o_2 \to [1, 2]$;
$a_3 \to loc_2$;
$o_3 \to [1, 3]$;
Game state after triggering the constraint: $a_1 \to loc_3, loc_1$;
$o_1 \to [-\infty, +\infty]$;
$a_2 \to loc_1$;
$o_2 \to [1, 2]$;
$a_3 \to loc_2$;
$o_3 \to [1, 3]$;

Example 10 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.
Functions in the dictionary: loc_1.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1, loc_4$;
$o_2 \to \emptyset$;

$a_3 \to loc_3$;
$o_3 \to [1, 4]$;
$(loc_1, [2, 3]) \to (loc_2, [5, 6])$.
Game state after triggering the constraint: $a_1 \to loc_1, loc_4$;
$o_1 \to [-\infty, +\infty]$;
$a_2 \to loc_1, loc_4$;
$o_2 \to \emptyset$;
$a_3 \to loc_3$;
$o_3 \to [1, 4]$;
$(loc_1, [2, 3]) \to (loc_2, [5, 6])$.

Example 11 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.
Functions in the dictionary: loc_1.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1, loc_4$;
$o_2 \to \emptyset$;
$a_3 \to loc_3$;
$o_3 \to [1, 4]$;
$(loc_1, [7, 8]) \to (loc_2, [5, 6])$.
Game state after triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [-\infty, +\infty]$;
$a_2 \to loc_1, loc_4$;
$o_2 \to \emptyset$;
$a_3 \to loc_3$;
$o_3 \to [1, 4]$;
$(loc_1, [7, 8]) \to (loc_2, [5, 6])$.

Example 12 Constraint: $(a_1, o_1) \supseteq deref(a_2, o_3)$.
Functions in the dictionary: loc_1, loc_4.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1, loc_4$;
$o_2 \to \emptyset$;
$a_3 \to loc_3$;
$o_3 \to [1, 4]$;
$(loc_1, [7, 8]) \to (loc_2, [5, 6])$.
Game state after triggering the constraint: $a_1 \to loc_1, loc_4$;
$o_1 \to [-\infty, +\infty]$;
$a_2 \to loc_1, loc_4$;
$o_2 \to \emptyset$;
$a_3 \to loc_3$;
$o_3 \to [1, 4]$;
$(loc_1, [7, 8]) \to (loc_2, [5, 6])$.

Rule 3

Example 1 Constraint: $(a_1, o_1) \supseteq f1'_r$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$f1'r \to (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$f1'_r \to (\emptyset, \emptyset)$;

Example 2 Constraint: $(a_1, o_1) \supseteq f1'_r$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$f1'_r \to (loc_1, [1, 3])$;
Game state after triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [1, 3]$;
$f1'_r \to (loc_1, [1, 3])$;

Example 3 Constraint: $(a_1, o_1) \supseteq f1'_r$.
Game state before triggering the constraint: $a_1 \to loc_2$;
$o_1 \to [2, 4]$;
$f1'_r \to (loc_1, [1, 3])$;
Game state after triggering the constraint: $a_1 \to loc_1, loc_2$;
$o_1 \to [1, 4]$;
$f1'_r \to (loc_1, [1, 3])$;

Rule 4

Example 1 Constraint: $(a_1, o_1) \supseteq f1'_n$.
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$f1'_n \to (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$f1'_n \to (\emptyset, \emptyset)$;

Example 2 Constraint: $(a_1, o_1) \supseteq f1'_n$. Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$f1'_n \to (loc_1, [1, 3])$;
Game state after triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [1, 3]$;

$f1'_n \to (loc_1, [1, 3])$;

Example 3 Constraint: $(a_1, o_1) \supseteq f1'_n$. Game state before triggering the constraint: $a_1 \to loc_2$;
$o_1 \to [2, 4]$;
$f1'_n \to (loc_1, [1, 3])$;
Game state after triggering the constraint: $a_1 \to loc_1, loc_2$;
$o_1 \to [1, 4]$;
$f1'_n \to (loc_1, [1, 3])$;

Rule 5

Example 1 Constraint: $(a_1, o_1) \supseteq retfp(a_2, o_2)$.
Functions in the dictionary: $\emptyset$
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 2 Constraint: $(a_1, o_1) \supseteq retfp(a_2, o_2)$
Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_r \to (loc_3, [7, 8])$
Game state after triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [7, 8]$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_r \to (loc_3, [7, 8])$

Example 3 Constraint: $(a_1, o_1) \supseteq retfp(a_2, o_2)$
Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;

$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_r \to (loc_3, [7, 8])$
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_r \to (loc_3, [7, 8])$

Example 4 Constraint: $(a_1, o_1) \supseteq \mathit{retfp}(a_2, o_2)$
Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_r \to (\emptyset, \emptyset)$
Game state after triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_r \to (\emptyset, \emptyset)$

Example 5 Constraint: $(a_1, o_1) \supseteq \mathit{retfp}(a_2, o_2)$
Functions in the dictionary: loc_1 is a function $f1'$
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f1'_r \to (loc_3, [7, 8])$
Game state after triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [7, 8]$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f1'_r \to (loc_3, [7, 8])$

Example 6 Constraint: $(a_1, o_1) \supseteq \mathit{retfp}(a_2, o_2)$
Functions in the dictionary: loc_1 is a function $f1'$
Game state before triggering the constraint: $a_1 \to \emptyset$;
$o_1 \to \emptyset$;
$a_2 \to loc_1$;
$o_2 \to [1, 3]$;

$loc_1, [2,4] \rightarrow loc_2, [5,6];$
$f1'_r \rightarrow (\emptyset, \emptyset)$
Game state after triggering the constraint: $a_1 \rightarrow \emptyset;$
$o_1 \rightarrow \emptyset;$
$a_2 \rightarrow loc_1;$
$o_2 \rightarrow [1,3];$
$loc_1, [2,4] \rightarrow loc_2, [5,6];$
$f1'_r \rightarrow (\emptyset, \emptyset)$

Example 7 Constraint: $(a_1, o_1) \supseteq \mathit{retfp}(a_2, o_2)$
Functions in the dictionary: $\emptyset$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset;$
$o_1 \rightarrow \emptyset;$
$a_2 \rightarrow loc_1;$
$o_2 \rightarrow [1,3];$
$loc_1, [2,4] \rightarrow loc_2, [5,6];$
$f1'_r \rightarrow (loc_3, [7,8])$
Game state after triggering the constraint: $a_1 \rightarrow \emptyset;$
$o_1 \rightarrow \emptyset;$
$a_2 \rightarrow loc_1;$
$o_2 \rightarrow [1,3];$
$loc_1, [2,4] \rightarrow loc_2, [5,6];$
$f1'_r \rightarrow (loc_3, [7,8])$

Example 8 Constraint: $(a_1, o_1) \supseteq \mathit{retfp}(a_2, o_2)$
Functions in the dictionary: $loc_1 isafunctionf1?andloc_2 isafunctionf2'$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset;$
$o_1 \rightarrow \emptyset;$
$a_2 \rightarrow loc_1;$
$o_2 \rightarrow [1,3];$
$loc_1, [2,4] \rightarrow loc_2, [5,6];$
$f2'_r \rightarrow (loc_3, [7,8])$
Game state after triggering the constraint: $a_1 \rightarrow loc_3;$
$o_1 \rightarrow [7,8];$
$a_2 \rightarrow loc_1;$
$o_2 \rightarrow [1,3];$
$loc_1, [2,4] \rightarrow loc_2, [5,6];$
$f2'_r \rightarrow (loc_3, [7,8])$

Example 9 Constraint: $(a_1, o_1) \supseteq \mathit{retfp}(a_2, o_2)$
Functions in the dictionary: $loc_1 isafunctionf1'$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset;$
$o_1 \rightarrow \emptyset;$
$a_2 \rightarrow loc_1;$
$o_2 \rightarrow [1,3];$
$loc_1, [2,4] \rightarrow loc_2, [5,6];$

$f1'_r \rightarrow (loc_4, [5,6])?f2'_r \rightarrow (loc_3, [7,8])$
Game state after triggering the constraint: $a_1 \rightarrow loc_4$;
$o_1 \rightarrow [7,8]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1,3]$;
$loc_1, [2,4] \rightarrow loc_2, [5,6]$;
$f1'_r \rightarrow (loc_4, [5,6])?f2'_r \rightarrow (loc_3, [7,8])$

Rule 6

Example 1 Constraint: $*(a_1, o_1) \supseteq (a_2, o_2)$.
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;

Example 2 Constraint: $*(a_1, o_1) \supseteq (a_2, o_3)$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 \rightarrow loc_1$;
$o_3 \rightarrow [1,3]$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 \rightarrow loc_1$;
$o_3 \rightarrow [1,3]$;

Example 3 Constraint: $*(a_1, o_1) \supseteq (a_2, o_3)$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1,3]$;
$a_3 \rightarrow \emptyset$;
$o_3 \rightarrow \emptyset$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;

$o_2 \rightarrow [1,3]$;
$a_3 \rightarrow \emptyset$;
$o_3 \rightarrow \emptyset$;

Example 4 Constraint: $*(a_1, o_1) \supseteq (a_2, [2,4])$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1,3]$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1,3]$;

Example 5 Constraint: $*(a_1, o_1) \supseteq (loc_2, o_2)$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;

Example 6 Constraint: $*(a_1, o_1) \supseteq (loc_2, [2,4])$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;

Example 7 Constraint: $*(loc_1, [1,3]) \supseteq (loc_2, [2,4])$
Game state before triggering the constraint: $\emptyset$
Game state after triggering the constraint: $(loc_1, [1,3]) \rightarrow (loc_2, [2,4])$;

Example 8 Constraint: $*(a_1, o_1) \supseteq (loc_2, [2,4])$
Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1,3]$;
Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1,3]$;
$(loc_1, [1,3]) \rightarrow (loc_2, [2,4])$;

Example 9 Constraint: $*(a_1, o_1) \supseteq (a_2, o_2)$.

Game state before triggering the constraint: $a_1 \rightarrow loc_1$;

$o_1 \rightarrow [1, 3]$;

$a_2 \rightarrow loc_2$;

$o_2 \rightarrow [2, 4]$;

Game state after triggering the constraint: $a_1 \rightarrow loc_1$;

$o_1 \rightarrow [1, 3]$;

$a_2 \rightarrow loc_2$;

$o_2 \rightarrow [2, 4]$;

$(loc_1, [1, 3]) \rightarrow (loc_2, [2, 4])$;

Example 10 Constraint: $*(loc_1, [1, 3]) \supseteq (a_2, o_2)$

Game state before triggering the constraint: $a_2 \rightarrow loc_2$;

$o_2 \rightarrow [2, 4]$;

Game state after triggering the constraint: $a_2 \rightarrow loc_2$;

$o_2 \rightarrow [2, 4]$;

$(loc_1, [1, 3]) \rightarrow (loc_2, [2, 4])$;

Example 11 Constraint: $*(a_1, o_1) \supseteq (a_2, o_2)$

Game state before triggering the constraint: $a_1 \rightarrow loc_1, loc_3$;

$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow loc_2, loc_4$;
$o_2 \rightarrow [2, 4]$;

Game state after triggering the constraint: $a_1 \rightarrow loc_1, loc_3$;

$o_1 \rightarrow [1, 3]$;

$a_2 \rightarrow loc_2, loc_4$;

$o_2 \to [2,4]$;

$(loc_1, [1,3]) \to (loc_2, [2,4])$;

$(loc_3, [1,3]) \to (loc_2, [2,4])$;

$(loc_1, [1,3]) \to (loc_4, [2,4])$;

$(loc_3, [1,3]) \to (loc_4, [2,4])$;

Rule 7

Example 1 Constraint: $f1'_n \supseteq (a_2, o_2)$.
Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
Game state after triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 2 Constraint: $f1'_n \supseteq (a_2, o_3)$.
Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
$a_3 \to loc_1$;
$o_3 \to [1,3]$;
Game state after triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
$a_3 \to loc_1$;
$o_3 \to [1,3]$;

Example 3 Constraint: $f1'_n \supseteq (a_2, o_3)$.
Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to loc_1$;
$o_2 \to [1,3]$;
$a_3 \to \emptyset$;
$o_3 \to \emptyset$;
Game state after triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to loc_1$;
$o_2 \to [1,3]$;
$a_3 \to \emptyset$;

$o_3 \to \emptyset$;

Example 4 Constraint: $f1'_n \supseteq (a_2, [2,4])$

Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to loc_1$;
$o_2 \to [1,3]$;
Game state after triggering the constraint: $f1'_n \to (loc_1, [2,4])$;
$a_2 \to loc_1$;
$o_2 \to [1,3]$;

Example 5 Constraint: $f1'_n \supseteq (loc_2, o_2)$

Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
Game state after triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 6 Constraint: $f1'_n \supseteq (loc_2, [2,4])$

Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
Game state after triggering the constraint: $f1'_n \to (loc_2, [2,4])$;

Example 7 Constraint: $f1'_n \supseteq (a_2, [2,4])$

Game state before triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Game state after triggering the constraint: $f1'_n \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 8 Constraint: $f1'_n \supseteq (a_2, o_3)$

Game state before triggering the constraint: $f1'_n \to (loc_4, [-\infty, +\infty])$;
$a_2 \to loc_1, loc_2$;
$o_2 \to [1,3]$;
$a_3 \to loc_3$;
$o_3 \to [0,0]$;
Game state after triggering the constraint: $f1'_n \to (loc_1, loc_2, loc_4, [-\infty, +\infty])$;
$a_2 \to loc_1, loc_2$;
$o_2 \to [1,3]$;
$a_3 \to loc_3$;

$o_3 \rightarrow [0,0]$;

Example 9 Constraint: $f1'_n \supseteq (loc_2, o_2)$

Game state before triggering the constraint: $f1'_n \rightarrow (loc_1, loc_2, [0,2])$;
$a_2 \rightarrow [loc_2]$;
$o_2 \rightarrow [1,3]$;
Game state after triggering the constraint: $f1'_n \rightarrow (loc_1, loc_2, [0,3])$;
$a_2 \rightarrow [loc_2]$;
$o_2 \rightarrow [-2,3]$;

Rule 8

Example 1 Constraint: $f1'_r \supseteq (a_2, o_2)$

Game state before triggering the constraint: $f1'_r \rightarrow (\emptyset, \emptyset)$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
Game state after triggering the constraint: $f1'_r \rightarrow (\emptyset, \emptyset)$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;

Example 2 Constraint: $f1'_r \supseteq (a_2, o_3)$

Game state before triggering the constraint: $f1'_r \rightarrow (\emptyset, \emptyset)$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 \rightarrow loc_1$;
$o_3 \rightarrow [1,3]$;
Game state after triggering the constraint: $f1'_r \rightarrow (\emptyset, \emptyset)$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 \rightarrow loc_1$;
$o_3 \rightarrow [1,3]$;

Example 3 Constraint: $f1'_r \supseteq (a_2, o_3)$

Game state before triggering the constraint: $f1'_r \rightarrow (\emptyset, \emptyset)$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1,3]$;
$a_3 \rightarrow \emptyset$;
$o_3 \rightarrow \emptyset$;
Game state after triggering the constraint: $f1'_r \rightarrow (\emptyset, \emptyset)$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [1,3]$;
$a_3 \rightarrow \emptyset$;

$o_3 \to \emptyset$;

Example 4 Constraint: $f1'_r \supseteq (a_2, [2,4])$

Game state before triggering the constraint: $f1'_r \to (\emptyset, \emptyset)$;
$a_2 \to loc_1$;
$o_2 \to [1,3]$;
Game state after triggering the constraint: $f1'_r \to (loc_1, [2,4])$;
$a_2 \to loc_1$;
$o_2 \to [1,3]$;

Example 5 Constraint: $f1'_r \supseteq (loc_2, o_2)$

Game state before triggering the constraint: $f1'_r \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;
Game state after triggering the constraint: $f1'_r \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 6 Constraint: $f1'_r \supseteq (loc_2, [2,4])$

Game state before triggering the constraint: $f1'_r \to (\emptyset, \emptyset)$;
Game state after triggering the constraint: $f1'_r \to (loc_2, [2,4])$;

Example 7 Constraint: $f1'_r \supseteq (a_2, [2,4])$

Game state before triggering the constraint: $f1'_r \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Game state after triggering the constraint: $f1'_r \to (\emptyset, \emptyset)$;
$a_2 \to \emptyset$;
$o_2 \to \emptyset$;

Example 8 Constraint: $f1'_r \supseteq (a_2, o_3)$

Game state before triggering the constraint: $f1'_r \to (loc_4, [-\infty, +\infty])$;
$a_2 \to loc_1, loc_2$;
$o_2 \to [1,3]$;
$a_3 \to loc_3$;
$o_3 \to [0,0]$;
Game state after triggering the constraint: $f1'_r \to (loc_1, loc_2, loc_4, [-\infty, +\infty])$;
$a_2 \to loc_1, loc_2$;
$o_2 \to [1,3]$;
$a_3 \to loc_3$;

$o_3 \rightarrow [0,0]$;

Example 9 Constraint: $f1'_r \supseteq (loc_2, o_2)$

Game state before triggering the constraint: $f1'_r \rightarrow (loc_1, loc_2, [0,2])$;
$a_2 \rightarrow [loc_2]$;
$o_2 \rightarrow [1,3]$;
Game state after triggering the constraint: $f1'_r \rightarrow (loc_1, loc_2, [0,3])$;
$a_2 \rightarrow [loc_2]$;
$o_2 \rightarrow [-2,3]$;

Rule 9

Example 1 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$

Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0,1]$;
$loc_1, [2,4] \rightarrow loc_2, [5,6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \rightarrow \emptyset$;
$o_1 \rightarrow \emptyset$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0,1]$;
$loc_1, [2,4] \rightarrow loc_2, [5,6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;

Example 2 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$

Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1,3]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [0,1]$;
$loc_1, [2,4] \rightarrow loc_2, [5,6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1,3]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [0,1]$;
$loc_1, [2,4] \rightarrow loc_2, [5,6]$;
$f2'_1 \rightarrow (loc_2, [0,1])$;

Example 3 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$

Functions in the dictionary: loc_2 is a function $f2'$

Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$loc_1, [2, 4] \rightarrow loc_2, [5, 6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;

Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$loc_1, [2, 4] \rightarrow loc_2, [5, 6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;

Example 4 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_3)$

Functions in the dictionary: loc_2 is a function $f2'$

Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [0, 2]$;
$a_3 > \emptyset$;
$o_3 \rightarrow \emptyset$;
$loc_1, [2, 4] \rightarrow loc_2, [5, 6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;

Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [0, 2]$;
$a_3 > \emptyset$;
$o_3 \rightarrow \emptyset$;
$loc_1, [2, 4] \rightarrow loc_2, [5, 6]$;
$f2'_1 \rightarrow (\emptyset, \emptyset)$;

Example 5 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_3)$

Functions in the dictionary: loc_2 is a function $f2'$

Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow \emptyset$;
$o_2 \rightarrow \emptyset$;
$a_3 > loc_2$;
$o_3 \rightarrow [0, 1]$;
$loc_1, [2, 4] \rightarrow loc_2, [5, 6]$;

$f2'_1 \rightarrow (\emptyset, \emptyset);$
Game state after triggering the constraint: $a_1 \rightarrow loc_1;$
$o_1 \rightarrow [1, 3];$
$a_2 \rightarrow \emptyset;$
$o_2 \rightarrow \emptyset;$
$a_3 > loc_2;$
$o_3 \rightarrow [0, 1];$
$loc_1, [2, 4] \rightarrow loc_2, [5, 6];$
$f2'_1 \rightarrow (\emptyset, \emptyset);$

Example 6 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_3)$
Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \rightarrow loc_1;$
$o_1 \rightarrow [0, 1];$
$a_2 \rightarrow \emptyset;$
$o_2 \rightarrow \emptyset;$
$a_3 > loc_2;$
$o_3 \rightarrow [0, 1];$
$loc_1, [2, 4] \rightarrow loc_2, [5, 6];$
$f2'_1 \rightarrow (\emptyset, \emptyset);$
Game state after triggering the constraint: $a_1 \rightarrow loc_1;$
$o_1 \rightarrow [0, 1];$
$a_2 \rightarrow \emptyset;$
$o_2 \rightarrow \emptyset;$
$a_3 > loc_2;$
$o_3 \rightarrow [0, 1];$
$loc_1, [2, 4] \rightarrow loc_2, [5, 6];$
$f2'_1 \rightarrow (\emptyset, \emptyset);$

Example 7 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_3)$
Functions in the dictionary: loc_2 is a function $f2'$
Game state before triggering the constraint: $a_1 \rightarrow loc_1;$
$o_1 \rightarrow [0, 1];$
$a_2 \rightarrow \emptyset;$
$o_2 \rightarrow \emptyset;$
$a_3 > loc_2;$
$o_3 \rightarrow [0, 1];$
$f2'_1 \rightarrow (\emptyset, \emptyset);$
Game state after triggering the constraint: $a_1 \rightarrow loc_1;$
$o_1 \rightarrow [0, 1];$
$a_2 \rightarrow \emptyset;$
$o_2 \rightarrow \emptyset;$
$a_3 > loc_2;$

$o_3 \rightarrow [0, 1]$;
$f2_1 \rightarrow (\emptyset, \emptyset)$;

Example 8 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$
Functions in the dictionary: $loc_1 isafunction f1'$
Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [3, 4]$;
$f1'_1 \rightarrow (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [3, 4]$;
$f1'_1 \rightarrow (loc_2, [3, 4])$;

Example 9 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$
Functions in the dictionary: $\emptyset$
Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [3, 4]$;
$f1'_1 \rightarrow (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [3, 4]$;
$f1'_1 \rightarrow (\emptyset, \emptyset)$;

Example 10 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_3)$
Functions in the dictionary: loc_1 is a function $f1'$
Game state before triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [3, 4]$;
$a_3 \rightarrow \emptyset$;
$o_3 \rightarrow \emptyset$;
$f1'_1 \rightarrow (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \rightarrow loc_1$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [3, 4]$;
$a_3 \rightarrow \emptyset$;

$o_3 \to \emptyset$;
$f1'_1 \to (\emptyset, \emptyset)$;

Example 11 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$
Functions in the dictionary: loc_2 is a function $f2'$ and loc_1 is a function $f1'$
Game state before triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [1, 3]$;
$a_2 \to loc_2$;
$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_1 \to (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [1, 3]$;
$a_2 \to loc_2$;
$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_1 \to (loc_2, [0, 1])$;

Example 12 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$
Functions in the dictionary: loc_1 is a function $f1'$
Game state before triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [1, 3]$;
$a_2 \to loc_2$;
$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_1 \to (\emptyset, \emptyset)$;
$f1'_1 \to (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \to loc_1$;
$o_1 \to [1, 3]$;
$a_2 \to loc_2$;
$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;
$f2'_1 \to (\emptyset, \emptyset)$;
$f1'_1 \to (loc_2, [0, 1])$;

Example 13 Constraint: $fp_1(a_1, o_1) \supseteq (a_2, o_2)$
Functions in the dictionary: loc_1 is a function $f1'$ and loc_2 is a function $f2'$ and loc_3 is a function $f3'$.
Game state before triggering the constraint: $a_1 \to loc_1, loc_3$;
$o_1 \to [1, 3]$;
$a_2 \to loc_2$;
$o_2 \to [0, 1]$;
$loc_1, [2, 4] \to loc_2, [5, 6]$;

$f2'_1 \rightarrow (\emptyset, \emptyset)$;
$f1'_1 \rightarrow (\emptyset, \emptyset)$;
$f3'_1 \rightarrow (\emptyset, \emptyset)$;
Game state after triggering the constraint: $a_1 \rightarrow loc_1, loc_3$;
$o_1 \rightarrow [1, 3]$;
$a_2 \rightarrow loc_2$;
$o_2 \rightarrow [0, 1]$;
$loc_1, [2, 4] \rightarrow loc_2, [5, 6]$;
$f2'_1 \rightarrow (loc_2, [0, 1])$;
$f1'_1 \rightarrow (\emptyset, \emptyset)$;
$f3'_1 \rightarrow (loc_2, [0, 1])$;

Rule 10

Example 1 Constraint: `memcpy`$(a_1, o_1, a_2, o_2, [4, 4])$
Game state before triggering the constraint: $a_1 \rightarrow loc_3$;
$o_1 \rightarrow [2, 3]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0, 1]$;
$(loc_1, [3, 4]) \rightarrow (loc_2, [5, 6])$;
Game state after triggering the constraint: $a_1 \rightarrow loc_3$;
$o_1 \rightarrow [0, 1]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0, 1]$;
$(loc_1, [3, 4]) \rightarrow (loc_2, [5, 6])$;
$(loc_3, [4, 7]) \rightarrow (loc_2, [5, 6])$

Example 2 Constraint: `memcpy`$(a_1, o_1, a_2, o_2, [4, 4])$
Game state before triggering the constraint: $a_1 \rightarrow loc_3$;
$o_1 \rightarrow [2, 3]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0, 1]$;
$(loc_1, [6, 7]) \rightarrow (loc_2, [5, 6])$;
Game state after triggering the constraint: $a_1 \rightarrow loc_3$;
$o_1 \rightarrow [2, 3]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0, 1]$;
$(loc_1, [6, 7]) \rightarrow (loc_2, [6, 6])$;

Example 3 Constraint: `memcpy`$(a_1, o_1, a_2, o_2, [4, 4])$
Game state before triggering the constraint: $a_1 \rightarrow loc_3$;
$o_1 \rightarrow [2, 3]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0, 1]$;

Game state after triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [2,3]$;
$a_2 \to loc_1$;
$o_2 \to [0,1]$;

Example 4 Constraint: $\texttt{memcpy}(a_1, o_1, a_2, o_2, [4,4])$
Game state before triggering the constraint: $a_2 \to loc_1$;
$o_2 \to [0,1]$;
$(loc_1, [3,4]) \to (loc_2, [5,6])$;
Game state after triggering the constraint: $a_2 \to loc_1$;
$o_2 \to [0,1]$;
$(loc_1, [3,4]) \to (loc_2, [5,6])$;

Example 5 Constraint: $\texttt{memcpy}(a_1, o_1, a_2, o_2, [4,4])$
Game state before triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [2,3]$;
$a_2 \to loc_1$;
$o_2 \to [0,1]$;
$(loc_1, [3,4]) \to (loc_2, [5,6])$;
$(loc_1, [1,2]) \to (loc_4, [7,8])$;
Game state after triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [0,1]$;
$a_2 \to loc_1$;
$o_2 \to [0,1]$;
$(loc_1, [3,4]) \to (loc_2, [5,6])$;
$(loc_1, [1,2]) \to (loc_4, [7,8])$;
$(loc_3, [4,7]) \to (loc_2, [5,6])$;
$(loc_3, [2,5]) \to (loc_4, [7,8])$;

Example 6 Constraint: $\texttt{memcpy}(a_1, o_1, a_2, o_2, [4,4])$
Game state before triggering the constraint: $a_1 \to loc_3, loc5$;
$o_1 \to [2,3]$;
$a_2 \to loc_1$;
$o_2 \to [0,1]$;
$(loc_1, [3,4]) \to (loc_2, [5,6])$;
Game state after triggering the constraint: $a_1 \to loc_3$;
$o_1 \to [0,1]$;
$a_2 \to loc_1$;
$o_2 \to [0,1]$;
$(loc_1, [3,4]) \to (loc_2, [5,6])$;
$(loc_1, [3,4]) \to (loc_4, [7,8])$;
$(loc_3, [4,7]) \to (loc_2, [5,6])$;

$(loc5, [4,7]) \rightarrow (loc_2, [5,6])$;

Example 7 Constraint: $\texttt{memcpy}(a_1, o_1, a_2, o_2, [4,4])$

Game state before triggering the constraint: $a_1 \rightarrow loc_3, loc5$;
$o_1 \rightarrow [2,3]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0,1]$;
$(loc_1, [3,4]) \rightarrow (loc_2, [5,6])$;
$(loc_1, [3,4]) \rightarrow (loc_4, [7,8])$;
Game state after triggering the constraint: $a_1 \rightarrow loc_3$;
$o_1 \rightarrow [0,1]$;
$a_2 \rightarrow loc_1$;
$o_2 \rightarrow [0,1]$;
$(loc_1, [3,4]) \rightarrow (loc_2, [5,6])$;
$(loc_1, [3,4]) \rightarrow (loc_4, [7,8])$;
$(loc_3, [4,7]) \rightarrow (loc_2, [5,6])$;
$(loc5, [4,7]) \rightarrow (loc_2, [5,6])$;
$(loc_3, [4,7]) \rightarrow (loc_2, [7,8])$;
$(loc5, [4,7]) \rightarrow (loc_2, [7,8])$;

3.5.2 Ordering Strategy

The auto-solver, or human player or players, will solve the program in fewer iterations if we solve the game instances in judicious order. In Dynamakr, wisely ordering game instances, and then wisely ordering the constraints in individual game helps bring more meaningfully elements to a player. Our ordering strategy is as follows:

Each constraint will be given a score, which is a 5-tuple of 0's and 1's. The score of a constraint *Con* will be defined as follows:

1. $C5 = 0$.
2. $C4 = 1$ if *Con* belongs to either rule #6, #9 or #10 and the right hand side of *Con* is a pair (n,e), where n is of type *Loc* and e is an interval. Otherwise $C_4 = 0$.
3. $C3 = 1$ if *Con* belongs to either rule #1, #2, #5, #7, #8 or #10 and the right hand side of Con is a pair (n, e), where n is of type *Loc* and e is an interval. Otherwise $C_3 = 0$.
4. $C2 = 1$ if *Con* belongs to either rule #1, #2, #5, #6, #7, #8, #9 or #10 and the right hand side of *Con* is a pair (n, e), where n is not of type *Loc* and e is not an interval. Otherwise, $C_2 = 0$.
5. $C_1 = 1$ if *Con* belongs to either rule #3 or #4. Otherwise $C_1 = 0$.

The following is a few examples of constraints and their scores

Constraint	Rule of the Constraint	Score of the Constraint
$(a_13, o_13) \supseteq f5_1$	3	$(0, 0, 0, 0, 1)$
$(a_12, o_12) \supseteq deref(a_10, o_10)$	2	$(0, 0, 0, 1, 0)$
$(loc_2, [0, 0]) \leftarrow (loc_3, [0, 0])$	6	$(0, 1, 0, 0, 0)$
$(a7, o7) \supseteq deref(loc_2, [0, 0])$	2	$(0, 0, 1, 0, 0)$

3.5.3 Scoring of a Game Instance

Scores are added pointwise. That is, If $A = (x_1, x_2, \ldots, x_5)$ and $B = (y_1, y_2, \ldots, y_5)$ are constraint scores, then $4A + B = (x_1 + y_1, x_2 + y_2, \ldots, x_5 + y_5)$. If a game instance has a set of n constraints $Cons = \{Con_1, Con_2, \ldots, Con_n\}$ and $S = [S_1, S_2, \ldots, S_n]$ is a list, where S_i is the score of the Con_i for $1 < i \leq n$. Then the score of the game instance is defined as sum of the scores of its constraints.

Following are examples of the score of a game instances.

Constraint	Rule of the Constraint	Score of the Constraint
$f6_1 \leftarrow (loc_5, [-\infty, +\infty])$	7	$(0, 0, 1, 0, 0)$
$(a7, o7) \supseteq deref(loc_2, [0, 0])$	2	$(0, 0, 1, 0, 0)$
$f6_2 \leftarrow (a7, o7)$	7	$(0, 0, 0, 1, 0)$
$f6_1 \leftarrow (loc_8, [-\infty, +\infty])$	7	$(0, 0, 1, 0, 0)$
$(a9, o9) \supseteq deref(loc_2, [4, 4])$	2	(0,0,1,0,0)
$f6_2 \leftarrow (a9, o9)$	7	$(0, 0, 0, 1, 0)$

The score of the game instance is $(0, 0, 4, 2, 0)$. The constraint scores are ordered lexicographically with the leftmost position most significant. For example $(0, 0, 4, 2, 2) < (0, 1, 0, 0, 0)$ and $(0, 0, 1, 0, 0) < (0, 0, 1, 0, 1)$. The auto-solver tries to solve the program by looping through each game instance in the program one by one. The following is one example of how applying ordering can help to solve the program in fewer iterations.

Example Suppose there are five game instances in a program. They are named `set1`, `set2`, `run`, `init`, and `main`. The constraints in the game instances and their corresponding constraint scores are given in the following tables:

`set1`

Constraint	Score of the Constraint
$(a_13, o_13) \supseteq f51$	$(0, 0, 0, 0, 1)$

`set2`

Constraint	Score of the Constraint
$(a_14, o_14) \supseteq f81$	(0,0,0,0,1)

`main`

Constraint	Score of the Constraint
$f6_1 \leftarrow (\texttt{loc}_5, [-\infty, +\infty])$	$(0,0,1,0,0)$
$(a7, o7) \supseteq \texttt{deref}(\texttt{loc}_2, [0,0])$	$(0,0,1,0,0)$
$f62 \leftarrow (a7, o7)$	$(0,0,0,1,0)$
$f61 \leftarrow (\texttt{loc}_8, [-\infty, +\infty])$	$(0,0,1,0,0)$
$(a9, o9) \supseteq \texttt{deref}(\texttt{loc}_2, [4,4])$	$(0,0,1,0,0)$
$f6_2 \leftarrow (a9, o9)$	$(0,0,0,1,0)$

`init`

Constraint	Score of the Constraint
$*(\texttt{loc}_2, [0,0]) \leftarrow (\texttt{loc}_3, [0,0])$	$(0,1,0,0,0)$
$*(\texttt{loc}_2, [4,4]) \leftarrow (\texttt{loc}_3, [20,20])$	$(0,1,0,0,0)$

`run`

Constraint	Score of the Constraint
$(a10, o10) \supseteq f61$	$(0,0,0,0,1)$
$(a11, o11) \supseteq f62$	$(0,0,0,0,1)$
$(a12, o12) \supseteq \mathit{deref}(a10, o10)$	$(0,0,0,1,0)$
$f6_1 \leftarrow (\texttt{loc}_8, [-\infty, +\infty])$	$(0,0,1,0,0)$

The scores of the game instances are given in decreasing order in the following table

Game Instance	Score
`init`	$(0,2,0,0,0)$
`main`	$(0,0,4,2,0)$
`run`	$(0,0,1,1,2)$
`set1`	$(0,0,0,0,1)$
`set2`	$(0,0,0,0,1)$

If the auto-solver tries to solve the program in the order given above it can solve the program in one iteration. Other orders will result in more than one iteration.

In the dynamic scoring strategy, the global points-to-graph is used. Each constraint will be given a score, which is a 5-tuple of 0's and 1's. The score of a constraint *Con* will be defined as follows:

1. $C_5 = 1$ if there is at least one arc (source,destination) in the points-to-graph such that the ID of source equals to the ID of right hand of *Con*, and otherwise $C_5 = 0$.

2. $C_4 = 1$ if *Con* belongs to either rule #6, #9 or #10 and the right hand side of *Con* is a pair (n, e), where n is of type *Loc* and e is an interval. Otherwise $C_4 = 0$.

3. $C_3 = 1$ if *Con* belongs to either rule #1, #2, #5, #7, #8 or #10 and the right hand side of *Con* is a pair (n, e), where n is of type *Loc* and e is an interval. Otherwise $C_3 = 0$.

4. $C_2 = 1$ if *Con* belongs to either rule #1, #2, #5, #6, #7, #8, #9 or #10 and the right hand side of *Con* is a pair (n, e), where n is not of type *Loc* and e is not an interval. Otherwise $C_2 = 0$.

5. $C_1 = 1$ if *Con* belongs to either rule #3 or #4. Otherwise $C_1 = 0$.

The scoring of a game instance and the ordering are the same as the *static* strategy.

3.6 Implementation of the Analyzer

3.6.1 Anchors File

The constraint generator produces an *anchors file* that lists all the anchor variables that the game backend should report on. An entry in the anchor file has the following syntax:

```
<anchor context="..." function="..." seqnr="...">
<pointer-pair>
  <address-component>
    <var-address uid="n"/>
  </address-component>
  <offset-component>
    <var-offset uid="m"/>
  </offset-component>
</pointer-pair>
</anchor>
```

The `context`, `function` and `seqnr` attributes are only relevant to the CodeHawk buffer-overflow analyzer and can be completely ignored by the backend. The results of the pointer analysis should only be reported for the variables that are listed under the `pointer pair component`. Each anchor refers to an address variable $\mathbf{A}n$ and an offset variable $\mathbf{O}m$. The variable identifiers n and m are often identical but not always. For the game back-end, they should just be considered as separate variables.

The format of the results file is very similar to that of the previous version. It consists of a list of entries that have the following syntax:

```
<address-variable-size uid="n" lower-bound="..." upper-bound="..." />
```

or

```
<offset-variable-size uid="m" lower-bound="..." upper-bound="..." />
```

An `address-variable-size` reports the size of all memory locations that are assigned to an address variable $\mathbf{A}n$. For example, if at the end of the game we find that the target set of address variable $\mathbf{A}n$ contains memory locations $M_1, M_2, \ldots, M_k$, then the range reported in the `address-variable-size` is

$$\text{size}(M_1) \cup \text{size}(M_2) \cup \ldots \cup \text{size}(M_k)$$

An `offset-variable-range` entry simply reports the range of an offset variable $\mathbf{O}m$ as it stands at the end of the game.

As an illustrative example, consider the following anchor file produced by the new constraints generator:

```
<?xml version="1.0" encoding="UTF 8"?>
  <anchor context="?;main@105" function="set1" seqnr="4">
    <pointer-pair>
      <address-component>
```

```
        <var-address uid="16"/>
      </address-component>
      <offset-component>
        <var-offset uid="16"/>
      </offset-component>
    </pointer-pair>
  </anchor>
  <anchor context="?;main@105;set1@74" function="set2" seqnr="1">
    <pointer-pair>
      <address-component>
        <var-address uid="18"/>
      </address-component>
      <offset-component>
        <var-offset uid="18"/>
      </offset-component>
    </pointer-pair>
  </anchor>
  <anchor context="?;main@121" function="set1" seqnr="4">
    <pointer-pair>
      <address-component>
        <var-address uid="13"/>
      </address-component>
      <offset-component>
        <var-offset uid="13"/>
      </offset-component>
    </pointer-pair>
  </anchor>
  <anchor context="?;main@121;set1@74" function="set2" seqnr="1">
    <pointer-pair>
      <address-component>
        <var-address uid="15"/>
      </address-component>
      <offset-component>
        <var-offset uid="15"/>
      </offset-component>
    </pointer-pair>
  </anchor>
  <anchor context="?" function="set1" seqnr="4">
    <pointer-pair>
      <address-component>
        <var-address uid="7"/>
      </address-component>
      <offset-component>
        <var-offset uid="7"/>
      </offset-component>
    </pointer-pair>
  </anchor>
  <anchor context="?;set1@74" function="set2" seqnr="1">
    <pointer-pair>
      <address-component>
        <var-address uid="12"/>
      </address-component>
      <offset-component>
        <var-offset uid="12"/>
      </offset-component>
    </pointer-pair>
  </anchor>
  <anchor context="?" function="set2" seqnr="1">
    <pointer-pair>
      <address-component>
        <var-address uid="11"/>
      </address-component>
      <offset-component>
        <var-offset uid="11"/>
      </offset-component>
    </pointer-pair>
  </anchor>
```

A possible results file produced by the game play, game model, and backend might be:

```
<?xml version="1.0" encoding="UTF 8"?>
  <address-variable-size uid="16" lower-bound="40" upper-bound="40" />
  <offset-variable-range uid="16" lower-bound="16" upper-bound="16" />
  <address-variable-size uid="18" lower-bound="40" upper-bound="40" />
  <offset-variable-range uid="18" lower-bound="20" upper-bound="20" />
  <address-variable-size uid="13" lower-bound="40" upper-bound="40" />
  <offset-variable-range uid="13" lower-bound="36" upper-bound="36" />
  <address-variable-size uid="15" lower-bound="40" upper-bound="40" />
  <offset-variable-range uid="15" lower-bound="40" upper-bound="40" />
```

Please note that anchors do not necessarily have to appear in the results file. In this case, the buffer-overflow verifier assumes that the range is $[-\infty, +\infty]$. Also, entries may be listed in any order and do not have to be grouped by UID or type. Address and offset variables are completely decoupled and do not need to be paired. The backend need only report values for anchors appearing in the anchor file as these are the values needed for subsequent verification runs. All context information is handled by the CodeHawk analyzer so the game backend has to manage only the variables (whether symbolic or numerical) as specified by their UIDs and listed in the CodeHawk-generated dictionary.

3.6.2 Detailed Example

In this section we present a small C program with an intentional complex verification defect and show how our approach finds that defect. We start with the following C-language program:

```
int A[10];
int B[20];

typedef int *(*fptr)(int *);

int *f1(int *p) {
  return p + 1;
}

int *f5(int *p) {
  return p + 5;
}
```

```
13
14  void exec(fptr f, int *p) {
15    int *pp = f(p);
16    *pp = 1;
17  }
18
19  main() {
20    exec(f1, &A[8]);
21    exec(f5, &B[15]);
22  }
```

After studying the small program we can work out that the call on line 21 will result in a buffer overflow in function `exec()` when the program performs the assignment at line 16. Let us see if our analyzer and game model can find this problem automatically.

Running the CodeHawk constraint generator produces the following dictionary and anchors files:

```
% dictionary
<memory-element uid="22">
  <function name="f1"/>
 </memory-element>
 <memory-element uid="23">
  <function name="f5"/>
 </memory-element>
 <memory-element uid="24">
  <function name="main"/>
 </memory-element>
 <memory-element uid="26">
  <global-variable name="B" size="80"/>
 </memory-element>
 <memory-element uid="25">
  <global-variable name="A" size="40"/>
 </memory-element>
 <memory-element uid="21">
  <function name="exec"/>
 </memory-element>
```

```
% anchors
<anchor context="?" function="exec" seqnr="5">
  <pointer-pair>
   <address-component>
        <var-address uid="9"/>
   </address-component>
   <offset-component>
        <var-offset uid="9"/>
   </offset-component>
  </pointer-pair>
 </anchor>
 <anchor context="?" function="exec" seqnr="6">
  <pointer-pair>
   <address-component>
        <var-address uid="10"/>
   </address-component>
   <offset-component>
```

```
        <var-offset uid="10"/>
   </offset-component>
  </pointer-pair>
 </anchor>
 <anchor context="?" function="exec" seqnr="8">
  <pointer-pair>
   <address-component>
        <var-address uid="11"/>
   </address-component>
   <offset-component>
        <var-offset uid="11"/>
   </offset-component>
  </pointer-pair>
 </anchor>
 <anchor context="?" function="exec" seqnr="11">
  <pointer-pair>
   <address-component>
        <var-address uid="12"/>
   </address-component>
   <offset-component>
        <var-offset uid="12"/>
   </offset-component>
  </pointer-pair>
 </anchor>
 <anchor context="?" function="f1" seqnr="1">
  <pointer-pair>
   <address-component>
        <var-address uid="20"/>
   </address-component>
   <offset-component>
        <var-offset uid="20"/>
   </offset-component>
  </pointer-pair>
 </anchor>
 <anchor context="?" function="f5" seqnr="3">
  <pointer-pair>
   <address-component>
        <var-address uid="17"/>
   </address-component>
   <offset-component>
        <var-offset uid="17"/>
   </offset-component>
  </pointer-pair>
 </anchor>
```

The constraint generator also produces a constraint file for each function in the program:

```
%  main.xml
<pointer-constraints>
  <function uid="24">
    <constraints>
      <offset-flow>
        <lhs>
          <var-offset uid="3"/>
        </lhs>
      <rhs>
        <offset>
          <interval-value lower-bound="-oo" upper-bound="+oo"/>
        </offset>
      </rhs>
    </offset-flow>
    <address-flow>
      <lhs>
        <var-address uid="3"/>
      </lhs>
      <rhs>
        <global-memory-location uid="22"/>
      </rhs>
    </address-flow>
    <offset-flow>
      <lhs>
        <var-offset uid="4"/>
      </lhs>
      <rhs>
        <offset>
          <constant-value value="32"/>
        </offset>
      </rhs>
    </offset-flow>
    <address-flow>
      <lhs>
        <var-address uid="4"/>
      </lhs>
      <rhs>
        <global-memory-location uid="25"/>
      </rhs>
    </address-flow>
    <pointer-flow>
      <lhs>
        <function-arg argument="1" function-uid="21"/>
      </lhs>
      <rhs>
        <pointer-pair>
          <address-component>
            <var-address uid="3"/>
          </address-component>
          <offset-component>
            <offset>
              <var-offset uid="3"/>
            </offset>
          </offset-component>
        </pointer-pair>
      </rhs>
    </pointer-flow>
    <pointer-flow>
      <lhs>
        <function-arg argument="2" function-uid="21"/>
      </lhs>
      <rhs>
        <pointer-pair>
          <address-component>
            <var-address uid="4"/>
          </address-component>
          <offset-component>
            <offset>
              <var-offset uid="4"/>
            </offset>
          </offset-component>
        </pointer-pair>
      </rhs>
    </pointer-flow>
    <offset-flow>
      <lhs>
        <var-offset uid="5"/>
      </lhs>
      <rhs>
        <offset>
          <interval-value lower-bound="-oo" upper-bound="+oo"/>
        </offset>
      </rhs>
    </offset-flow>
    <address-flow>
      <lhs>
        <var-address uid="5"/>
      </lhs>
      <rhs>
        <global-memory-location uid="23"/>
      </rhs>
    </address-flow>
    <offset-flow>
      <lhs>
        <var-offset uid="6"/>
      </lhs>
      <rhs>
        <offset>
          <constant-value value="60"/>
        </offset>
      </rhs>
    </offset-flow>
    <address-flow>
      <lhs>
        <var-address uid="6"/>
      </lhs>
      <rhs>
        <global-memory-location uid="26"/>
      </rhs>
    </address-flow>
    <pointer-flow>
      <lhs>
        <function-arg argument="1" function-uid="21"/>
      </lhs>
      <rhs>
        <pointer-pair>
```

```
            <address-component>
              <var-address uid="5"/>
            </address-component>
            <offset-component>
              <offset>
                <var-offset uid="5"/>
              </offset>
            </offset-component>
          </pointer-pair>
        </rhs>
      </pointer-flow>
      <pointer-flow>
        <lhs>
          <function-arg argument="2" function-uid="21"/>
        </lhs>
        <rhs>
          <pointer-pair>
            <address-component>
              <var-address uid="6"/>
            </address-component>
            <offset-component>
              <offset>
                <var-offset uid="6"/>
              </offset>
            </offset-component>
          </pointer-pair>
        </rhs>
      </pointer-flow>
    </constraints>
  </function>
</pointer-constraints>
```

```
% exec.xml
<pointer-constraints>
  <function uid="21">
    <constraints>
      <pointer-flow>
        <lhs>
          <pointer-pair>
            <address-component>
              <var-address uid="9"/>
            </address-component>
            <offset-component>
              <var-offset uid="9"/>
            </offset-component>
          </pointer-pair>
        </lhs>
        <rhs>
          <function-arg argument="1" function-uid="21"/>
        </rhs>
      </pointer-flow>
      <pointer-flow>
        <lhs>
          <pointer-pair>
      <address-component>
        <var-address uid="10"/>
      </address-component>
      <offset-component>
        <var-offset uid="10"/>
      </offset-component>
    </pointer-pair>
  </lhs>
  <rhs>
    <function-arg argument="2" function-uid="21"/>
  </rhs>
</pointer-flow>
<pointer-flow>
  <lhs>
    <pointer-pair>
      <address-component>
        <var-address uid="11"/>
      </address-component>
      <offset-component>
        <var-offset uid="11"/>
      </offset-component>
    </pointer-pair>
  </lhs>
  <rhs>
    <deref>
      <pointer-pair>
        <address-component>
          <var-address uid="9"/>
        </address-component>
        <offset-component>
          <offset>
            <var-offset uid="9"/>
          </offset>
        </offset-component>
      </pointer-pair>
    </deref>
  </rhs>
</pointer-flow>
<offset-flow>
  <lhs>
    <var-offset uid="12"/>
  </lhs>
  <rhs>
    <offset>
      <var-offset uid="13"/>
    </offset>
  </rhs>
</offset-flow>
<address-flow>
  <lhs>
    <var-address uid="12"/>
  </lhs>
  <rhs>
    <var-address uid="13"/>
  </rhs>
</address-flow>
<offset-flow>
  <lhs>
    <var-offset uid="14"/>
  </lhs>
  <rhs>
    <offset>
      <var-offset uid="10"/>
    </offset>
  </rhs>
</offset-flow>
<address-flow>
  <lhs>
    <var-address uid="14"/>
  </lhs>
  <rhs>
    <var-address uid="10"/>
  </rhs>
</address-flow>
<pointer-flow>
  <lhs>
    <arg-fp arg="1">
      <pointer-pair>
        <address-component>
          <var-address uid="11"/>
        </address-component>
        <offset-component>
          <offset>
            <var-offset uid="11"/>
          </offset>
        </offset-component>
      </pointer-pair>
    </arg-fp>
  </lhs>
  <rhs>
    <pointer-pair>
      <address-component>
        <var-address uid="14"/>
      </address-component>
      <offset-component>
        <offset>
          <var-offset uid="14"/>
        </offset>
      </offset-component>
    </pointer-pair>
  </rhs>
</pointer-flow>
<pointer-flow>
  <lhs>
    <pointer-pair>
      <address-component>
        <var-address uid="13"/>
      </address-component>
```

```
            <offset-component>
              <var-offset uid="13"/>
            </offset-component>
          </pointer-pair>
        </lhs>
        <rhs>
          <ret-fp>
            <pointer-pair>
              <address-component>
                <var-address uid="11"/>
              </address-component>
              <offset-component>
                <offset>
                  <var-offset uid="11"/>
                </offset>
              </offset-component>
            </pointer-pair>
          </ret-fp>
        </rhs>
      </pointer-flow>
    </constraints>
  </function>
</pointer-constraints>
```

```
% f1.xml
<pointer-constraints>
  <function uid="22">
    <constraints>
      <pointer-flow>
        <lhs>
         <pointer-pair>
           <address-component>
             <var-address uid="20"/>
           </address-component>
           <offset-component>
             <var-offset uid="20"/>
           </offset-component>
         </pointer-pair>
       </lhs>
       <rhs>
         <function-arg argument="1" function-uid="22"/>
      </rhs>
    </pointer-flow>
    <offset-flow>
      <lhs>
        <var-offset uid="19"/>
      </lhs>
      <rhs>
        <offset>
          <linear-expression>
            <constant-factor value="4"/>
              <component coefficient="1" var-uid="20"/>
          </linear-expression>
        </offset>
      </rhs>
    </offset-flow>
    <address-flow>
      <lhs>
        <var-address uid="18"/>
      </lhs>
      <rhs>
        <var-address uid="20"/>
      </rhs>
    </address-flow>
    <pointer-flow>
      <lhs>
        <function-return function-uid="22"/>
      </lhs>
      <rhs>
        <pointer-pair>
          <address-component>
            <var-address uid="18"/>
          </address-component>
          <offset-component>
            <var-offset uid="19"/>
          </offset-component>
        </pointer-pair>
      </rhs>
    </pointer-flow>
  </constraints>
 </function>
</pointer-constraints>
```

```
% f5.xml
<pointer-constraints>
  <function uid="23">
    <constraints>
      <pointer-flow>
        <lhs>
          <pointer-pair>
            <address-component>
              <var-address uid="17"/>
            </address-component>
            <offset-component>
              <var-offset uid="17"/>
            </offset-component>
          </pointer-pair>
        </lhs>
        <rhs>
          <function-arg argument="1" function-uid="23"/>
        </rhs>
      </pointer-flow>
      <offset-flow>
        <lhs>
          <var-offset uid="16"/>
        </lhs>
        <rhs>
          <offset>
```

```
            <linear-expression>
              <constant-factor value="20"/>
              <component coefficient="1" var-uid="17"/>
            </linear-expression>
          </offset>
        </rhs>
      </offset-flow>
      <address-flow>
        <lhs>
          <var-address uid="15"/>
        </lhs>
        <rhs>
          <var-address uid="17"/>
        </rhs>
      </address-flow>
      <pointer-flow>
        <lhs>
          <function-return function-uid="23"/>
        </lhs>
        <rhs>
          <pointer-pair>
            <address-component>
              <var-address uid="15"/>
            </address-component>
            <offset-component>
              <var-offset uid="16"/>
            </offset-component>
          </pointer-pair>
        </rhs>
      </pointer-flow>
    </constraints>
  </function>
</pointer-constraints>
```

In order to follow game play more easily we reproduce the constraints in plain text with cross reference numbers:

```
// function main
1:  offset-flow(O#3, [-oo; +oo])  // offest of #3 is an integer
2:  address-flow(A#3, &f1)        // if #3 pointx to x then f1 does
3:  offset-flow(O#4, [32; 32])
4:  address-flow(A#4, &A)
5:  pointer-flow(exec@1, (A#3, O#3))
6:  pointer-flow(exec@2, (A#4, O#4))
7:  offset-flow(O#5, [-oo; +oo])
8:  address-flow(A#5, &f5)
9:  offset-flow(O#6, [60; 60])
10: address-flow(A#6, &B)
11: pointer-flow(exec@1, (A#5, O#5))  // If 1st parm of exec points  to
    x, then node #5 points to x
12: pointer-flow(exec@2, (A#6, O#6))
```

```
// function exec
13: pointer-flow((A#9, O#9), exec@1)
14: pointer-flow((A#10, O#10), exec@2)
```

```
15: pointer-flow((A#11, O#11), deref(A#9, O#9))
16: offset-flow(O#12, O#13)
17: address-flow(A#12, A#13)
18: offset-flow(O#14, O#10)
19: address-flow(A#14, A#10)
20: pointer-flow(arg-fp@1(A#11, O#11), (A#14, O#14))
21: pointer-flow((A#13, O#13), ret-fp(A#11, O#11))
```

```
// function f1
22: pointer-flow((A#20, O#20), f1@1)
23: offset-flow(O#19, 4 + O#20)
24: address-flow(A#18, A#20)
25: pointer-flow(f1@ret, (A#18, O#19))
```

```
// function f5
26: pointer-flow((A#17, O#17), f5@1)
27: offset-flow(O#16, 20 + O#17)
28: address-flow(A#15, A#17)
29: pointer-flow(f5@ret, (A#15, O#16))
```

We now proceed with sample game play. For the sake of simplicity, we apply the optimal strategy that leads to the best solution for the system of constraints. We could of course fire the constraints in no particular order.

Initial game configuration:

Addresses	Offsets	Functions
$A\#3 = \emptyset$	$O\#3 = \emptyset$	exec@1 $= (\emptyset, \emptyset)$
$A\#4 = \emptyset$	$O\#4 = \emptyset$	exec@2 $= (\emptyset, \emptyset)$
$A\#5 = \emptyset$	$O\#5 = \emptyset$	f1@1 $= (\emptyset, \emptyset)$
$A\#6 = \emptyset$	$O\#6 = \emptyset$	f1@ret $= (\emptyset, \emptyset)$
$A\#9 = \emptyset$	$O\#9 = \emptyset$	f5@1 $= (\emptyset, \emptyset)$
$A\#10 = \emptyset$	$O\#10 = \emptyset$	f5@ret $= (\emptyset, \emptyset)$
$A\#11 = \emptyset$	$O\#11 = \emptyset$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

After firing constraints 1 through 4 we assign explicit locations and offsets to four variables. Applying the rules we obtain the following game configuration:

Firing constraints 1-4:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	exec@1 = $(\emptyset, \emptyset)$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	exec@2 = $(\emptyset, \emptyset)$
$A\#5 = \emptyset$	$O\#5 = \emptyset$	f1@1 = $(\emptyset, \emptyset)$
$A\#6 = \emptyset$	$O\#6 = \emptyset$	f1@ret = $(\emptyset, \emptyset)$
$A\#9 = \emptyset$	$O\#9 = \emptyset$	f5@1 = $(\emptyset, \emptyset)$
$A\#10 = \emptyset$	$O\#10 = \emptyset$	f5@ret = $(\emptyset, \emptyset)$
$A\#11 = \emptyset$	$O\#11 = \emptyset$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

After firing constraints 5 and 6 we assign values to the formal parameters of function `exec`.

Firing constraints 5-6:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	exec@1 = $(\{\&\mathrm{f1}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	exec@2 = $(\{\&\mathrm{A}\}, [32, 32])$
$A\#5 = \emptyset$	$O\#5 = \emptyset$	f1@1 = $(\emptyset, \emptyset)$
$A\#6 = \emptyset$	$O\#6 = \emptyset$	f1@ret = $(\emptyset, \emptyset)$
$A\#9 = \emptyset$	$O\#9 = \emptyset$	f5@1 = $(\emptyset, \emptyset)$
$A\#10 = \emptyset$	$O\#10 = \emptyset$	f5@ret = $(\emptyset, \emptyset)$
$A\#11 = \emptyset$	$O\#11 = \emptyset$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Firing constraints 7 through 10 we assign explicit locations and offsets to four other variables. Applying the rules, we obtain the following game configuration:

Firing constraints 7-10:

Addresses	Offsets	Functions
$A\#3 = \{\&\text{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\text{exec@1} = (\{\&\text{f1}\}, [-\infty, +\infty])$
$A\#4 = \{\&\text{A}\}$	$O\#4 = [32, 32]$	$\text{exec@2} = (\{\&\text{A}\}, [32, 32])$
$A\#5 = \{\&\text{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\text{f1@1} = (\emptyset, \emptyset)$
$A\#6 = \{\&\text{B}\}$	$O\#6 = [60, 60]$	$\text{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \emptyset$	$O\#9 = \emptyset$	$\text{f5@1} = (\emptyset, \emptyset)$
$A\#10 = \emptyset$	$O\#10 = \emptyset$	$\text{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \emptyset$	$O\#11 = \emptyset$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Continuing with constraints 11 and 12 we assign values to the formal parameters of function `exec`.

Firing constraints 11-12:

Addresses	Offsets	Functions
$A\#3 = \{\&\text{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\text{exec@1} = (\{\&\text{f1}, \&\text{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\text{A}\}$	$O\#4 = [32, 32]$	$\text{exec@2} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#5 = \{\&\text{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\text{f1@1} = (\emptyset, \emptyset)$
$A\#6 = \{\&\text{B}\}$	$O\#6 = [60, 60]$	$\text{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \emptyset$	$O\#9 = \emptyset$	$\text{f5@1} = (\emptyset, \emptyset)$
$A\#10 = \emptyset$	$O\#10 = \emptyset$	$\text{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \emptyset$	$O\#11 = \emptyset$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Firing constraints 13 and 14 and applying rules assign values to four more variables, producing the following game configuration:

Firing constraints 13-14:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\emptyset, \emptyset)$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\emptyset, \emptyset)$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \emptyset$	$O\#11 = \emptyset$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Firing constraint 15 tries to dereference a pointer to a function. Following the rule for dereference in the game model we obtain the following configuration:

Firing constraint 15:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\emptyset, \emptyset)$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\emptyset, \emptyset)$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \emptyset$	$O\#14 = \emptyset$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Firing constraints 18 and 19 simply move around sets of addresses and offsets across variables.

Firing constraints 18-19:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\emptyset, \emptyset)$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\emptyset, \emptyset)$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Firing constraint 20 assigns a value to the argument of a function called by pointer. The function pointer is given by variable `A#11`, which refers to functions `f1` and `f5`. Applying the rule for indirect function calls, we obtain the following configuration:

Firing constraint 20:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \emptyset$	$O\#20 = \emptyset$	

Firing constraint 22 retrieves the value for the argument of function `f1`.

Firing constraint 22:

Addresses	Offsets	Functions
$A\#3 = \{\&\text{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\text{exec@1} = (\{\&\text{f1}, \&\text{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\text{A}\}$	$O\#4 = [32, 32]$	$\text{exec@2} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#5 = \{\&\text{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\text{f1@1} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#6 = \{\&\text{B}\}$	$O\#6 = [60, 60]$	$\text{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \{\&\text{f1}, \&\text{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\text{f5@1} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#10 = \{\&\text{A}, \&\text{B}\}$	$O\#10 = [32, 60]$	$\text{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \{\&\text{f1}, \&\text{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\text{A}, \&\text{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \emptyset$	$O\#19 = \emptyset$	
$A\#20 = \{\&\text{A}, \&\text{B}\}$	$O\#20 = [32, 60]$	

Firing constraints 23 and 24 move around symbolic and offset values, performing some arithmetic operations on the offsets.

Firing constraints 23-24:

Addresses	Offsets	Functions
$A\#3 = \{\&\text{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\text{exec@1} = (\{\&\text{f1}, \&\text{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\text{A}\}$	$O\#4 = [32, 32]$	$\text{exec@2} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#5 = \{\&\text{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\text{f1@1} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#6 = \{\&\text{B}\}$	$O\#6 = [60, 60]$	$\text{f1@ret} = (\emptyset, \emptyset)$
$A\#9 = \{\&\text{f1}, \&\text{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\text{f5@1} = (\{\&\text{A}, \&\text{B}\}, [32, 60])$
$A\#10 = \{\&\text{A}, \&\text{B}\}$	$O\#10 = [32, 60]$	$\text{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \{\&\text{f1}, \&\text{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\text{A}, \&\text{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \{\&\text{A}, \&\text{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\text{A}, \&\text{B}\}$	$O\#20 = [32, 60]$	

Firing constraint 25 assigns a return value to function `f1`.

Firing constraint 25:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	exec@1 = $(\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	exec@2 = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	f1@1 = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	f1@ret = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [36, 64])$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	f5@1 = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	f5@ret = $(\emptyset, \emptyset)$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \emptyset$	$O\#17 = \emptyset$	
$A\#18 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#20 = [32, 60]$	

Firing constraint 26 retrieves the value for the argument of function `f5`.

Firing constraint 26:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	exec@1 = $(\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	exec@2 = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	f1@1 = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	f1@ret = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [36, 64])$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	f5@1 = $(\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	f5@ret = $(\emptyset, \emptyset)$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \emptyset$	$O\#16 = \emptyset$	
$A\#17 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#17 = [32, 60]$	
$A\#18 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#20 = [32, 60]$	

Constraints 27 and 28 move around symbolic and offset values, performing some arithmetic operations on the offsets.

Firing constraints 27-28:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [36, 64])$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\emptyset, \emptyset)$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#16 = [52, 80]$	
$A\#17 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#17 = [32, 60]$	
$A\#18 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#20 = [32, 60]$	

Constraint 29 assigns a return value to function `f5`.

Firing constraint 29:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [36, 64])$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [52, 80])$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \emptyset$	$O\#13 = \emptyset$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#16 = [52, 80]$	
$A\#17 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#17 = [32, 60]$	
$A\#18 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#20 = [32, 60]$	

Firing constraint 21 retrieves the value of a function called by pointer. The function pointer is given by variable `A#11`, which refers to functions `t1` and `f5`. Applying the rule for indirect function returns, we obtain the following configuration:

Firing constraint 21:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [36, 64])$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [52, 80])$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \emptyset$	$O\#12 = \emptyset$	
$A\#13 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#13 = [36, 80]$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#16 = [52, 80]$	
$A\#17 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#17 = [32, 60]$	
$A\#18 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#20 = [32, 60]$	

Firing constraints simply transfer values from variable to variable.

Firing constraints 16-17:

Addresses	Offsets	Functions
$A\#3 = \{\&\mathrm{f1}\}$	$O\#3 = [-\infty, +\infty]$	$\mathrm{exec@1} = (\{\&\mathrm{f1}, \&\mathrm{f5}\}, [-\infty, +\infty])$
$A\#4 = \{\&\mathrm{A}\}$	$O\#4 = [32, 32]$	$\mathrm{exec@2} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#5 = \{\&\mathrm{f1}\}$	$O\#5 = [-\infty, +\infty]$	$\mathrm{f1@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#6 = \{\&\mathrm{B}\}$	$O\#6 = [60, 60]$	$\mathrm{f1@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [36, 64])$
$A\#9 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#9 = [-\infty, +\infty]$	$\mathrm{f5@1} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [32, 60])$
$A\#10 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#10 = [32, 60]$	$\mathrm{f5@ret} = (\{\&\mathrm{A}, \&\mathrm{B}\}, [52, 80])$
$A\#11 = \{\&\mathrm{f1}, \&\mathrm{f5}\}$	$O\#11 = [-\infty, +\infty]$	
$A\#12 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#12 = [36, 80]$	
$A\#13 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#13 = [36, 80]$	
$A\#14 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#14 = [32, 60]$	
$A\#15 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#16 = [52, 80]$	
$A\#17 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#17 = [32, 60]$	
$A\#18 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#19 = [36, 64]$	
$A\#20 = \{\&\mathrm{A}, \&\mathrm{B}\}$	$O\#20 = [32, 60]$	

The system has been solved. Firing any constraint leaves the previous configuration unchanged. We have therefore reached a fixpoint and obtained a solution of the system of pointer constraints. We now need to report the values of the variables listed in the anchor file, *i.e.,* the variables `A#9`, `O#9`, `A#10`, `O#10`, `A#11`, `O#11`, `A#12`, `O#12`, `A#17`, `O#17`, `A#20`, `O#20`. According to the dictionary, the size of variable `A` is 40 and the size of variable `B` is 80. The size of a function is 4 bytes by convention in the analyzer.

We generate the following result file:

```
<address-variable-size uid="9" lower-bound="4" upper-bound="4"/>
<offset-variable-range uid="9" lower-bound="-oo" upper-bound="+oo"/>
<address-variable-size uid="10" lower-bound="40" upper-bound="80"/>
<offset-variable-range uid="10" lower-bound="32" upper-bound="60"/>
```

```
<address-variable-size uid="11" lower-bound="4" upper-bound="4"/>
<offset-variable-range uid="11" lower-bound="-oo" upper-bound="+oo"/>
<address-variable-size uid="12" lower-bound="40" upper-bound="80"/>
<offset-variable-range uid="12" lower-bound="36" upper-bound="80"/>
<address-variable-size uid="17" lower-bound="40" upper-bound="80"/>
<offset-variable-range uid="17" lower-bound="32" upper-bound="60"/>
<address-variable-size uid="20" lower-bound="40" upper-bound="80"/>
<offset-variable-range uid="20" lower-bound="32" upper-bound="60"/>
```

We now pass this result file to the CodeHawk CircuitBot buffer overflow analyzer together with the dictionary, anchors file, constraints files, and original CIL representation of the small program in order to obtain the verification results. The verification result yields overflow and underflow checks with safe, warning and error flags for each context and location by function. Let us first examine the verification result without the benefit of the pointer analysis. We see that the analyzer cannot resolve the proof obligations and yields warnings (unknowns) for every check:

```
<?xml version="1.0" encoding="UTF-8"?>
 <buffer-overflow-checks>
  <function uid="33">
   <checks>
    <overflow-check context="?;main@272" line="14" status="warning"/>
    <underflow-check context="?;main@272" line="14" status="warning"/>
    <overflow-check context="?;main@272" line="13" status="warning"/>
    <underflow-check context="?;main@272" line="13" status="warning"/>
    <overflow-check context="?;main@272" line="13" status="warning"/>
    <underflow-check context="?;main@272" line="13" status="warning"/>
    <overflow-check context="?;main@254" line="14" status="warning"/>
    <underflow-check context="?;main@254" line="14" status="warning"/>
    <overflow-check context="?;main@254" line="13" status="warning"/>
    <underflow-check context="?;main@254" line="13" status="warning"/>
    <overflow-check context="?;main@254" line="13" status="warning"/>
    <underflow-check context="?;main@254" line="13" status="warning"/>
    <overflow-check context="?" line="14" status="warning"/>
    <underflow-check context="?" line="14" status="warning"/>
    <overflow-check context="?" line="13" status="warning"/>
    <underflow-check context="?" line="13" status="warning"/>
    <overflow-check context="?" line="13" status="warning"/>
    <underflow-check context="?" line="13" status="warning"/>
   </checks>
  </function>
 </buffer-overflow-checks>
```

Now let us rerun the analyzer with the benefit of the pointer analysis results. We find now that the analyzer can identify the overflow error condition. Line 5 of the XML result file below shows the verification result showing the overflow error, found in the `exec` function, at line 14 of the source file (which corresponds to line 16 of our representation above). The game model and procedure has been successful.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2  <buffer-overflow-checks>
3   <function uid="33">
4    <checks>
```

```
   <overflow-check context="?;main@272" line="14" num-var="tmp_3N"
      range-max="79" span-min="80" status="error" sym-var="tmp_3S"/>
   <underflow-check context="?;main@272" line="14" status="safe"/>
   <overflow-check context="?;main@272" line="13" status="warning"/>
   <underflow-check context="?;main@272" line="13" status="warning"/>
   <overflow-check context="?;main@272" line="13" status="warning"/>
   <underflow-check context="?;main@272" line="13" status="warning"/>
   <overflow-check context="?;main@254" line="14" status="safe"/>
   <underflow-check context="?;main@254" line="14" status="safe"/>
   <overflow-check context="?;main@254" line="13" status="warning"/>
   <underflow-check context="?;main@254" line="13" status="warning"/>
   <overflow-check context="?;main@254" line="13" status="warning"/>
   <underflow-check context="?;main@254" line="13" status="warning"/>
   <overflow-check context="?" line="14" status="warning"/>
   <underflow-check context="?" line="14" status="warning"/>
   <overflow-check context="?" line="13" status="warning"/>
   <underflow-check context="?" line="13" status="warning"/>
   <overflow-check context="?" line="13" status="warning"/>
   <underflow-check context="?" line="13" status="warning"/>
  </checks>
 </function>
</buffer-overflow-checks>
```

3.7 Software Architecture

Early in the project the team members conducted a typical software engineering effort to gather requirements and prepare its software architecture. The team prepared an *Architecture Description Document* (ADD) to describe these engineering activities, the driving requirements and constraints, views and tactics, the trade studies and quality attributes, design patterns, and themes for communication for the project ([2, 3]). The ADD was a deliverable item under the contract.

There were a few main results of the architectural analysis that proved valuable throughout the project. The first was to implement a stateless game and backend workflow to avoid having to keep track of progress relationships between game elements and players. The second was to preprocess as much of the game data as possible in advance, preparing and positioning a large volume of data for play to avoid computing values at play time. The third was to create an interactive game for client-neutral platform. The team did not change or undo any of these decisions and they served well for all three games.

Figure 11 shows the resulting system context diagram. A special version of the CodeHawk program analyzer designed for this CSFV program creates a set of artifacts describing the source target under analysis. We preprocess these artifacts to populate the databases for the game and game model. Once populated, the game services can deliver game instances for play to the game client and player. The source target never is shown or identified in the databases or client. The game model, situated inside the game itself together in the player's browser, ensures the player makes only legal and evaluates the results. The game returns results to the databases through the backend services. After some time a fixed-point solution is reached and we can dump the databases for post-processing by a second special-purpose CodeHawk analyzer which reads the points-to graph and performs program verification.

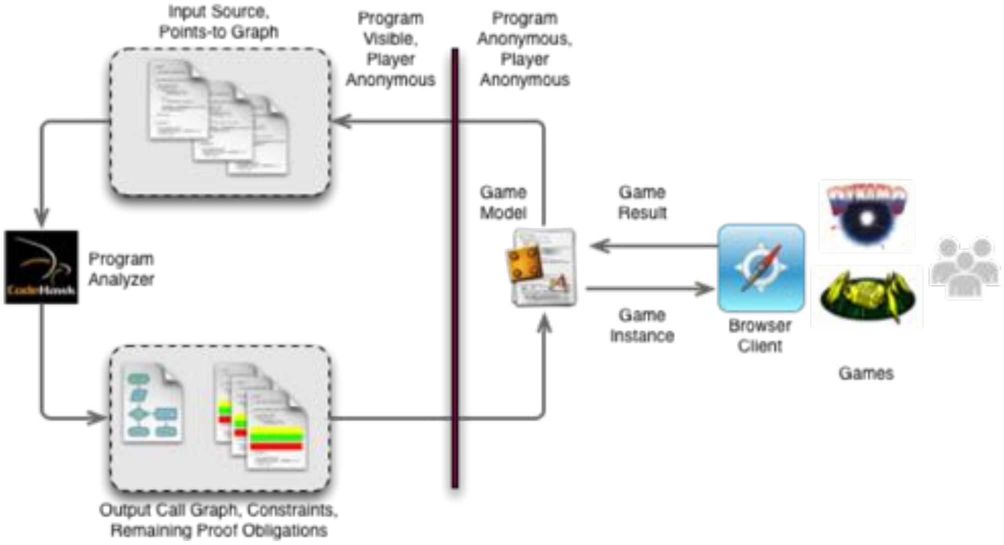


Figure 11: System context diagram

To meet the stateless workflow goal we chose to pursue a service-oriented architecture following the representational state transfer (REST) style. A RESTful architecture is resource-focused and applies six constraints:

- **Client-Server** The client-server constraint separates the concerns of the client and server. In our case this separation implies that servers provide access to data stores and clients are not concerned with these details. Many clients can come or go during an analysis operation, including by cloning or multiplication, and can be written for different platforms than the serves. Servers do not have to keep client state to communicate with each one. Servers can be optimized to a particular backend to meet scalability, security, availability, and other attributes.

- **Stateless** No client context is stored on the server between contexts. Each request from a client contains all of the information the server needs to process the request.

- **Cacheable** Clients can cache responses to improve scalability or performance.

- **Layered** Intermediary servers may improve system scalability by enabling load-balancing and shared caches. The client should not be able to tell what server it is working with along the way to the backend server.

- **Uniform Interface** The uniform interface between clients and servers decouples the architecture so that the two sides can evolve independently. This interface must provide for the identification of resources; in our case we use uniform resource identifiers (URIs) as the global identifier. Our clients and servers must manipulate the resources using these representations, and the URI must be sufficient to modify or delete the resource if it has permission. The interface must use self-descriptive messages. We use JSON for network traffic while the underlying MongoDB database services use BSON for storage representation.

- **Code on Demand** Servers can extend functionality of a client by issuing executable code. This RESTful constraint is optional and we do not apply any code-on-demand services, such as service adapters or applets.

We apply these architectural constraints using the HTTP protocol for client-server communication, including its GET, POST, PUT and DELETE methods.

Providing some more detail for the chosen design, we refer to the reference architecture of Figure 12. Following the step numbers in the diagram, the description follows:

1. (a) Administrator service invokes game instance service to load analysis output. (b) Game instance job reads game variable, call graph, and constraint data from CodeHawk XML files.

2. (a) Administrator service invokes statistics service to load analysis output. (b) Statistics service reads CodeHawk `*.xbo` files and writes statistics to document store.

3. (a) Game pulls game by ID from game instance service. (b) TA3 performer (TopCoder) provides the game ID to play, originally through a *resource allocator* (RA) service (later abandoned for our own priority scheme). The method employed to gather RA game ID candidates is not shown, but the reference content is populated by our game instance service.

4. Administrator service queries game instance service for game progress.

5. Occasionally, or when game play is complete and the fixpoint iteration stabilizes, the administrator service requests the game instance service to export game results as an XML file for CodeHawk as subsequent analysis input.

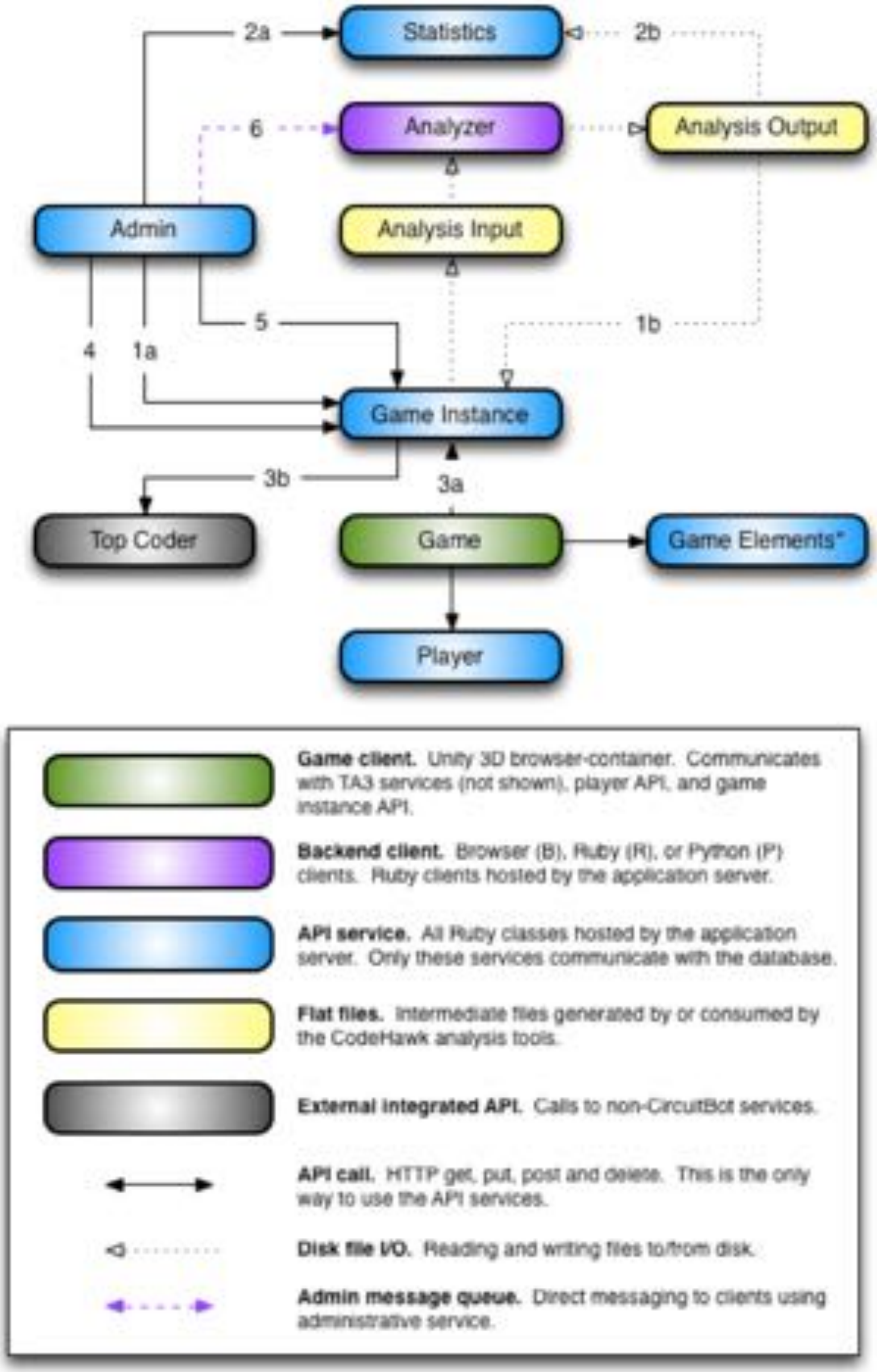
6. The administrator service requests from the analyzer client another round of analysis yielding analysis output files. From this point back to step 2(a) for refreshed statistics.

3.8 Backend Services

Our *backend services* typically run in the cloud to provide data to distributed game clients. Each of our backend services provides access in one way or another to a MongoDB document store collection. We expect our clients to access these stores only through these services, not trying to access the document store directly. A standard application server, named Thin, hosts the run-time backend services as well as a web server for simple user interface tasks. We adopted a URI naming scheme that included the service version number as well as the resource name. Our scheme was `/api/v<version>/<resource>`. Here the version was a major-minor number indicator such as `0.1`, and the resource was a service name such as `statistics`. A complete GET service call might resemble

```
GET http://localhost:8080/api/v1.2/analysis/L8LN44332
```

We incremented the minor version numbers to indicate new features or repairs to existing services. We incremented the major version number when the changes are substantial enough



Game client. Unity 3D browser-container. Communicates with TA3 services (not shown), player API, and game instance API.

Backend client. Browser (B), Ruby (R), or Python (P) clients. Ruby clients hosted by the application server.

API service. All Ruby classes hosted by the application server. Only these services communicate with the database.

Flat files. Intermediate files generated by or consumed by the CodeHawk analysis tools.

External integrated API. Calls to non-CircuitBot services.

API call. HTTP get, put, post and delete. This is the only way to use the API services.

Disk file I/O. Reading and writing files to/from disk.

Admin message queue. Direct messaging to clients using administrative service.

Figure 12: Reference architecture diagram

to break out-of- date clients. Consequently, the services admitted requests with an out-of-date minor version number, but not a mismatched major version number. The service request must never exceed either the major or minor version number. Requests that fail a version check received an HTTP 400 response.

We had twenty-two backend services which ran in the application server. We used the modular style of the Ruby Sinatra framework to provide a lightweight and modular solution for implementing our HTTP-based RESTful API. We hosted the Sinatra applications in a Thin server which also provided an HTTP server single-point client access and route distribution with Rack middleware. For scale-up and load-balancing, when needed, we added an HAProxy layer above several instances of Thin for testing, and routed HTTP traffic to our servers via the TA3 performer's Nginx host for deployment. All of the back-end services including the CodeHawk analyzer ran on the CentOS operating system which we coordinated in discussion with the TA3 performer.

Our backend services mostly corresponded to our document store collections:

Admin Message The administrative messages service handles transactions for the backend clients and the event machine pool. The transactions originate with the administrative web page GUI for control instructions by the administrator. The service dispatches the messages and invokes operations in separate threads. This service has no corresponding document collection or object model.

Administration The admin service provides web pages describing some simple states of the services and document collections along with the some help pages. The admin service provides GET operations for embedded data but does not integrate a document collection.

Award The award service manages access to the player award document collection.

Badge The badge service manages access to the player badge document collection.

Call Graph The call graph services manages access to the function UID connection document collection which serves as our call graph information for the game instance relationship queries.

Dictionary The dictionary service manages access to the dictionary document collection. CodeHawk generates the dictionary when it generates the constraint files that comprise the game instances. The dictionary identifies the variable type and location information in a non-identifiable manner. Subsequent analysis with the game results requires the original dictionary to be provided to CodeHawk.

Factory Definition The factory definition service manages access to the factory definition document collection. The factory definitions describe a factory and its relationship to its planet.

Factory The factory service manages access to the factory document collection. Factories are associated with game missions and instances and describe the status of the factory completion.

Game Instance The game instance service manages access to the game instance document collection. The game instances are the problems to be solved as defined by the CodeHawk constraint generator; there is one unique instance per problem, and these translate into game levels for the player.

Goal The goal service manages access to the goal document collection. Goals are game play concepts that map achievements to rewards.

Graph The graph service manages access to the graph document collection. The graph represents the collection of points-to graph arcs the players produce as a result of game play. We collect all of these arcs and pass them to the CodeHawk analyzer for subsequent analysis.

History The history service manages access to the history document collection. The history records snapshots of verification progress over time.

Mission The mission service manages access to the mission document collection. The missions describe play structures for the player and game engine.

Planet The planet service manages access to the planet document collection. The planets describe locations for missions.

Player History The player history service manages access to the player history document collection.

Player The player service manages access to the player document collection. We use the player records to record progress and rewards, but no personally identifiable information; the player IDs come from the TA3 enrollment services.

Resource Definition The resource definition service manages access to the resource definition document collection. Resource definitions describe characteristics of resources as employed during missions.

Resource The resource service manages access to the resource document collection. Resources are generated and consumed by player decisions during missions.

Statistics The statistics service manages access to the statistics document store. The statistics are verification results for each function in the program under analysis.

UID The UID service manages access to the UID data which provides a mapping of game instances to UID appearances.

Worker History The worker history service manages access to the worker history document collection.

Worker The worker services manages access to the worker document collection. We use the worker records to record progress and assignments, but no personally identifiable information; the worker IDs come from the TA3 enrollment services.

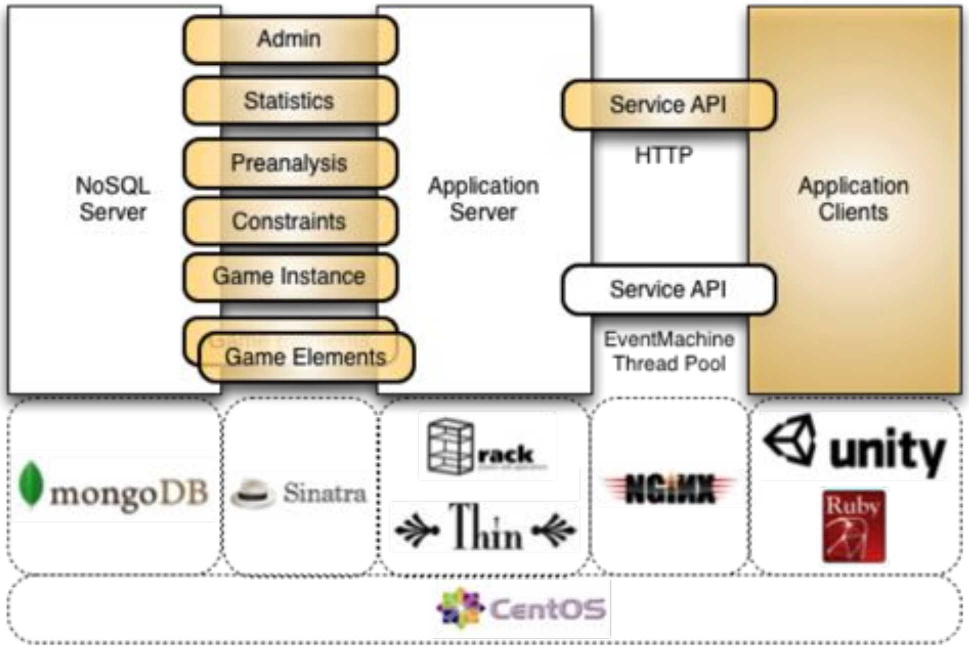


Figure 13: Backend service implementation diagram. Developed elements are shown in orange; non-developed elements are shown in white.

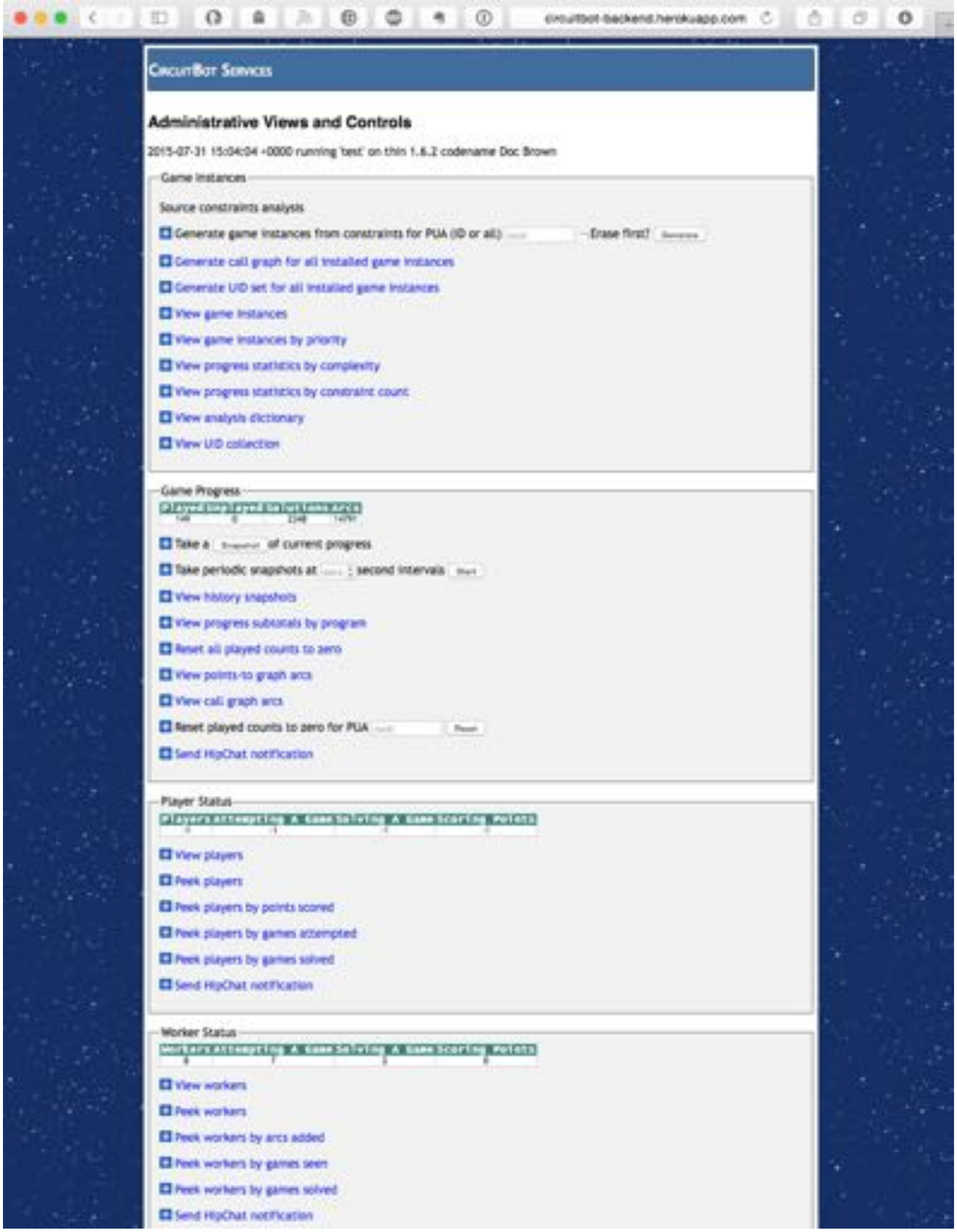
Appendix B provides the details of our document collection designs.

Figure 13 shows a block architecture of the backend. The services described above connect the application server with the database (NoSQL, MongoDB document store) server. Our REST service API connects the application clients with the application server. Our specific implementation of the application server used off-the-shelf Rack and Thin products. We wrote the application services in Ruby according to Sinatra library frameworks, connecting the data models to MongoDB using the `mongomodel` gem. We used both HAProxy and Nginx for load management during testing and deployment. Our clients used Unity and C# for the game, PHP for web services, and a mix of OCaml, Ruby, and C# for analysis. The CodeHawk verification engine and special-purposes analysis tools were written in OCaml. Nearly all of our file formats were XML.

For development and testing we used Amazon Web Services (AWS) elastic compute cloud (EC2) virtual machines. We created Amazon machine instances containing our working configurations. For deployment we shared these with the TA3 performer for additional configuration in specific virtual private cloud (VPC) setups.

The backend provided a service for administrator control and status information through a browser client. Figure 14 shows the landing page for the administrator client. After authenticating with the administrator user ID and password, the administrator could inspect and sort most of the document stores in legible formats, collect progress statistics, and recall some status and configuration information.

During the two years of game deployment with three games we experienced *zero* downtime associated with the backend services or document stores.



CircuitBot Services

Administrative Views and Controls

2015-07-31 15:04:04 +0000 running 'test' on thin 1.6.2 codename Doc Brown

Game Instances

Source constraints analysis

- Generate game instances from constraints for PUA (ID or all) ____ —Erase first? Generate
- Generate call graph for all installed game instances
- Generate UID set for all installed game instances
- View game instances
- View game instances by priority
- View progress statistics by complexity
- View progress statistics by constraint count
- View analysis dictionary
- View UID collection

Game Progress

Played	Unplayed	Solutions	Arcs
146	0	2248	14797

- Take a Snapshot of current progress
- Take periodic snapshots at ____ second intervals Start
- View history snapshots
- View progress subtotals by program
- Reset all played counts to zero
- View points-to graph arcs
- View call graph arcs
- Reset played counts to zero for PUA ____ Reset
- Send HipChat notification

Player Status

Players	Attempting A Game	Solving A Game	Scoring Points
0	1	1	0

- View players
- Peek players
- Peek players by points scored
- Peek players by games attempted
- Peek players by games solved
- Send HipChat notification

Worker Status

Workers	Attempting A Game	Solving A Game	Scoring Points
8	7	0	0

- View workers
- Peek workers
- Peek workers by arcs added
- Peek workers by games seen
- Peek workers by games solved
- Send HipChat notification

Figure 14: Backend services administration client.

3.9 Web Services

We used a web-based middleware layer to serve as a broker between the game, the backend, and the scoring and registration services. We designed and implemented this layer with fairly typical PHP (personal home page) scripts. The PHP scripts primarily served as a filter translating game constructs into backend service API calls. The game, developed in Unity, ran in the Unity plugin inside the player's browser. The game file and the web page that hosted it were on our game server. Our PHP scripts were also on this server. The game communicated with our PHP scripts via HTTP requests, and the PHP scripts likewise communicated with the backend via HTTP requests.

Our PHP layer design and implementation offered substantial flexibility in developing and testing the game. We could change the `host` and `port` settings used by the PHP scripts to direct them to any backend machine. Switching from a *staging* to a *production* backend was a matter of changing a PHP configuration file. Because the browser-based game only communicated with the game server, it did not contain the host and port location of our backend services, thus offering a layer of security as the game players could not find these locations by embedded in the game. Moreover, the PHP layer acted as a bridge between the game and our databases, which allowed us to filter, compress, and otherwise organize data transfers with the backend. When TA3-provided services were later added for player registration and for the scoring and awards API, we added these calls to our PHP layer, not to our backend.

3.10 CircuitBot Game

In our CSFV proposal we employed the legacy *Pipe Jam* game as a springboard for our CircuitBot (Figure 15 on page 109) concept, and envisioned our game unfolding as follows:

> *Like Pipe Jam, a CircuitBot game level is a directed graph, with certain edges labeled (by the player) with properties a cargo must have in order to traverse them. In the case of Pipe Jam the allowed properties are linearly ordered, corresponding to the possible widths of balls and pipes, in such a way that if a cargo has property p then it also has property q for any $q > p$ (that is, a ball will fit through a pipe its size, or through any larger pipe). From a mathematical standpoint, CircuitBot enriches the Pipe Jam model in two ways. First, the properties of edges and cargoes need not be linearly ordered. Second, cargoes may mutate when they pass through a node, either singly or in combination with other cargoes, to form new cargoes with new properties. Each game level represents a correctness problem for a particular line of code with respect to a particular error that may occur in that line according to a CodeHawk warning. The abstract generated graph corresponds to relevant aspects of data flow with nodes playing the role of function calls, arcs playing the role of variables, and labels playing the role of properties or types that a variable may have.*
>
> *From a gameplay standpoint, a game level is a room containing several small tables, with one or more portholes in the walls, and with directed paths drawn on the floor from certain tables and/or portholes to certain others. Portholes are marked visually as to whether cargo passes into the room or out through them.*

> *Each table or porthole is marked with properties a cargo must have in order to occupy it, and with how it can transform or combine cargo that moves across it. For example, a table may have a funnel that will turn a large ball into a small one or vice versa, a paint brush that turns any colored ball into a green one, a red laser that zaps any ball that is not red, etc. A table may also have a pipe network that accepts two or more cargoes and creates a new one whose properties depend on the inputs. For example a cargo of a certain color and another cargo of a certain size may be required to form a new cargo with some specified property. The outgoing portholes are similarly marked for what sorts of cargo may safely exit through them.*
>
> *The game is played by selecting one or more types of robots for each input porthole. Generally speaking, robot types are defined by the kinds of cargo they may carry, with the simplest case being all cargoes of a given size or smaller; however more intricate combinations are possible, such as all cargoes of a given color and of a given size or smaller, or all rectangular cargoes of a given width or smaller and a given length or smaller. Once the player selects robots, he or she hits* **go** *and the factory begins to operate in a frenzy. Each robot morphs into a swarm of identical robots, as cargoes of various sorts pour out of the incoming port holes. If a robot attempts to carry a cargo it cannot handle, that cargo drops onto the floor. The robots carry the ones they can handle to every destination they can reach, where the cargoes are transmuted and passed to other robots (who may or may not be able to carry them) and the frenzy continues until all the possible deliveries have been made. Each delivery of a cargo through an outgoing porthole scores points for the player; but attempting to deliver a disallowed cargo causes the room to explode. Thus the player is engaging in an optimization problem trying to deliver all and only acceptable cargoes, leaving the "poisonous" ones lying on the floor.*
>
> *Robot types actually correspond to restrictions on variables in a function, and higher scores in the game give more permissive preconditions thus allowing a larger input space among the safe inputs. The absence of explosions corresponds provably to the absence of errors. The attributes a cargo or robot may have, along with the visual representations of these attributes and their allowed transmutations and interactions, will potentially be unique to each error type, though with a substantial degree of reuse.*

As explained in the Introduction, within the first few months of the project our approach to the problem changed from *assertions and counter-examples* to *constructing a points-to graph for the program under analysis.* The reason for this change was that we concluded our static analyzer already generated automatically the vast majority of assertions and counter-examples needed for the memory safety properties, so game play was unlikely to add value. What was lacking, in particular for the BIND program, was information about the use of pointers. We therefore changed our approach to generate facts about the use of pointers and memory locations within the program. The new description of the abstract game model envisioned a universe of connected rooms, with each room containing one or more crates. There were one-dimensional paths or *bridges* between crates and between rooms. A set of rules described when new bridges could be created. The player made a move by selecting a bridge. The existing bridge and its endpoints were examined to determine which bridge-building rules applied and then added new bridges if possible. When bridges were added, any extant bridges which shared an endpoint with a new bridge would be reexamined to determine if that bridge could generate new paths. The game was over when all bridges were examined and no new bridges could be built.

From a game-making perspective, the work of adding connections to a points-to graph appeared too much like *work* so we were concerned that Internet gamers would be turned away by a game that involved serious work. To remedy the situation we decided to build a game-within-a-game, with a fun *outer game* surrounding a more work-like *inner game*. From the player's perspective, playing the inner game would be necessary to unlock game elements, or to provide resources to continue the outer game.

Maintaining the robots theme we envisioned the game taking place in the near future. CircuitBot became a resource management game, where players carry out missions to destinations in space, guiding robots to build facilities which prepare the way for human colonization. The player needed to program the robots to build the needed factories, research facilities, etc. This programming task is in fact the work that produces the arcs for the desired points-to graph.

From a game theme perspective, the play description would appear as follows:

> It costs $10,000+ per kilo to lift any payload from the Earth to a stable Earth orbit. This cost applies to people, equipment, and fuel.
>
> A near Earth asteroid is not a rare thing. Many asteroids travel in a zone near the Earth's orbit. Many are small rocks between the orbits of Venus and Mars which make a orbit of the sun at a rate that places them nearest to Earth with predictable frequency. There are many types of asteroids depending on their origin. Their composition is similar to the elements of the inner solar system. Some have high levels of metals, organic elements, and water. Detecting the composition is critical to determining the mission value.
>
> A Near Earth asteroid which contains an ice core can contain 100s of metric tons of water in the form of ice surrounded by a surface of other materials which prevent solar radiation from sublimating the ice core. A robotic expedition can travel to one of these asteroids, in an orbit that places the group within a close range to Earth every one to two years.
>
> Extraction of water from such a core require a system which can drill below the surface, melt the H_2O, and collect the water. Combining the collected water with a electrolysis system can separate the water into O_2 and H_2 and store them separately. This is the basis of fuel production. In space maintaining the temperatures required to efficiently store these resources is simplified. Recombining hydrogen to create more stable fuel sources is also simplified, but potentially not required. Oxygen is valuable as a reaction agent or for human life support.
>
> During the time that a robotic team is out of Earth's ideal range, the mission can retrieve metric tons of water, and convert this into fuel using solar energy for delivery to Earth orbit. This translates into hundreds of millions worth of fuel every approximately 1.5 years. This fuel is transported by a dedicated trans-

portation system, leaving the robotic team free to focus on production and to relocation when a fuel source is depleted.

The cost of a robot team capable of setting up a fuel production station would be high initially, but the same set of robots can be repositioned to set up additional stations to multiply the effort. Once the technology has been refined, cost for additional missions is reduced. Efficiency is increased over time. This effort simplifies space travel considerably. Earth can lift humans and mechanical resources to low Earth orbit for reduced payload cost, and be assisted to high orbit using fuel provided at a much lower cost by these sources.

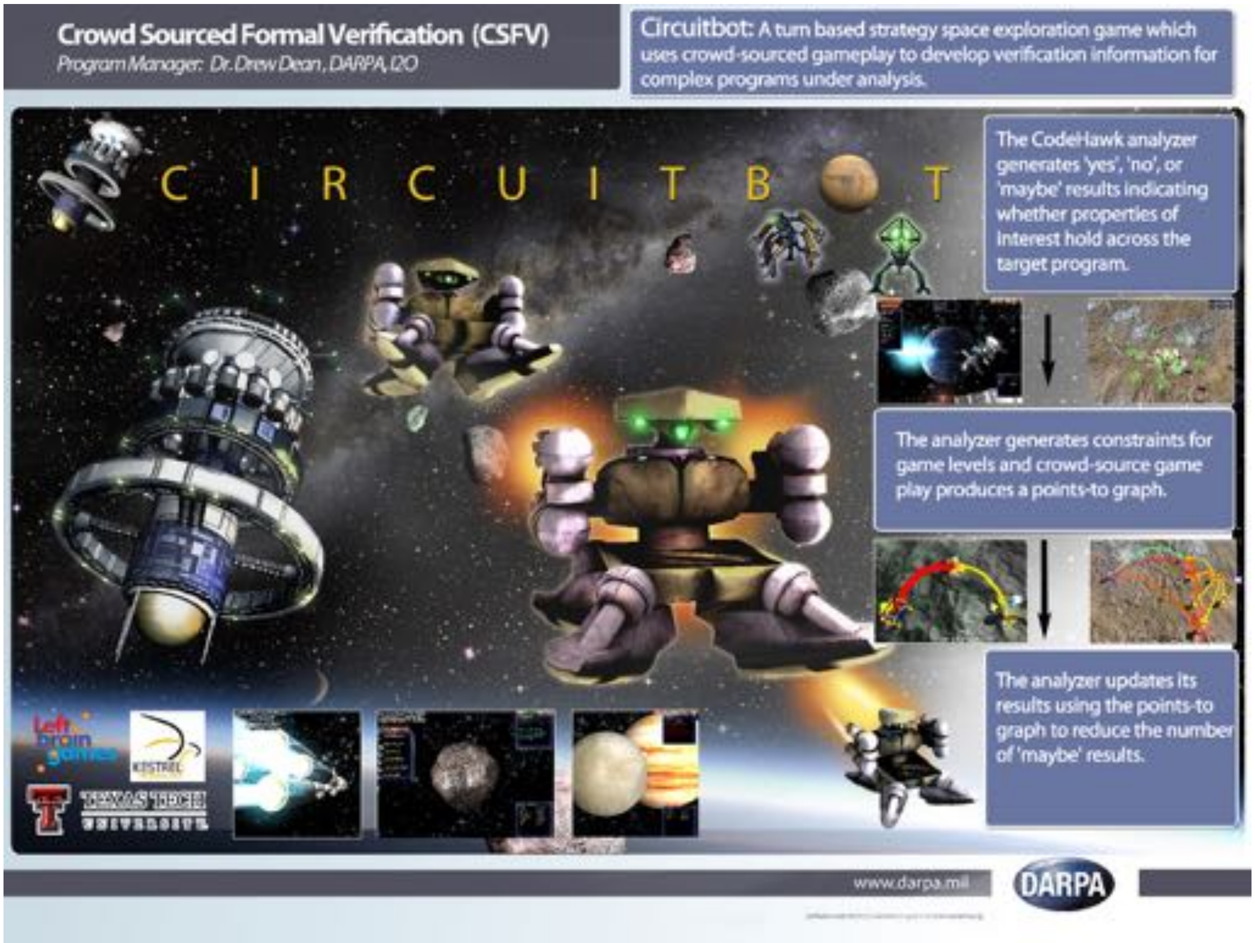
Figure 15: CircuitBot logo banner.

Figure 16: CircuitBot promotional banner for meeting booths.

3.10.1 Design Goals

Our design goals for the game were to

- Provide an entertaining game structure disguising the verification work



Crowd Sourced Formal Verification (CSFV)

Program Manager: Dr. Drew Dean, DARPA, I2O

Circuitbot: A turn based strategy space exploration game which uses crowd-sourced gameplay to develop verification information for complex programs under analysis.

The CodeHawk analyzer generates 'yes', 'no', or 'maybe' results indicating whether properties of interest hold across the target program.

The analyzer generates constraints for game levels and crowd-source game play produces a points-to graph.

The analyzer updates its results using the points-to graph to reduce the number of 'maybe' results.

Figure 17: CircuitBot poster produced for I2O Demo Day.

- Provide the player with short, medium and long term goals, which provides motivation for the player to return.
- Create an environment where the player is always planning or maintaining his settlements.
- Allow for unexpected opportunities and disasters to keep players engaged.
- Create a casual game in a science fiction setting, incorporate hard science themes using real science and technology as a foundation for the descriptive elements in the game.

Each building (factories, mines, research facilities, etc) represented a game level (game instance) and embodied the verification work we needed the player to do. The robots represented variables and addresses, abstracted from source code, and had relationships with other robots based on constraints also abstracted from the source code. These relationships were represented by arrows pointing from one robot to other robots. The player was provided with a tool panel for applying constraints which triggered new connection arrows to appear between robots when new arcs were added to the points-to graph. When the player applied constraints until all were satisfied, the level was completed. The connection information was reported back to the server, and the player was able to continue managing his settlements.

The world of CircuitBot relied on a turn-based time system. On a given mission, the player selected which factories or buildings to construct. After completing the robot programming task, there was an imagined period of time while the building was under construction and not yet capable of producing resources. The more advanced a building, the longer this build time. The game player made decisions while the clock was frozen. When he was ready, he pressed the **end turn** button to allow time to pass (10 in-game days). This allowed the player to advance time, react to the state of the universe, and then advance time again. The in-game time had no relationship to the player's real life time.

Missions The CircuitBot player began on missions to near-Earth asteroids and after completing these missions unlocked more complex missions farther away (Figure 18 on the following page). Each mission had about five goals to produce amounts of specific resources. Producing the resources required building some number of factories. Mission locations included:

- Near Earth Asteroid
- Carbonaceous Chondrite
- Large Belt Asteroid
- Ceres
- The Moon
- Mars
- Ganymede
- Europa

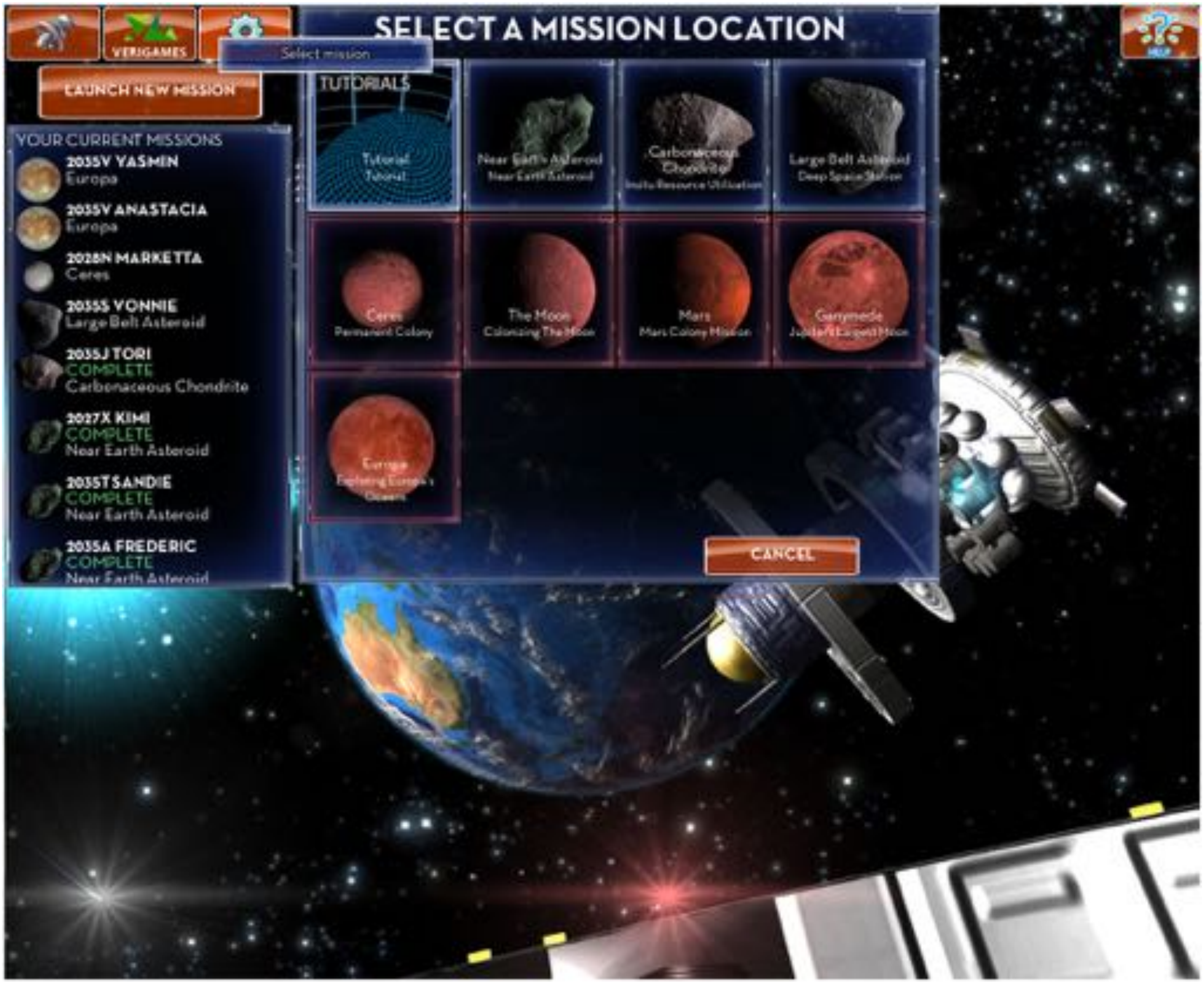
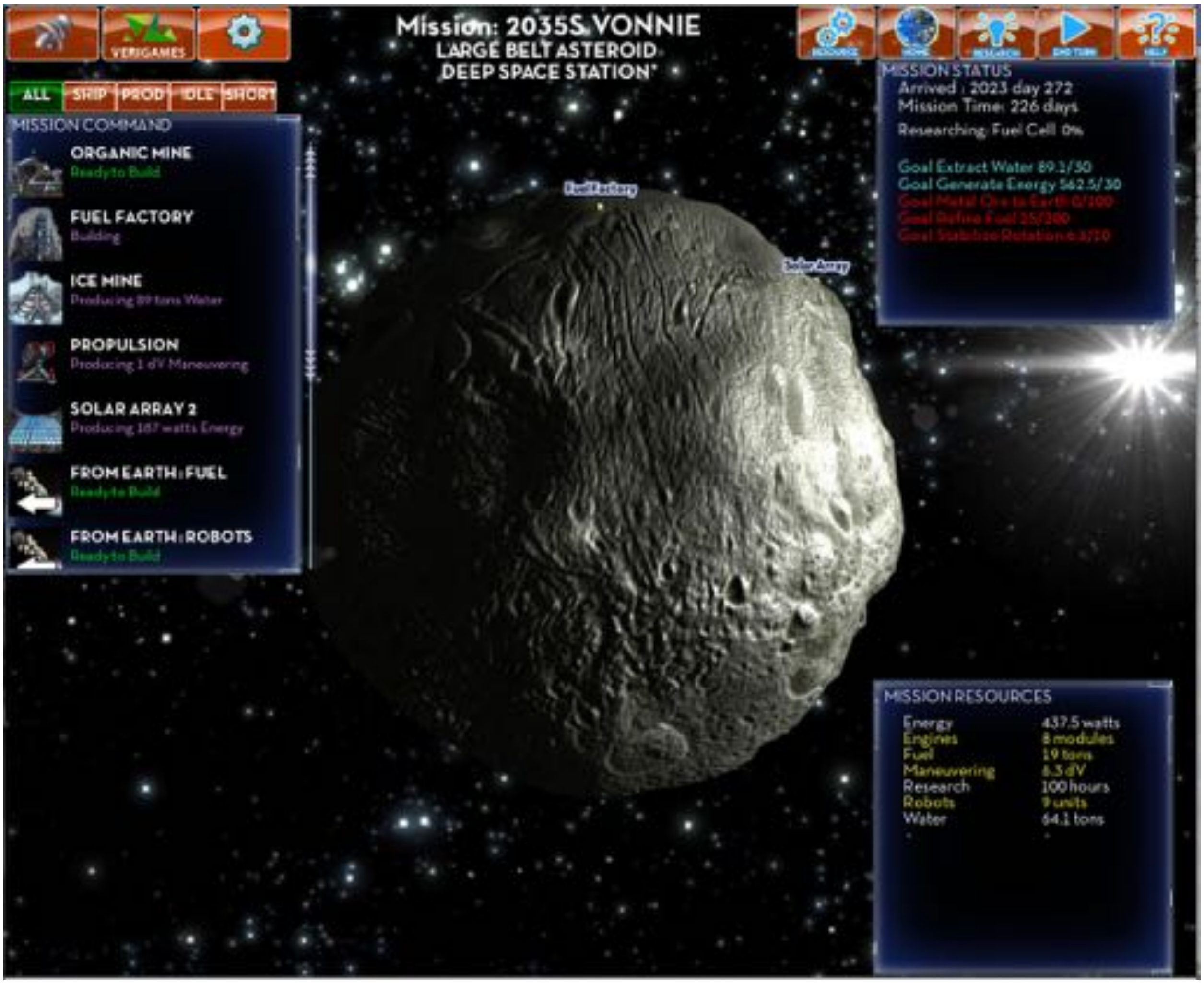


Figure 18: CircuitBot *Mission Selection* screen. Some missions are not available until the player has collected prerequisite resources during previous missions.

Factories Although we used the word *factory* to describe the things built at the mission location, to the player these were various types of buildings (Figure 19 on page 115). Programmatically, the game treated them all like generic factories in the sense that each requires an input resource and produces an output resource. This makes some factories dependent on the resources produced by other factories. We used this dependency structure to provide the player with mission goals (for example, *build a reactor*, which required that a sequence of other factories first be built to provide the raw materials needed to build the reactor). For any given mission, we offered the player a choice of factories he could build. In the factory interface panel, factories that required resources that did not exist did not include a **build** button. Resource requirements about each factory were provided so the player could determine a build sequence (Figure 20 on page 116). The interface display for factories that were built included production information as well as status indicators that alerted the player to resource shortages. Factories included:

- Biodome
- Biodome Upgrade
- Nano
- Nano Upgrade
- Centrifuge
- Centrifuge Upgrade
- From-Earth Fuel
- From-Earth Robots
- Storage
- Storage Upgrade
- Refinery
- Refinery Upgrade
- Refinery Upgrade 2
- Greenhouse
- Greenhouse Upgrade
- Heavy
- Heavy Upgrade
- Hospital
- Hospital Upgrade

- Housing
- Housing Upgrade
- Ice
- Ice Upgrade
- Leisure
- Leisure Upgrade
- Manufacturing
- Manufacturing Upgrade
- MetalMine
- MetalMine Upgrade
- Organic Mine
- Organic Mine Upgrade
- Linear Accelerator
- Linear Accelerator Upgrade
- Radiation Shielding
- Radiation Shielding Upgrade
- Reactor
- Reactor Upgrade
- Research
- Robot
- Robot Upgrade
- Solar
- Solar Upgrade
- Starship
- Starship Upgrade
- Telescope
- Telescope Upgrade



Mission: 2035S VONNIE
LARGE BELT ASTEROID
DEEP SPACE STATION*

MISSION COMMAND

ORGANIC MINE
Readyto Build

FUEL FACTORY
Building

ICE MINE
Producing 89 tons Water

PROPULSION
Producing 1 dV Maneuvering

SOLAR ARRAY 2
Producing 187 watts Energy

FROM EARTH : FUEL
Readyto Build

FROM EARTH : ROBOTS
Readyto Build

MISSION STATUS
Arrived : 2023 day 272
Mission Time: 226 days
Researching: Fuel Cell 0%

Goal Extract Water 89.1/30
Goal Generate Energy 562.5/30
Goal Metal Ore to Earth 0/100
Goal Refine Fuel 35/200
Goal Stabilize Rotation 6.3/10

MISSION RESOURCES

Energy	437.5 watts
Engines	8 modules
Fuel	19 tons
Maneuvering	6.3 dV
Research	100 hours
Robots	9 units
Water	64.1 tons

Figure 19: View of the CircuitBot *Mission Status* screen showing factories built or available for building, mission goals, and resources available for the mission. The player selects items from the *Mission Command* list to access project details.

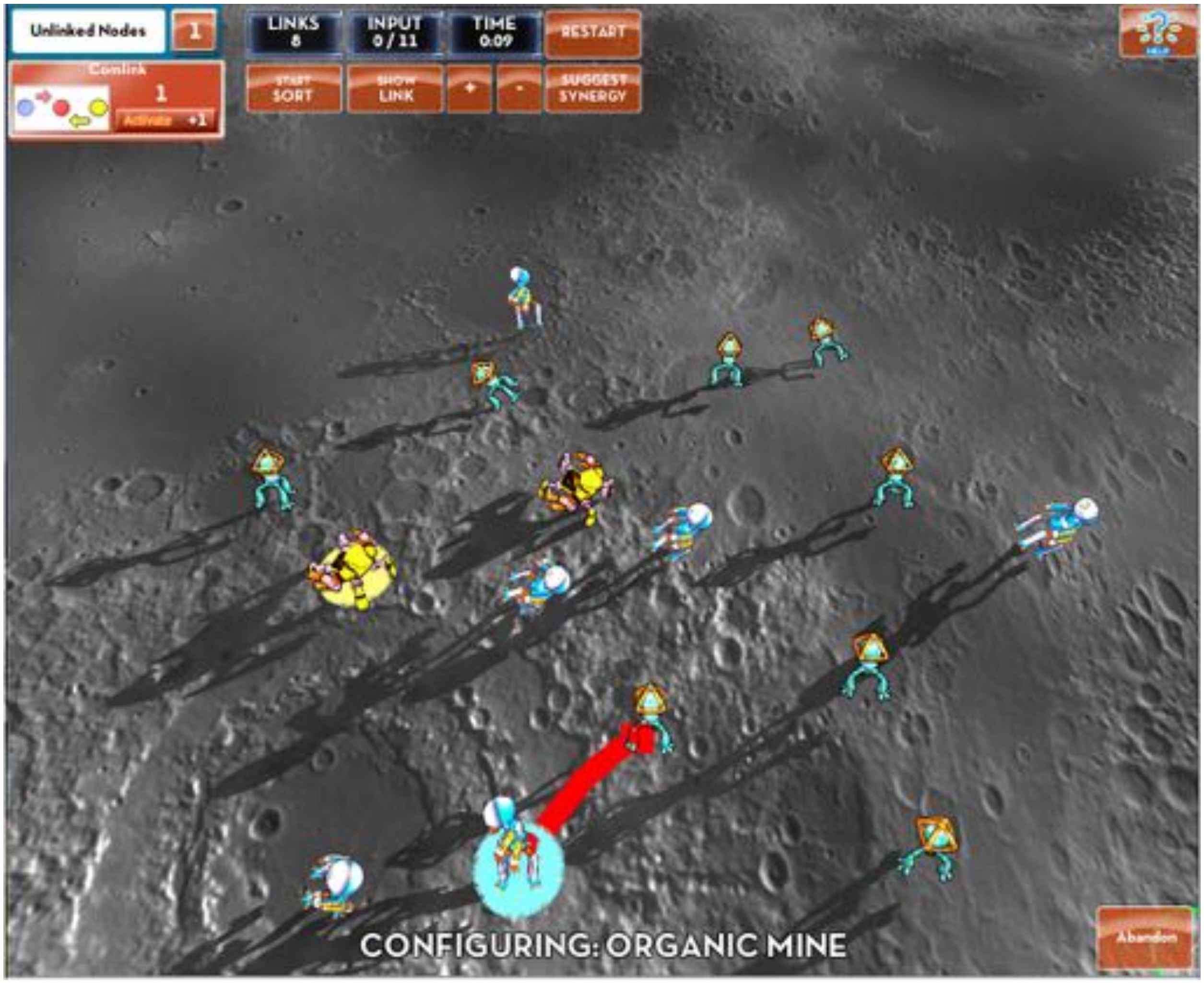


Figure 20: View of a CircuitBot landing site where the player will complete a factory or other project. Robots represent nodes in the underlying points to graph. Links drawn between robots represent arcs in the underlying graph.

Resources Factories in CircuitBot consumed and produced resources. The player managed resources for each mission, choosing when to sell, buy or ship resources to and from Earth. Missions could require that certain resources exist on Earth before the player could launch that mission. This type of resource requirement enabled us to require the player to complete one mission, before unlocking another mission. Awards and mission goals were tied to producing a minimal quantity of a particular resource. Resources included:

- Asteroid Protection
- Bring Comets to planet
- Colony
- Launch
- Modifies Atmosphere
- Modifies Planet Orbit
- Orbital Insertion
- Radiation Protection
- Research
- Tourism
- Maneuvering
- Energy
- Water
- Fuel
- Organic Ore
- Metal Ore
- Nanotubes
- Components
- Robots
- Heavy Metals
- Plant Life
- Animal Life
- Colonists Goal

- Credits
- Engines
- Transport
- Reactor
- Fissionables
- Food
- Exotic Particles
- HRS Robots
- Repair Robots
- Ship Facilities
- Starship

Research In CircuitBot, after the player built a *research facility*, he could select a research topic from the bottom layer of a research dependency tree. Research required elapsed time, with higher level research topics taking more time than lower level topics. When the research on a particular topic was complete, that topic was considered to be known. Internally, the game created a hidden resource related to this topic. Like any other resource, factories could require this research topic to be completed (known) before the factory could be built. This provided another information thread for the player to manage as he planned how to complete missions. Research topics included:

- Organic Mine
- Greenhouses
- Solar Array 2
- Metal Mine
- Human Habitation
- Fuel Cell
- Fuel Factory
- Bucky factory
- Heavy Metal Mine
- Manufacturing

- Bio Dome
- Centrifuge
- Telescope
- HRS Factory
- Radiation Shielding
- Spaceport
- Luxury Complex
- Near Earth Asteroid Mapping
- Reactor
- Starship
- Quantum Accelerator
- Hospital
- Main Asteroid Belt Mapping
- Improved Fuel Factory
- Martian Explorer
- Sub Kuiper Mapping
- Oort Cloud Mapping

Market The CircuitBot market offered a way for the player to convert a given resource into money (credits) which he could use to buy resources which he had not produced. Not all resources were available for purchase, since this might allow a player to shortcut the mission dependency setup of the game. Resources harvested on a mission must be sent back to Earth in order to appear in the market. Shipping resources from a mission location required elapsed time. This time-delay added to the realism of the game and promoted the idea that missions are taking place on far away locations. Future plans for the market, in a subsequent version of CircuitBot, were to add a microeconomics model that would allow the prices for resources to fluctuate based on supply and demand.

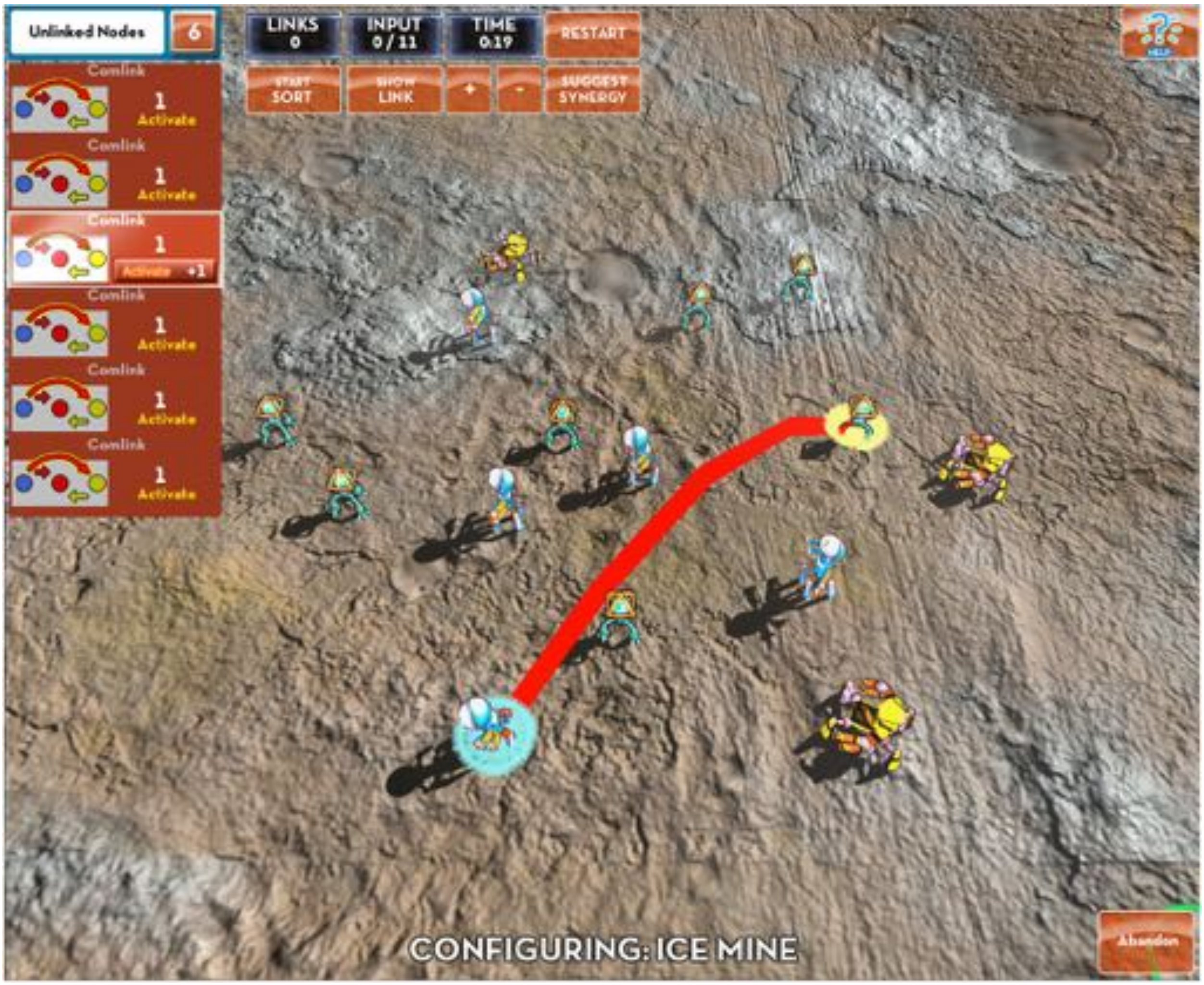
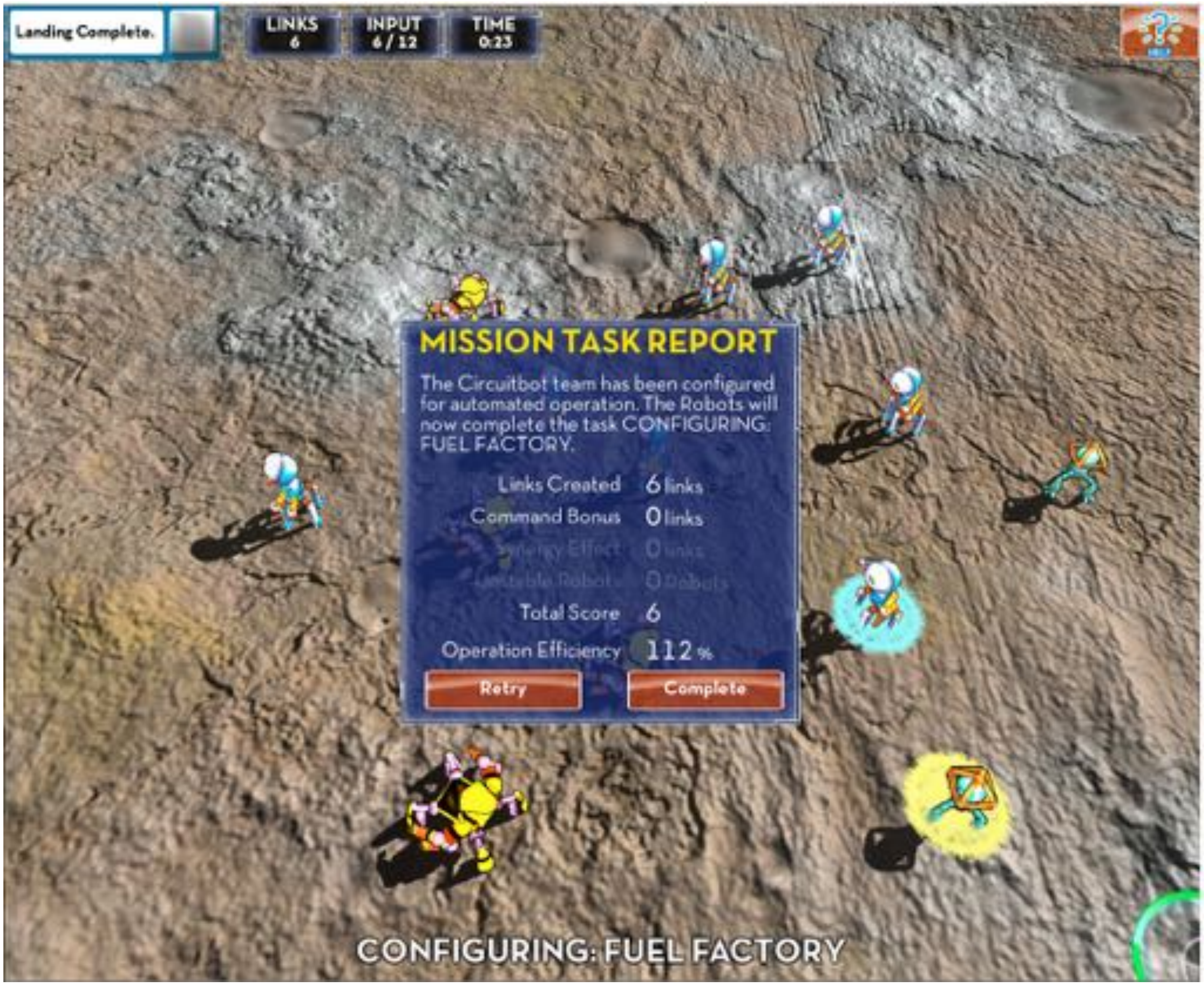


Figure 21: CircuitBot player command panel. Along the left-hand side of the screen are command buttons which represent game model constraints which would provide new graph arcs when activated. Constraints that have already provided information or would otherwise not add information are hidden.



Landing Complete.

LINKS 6

INPUT 6 / 12

TIME 0:23

MISSION TASK REPORT

The Circuitbot team has been configured for automated operation. The Robots will now complete the task CONFIGURING: FUEL FACTORY.

Links Created	6 links
Command Bonus	0 links
Synergy Effect	0 links
Unstable Robots	0 Robots
Total Score	6
Operation Efficiency	112 %

CONFIGURING: FUEL FACTORY

Figure 22: Final report for an individual CircuitBot game mission. The player's score is reflected in the operational efficiency of his completed project in ongoing game play.

Operations In order for a player to build a factory, which was any sort of building like solar panels or ice mines, or to prepare a resource to ship back to Earth orbit, the player needed to prepare the robots to do the work. The robots needed to be *programmed.*

Each robot represented a single node, variable, memory location, or parameters passed between functions, in the domain of a program under analysis. Along the side of the screen are the sets of rules which the player fires to create links between robots (Figure 21 on the previous page).

Conceptually, robots were autonomous workers which have been given a procedure. Each robot was responsible for completing distinct steps in the procedure. The player applies the available rules which designate that one robot depends on one or more other robots, or that one of more robots depends on them. Imagine an *ad hoc* assembly line, where robots complete a smaller task and then the next robots continue to follow through on a task, from creating a group of structures and actuators and combining them into larger components and then into the final construction (Figure 22). That is the concept of the robotic team.

In applying these rules, the player was in fact activating pointer flow constraints inside game instances, which produced new points-to graph arcs and modified the graph nodes. All of this information was stored in the backend document stores to inform related game instances and for subsequent verification analysis.

3.10.2 Resource Allocation Integration

We performed early integration testing with the TA3-provided Resource Allocator (RA). The main purpose of the RA was to assign game levels to players according to skills and experience. In theory, this would maximize throughput of all the games by giving players levels that they could play and complete, rather than players encountering levels that were either too easy (boring) or too difficult (frustrating). In practice, for the system to work, each TA1 team needed to establish a set of criteria and metrics to assess game level difficulty as well as metrics to measure how well a player did on a given level. Reaching a workable universal set of criteria proved difficult because TA1 game design and play approaches differed substantially.

Our backend services and API were available very early in the project. The flow of game levels between backend and game was handled by PHP scripts running in our game server. Our game, running in the browser, made requests to these scripts, which in turn accessed the backend services. The implementation of the RA was occurring in parallel, and by the time it was available for requesting game levels we needed to redesign the PHP code to integrate with it. This redesign and testing was complicated because in order to continue development we needed a reliable source of game levels. The server which hosted the RA was out of our control, so any outage or bugs would keep our game from working. To mitigate this risk, we added logic to the PHP scripts to either use the RA or use our backend. To further complicate the situation, some of the TA3-provided scoring system APIs, under development in parallel, required that game level identifiers co-exist in the scoring database and RA database. Eventually the RA was discontinued but it was a simple matter for our team to reset the configuration of our backend to substitute its own service equivalents.

3.10.3 Launch

We developed eleven missions of content although we deployed only eight missions during the CircuitBot game phase. The non-deployed missions consisted of additional factory types required to complete each mission representing new technologies available. In total, a player would need to complete 185 factories in order to complete all available missions. This represented developing points-to graph data for the same number of game instances.

CircuitBot was developed on best guesses of the data which had not been generated from CodeHawk until just before launch. Analysis of data collected from CircuitBot gameplay lead to significant improvements to the quality of the entire process. After launch, we modified the game according to quality of game instances being delivered in order to run an auto-solver. This ran in the background to detect challenging game instances, and to solve them partially before presenting them to the player. Multiple loaded instances would be combined into a single game experience greatly increasing the rate at which players could contribute productive results.

We had several ways to track player progress. Because players played the missions sequentially, and because significant milestones indicated accomplishments, we could detect which players were returning, how far they were proceeding, the points where they more likely to leave, and so on. We collected logs for every player event during game play as well as the quality of player contributions. We accumulated metrics for total work generated and time required, and computed allocations of work time to play time activity. We also used this data to award player badges for the in-game milestones such as *First Mission Launch Completed* and *Moon Mission Completed*.

3.10.4 Lessons Learned

This section captures some of the lessons learned, some of which we incorporated into our subsequent games, and some of which we leave for future programs.

1. **It was a challenge to create a *simple* game that included the verification work.** Creating tutorials was difficult. The concepts were not easy to convey in an engaging form without becoming long winded. Feedback from testing was contradictory at times. Although players wanted more information about how to play, they also complained that tutorials were long, boring, and so on. Many players do not like being forced to play tutorials in any game. The combination of an outer, exploration/strategy game with an inner puzzle-like/verification related game was effective at motivating certain types of players. Players could deal with much larger sets of rules if trivial content were filtered away. Figures 23 on the next page, 24 on page 125, 25 on page 126, 26 on page 126 and 27 on page 127 present examples of our tutorials.

2. **A variety of test data is important during development.** Due to upgrades to the CodeHawk analyzer, the source of data for the game, the nature of the data changed during the course of development. We were forced to make assumptions about how frequently certain types of data would appear, as well as speculated on which properties in the data would be most valuable for ordering the game levels. Some of our assumptions did not match up well. We built-in functionality for data attributes that we believed would be important or frequent situations, but turned out to be neither. There was a belief that players could encounter game levels where satisfying some constraints causes others to become unsatisfied, and satisfying the second group causing the first group to become unsatisfied, and repeating endlessly. We never encountered this situation during gameplay. We assumed that players wouldn't be able to cope with large quantities of nodes and arcs, so they would need a way to reduce this information. To address this we developed the process of *synergy* in which the player could treat a group of nodes and the arcs that join them as a single entity. Although we built this functionality into the game, in the end the player did not need it. In hindsight, the time and resources that went into adding an interface to the game for synergy, as well as creating tutorials to explain it was unnecessary.

3. **Anonymous players turned out to be important.** Accommodating anonymous players in our first game was a bit of an afterthought. We had assumed the number of anonymous players would be small compared to registered players, and therefore

Figure 23: The CircuitBot game's Mr. Scott character appears with messages for the player during early game play tutorial systems.

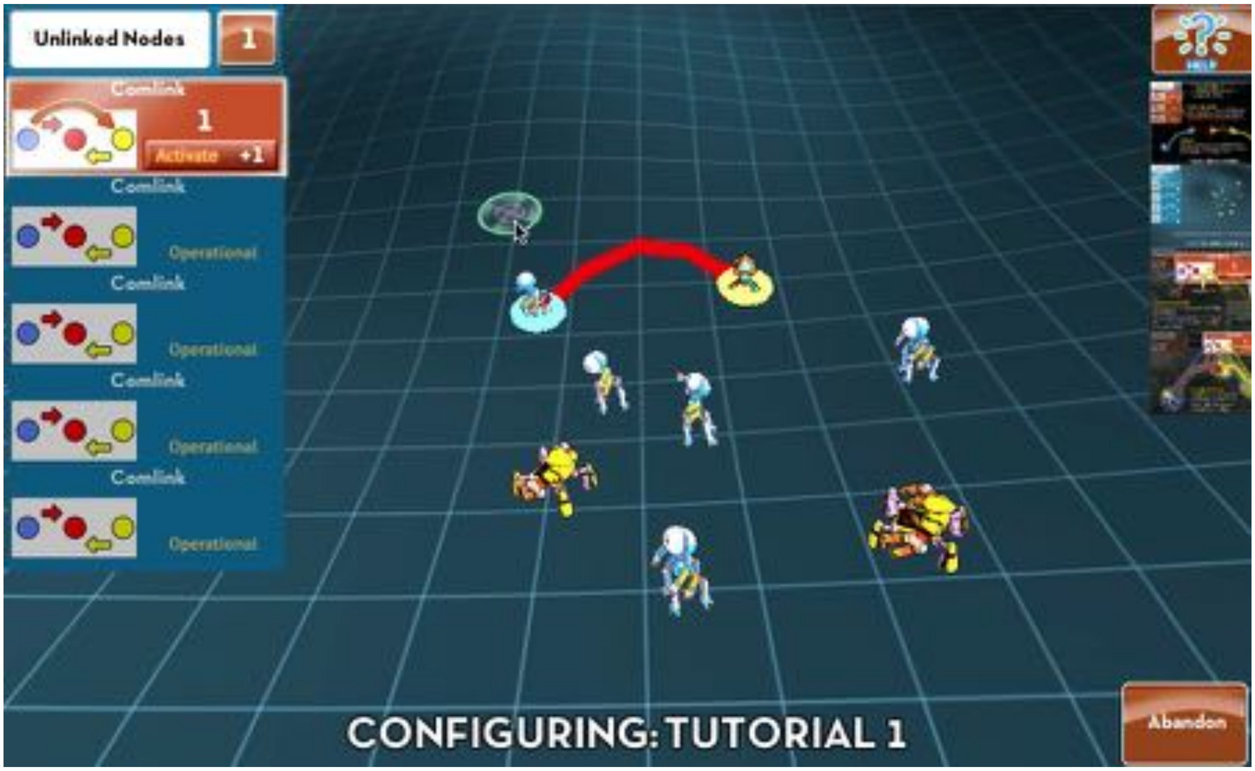


Figure 24: CircuitBot tutorial screen shot, presenting concepts for commands and producing simple *alpha* connections.

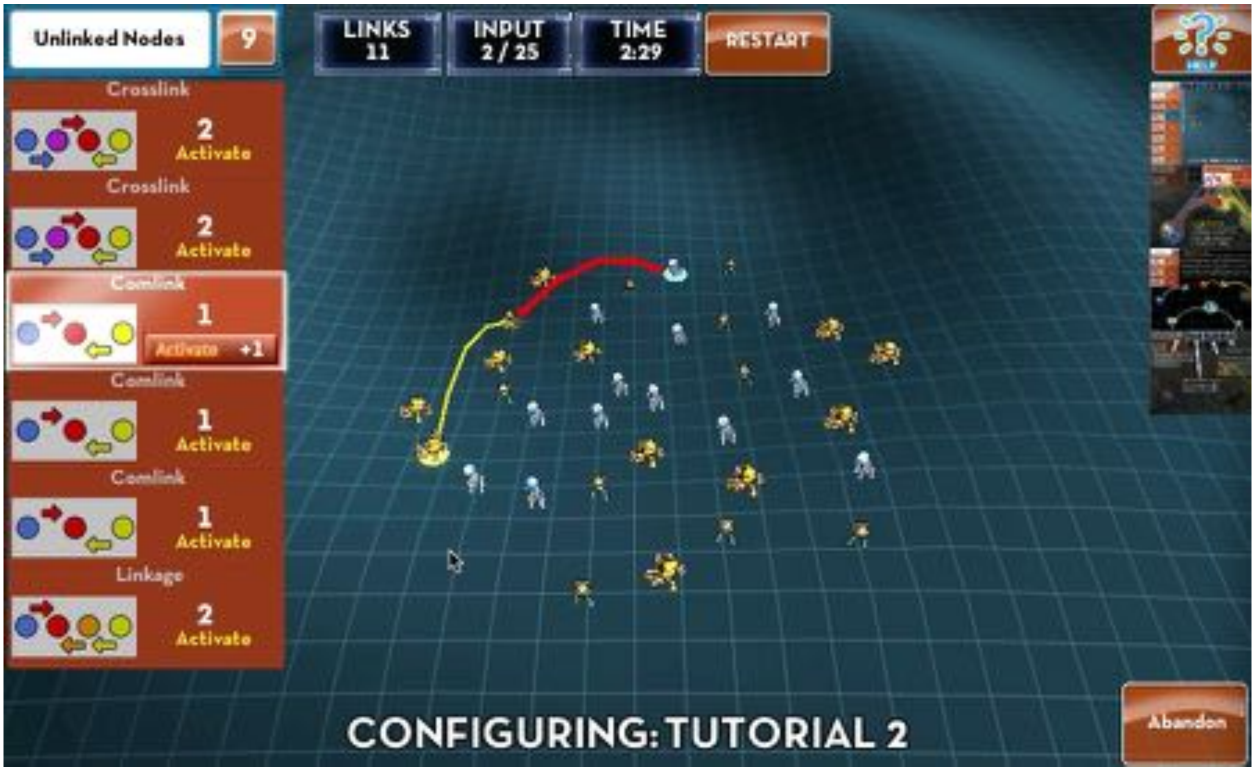


Figure 25: CircuitBot tutorial screen shot, presenting *gamma* connections.

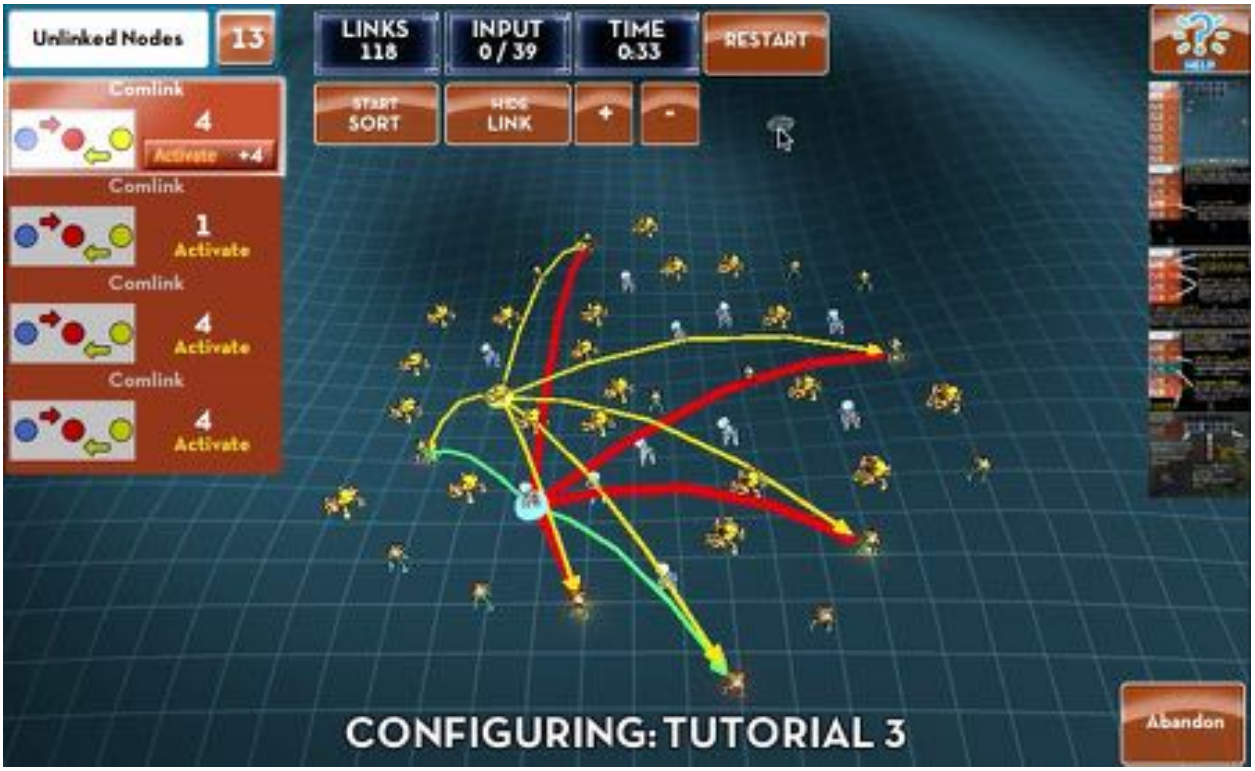


Figure 26: CircuitBot tutorial screen shot, presenting *beta* constraints, seen at bottom of the command list.

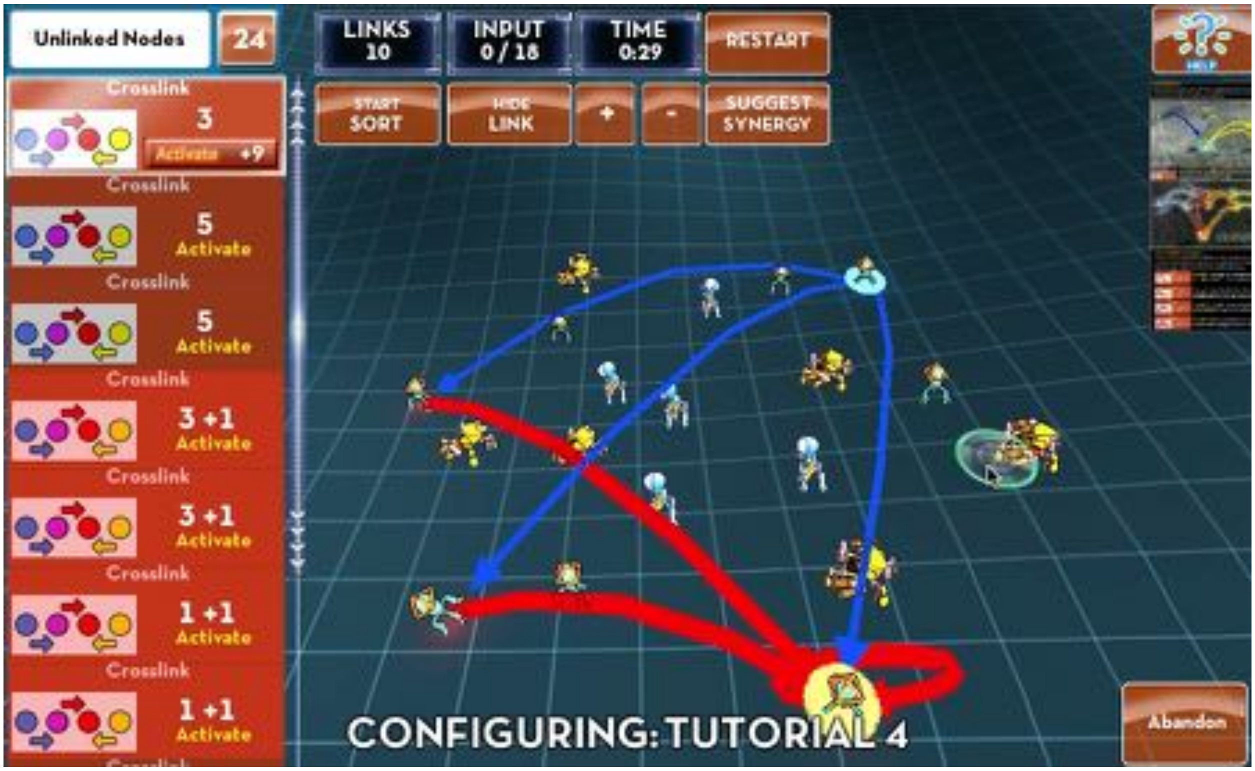


Figure 27: CircuitBot tutorial screen shot, presenting *delta* constraints.

discounted any contributions anonymous players could provide. In CircuitBot, anonymous players played a tutorial version of the game which used a set of canned game levels. Most players (by a 15:1 ratio) preferred to try the game anonymously without creating a user account. In reviewing the number of milestones reached by anonymous players, where more milestones indicates a longer play experience, we note a steady drop off. After reviewing the number of milestones reached by players — more milestones indicating a longer play experience — we noted a steady drop off of players. Some of the possible causes for drop-off before launching the game may have included:

- Unity plugin installation. Many players are reluctant to install new software in order to try a new game.
- Apathetic netbots. Bots that find the page would move on before the game launches
- Players impatient with game download (about 10 seconds).

Some of the possible causes for drop-off during the game may have included:

- Tutorials off-putting. The player numbers decline steadily with each tutorial. Many players prefer to "just play" and then visit tutorials if they do not understand something. Our game forced players through tutorial levels before allowing them free access to the complete interface.

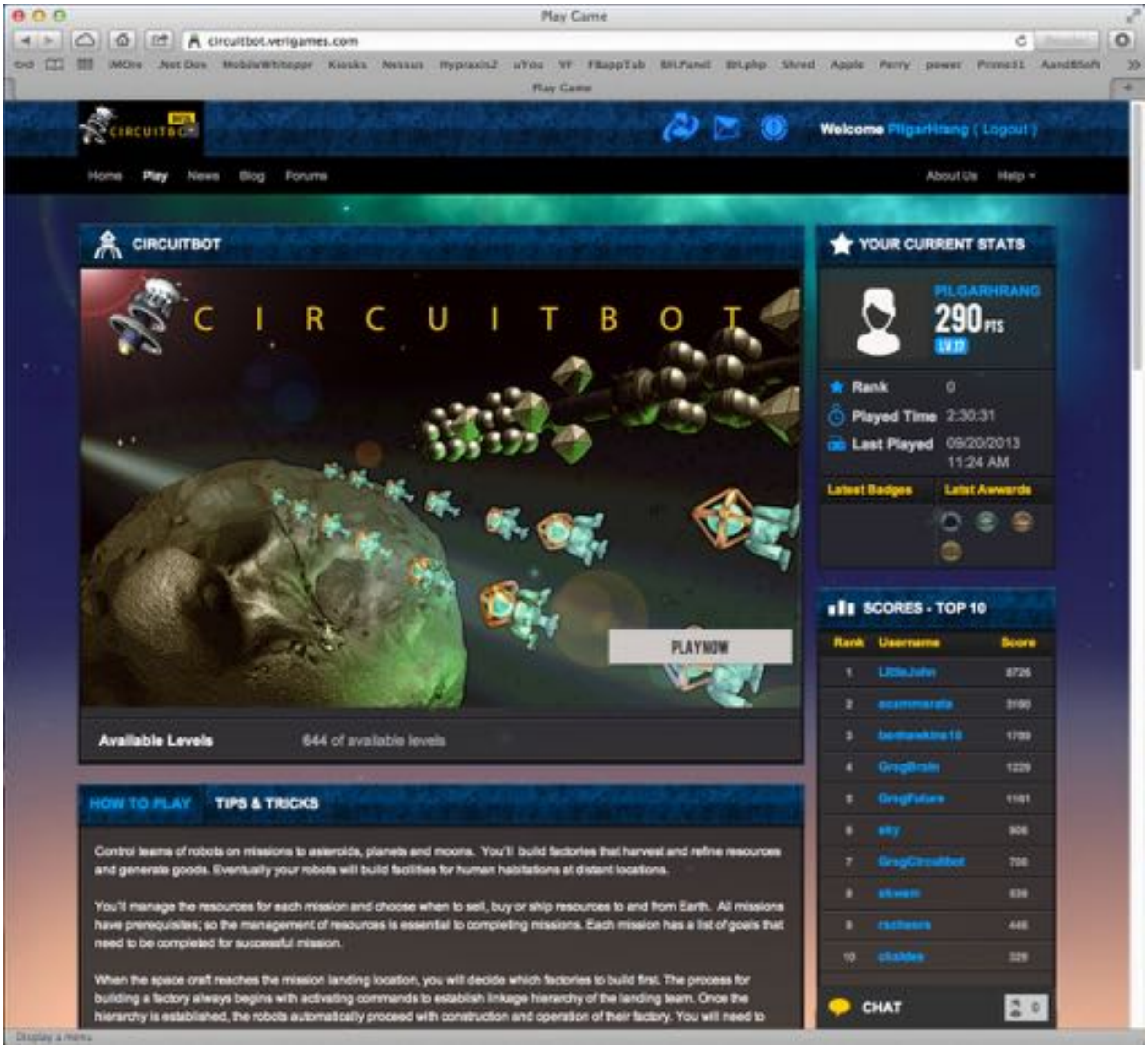
- Game style. Players who do not enjoy space-based resource management games may be turned off early in the tutorial process
- Game difficulty. CircuitBot was not a simple game.
- Return as registered players - An anonymous player who enjoyed playing CircuitBot might have left the game to visit the registration page and then would be counted among the registered players.

4. **Lower player traffic than expected demanded alternative production.** The hypothetical "best" outcome for the project was to produce a game that was as popular as "Angry Birds." More players meant more work done. By the end, CircuitBot had seen nearly 1,000 registered players, and had been played anonymously about 13,000 times. These unexpectedly low player visit numbers hindered the solving process. Due to the low number of players, our team worked on automating game play, driven by the practical need to generate data to use for analysis. One side benefit of creating this auto-solver was that by studying the data, and looking for ways to speed it up, we gained a better understanding of the data in the game instances.

5. **Forums were a good way to reach the registered players.** The forums on the Verigames site recorded the number of players that read each forum post. This was a large number, compared to the number of players who actually posted a message. We conclude that the forums were a good way for the developers to communicate with registered players.

6. **Social network integration could have been expanded.** The TA3 performer developed a web portal incorporating many social elements (Figure 28 on the following page). This was intended to provide a method for players to share information and collaborate. The forums were a good way for players to report problems and to share experiences. Unfortunately, this took place outside the games. The website included buttons for "liking" the web pages for each game (via Facebook, Twitter, Google+). This provided some social engagement, but there was room for more. If players could have made social posts from within the games, including artwork and event messages from the game, and linking directly back into the game, there might have been more traffic to the games. The difficulty in adding this deep level of social engagement arose from the fact that each game was developed differently, using different technologies, and the TA3 performer would have been tasked with creating a generic API for all TA1 performers to use which linked to the various social networks. There wasn't time or budget for such a development effort. For future development it might be helpful to allow developers to create their own social systems which could be integrated into their games at a deeper level.

7. **Replay level assumptions wedged.** In our game a single game level needed to be played multiple times, so it was permissible for a single user to encounter the same game level multiple times. This ran contrary to assumptions used by the TA3 performer when it designed systems that recorded data about players. In several subsystems it assumed the combination of `player_id` and `game_level_id` was a unique key used



Rank 0
Played Time 2:30:31
Last Played 09/20/2013 11:24 AM

Available Levels 644 of available levels

Rank	Username	Score
1	LittleJohn	8726
2	scammarata	3190
3	bmhawkins10	1709
4	GregBrain	1229
5	GregFuture	1161
6	sky	906
7	GregCircuitbot	706
8	skwan	639
9	rschears	446
10	cliaddes	329

Control teams of robots on missions to asteroids, planets and moons. You'll build factories that harvest and refine resources and generate goods. Eventually your robots will build facilities for human habitations at distant locations.

You'll manage the resources for each mission and choose when to sell, buy or ship resources to and from Earth. All missions have prerequisites; so the management of resources is essential to completing missions. Each mission has a list of goals that need to be completed for successful mission.

When the space craft reaches the mission landing location, you will decide which factories to build first. The process for building a factory always begins with activating commands to establish linkage hierarchy of the landing team. Once the hierarchy is established, the robots automatically proceed with construction and operation of their factory. You will need to

Figure 28: A view of the CircuitBot *mini-site* the player used to access the game through the Verigames web site. The mini-site displays player status, summary instructions, leader board, and access to other social features.

without replacement. For our games it was unique but used with replacement. This lead to problems in accumulating game play data about CircuitBot players to the RA and the Scoring system. For the Scoring system - which fed the game web site and player ranking - we established subfields in the record to track the individual scores achieved by the player for this game level, a “best” field. The PHP code would overwrite the top level score value for the `player_id + game_level_id` record with the “best” value.

8. **More player data is better.** As a result of work of integrating with the RA, we included code to record a large amount of information about the player during gameplay. This included the time each player spent on various screens, session duration, etc. We also recorded when players reached certain internal milestones (the first time they reached each major screen, first mission launch, completion of each tutorial, etc). Because the missions were played sequentially, and significant milestones were tracked related to player progress, we could detect which players were returning, how far they were getting, the points where they would drop off, and analyze this data. We collected logs of player events and contributions during game play (these were not attributable to identifiable individuals unless later volunteered). We maintained tallies for total productive work generated and time required, and developed assessments of the proportion of play-time to work-time. We gathered a number of internal game and backend metrics for progress and performance tuning.

3.11 Dynamakr Game

The low player volume seen during the Phase I games refocused the project to target the *whales* or *gurus*; that is, a handful of interested players who are willing to take the time to learn the game and be especially productive. In particular, our team’s experience with CircuitBot led us to make some course-correcting decisions for our Phase II game:

- The game would not include an *outer game* such as exploration missions, originally intended for player retention.
- Full focus of the game should be the symbolic node relationships of our game model.
- The game should auto-activate constraints whenever possible, instead of asking the player to take an extra step to do this.
- The game would present just-in-time tutorials to eliminate the qualification procedure.
- We would enable anonymous players to contribute to the verification effort.

As we worked with the verification data in CircuitBot and gained a better understanding of the components of a game instance — constraint types, memory offsets, memory intervals — we spent time exploring different ways of ordering the game instances, grouping them and exploring relationships between the elements. It was this basic idea of *exploration* that started our search for our Phase II game design. During CircuitBot development we created tools for automating different aspects of gameplay and wanted to include automation in the new game. We saw automation as the means for clearing away the large volume of trivial and tedious work and leaving the player to spend time on meaningful tasks.

The process of exploration was at the heart of what we were doing as we examined the data, and we wanted to put the player in charge of this exploration. The design began as first-person exploration game, with the player flying through many visually interesting environments. This became our second game which we called *Dynamakr* (29). The main conceptual differences between Dynamakr and CircuitBot were the following:

- CircuitBot players would work on a single game level at a time. In Dynamakr the player would work with multiple game levels simultaneously, with an auto-solver doing some of the work. Figure 30 on the following page captures some of this idea.
- CircuitBot wrapped the verification-based work into the universe of the game. In Dynamakr we twisted the focus of game play to encourage the player to make direct contributions to verification rather than indirect contributions.
- In Dynamakr we revealed some of the game model by way of rules and constraints, without exposing the program identification or its vulnerabilities.
- In Dynamakr we used simplified visual objects to enable many more important elements to be displayed in the player's browser.

Figure 29: Dynamakr game logo.

3.11.1 Design Goals

The Dynamakr design themes captured the conceptual changes. Dynamakr was a game of action and puzzle solving. It adopted the metaphor of a universe containing warped fabrics composed of knots and threads. The player traveled through the alternate reality collecting power knots and avoiding everything else. The player piloted his way through this environment using first person controls, while constantly moving forward. The player controlled an avatar which must perform game actions and collect energy required for additional travel. The player's primary goal was to search the fabric universe for certain special knots which

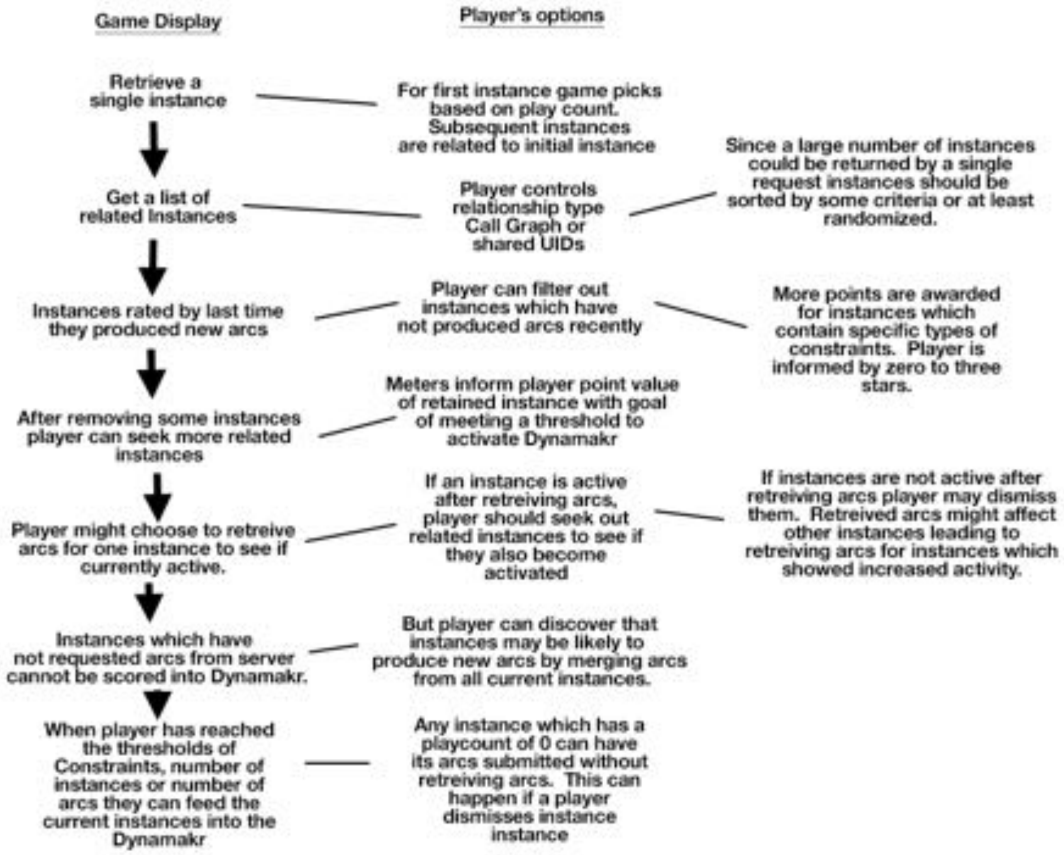

Game Display
Player's options
Retrieve a single instance
For first instance game picks based on play count. Subsequent instances are related to initial instance
Get a list of related Instances
Player controls relationship type Call Graph or shared UIDs
Since a large number of instances could be returned by a single request instances should be sorted by some criteria or at least randomized.
Instances rated by last time they produced new arcs
Player can filter out instances which have not produced arcs recently
More points are awarded for instances which contain specific types of constraints. Player is informed by zero to three stars.
After removing some instances player can seek more related instances
Meters inform player point value of retained instance with goal of meeting a threshold to activate Dynamakr
Player might choose to retreive arcs for one instance to see if currently active.
If an instance is active after retreiving arcs, player should seek out related instances to see if they also become activated
If instances are not active after retreiving arcs player may dismiss them. Retreived arcs might affect other instances leading to retreiving arcs for instances which showed increased activity.
Instances which have not requested arcs from server cannot be scored into Dynamakr.
But player can discover that instances may be likely to produce new arcs by merging arcs from all current instances.
When player has reached the thresholds of Constraints, number of instances or number of arcs they can feed the current instances into the Dynamakr
Any instance which has a playcount of 0 can have its arcs submitted without retreiving arcs. This can happen if a player dismisses instance instance

Figure 30: Early design flow detailing the change in game play, from focusing on the elements inside a single instance, to focusing on the relationships between game instances, the decisions available to the player, and the consequences of those decisions.

were not connected to threads. The player collected these unconnected knots and carried them during travel through the universe. The player needed to avoid collisions with other objects lest he lose travel energy. Over time as well the player's travel energy would be depleted so he must pause and accumulate more energy with the special play actions. When the energy was depleted the *power knot* converter would appear for the player. The player would attempt to combine, or tie, power knots to accumulate energy and earn points. The game determined which power knots were similar enough to be combined and would assign an energy and point value. Thoughtful players would examine the possible combinations and only tie power knots in such a way to achieve the maximum energy value (see for instance Figure 31 on the next page). The player could choose to save power knots to use in future games.

The game design goals included the following:

- Take advantage of Unity 5 graphical features.
- Play multiple game instances simultaneously, illustrating how game results influenced each other.
- Find related game instances dynamically and determine play priorities for efficiency.
- Auto-solver analyzers were embedded in the game, working alongside the player.
- Cases where infinite loops were predicted from automated solving of game instances did not materialize, so the need for a player to detect and deal with these situations was no longer present. Before the second game development period started, we ran the auto-solver many times analyzing types of constraints, numbers of arcs, CPU load, number of iterations over the entire set, and many other factors to try to determine what a player could do to improve on any aspect of the auto solution process.
- CircuitBot revealed that the number of arcs in a completed points to graph would be in the millions so there needed to be special tools to allow each game instance to be solved with only a subset of the required arcs. A player could use information about each instance to help him determine when or if each instance needed to be solved before committing to the time consuming and memory intensive process of downloading the arcs already associated with the instance.
- We envisioned a Mechanical Turk derivative while developing the Dynamakr game. Some of the development effort for Mechanical Turk was devoted to designing a system which would support both the game and the Turk system. We developed a prototype system which exposed as much information as possible for a player to work with, which helped us to determine the important factors having impact on game play. The two finished games shared over 80% of the underlying code, with the main differences being how the data is presented to the player and the addition of the Dynamakr reward game. The two games shared 100% of the backend code.
- The arcade style reward game was designed so that if any final changes to the game model code or the CodeHawk-generated levels wound up introducing the predicted infinite loop states, we might detect the condition and incorporate some remedy player

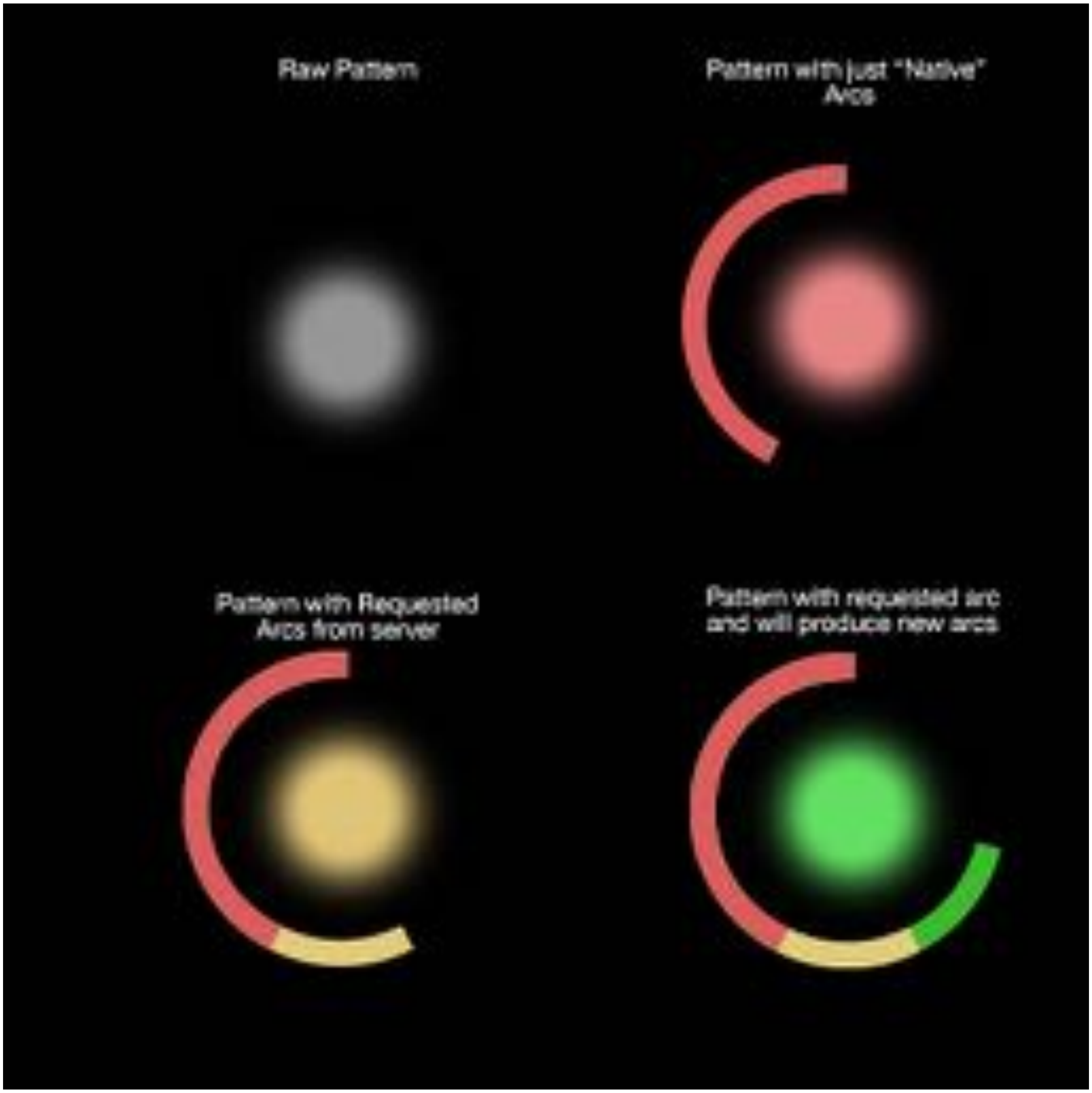

Raw Pattern
Pattern with just "Native" Arcs
Pattern with Requested Arcs from server
Pattern with requested arc and will produce new arcs

Figure 31: Dynamakr game play visualization of individual game instances measuring arcs generated with and without merging relevant arcs from the underlying points-to graph.

action into the shooter game. This would have been reflected by continuous, increasingly fast game play with the player encountering larger knots of arcs repeatedly. This optional element was never integrated because the looping situation did not occur.

The Dynamakr game was graphically intense. It used the most advanced features of the Unity 5 game engine to yield a play experience that was abstract, interesting, and fun to explore. The player searched through under-water, outer space, geometric, whirlwind and other abstract environments to obtain the power knots that were the player's primary source of energy. The universe of the game was generated procedurally. Each game session introduced undiscovered worlds for the player to explore.

3.11.2 Launch

The Dynamakr game was composed of two phases: a working *analysis phase* and a rewarding *dynamo phase*. We defined these as follows:

- **Analysis Phase**. The player's goal was to generate new arcs (*energy* in the game). The analysis phase was complete when the player had added the target amount of energy or more. Arcs are added when the player satisfies constraints in a game instance. The player directed the search for related game instances based on the assumption that arcs added to one game instance will trigger new arcs in related instances. There were three types of relationships that instances could share based on elements they have in common. Figure 32 on the following page describes the weaving logic.

- **Dynamo Phase**. Using the data from the game instances encountered the analysis phase, the game generated a first-person flying environment. The player steered a ship which required energy to fly and had a limited amount of shield energy which protected the player from flying obstacles. The player shot the obstacles having differing point values based on how difficult or dangerous they were. By shooting obstacles the player replenished his power and shields. He could also collect *powerups* which improved his weapons, which allowed the player to extend play time, and difficulty increased with additional obstacles and hardening. Some obstacles even avoided the player's shots and returned fire. The dynamo phase time was proportional to the number of game instances encountered during the analysis phase. At the end of the dynamo phase, the player resumed play in the analysis phase. The game continued until the player decided to leave. Figures 33 on page 137, 34 on page 137, and 35 on page 138 capture the dynamo phase action.

On-the-Fly Tutorial The Dynamakr game learned from the CircuitBot player drop-off lesson and used on-the-fly tutorials. Rather than forcing players through an initial static and somewhat contrived tutorial level we chose to integrate text displays and indicators that appeared just at the moment they are needed. In most cases, these indicators appeared when the player encountered game elements for the first time. The feedback from player testing was positive. Figures 36 on page 139 through 42 on page 145 show screen shots of these tutorials.

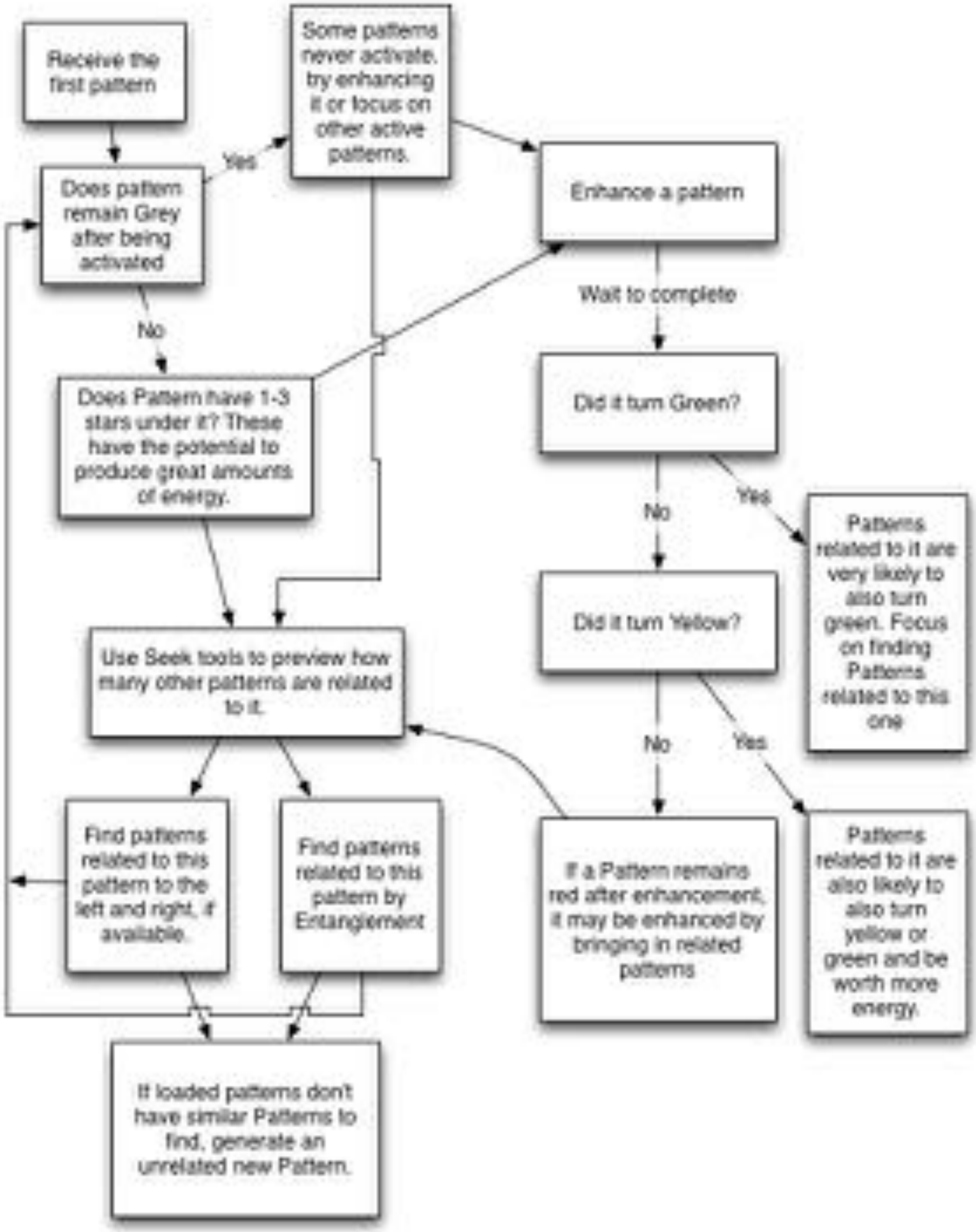

Receive the first pattern
Does pattern remain Grey after being activated
Yes
Some patterns never activate. try enhancing it or focus on other active patterns.
Enhance a pattern
Wait to complete
No
Does Pattern have 1-3 stars under it? These have the potential to produce great amounts of energy.
Did it turn Green?
No
Yes
Patterns related to it are very likely to also turn green. Focus on finding Patterns related to this one
Did it turn Yellow?
Use Seek tools to preview how many other patterns are related to it.
No
Yes
Find patterns related to this pattern to the left and right, if available.
Find patterns related to this pattern by Entanglement
If a Pattern remains red after enhancement, it may be enhanced by bringing in related patterns
Patterns related to it are also likely to also turn yellow or green and be worth more energy.
If loaded patterns don't have similar Patterns to find, generate an unrelated new Pattern.

Figure 32: Dynamakr game weaving process flow showing more refined player actions related to state of instances, focusing player on generating new graph arcs.

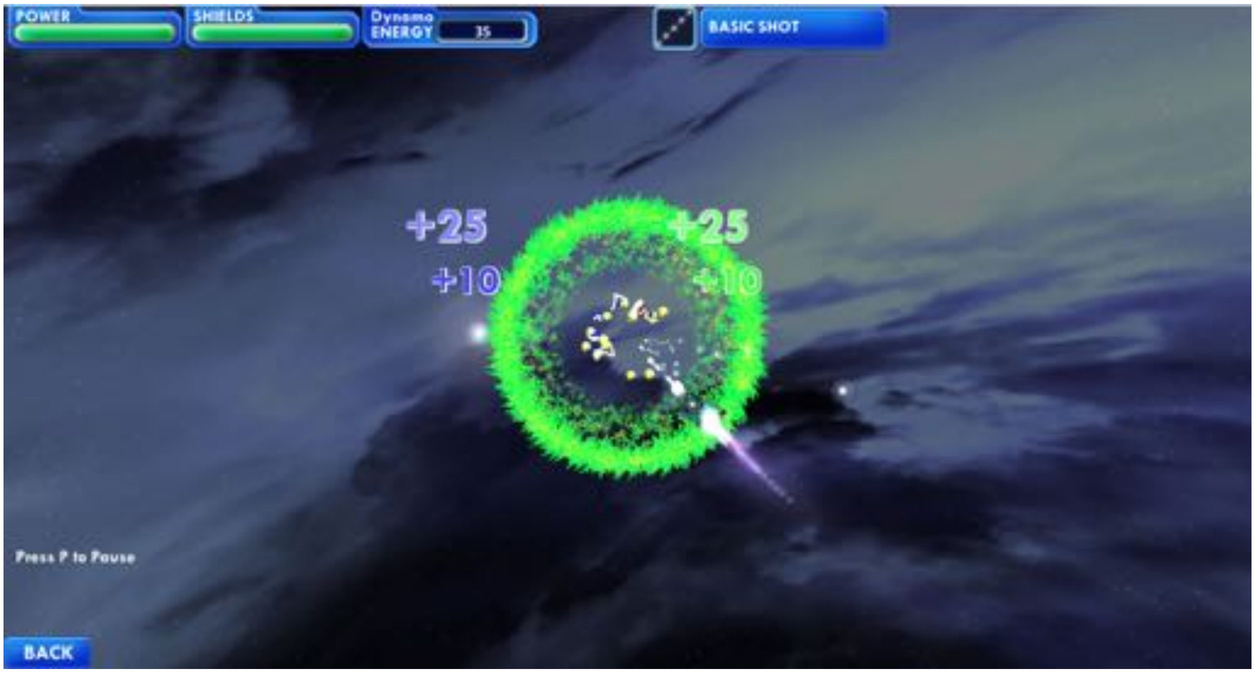


Figure 33: Dynamakr game's arcade-style reward game *inside the dynamo*. For each instance the player fed into the dynamo there was a section of arcade objects presented that he could shoot.

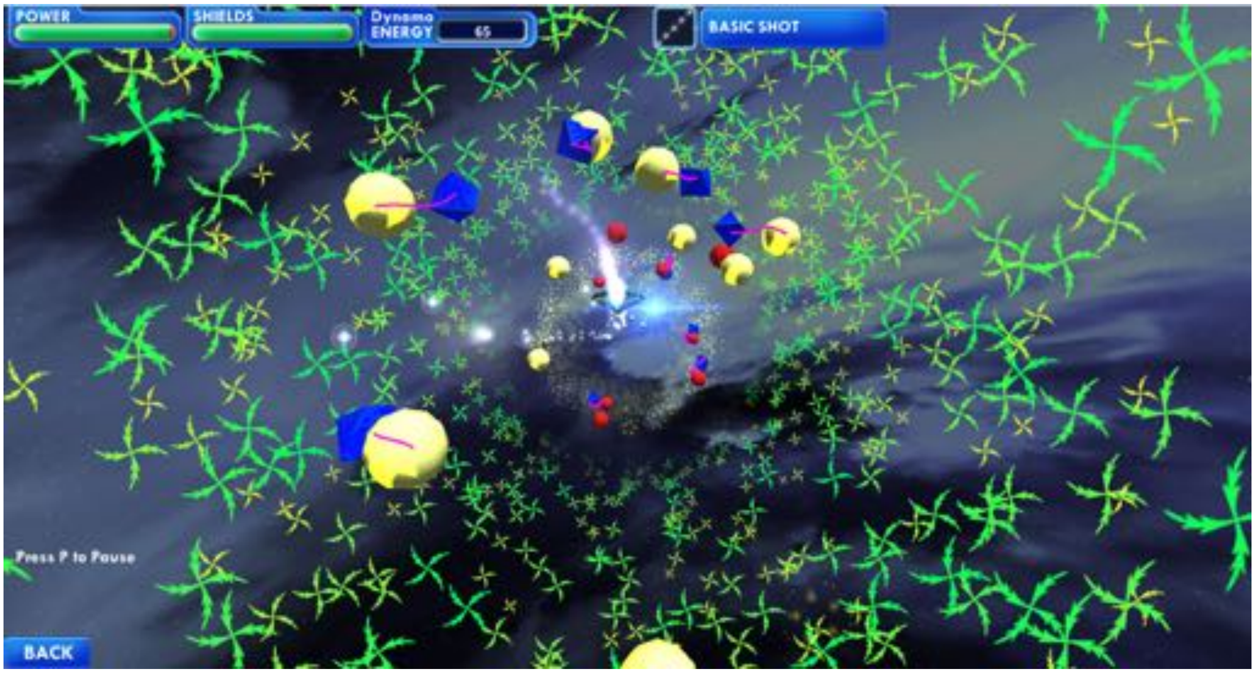


Figure 34: Additional view of the arcade reward phase of the Dynamakr game. The player shot the energized nodes to collect their energy.

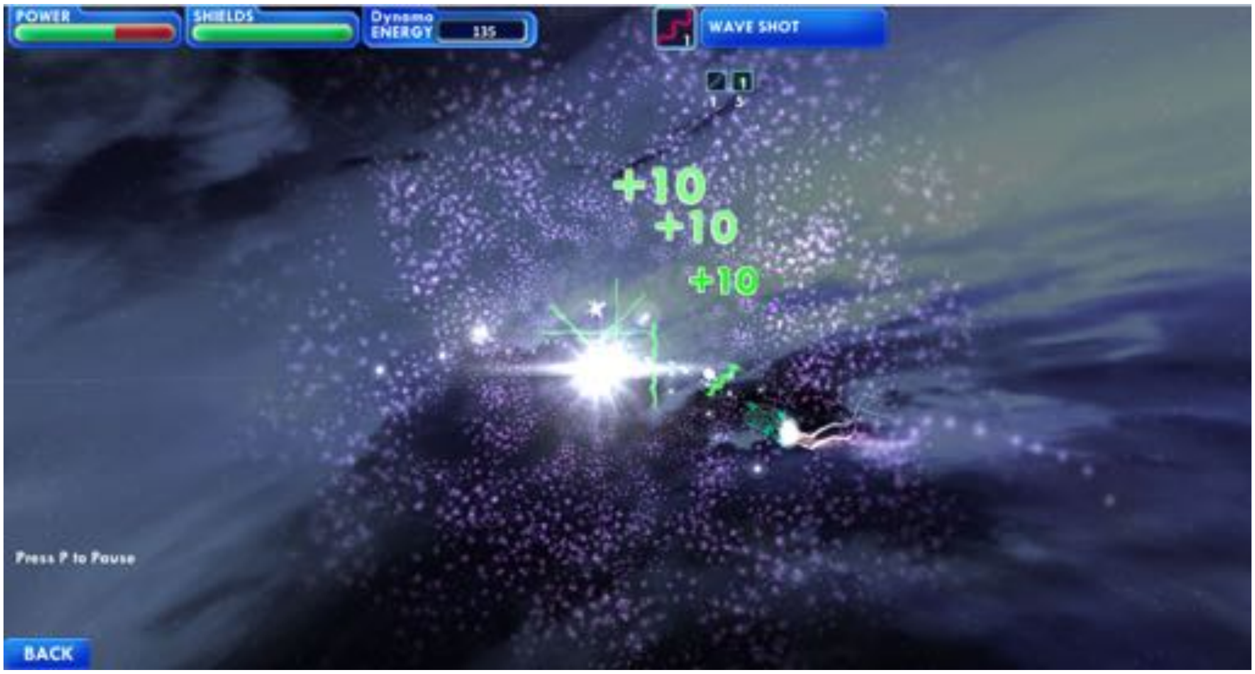


Figure 35: View of an Dynamakr game phase where the player shot obstacles which the game generated based on the number of inactive pointer-flow constraints.

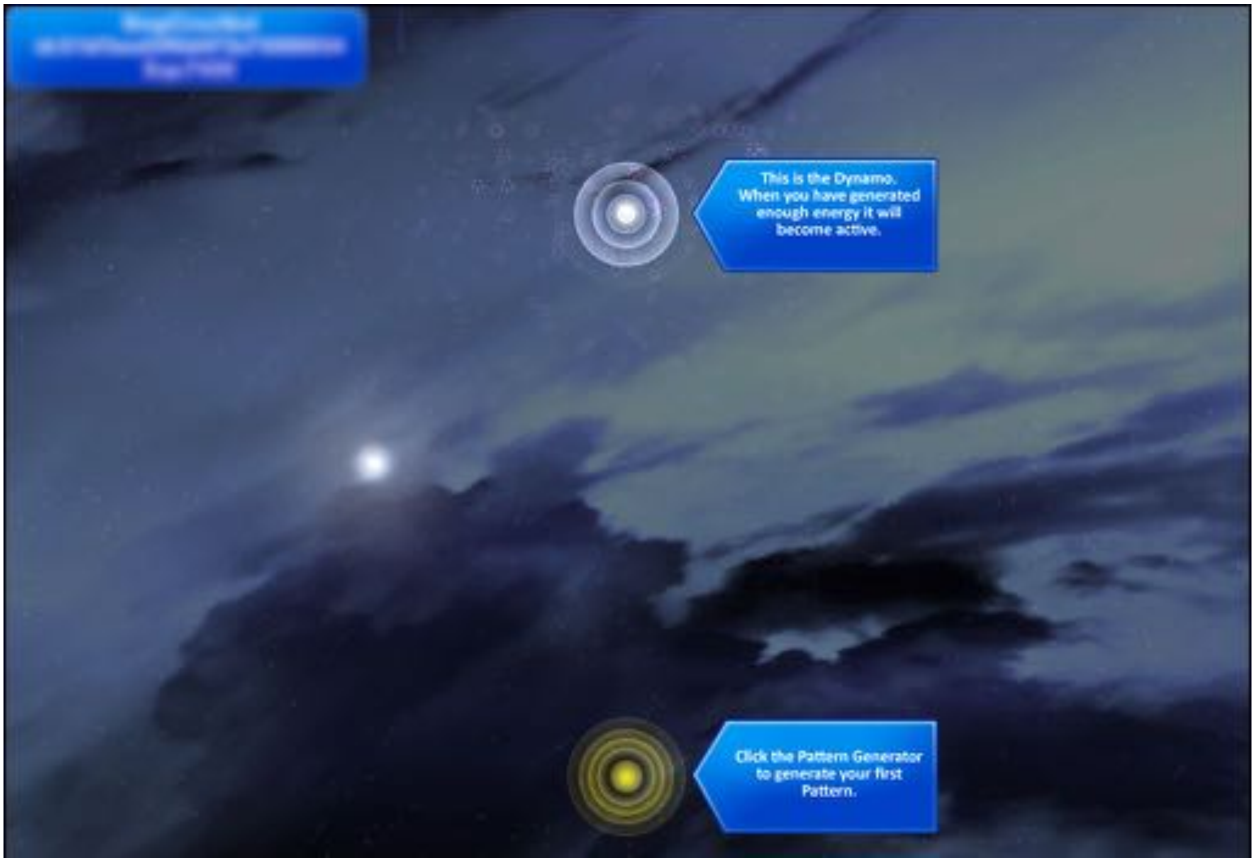
Citizen Scientists One of the goals during Phase II was to engage with *Citizen Scientists*. One of the conclusions reached analyzing the visitor traffic during the Phase I game was that people were visiting the Verigames site not to play the games, but to learn more about the project. The TA3 performer therefore revamped the Verigames website to include more detailed information about the CSFV project from an academic perspective. To support this with more academic information about our game, we exposed the number of arcs added to our points-to graph so that number could be presented to the Citizen Science audience as a measure of progress for our game. This number is labeled as the *verification facts discovered* on the Citizen Scientist[2] landing page.

3.11.3 Lessons Learned

This section captures some of the lessons learned, some of which were reinforced from the previous game, some of which we incorporated into our subsequent games, and some of which we leave for future programs.

1. **It was difficult to create a pure exploration game.** Our original vision was to create a first person flying game where the player would direct the flight of a ship through strange environments. However, in order to do meaningful verification work in the game, the player needed to make decisions about which game instances to load. This could not be done while flying the ship, so we introduced an analysis phase. This allowed us to present information about the already-loaded instances – through color,

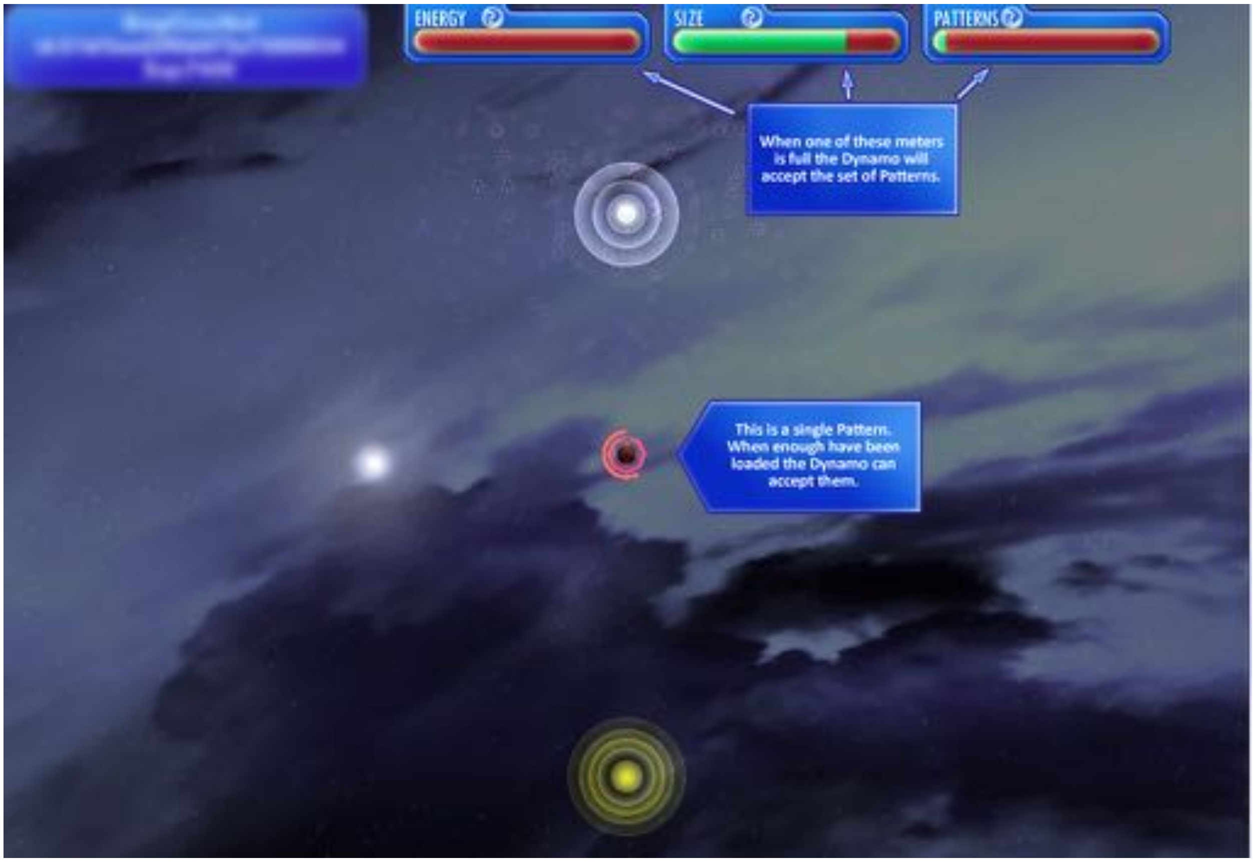
[2] http://verigames.com/science/.



This is the Dynamo. When you have generated enough energy it will become active.

Click the Pattern Generator to generate your first Pattern.

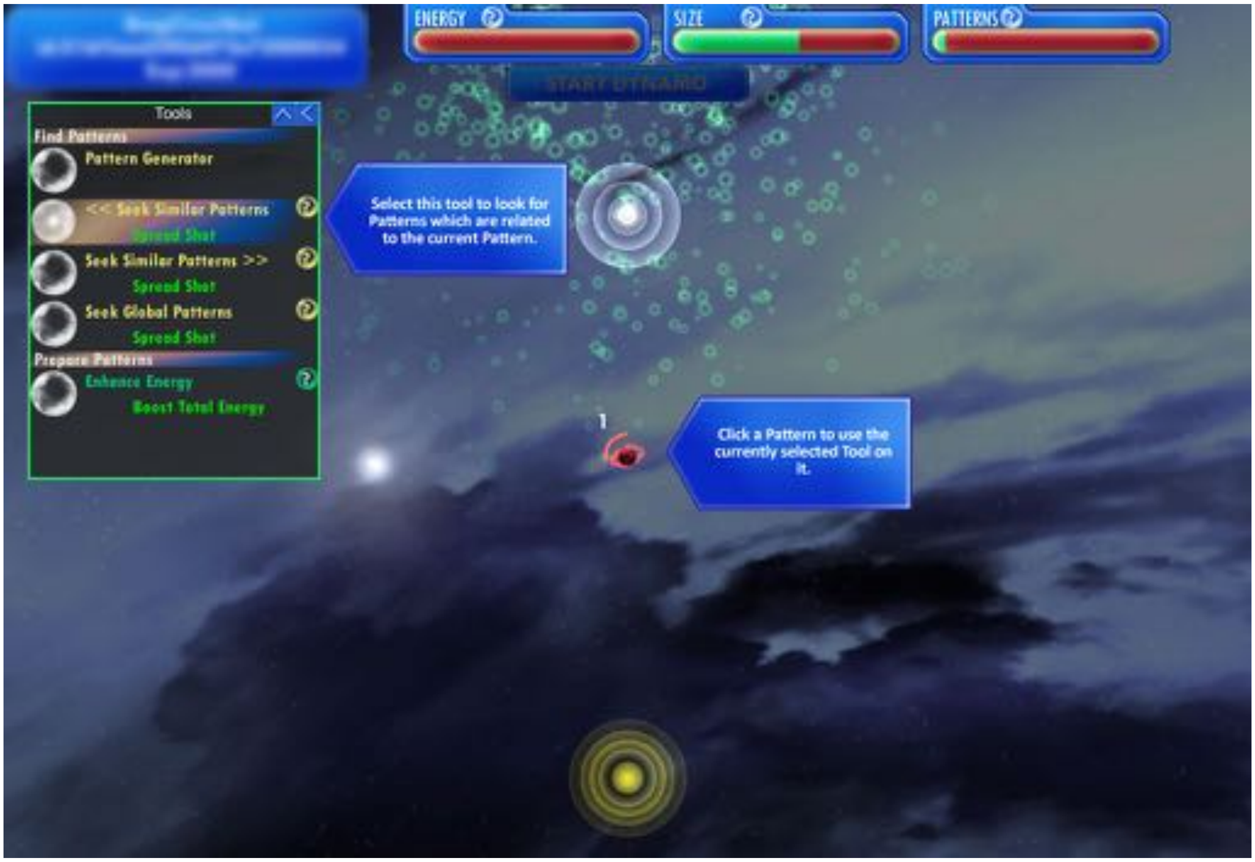
Figure 36: Dynamakr game tutorial 1. Presenting the instance generator and dynamo that will process the instances.



When one of these meters is full the Dynamo will accept the set of Patterns.

This is a single Pattern. When enough have been loaded the Dynamo can accept them.

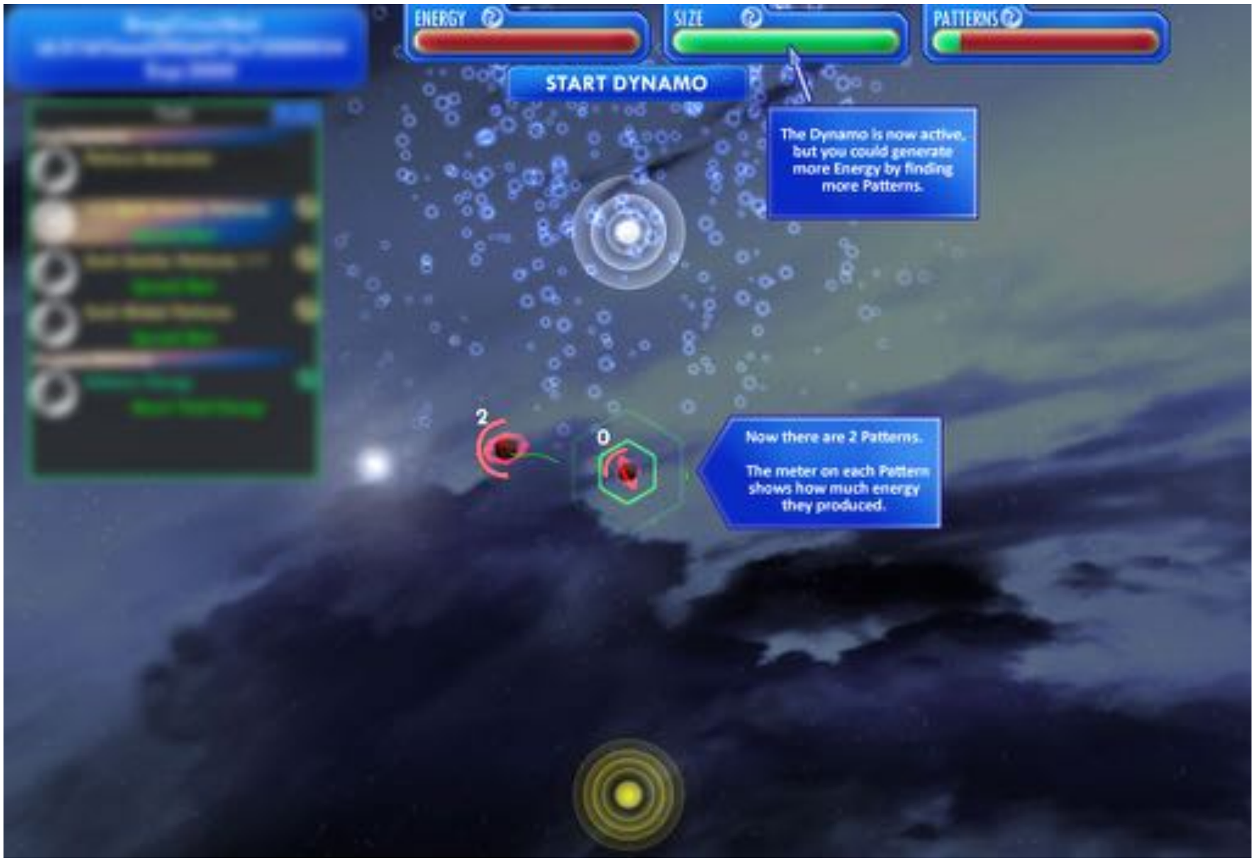
Figure 37: Dynamakr game tutorial 2. First instance presented and scoring for measuring progress in game.



Select this tool to look for Patterns which are related to the current Pattern.

Click a Pattern to use the currently selected Tool on it.

Figure 38: Dynamakr game tutorial 3. Presents tools leading the player through process of using *Seek Similar* patterns on a game instance. This tool uses the underlying pointer flow constraint call graph to find related game instances exchanging information via function parameters.

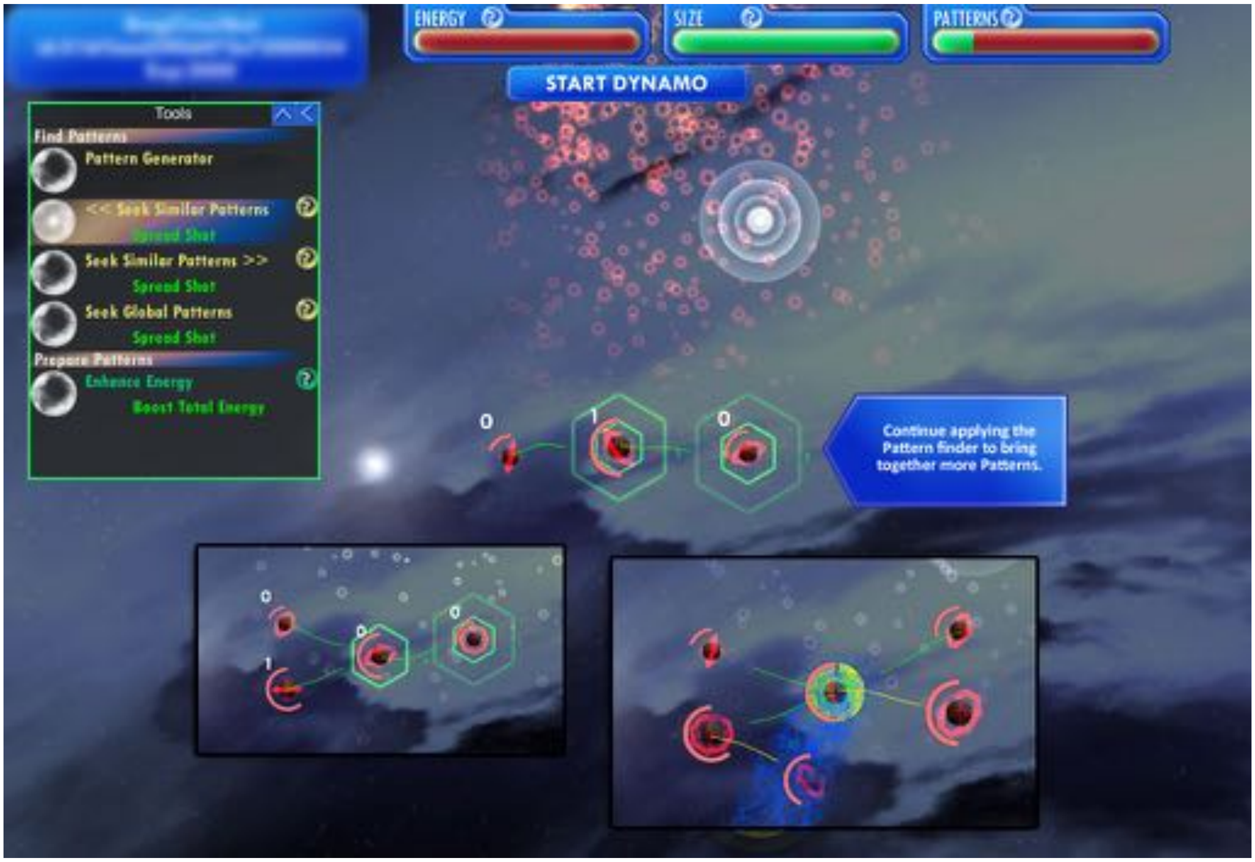


The Dynamo is now active, but you could generate more Energy by finding more Patterns.

Now there are 2 Patterns.

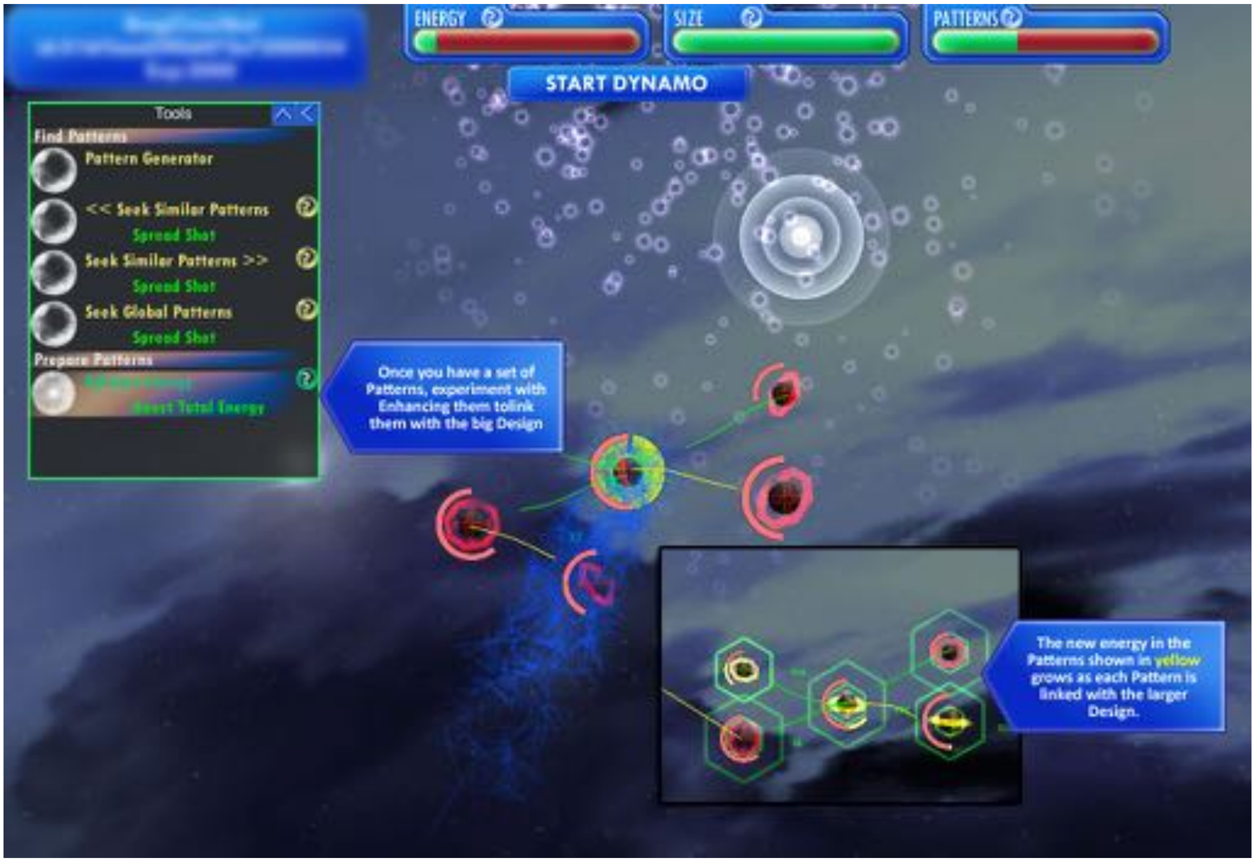
The meter on each Pattern shows how much energy they produced.

Figure 39: Dynamakr game tutorial 4. Adding new instances to the workspace. These instances are automatically solved in isolation by the auto-solvers.



Continue applying the Pattern finder to bring together more Patterns.

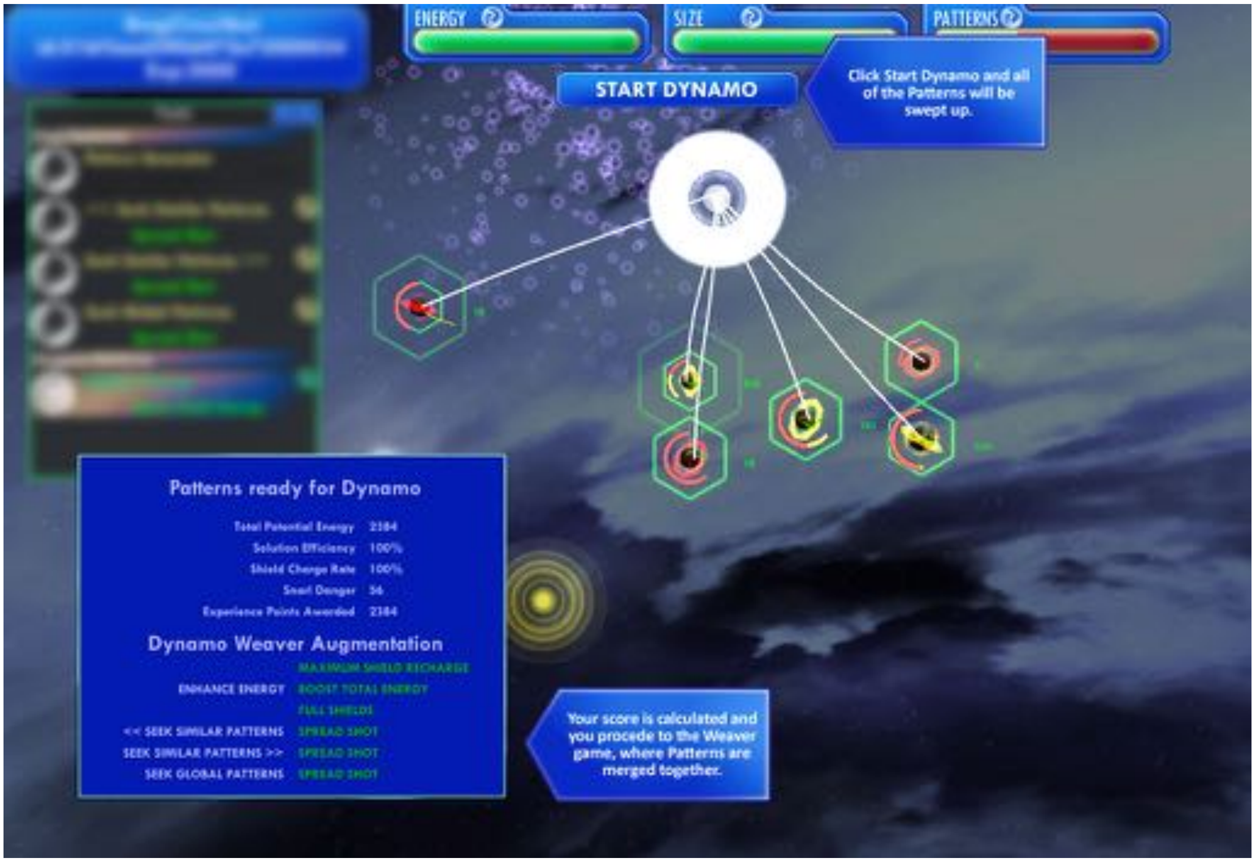
Figure 40: Dynamakr game tutorial 5. Visualization showing the accumulation of more instances into the work space. In this sequence, arcs from one instance are causing additional arcs to be generated in other instances, causing the yellow meters to appear.



Once you have a set of Patterns, experiment with Enhancing them tolink them with the big Design

The new energy in the Patterns shown in yellow grows as each Pattern is linked with the larger Design.

Figure 41: Dynamakr game tutorial 6. The player uses the *Enhance Energy* tool in instances to pull in points-to graph elements from the server in order to activate more constraints in each game instance. In this case, doing so causes more instances to change to yellow meters highlighting additional information brought in from outside. If this caused new graph arcs to be generated, and these arcs were not already present in the global points-to graph solution, then the player's instances would turn green to reflect his contribution to the solution.



Click Start Dynamo and all of the Patterns will be swept up.

Patterns ready for Dynamo

Total Potential Energy 2384
Solution Efficiency 100%
Shield Charge Rate 100%
Snail Danger 56
Experience Points Awarded 2384

Dynamo Weaver Augmentation

MAXIMUM SHIELD RECHARGE
ENHANCE ENERGY BOOST TOTAL ENERGY
FULL SHIELDS
<< SEEK SIMILAR PATTERNS SPREAD SHOT
SEEK SIMILAR PATTERNS >> SPREAD SHOT
SEEK GLOBAL PATTERNS SPREAD SHOT

Your score is calculated and you procede to the Weaver game, where Patterns are merged together.

Figure 42: Dynamakr game tutorial 7. The player has fulfilled the requirements to submit the set of game instances into the dynamo.

size, animation, and other attributes – and allow the player to choose what to load next. This two-phase game worked well, with a clear *work* phase, followed by a *play* phase.

2. **Anonymous players are important.** The player counts from CircuitBot clearly indicated that users wanted to visit the game without registering, so we revised our PHP scripts to allow anonymous players to contribute results without participating in rewards. If the player was not logged in when he reached our index page, we offered him a choice of going to the Verigames login page or playing anonymously. If he chose to remain anonymous, our code generated a new player record in our backend and assigned it a player ID which was identifiable as an anonymous player but not otherwise associated to the person. This helped when analyzing player data to distinguish anonymous players and registered players. We reused this approach for the VIPER version of the game because we could treat Amazon Mechanical Turk workers as a separate class of players during subsequent player analysis. While the changes to Dynamakr did increase the rate at which a player could contribute, the overall game appeal was not as effective as CircuitBot. If the project were revisited the CircuitBot game could be combined with Dynamakr's improvements to provide a storyline to push players to replay. The focus of Dynamakr was to allocate enough engineering hours to the technical goal and focus less on motivational game play. The main difference with the CircuitBot 2.0 game would be that instead of each robot representing a single node in a game instance, each robot would represent a game instance. The player would need to activate a certain number of robots in order to complete a factory. Whereas completing all eight missions in CircuitBot originally would have solved 180 game instances, a player of CircuitBot 2.0 would have needed to find new arcs in over 2,500 instances, and may solve thousands more in the process. It would combine the motivation and sequential play of the original game with the huge improvement to player productivity.

3.12 VIPER Game

Part way through Phase II we proposed and were tasked to produce a *mechanical turk* version of our verification game. This version would enlist *paid crowd source* (PCS) or *pay for play* participants, an experiment to determine whether we could interest a different community to contribute to the crowd sourced verification solutions. Having anticipated this version our game was largely a *revealed* or *stripped-away* version of Dynamakr, our third game which we called VIPER (verification improvement by PCS enhanced results) (Figure 43). Eventually the mechanical turk operation landed fully on the Amazon Mechanical Turk (AMT) foundation, with a new and convenient TA3-provided tool for loading *human intelligence tasks* (HITs) into the work queue and assigning payout values. Our HITs were batches of game instances that we would prefetch and load into the VIPER game for the player as a starting point.

Owing to the different player motivation, the VIPER version of our game had one major difference from both Dynamakr and CircuitBot. When playing VIPER, the players were paid for successful results. One of our primary building principles was to have the paid

Figure 43: VIPER game logo.

players do as much direct verification work as possible, and to do this without a need in the game for an overt entertainment element or an obfuscation of purpose. We dropped the overt entertainment elements of CircuitBot and Dynamakr because those were designed for player retention. In VIPER the monetary payout was the retention device. The VIPER player working time and retention statistics of Section 4.4 on page 183 show the success of this approach.

3.12.1 Design Goals

Having anticipated the VIPER game during Dynamakr game development, our design and implementation of the two were similar underneath but quite different in appearance. Our design approach for VIPER was to start with the Dynamakr game then strip it down to the minimum user interface necessary for a player to choose which game instances to load and which game instances to solve (by applying its constraints). The minimum feedback would indicate when solving a game instance produced new arcs and yielded points.

For VIPER we elevated the exploration of inter-game-level relationships to a primary activity. Our intuition about how to search through the set of game instances, which developed during the construction of Dynamakr, was to move from one game instance to others by following *neighbor* relationships. Three types of relationships were relevant:

- **Left Related** – game instances that are called (via function calls) from the target game instance
- **Right Related** – game instances that call to the target instance
- **Common Related** – game instances whose constraints refer to memory locations found within the constraints of the target game instance

In VIPER the game instances were referred to as *patterns* and were displayed on the screen as rectangular information panels such as Figure 44 on the next page. The background color of a panel was a visual indicator of the state of that pattern. Red background indicated that the instance had been loaded, but that the arcs leading to and from the memory locations in its constraints had not been loaded. Arc counts could be very high and a single game instance might have referenced hundreds of arcs, so the game engine did not load them automatically. The player chose if and when to load the arcs. The presumption was that the player would examine the detailed information for the pattern to decide whether it was worth the time to load the arcs for that instance. Some strategies the player may have considered, none of which were guaranteed to succeed in any specific situation, included:

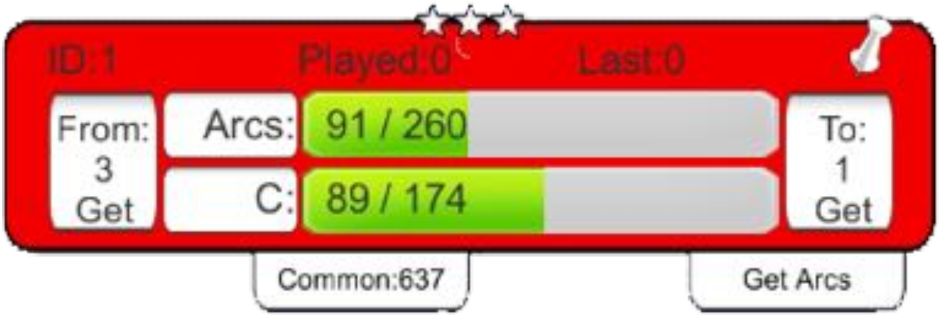


Figure 44: VIPER game play display panel showing detail information for a single pattern (a single game instance). The panel shows the number of related game instances based on different relationships (From=3, To=1, and Common=637), number of arcs (Arcs=91/260, showing the ratio of *arcs loaded* over the *estimated total arcs connected to this instance*) and the number of constraints (C=89/174, showing the ratio of *satisfied constraints* to the *total constraints*.

- When a pattern background turned green, focus on loading the neighbors of that pattern, with the understanding that arcs added for one game instance may allow for new arcs to be generated for its neighbors.
- The pattern panel showed an estimate for the number of arcs that would be loaded if the **add arcs** button for this pattern was pressed. Loading large quantities of arcs tended to slow down the game, so it was in the player's time and economic interest to be thoughtful about this action.

The player was given a target number of patterns to turn green for success. Once he reached that goal, the game provided a *completion code* which was sent automatically to the AMT system through the TA3-provided API. To be paid for his effort, the player submitted his completion code into his AMT form.

In summary, the VIPER game worked like Dynamakr behind the scene, but did not resemble it at all:

- VIPER presented none of Dynamakr's graphical gameplay elements. In VIPER, only the most important aspects of a game instance itself were presented to the player. Essentially, VIPER revealed the game content directly to the player, while Dynamakr hid this content.
- In VIPER, game instances were presented to the player as graphical panels showing the key properties, including numerical values, of that game instance.
- In VIPER the player was given only a few actions to take. A player could load instances related to an on-screen instance, or he could request arcs for a game instance. Requesting arcs caused the given game instance to be evaluated (its pointer flow constraints were triggered) to determine if any new arcs could be added. If so, the display for that game instance turned green and the player was awarded one point. Figures 45 on page 150 through 50 on page 155 show the instance tagging and arc fetching activities for the game.

3.12.2 Lessons Learned

The following were some lessons learned during our experience with the Mechanical Turk version of our verification game.

- **By comparison, Turk workers were plentiful.** In a strong contrast to the small population of players for either of the CircuitBot or Dynamakr games, the population of AMT workers was plentiful. Because they were motivated by the possibility of financial payment associated with completing a HIT, the population of *turk workers* immediately descended on any new HIT candidate that appeared ripe for the taking. Turk workers had the option of examining a HIT and then returning it if they did not want to complete the work. We had thousands of turk workers who did this with our VIPER game.

- **Turk workers would endure inconveniences for potential payouts.** With the player directing the search through the collection of game instances, some players took longer to find the target number of instances that could be improved. Longer searches resulted in the game having to manage a large number of instances, which tended to make the game run more slowly. The longer the game took to complete, the more negative feedback we received from the turk workers. Such feedback focused on the amount of time it took to play the game, which translated directly into less pay per hour of effort. Although this negative feedback — email messages — gave the impression that the game was not working, this impression was misplaced. The VIPER game simply took more time to process the game content because more game instances were loaded into memory and consuming network traffic. Overall, given the large number of turk workers who played the game, the percentage of negative feedback we received was quite small. Section H on page 306 provides an example of the feedback we followed during the AMT sessions.

4.0 RESULTS AND DISCUSSION

In this section we present statistical results gathered for each of our three games. Each result collection presents player counts and play time as well as various game productivity measures. Section 4.1 on page 156 first presents the economic model we derived to assess the value of an improved verification result, comparing the cost of obtaining that result automatically or with the crowd, with the cost avoidance of human experts manually computing the same result. Section 4.2 on page 157 presents the CircuitBot game results. Section 4.3 on page 169 presents the Dynamakr game results. Section 4.4 on page 183 presents the VIPER game results. For better comparison of paid and unpaid players, we have also produced a set of combined results for the Dynamakr and VIPER games, presented in Section 4.5 on page 207.

In addition to game play results, we also present here our auto-solver results. Section 4.6 on page 215 presents the production and economic statistics for our Arcweaver auto-solver. Section 4.7 on page 257 provides the verification cases and data tables accompanying these results.

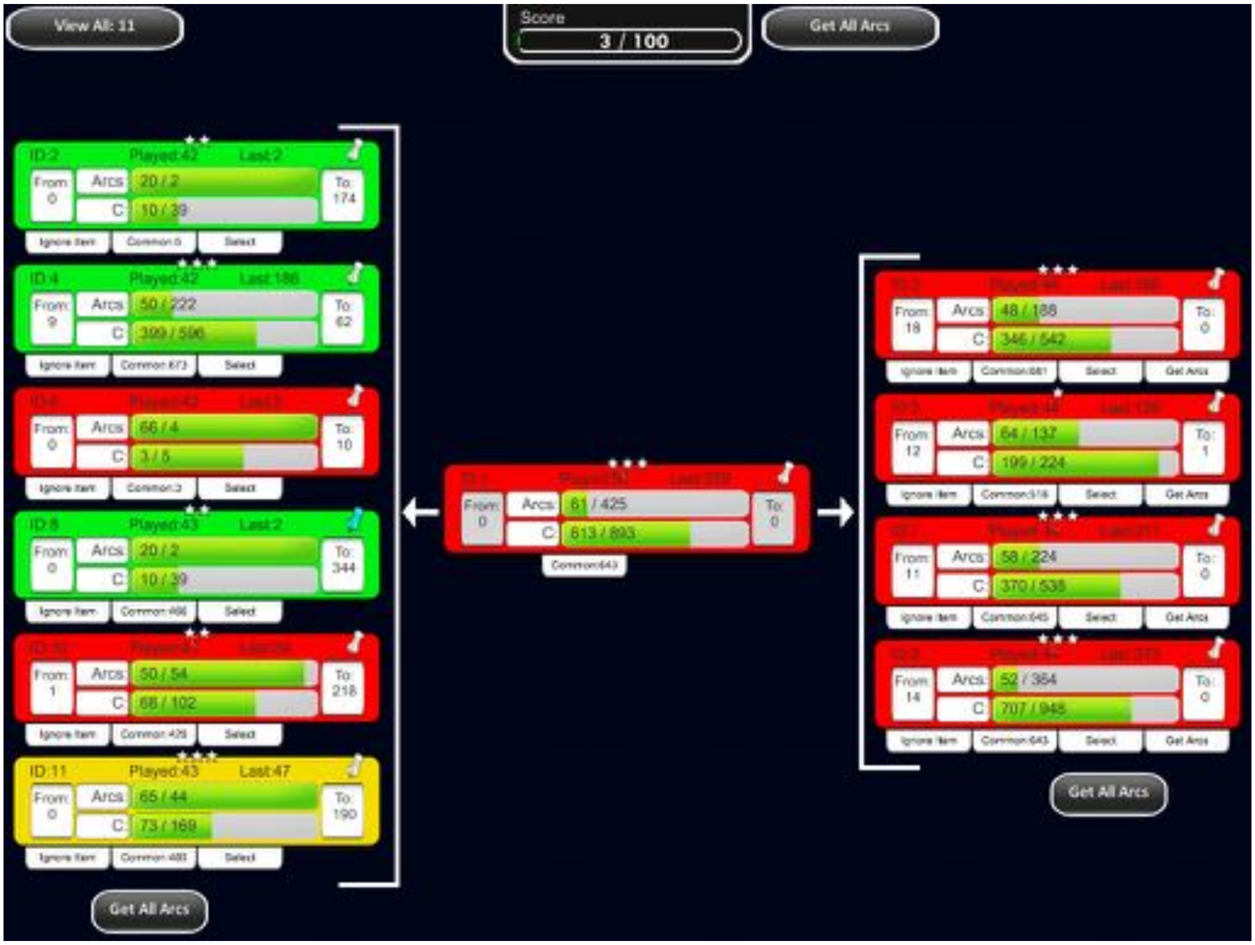

View All: 11
Score
3 / 100
Get All Arcs
ID:2 Played:42 Last:2
From 0 Arcs 20 / 2 C 10 / 39 To 174
Ignore Item Common:6 Select
ID:4 Played:42 Last:186
From 9 Arcs 50 / 222 C 399 / 596 To 62
Ignore Item Common:673 Select
From 0 Arcs 66 / 4 C 3 / 5 To 10
Ignore Item Common:3 Select
ID:8 Played:43 Last:2
From 0 Arcs 20 / 2 C 10 / 39 To 344
Ignore Item Common:486 Select
From 1 Arcs 50 / 54 C 68 / 102 To 218
Ignore Item Common:428 Select
ID:11 Played:43 Last:47
From 0 Arcs 65 / 44 C 73 / 169 To 190
Ignore Item Common:480 Select
Get All Arcs
From 0 Arcs 61 / 425 C 813 / 893 To 0
Common:643
From 18 Arcs 48 / 188 C 346 / 542 To 0
From 12 Arcs 64 / 137 C 199 / 224 To 1
From 11 Arcs 58 / 224 C 370 / 538 To 0
From 14 Arcs 52 / 384 C 707 / 948 To 0
Ignore Item Select Get Arcs
Get All Arcs

Figure 45: VIPER game play. This workspace shows a single instance in the middle and the *related* instances up and down the call graph. Color coding shows red for instances which link to or produce no arcs beyond the ones contained initially. Yellow shows arcs from the server points-to graph, and green represents instances which have produced additional arcs after importing arcs from server's global points-to graph.

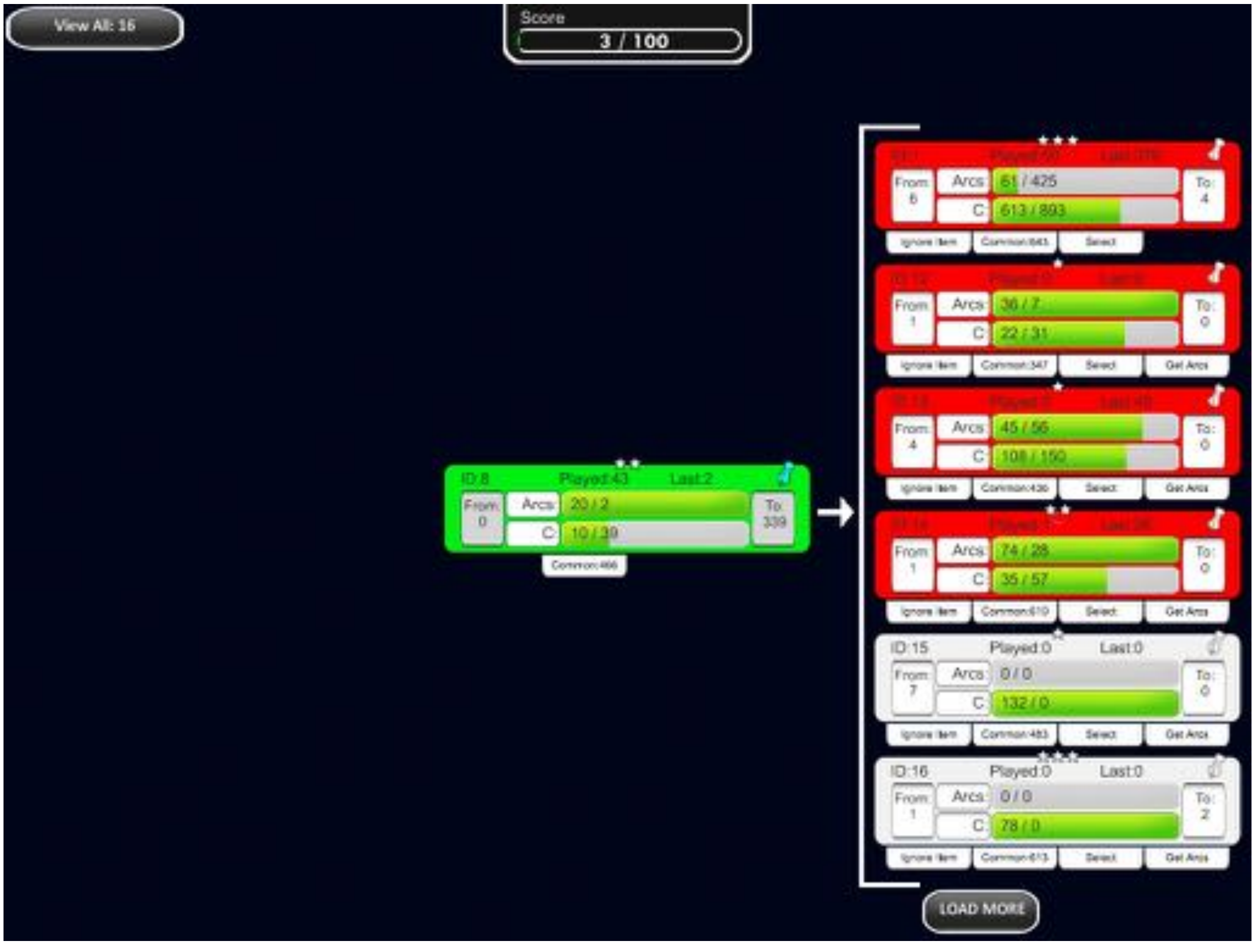


Figure 46: VIPER game play. The grey instances indicate the initial state of instances before being processed by the auto-solver.

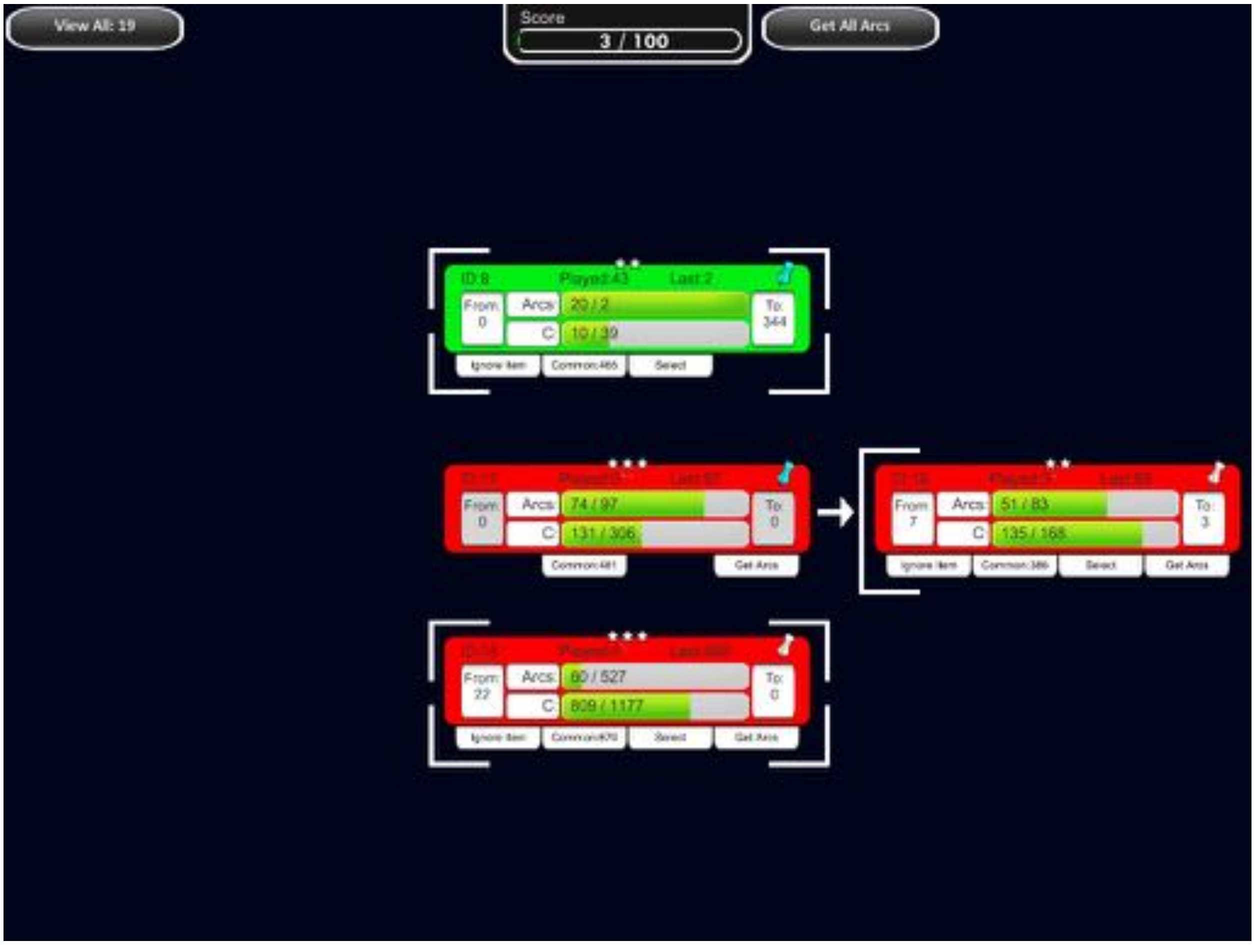


Figure 47: VIPER game play. Workers can tag instances they find interesting so that they remain in the workspace while changing views, as shown with the instance above center. Below center is an instance which contains one or more common node identifiers (called *universal IDs*).

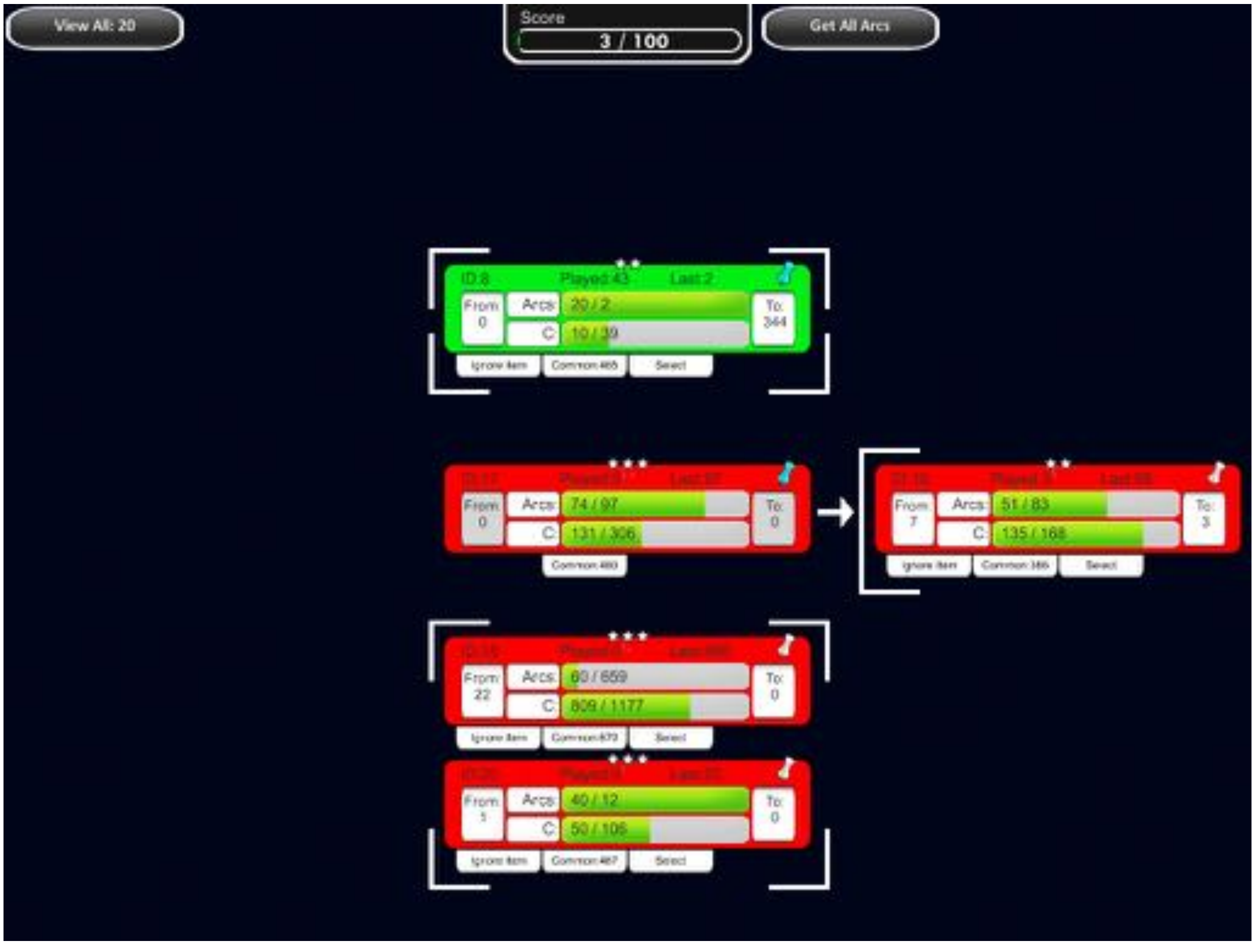
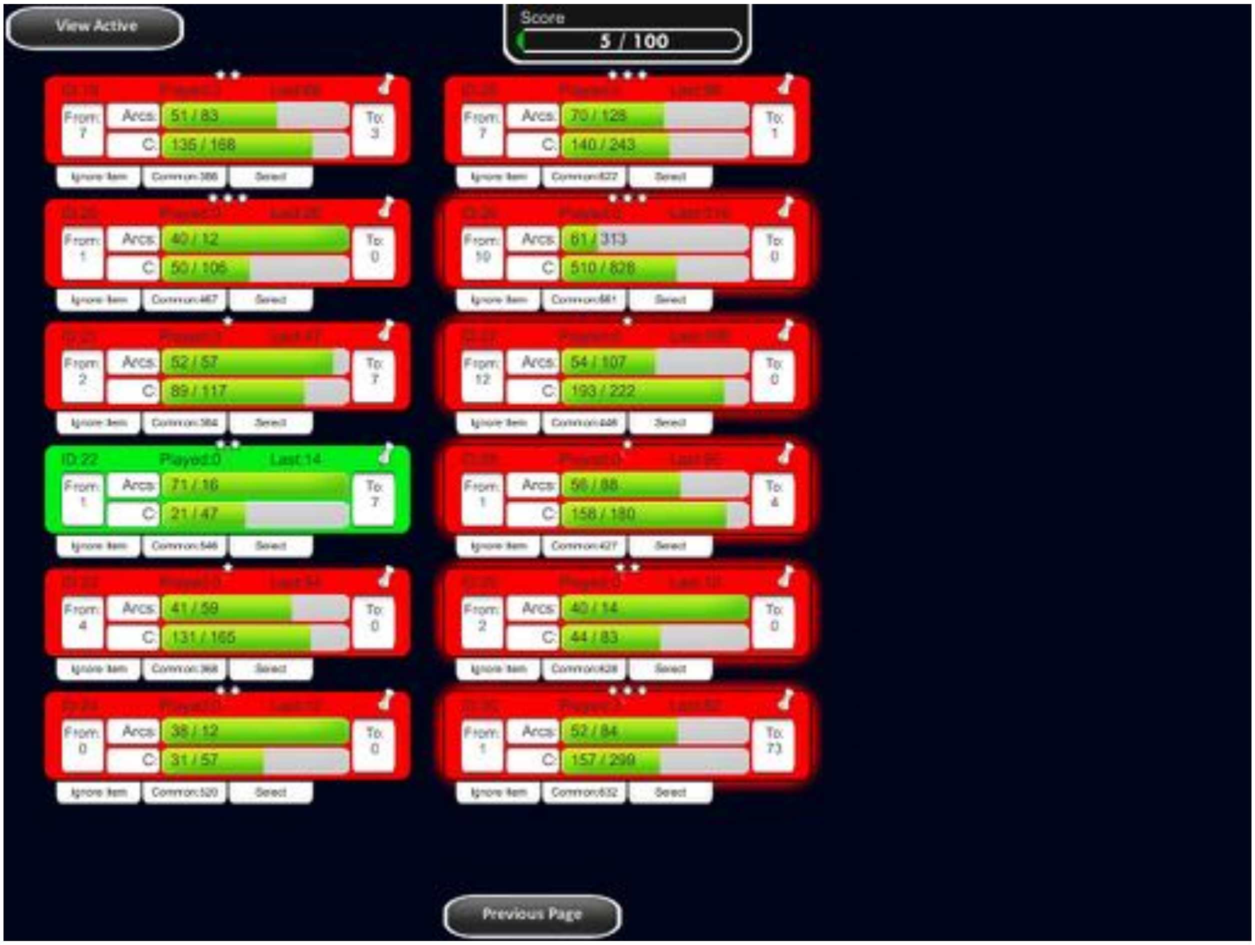


Figure 48: VIPER game play. This worker is importing more instances from the server as referenced by shared graph nodes.



From: 7 Arcs: 51 / 83 C: 135 / 168 To: 3
Ignore Item Common:386 Select

From: 1 Arcs: 40 / 12 C: 50 / 106 To: 0
Ignore Item Common:467 Select

From: 2 Arcs: 52 / 57 C: 89 / 117 To: 7
Ignore Item Common:384 Select

ID:22 Played:0 Last:14
From: 1 Arcs: 71 / 16 C: 21 / 47 To: 7
Ignore Item Common:546 Select

From: 4 Arcs: 41 / 59 C: 131 / 165 To: 0
Ignore Item Common:368 Select

From: 0 Arcs: 38 / 12 C: 31 / 57 To: 0
Ignore Item Common:520 Select

From: 7 Arcs: 70 / 128 C: 140 / 243 To: 1
Ignore Item Common:622 Select

From: 10 Arcs: 61 / 313 C: 510 / 828 To: 0
Ignore Item Common:661 Select

From: 12 Arcs: 54 / 107 C: 193 / 222 To: 0
Ignore Item Common:446 Select

From: 1 Arcs: 56 / 88 C: 158 / 180 To: 4
Ignore Item Common:427 Select

From: 2 Arcs: 40 / 14 C: 44 / 83 To: 0
Ignore Item Common:428 Select

From: 1 Arcs: 52 / 84 C: 157 / 299 To: 73
Ignore Item Common:632 Select

Figure 49: VIPER game play. An alternative view of the imported instances to search for pastes all instances together.

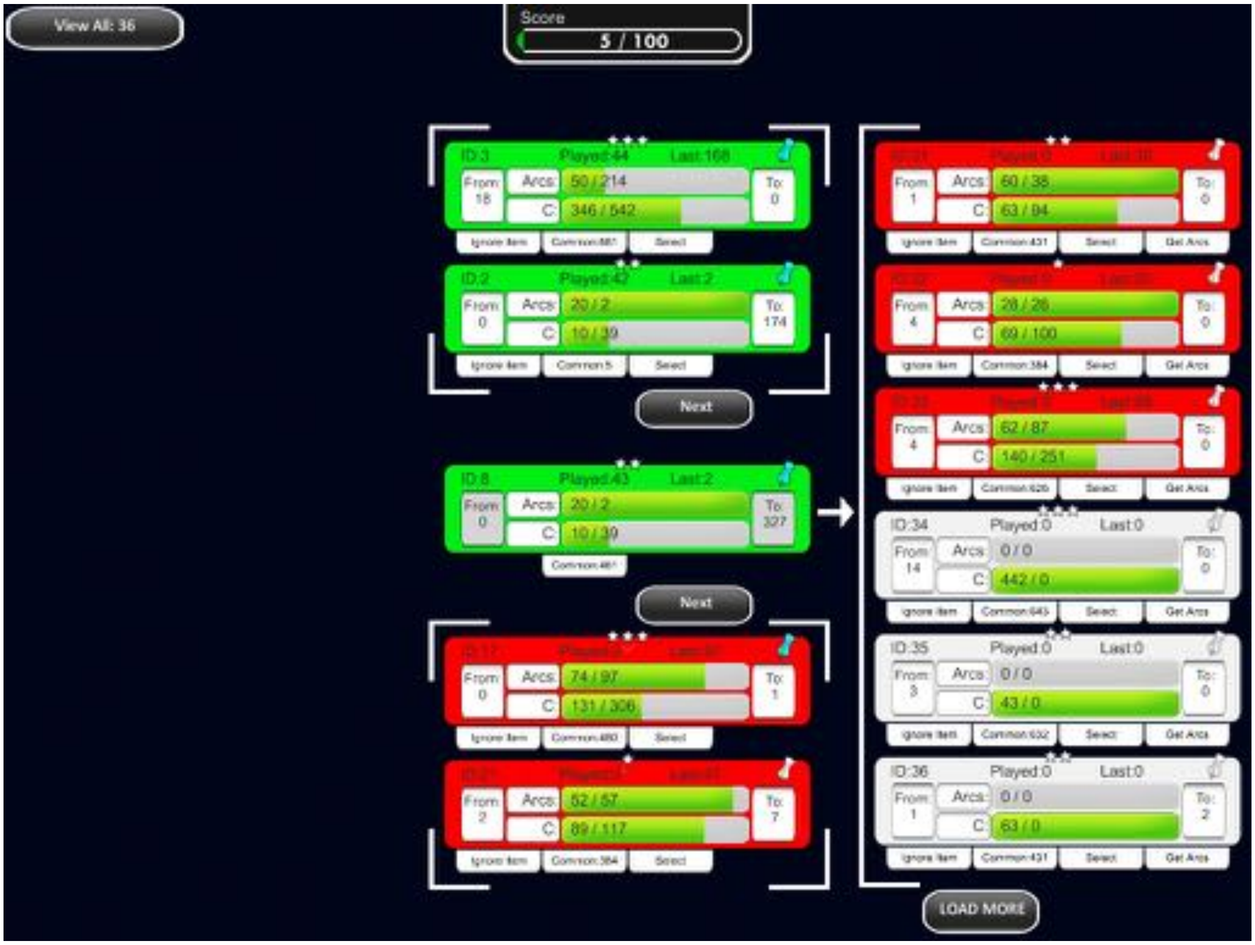


Figure 50: VIPER game play. The player has tagged more instances of interest and is importing the related arcs for one of them.

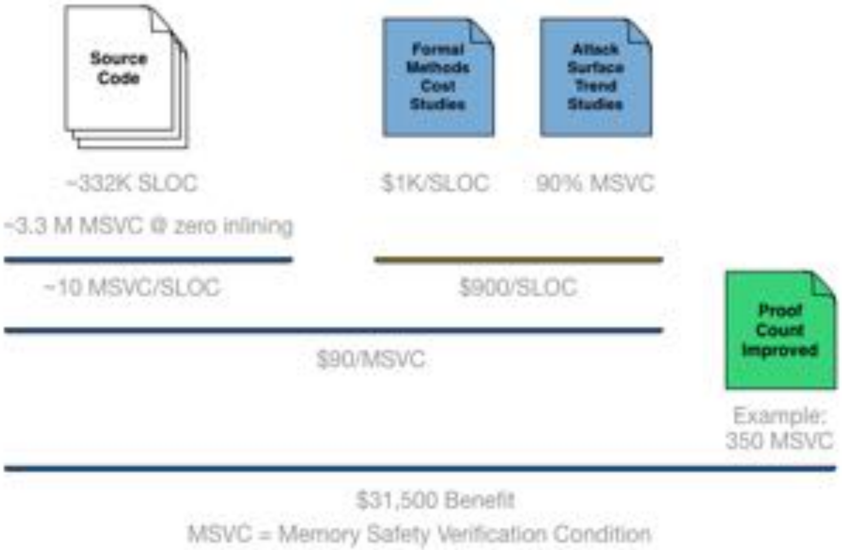

Source Code
~332K SLOC
~3.3 M MSVC @ zero inlining
~10 MSVC/SLOC
Formal Methods Cost Studies
$1K/SLOC
Attack Surface Trend Studies
90% MSVC
$900/SLOC
$90/MSVC
Proof Count Improved
Example: 350 MSVC
$31,500 Benefit
MSVC = Memory Safety Verification Condition

Figure 51: A straight-forward model for the value of a memory safety verification condition (MSVC) based on formal methods studies claiming a cost of $1 thousand per source line of code (SLOC).

4.1 Economic Model

In order to assign a rough monetary value to a verification result obtained with our automated systems we derived some models. Where one agrees with the premises, these models are straight-forward. Our first economic benefit model was based on formal methods lore claiming a cost of one thousand dollars per source line of code ($1K/SLOC) (see Figure 51). If memory safety verification conditions (MSVC) comprise say 90% of the cyber attack surface and associated formal methods effort (including vulnerabilities such as stack corruption, heap corruption, use-after-free, and so on) then the value of the formal methods verification related to memory safety is $900/SLOC. The number of MSVC for a body of source code is fixed, so for BIND we have roughly 332 KSLOC and with zero levels of inlining about 3.3 million MSVC, or an average of 10 MSVC/SLOC. Canceling the SLOC terms yields $90/MSVC. If an automated verification system obtains 350 new proven safe MSVC results at zero cost, its net economic benefit is $31,500. If there was cost associated with computer time or worker reward payments, we would subtract this from the benefit.

Our second model is similar but based on a different premise and study. The study reported expert time rather than expert cost for formal methods at 3.27 SLOC/hr. If we assumed fully burdened expert cost might cost $200/hr, then the formal method effort cost would be $654/SLOC (Figure 52 on the next page). With 10 MSVC/SLOC the cancellation yields $65.40/MSVC and our small case example benefit becomes $22,890.

For an example that includes economic cost consider the case shown in Figure 53 on page 158. The Arcweaver auto-solver solution rate on a particular Amazon Web Services (AWS) Elastic Compute Cloud (EC2) host type `c3.large` is 10,000 MSVC/hr at a computer time cost of $0.10/hr. For the previous example with 10 MSVC/SLOC at the study rate of 3.27 SLOC/hr, this yields a required 32.7 MSVC/hr at $200/hr expense for the expert

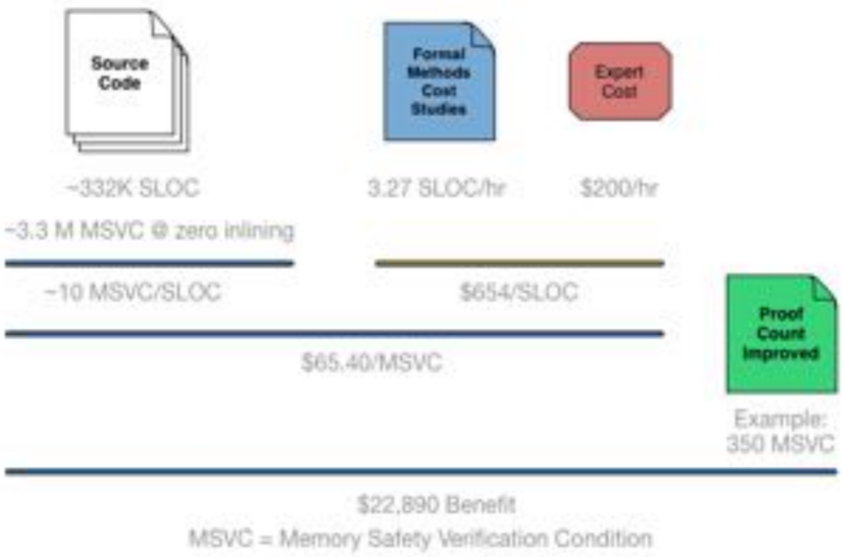


Figure 52: A straight-forward model for the value of a memory safety verification condition (MSVC) based on a source line of code (SLOC) per hour rate for formal methods.

time. The Arcweaver production rate is 10,000 MSVC/hr at a cost of $0.10/hr. For the example study case of 350 MSVC improvements, the expert cost would be $21,406.73 while the equivalent Arcweaver cost would be $0.0035. In fact the EC2 lease time is rounded up to the next whole hour so the imposed cost for a solitary job would be ten cents.

With CodeHawk analysis we usually employ inlining to increase the verification resolution. When deeper levels of inlining are used the number of MSVC per SLOC increases as will be seen below in the Arcweaver results (Section **??** on page ??). Consequently the dollar value per verification condition decreases by these models, but the total number of verification conditions increases as well. Generally the results improve as well and we expect to obtain more verification conditions improved even at a reduced individual value for an overall larger economic benefit. The Arcweaver results tables illustrate this difference.

4.2 CircuitBot Results

CircuitBot was our Phase I game. The game provided a space colonization mission backstory intended to retain and incentivize players and thereby generate increased program verification production. We loaded a large volume of verification game candidates onto the backend to initialize the content, and the CircuitBot backend delivered these candidates to the players using its game instance prioritization algorithm. This section presents a number of statistical results collected from game play through our backend and the TA3-provided Logaholics server. These data were collected starting with the "friends and family" release on 24 October 2013 or the public live release date of 2 December 2013.

Figure 54 on page 160 shows the spot statistics for the number of unique visitors (by IP address), page views, and pages per user over time for the CircuitBot game beginning with its live release on 2 December 2013. Figure 55 on page 161 provides the cumulative

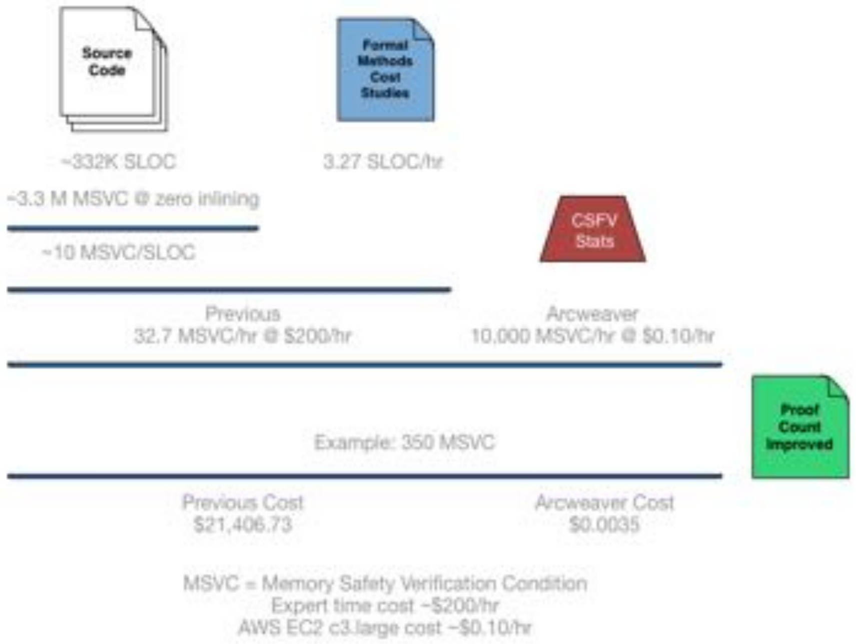

Source Code
Formal Methods Cost Studies
~332K SLOC
3.27 SLOC/hr
~3.3 M MSVC @ zero inlining
CSFV Stats
~10 MSVC/SLOC
Previous
32.7 MSVC/hr @ $200/hr
Arcweaver
10,000 MSVC/hr @ $0.10/hr
Proof Count Improved
Example: 350 MSVC
Previous Cost
$21,406.73
Arcweaver Cost
$0.0035
MSVC = Memory Safety Verification Condition
Expert time cost ~$200/hr
AWS EC2 c3.large cost ~$0.10/hr

Figure 53: A straight-forward model for the value of a memory safety verification condition (MSVC) based on a source line of code (SLOC) per hour rate for formal methods. The Arcweaver auto-solver solution rate on a particular host type is 10,000 MSVC/hr at a computer time cost of $0.10/hr.

count representation of these statistics. Clearly the visitor traffic was highest just after the live release, likely associated with the news generated by the CSFV Program and the Verigames site, press releases, and social media. There were a few ephemeral traffic spikes in traffic later on, and these also were directly associated with news releases. While playing the game the player contributes verification results in the form of points-to graph arcs. Figure 56 on page 162 shows the spot statistics for graph arcs generated by day along with the number of game instances (game levels) played each day. Figure 57 on page 163 provides the cumulative count representation of these statistics. Because there were hundreds of thousands of unique game instances that must be played several times over in order to complete a fixpoint iteration, these data show that the volume of traffic was insufficient to achieve a fixpoint solution in a reasonable amount of time; this finding informed our subsequent game designs. Figure 58 on page 164 shows some of the internal metrics we collected about the game mechanics and play experience, intended to help us determine where bottlenecks or obstacles might occur. The figure shows no particular problems, although we did use a figure like this early in the project to collect data about player loss during the tutorial flow.

Ranking the players by contributions we can learn whether a few players contributed most of the results. We measured verification productivity the number of arcs the player contributed to the points-to graph for the program under analysis. After the graph reached a fixed point solution (which we measured by both arc and interval stability) then we could compute the total size of the graph and each player's individual contribution. The measure also enabled an in-progress contribution merely by computing the player's fraction of the total graph arcs generated so far. Figure 59 on page 165 ranks the individual registered players by their total arc production for all games played. There were over 300 registered players for CircuitBot, as well as over 1,000 anonymous players for which we did not collect data. The figure highlights a region in orange to indicate that a small proportion of registered players produced 80% of the total volume of arcs for the game's graph iteration. The *top producers* annotation provides the actual count included in this region. Similarly, because we know we have a finite number of game levels contributing constraints to the constraint problem the players are solving, we can compute how many of these game levels have been consumed. Figure 60 on page 166 ranks the registered players by their total game instance consumption for all days played. It highlights a region in orange to indicate that a small proportion of players is required to consume 80% of the game instances; the annotation *top consumers* provides the actual count included in this region. Figure 61 on page 167 depicts the cumulative play time (in seconds) experienced to produce these arcs and consume these game instances, measured as the total session time including the backstory and analysis time. Finally, Figure 62 on page 168 shows one measure of CircuitBot player retention reported during the course of the program. For each registered player ID we counted the number of different days that player contributed game results. The player may have contributed many results on a given day but we do not distinguish single-day contributions. For CircuitBot we did not track or collect verification results for anonymous player contributions. These data reveal that the CircuitBot game had a few "whale" players that returned many days and contributed most of the productive work.

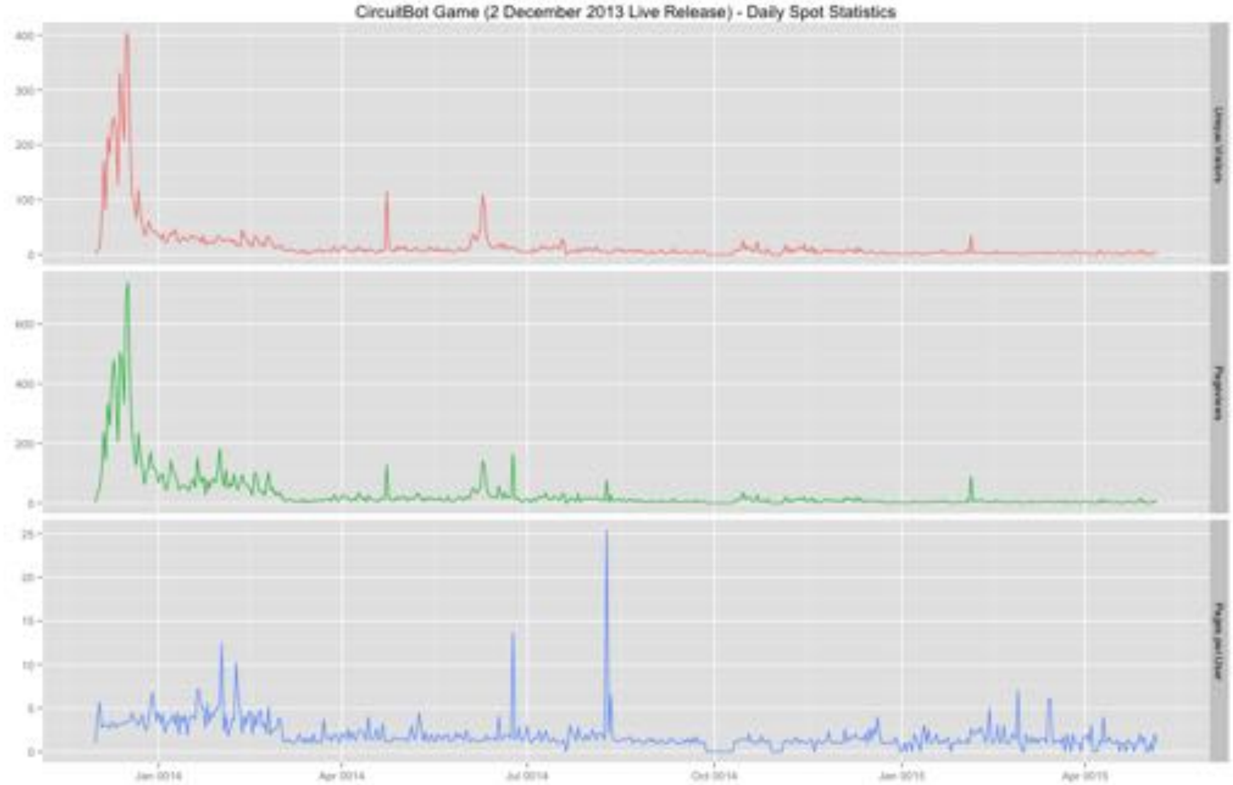


Figure 54: CircuitBot game daily visitor and page view counts over time. Samples from 2 December 2013 live release through 15 August 2015.

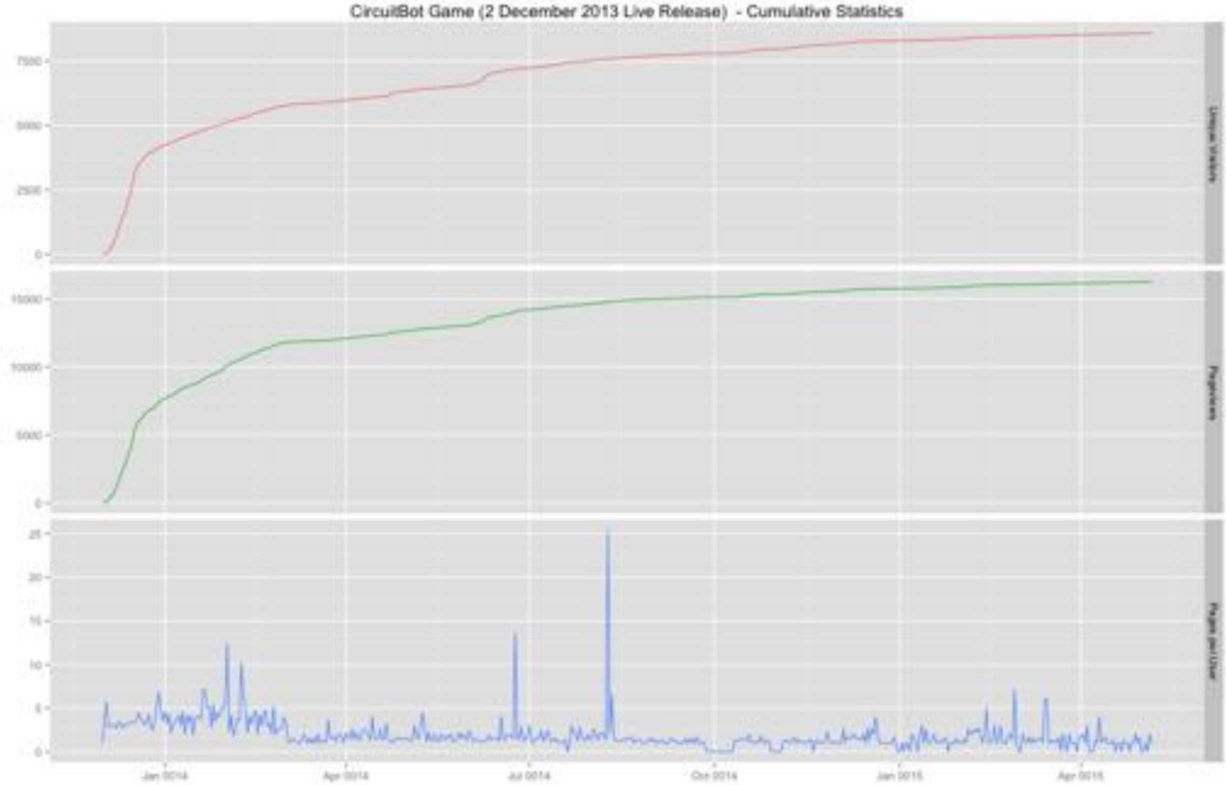


Figure 55: CircuitBot game daily visitor and page view counts accumulated over time. Samples from 2 December 2013 live release through 15 August 2015.

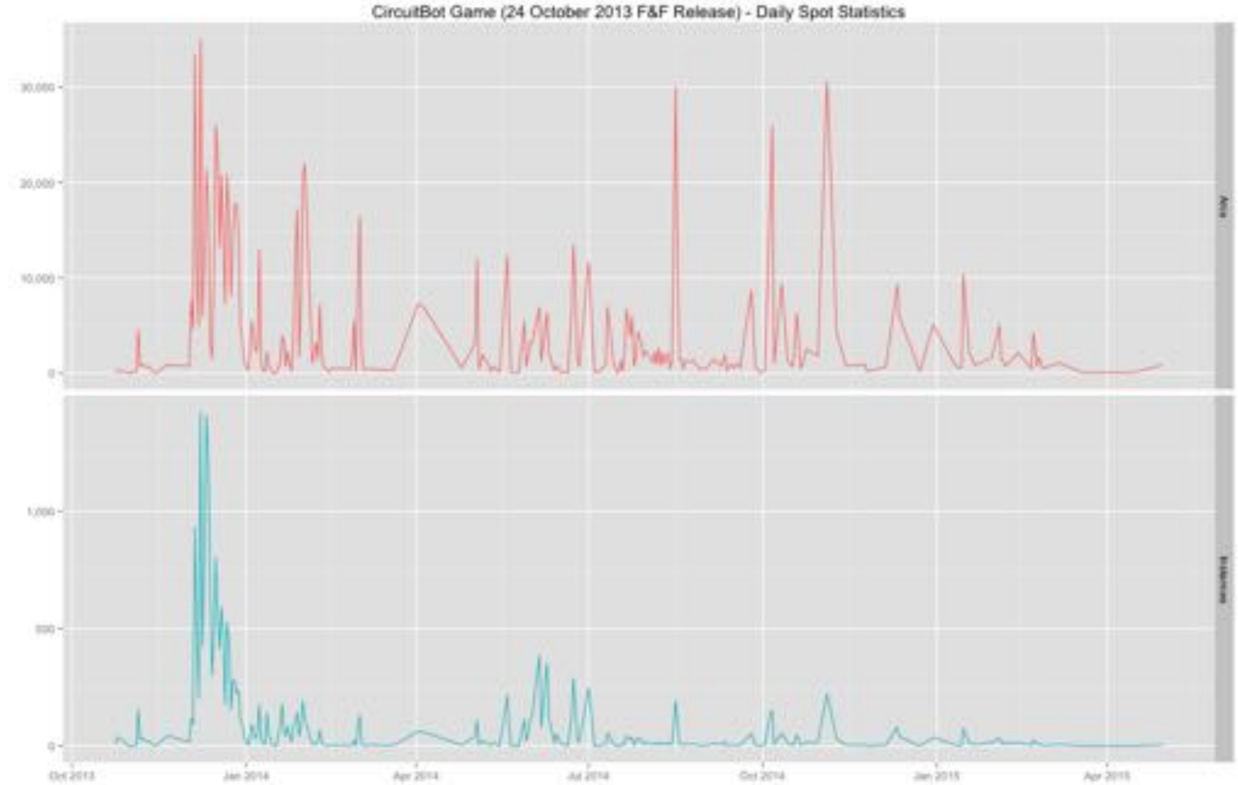


Figure 56: CircuitBot arc generation and game instance consumption daily counts over time. Samples from 24 October 2013 Friends and Family release through 15 August 2015.

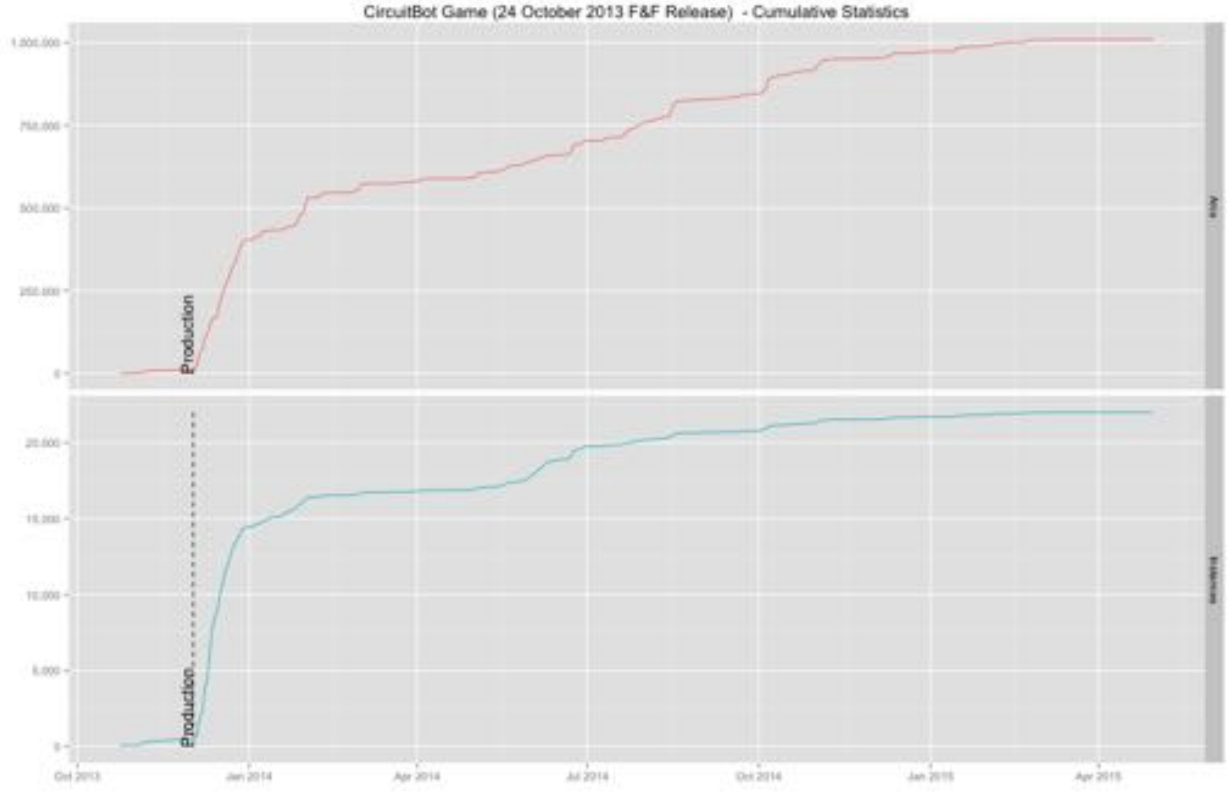


Figure 57: CircuitBot arc generation and game instance consumption daily counts accumulated over time. Samples from 24 October 2013 Friends and Family release through 15 August 2015.

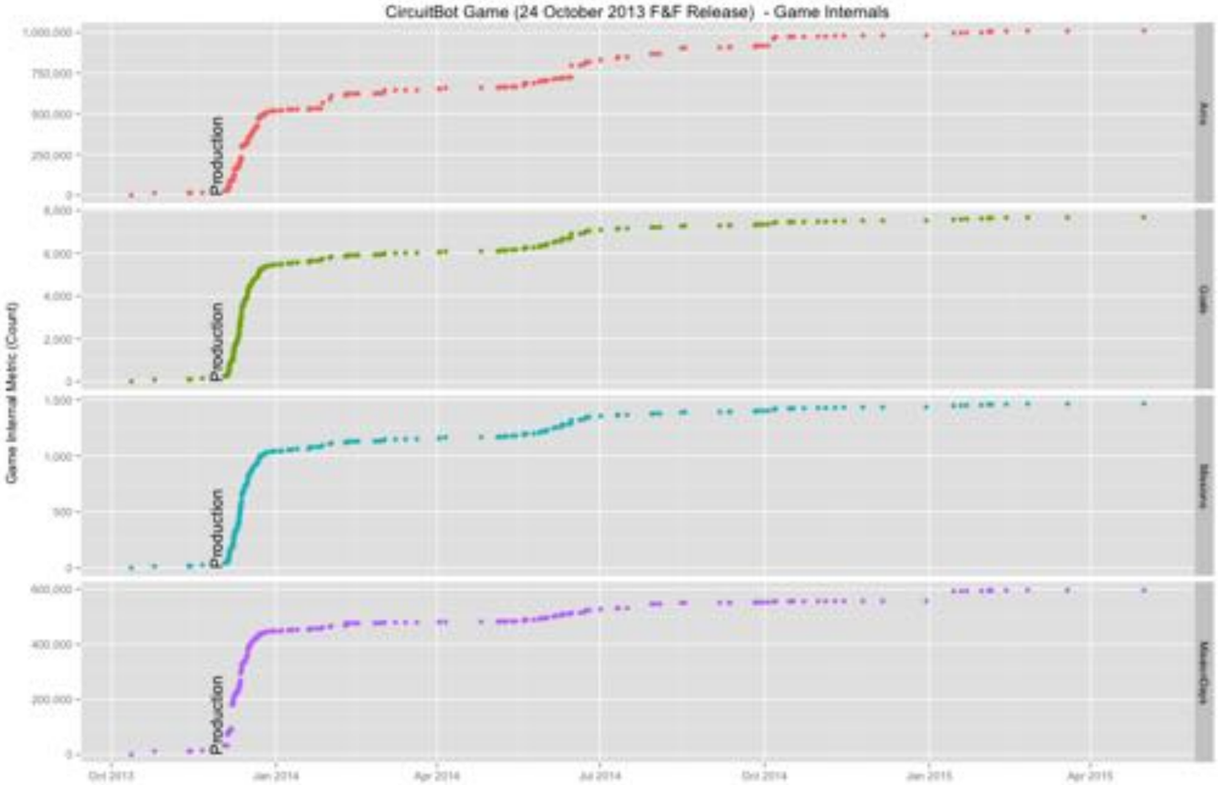


Figure 58: CircuitBot game internal metrics over time. Internal metrics tracked regarding game instance activity and back story progress included arcs generated, mission goals achieved, missions completed, and mission days played. Samples from 24 October 2013 Friends and Family release through 15 August 2015.

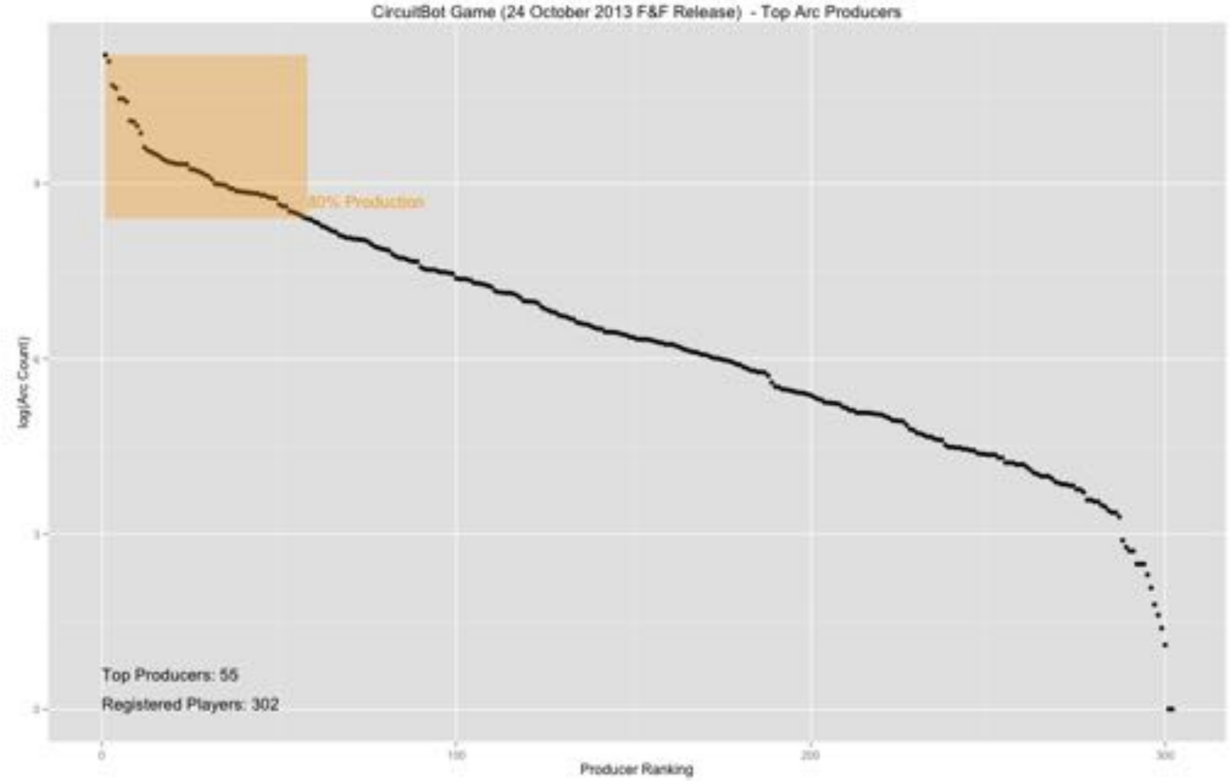


Figure 59: CircuitBot game top arc producers. Ranked high to low from left to right, the top few players produced 80% of all the productive results (points-to graph arcs) for the game. The CircuitBot game did not enable anonymous players to contribute game results so these data were not collected. Samples from 24 October 2013 live release through 15 August 2015.

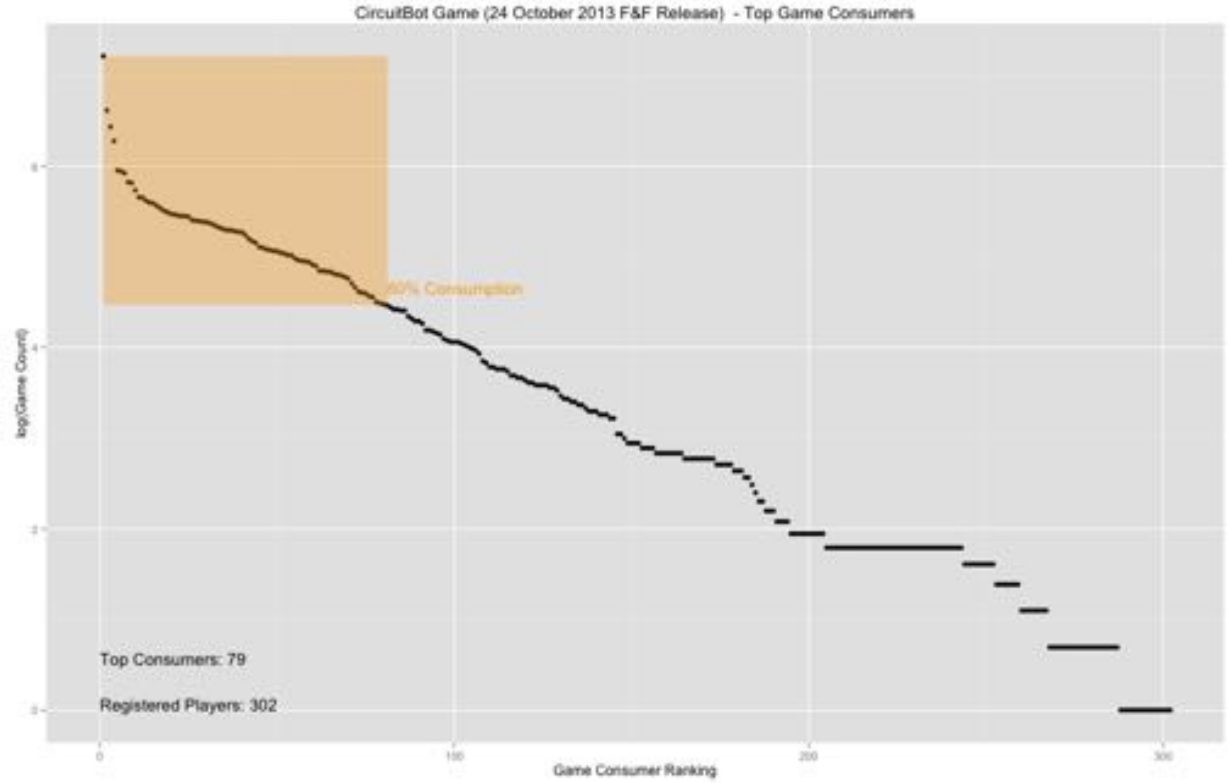


Figure 60: CircuitBot game top game consumers. Ranked high to low from left to right, the top few players consumed 80% of all the constraints problem game instances. The CircuitBot game did not enable anonymous players to contribute game results so these data were not collected. Samples from 24 October 2013 Friends and Family release through 15 August 2015.

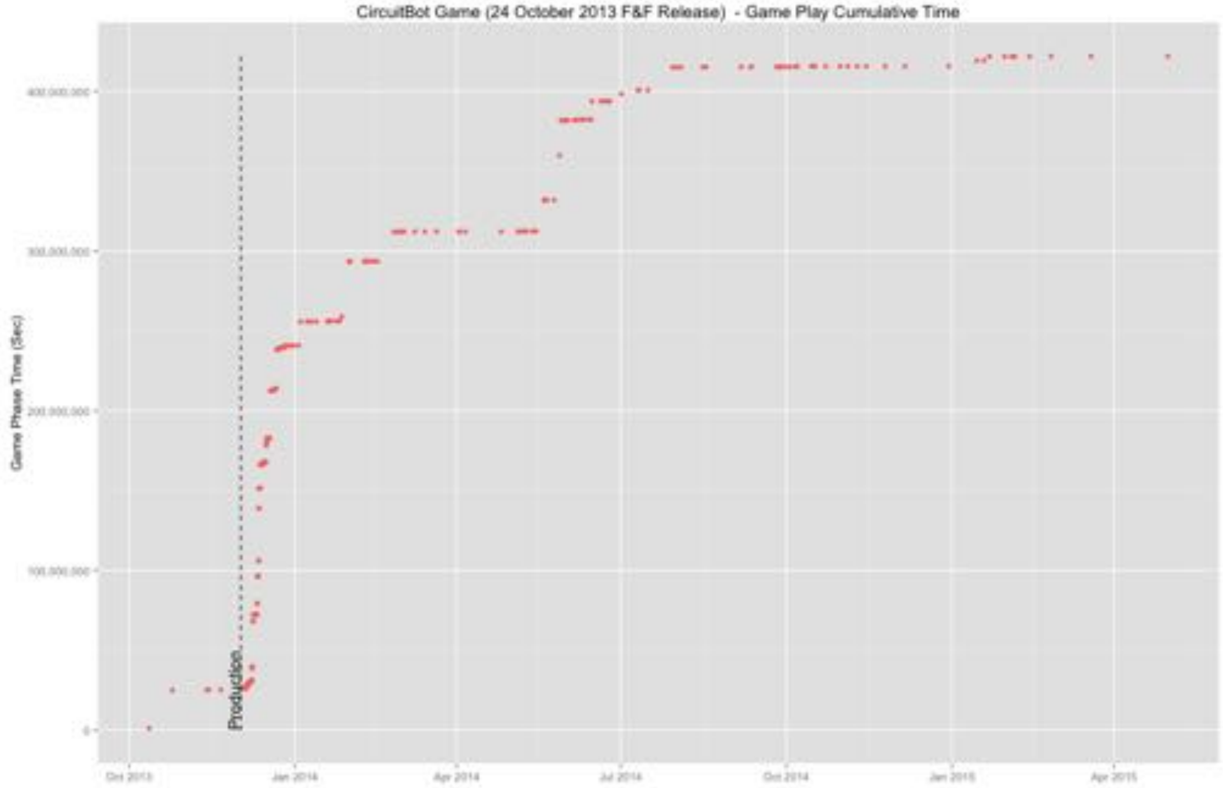


Figure 61: CircuitBot game cumulative play time (seconds). In the CircuitBot game the play time collected was the combined session time which included backstory time and analysis time. Samples from 24 October 2013 Friends and Family release through 15 August 2015.

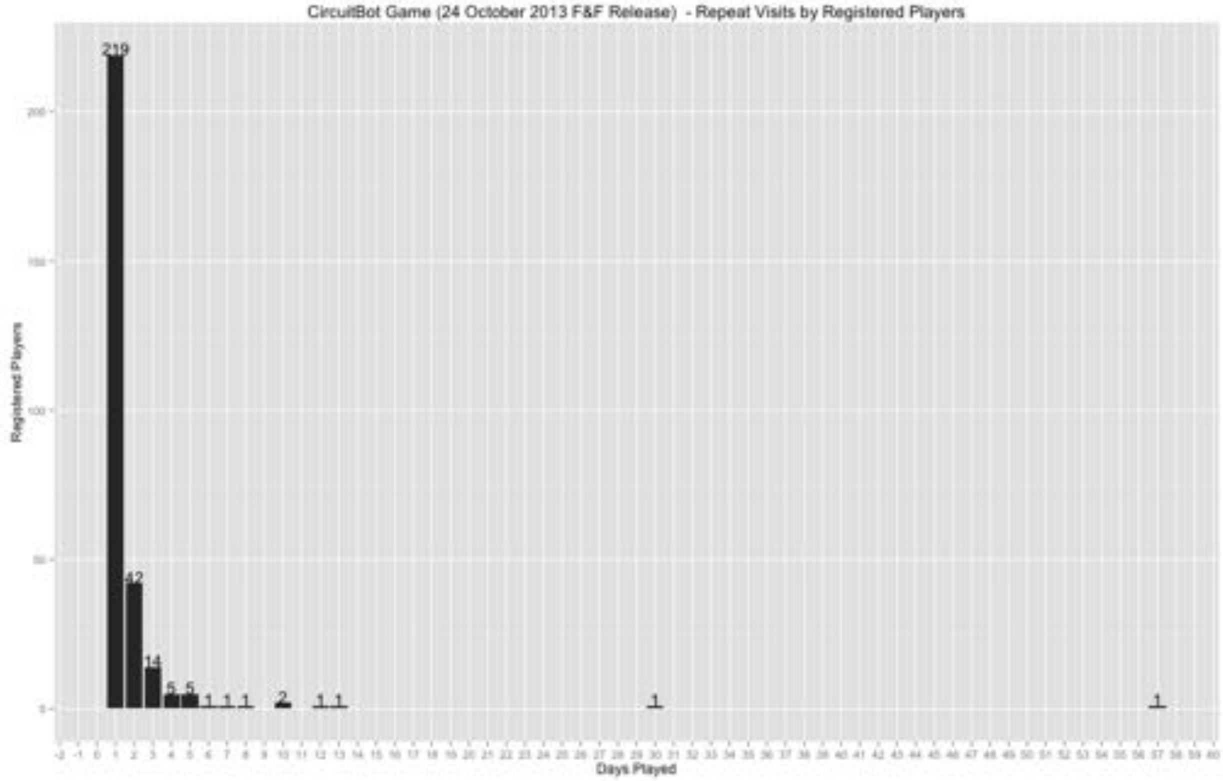


Figure 62: CircuitBot game player retention represented as number of days played (repeat visits) per unique player identifier. Within each day a player may have played the game one or more times. Samples from 24 October 2013 Friends and Family release through 15 August 2015.

4.3 Dynamakr Results

Dynamakr was our Phase II game. We designed it with lessons learned from CircuitBot in mind, and with an eye toward a possible Amazon Mechanical Turk game in the future. Dynamakr stripped away much of the backstory elements of the game, which were designed for player retention, and set out for more immediate capture of productive work with higher integration between human players and robot auto-solvers. This section presents a number of statistical results collected from game play through our backend and the TA3-provided Logaholics server. These data were collected starting with the "friends and family" release on 10 April 2015 or the public live release date of 27 May 2015.

Figure 63 on page 171 shows the spot statistics for the number of unique visitors (by IP address), page views, and pages per user over time for the Dynamakr game beginning with its live release on 27 May 2015. Figure 64 on page 172 provides the cumulative count representation of these statistics. The visitor traffic was high just after the live release, likely associated with the news generated by the CSFV Program and the Verigames site, but there was also a nearly equally-sized spike in traffic associated with a BBC article. There were a few ephemeral traffic spikes in traffic later on, and these also were directly associated with news releases. As with the CircuitBot game, while playing the Dynamakr game the player contributes verification results in the form of points-to graph arcs. Unlike the CircuitBot game, during the Dynamakr game the anonymous players can contribute valid results to the collection but we do not associate them with particular persons. Figure 65 on page 173 shows the spot statistics for graph arcs generated by day along with the number of game instances (game levels) played each day. Figure 66 on page 174 provides the cumulative count representation of these statistics. Interestingly, even though the BBC-stimulated traffic perked up the visitors, these visitors did not contribute to game results. Like CircuitBot, because there were hundreds of thousands of unique game instances that must be played several times over in order to complete a fixpoint iteration, these data show that the volume of traffic was insufficient to achieve a fixpoint solution in a reasonable amount of time; this finding informed our subsequent game designs. However, unlike CircuitBot, during Dynamakr we used auto-solvers to work off a large proportion of the easier game problems in order to eliminate the tedium from the human game play experience. Consequently, the games were more interesting for the players and the problems presented required more human intuition to solve. During Dynamakr we loaded only one or two BIND modules at a time in order to try to achieve a fixed point solution for a verifiable module, rather than loading all BIND modules at one time and trying to solve all of them to fixpoint solution simultaneously.

As with CircuitBot, by ranking the Dynamakr players by contributions we learn whether a few players contributed most of the results. We measured verification productivity the number of arcs the player contributed to the points-to graph for the program under analysis. Figure 67 on page 175 ranks both the anonymous and the individual registered players by their total arc production for all games played. The figure shows all anonymous players (over 750 as shown in the annotation) bundled into one data point, while ranking the registered players individually. The figure highlights a region in orange to indicate that a small proportion of contributors produced 80% of the total volume of arcs for the game's graph iteration. The *top producers* annotation provides the actual count included in this region. If we unbundle the anonymous players as in Figure 68 we can see the contribution of a small number of registered players exceeds the relative contribution of anonymous contributors. Similarly, because we know we have a finite number of game levels contributing constraints to the constraint problem the players are solving, we can compute how many of these game levels have been consumed. Figure 69 on page 177 ranks the anonymous and registered players by their total game instance consumption for all days played, with anonymous players bundled into one point. It highlights a region in orange to indicate that a small proportion of players is required to consume 80% of the game instances. Figure 70 on page 178 provides the game consumption ranking with anonymous players unbundled.

To help examine game productivity, Figure 71 on page 179 shows some of the internal metrics we collected about the game mechanics and play experience. Figure 72 on page 180 shows the cumulative graph arc production over the course of new game instances seen by players. Eventually the game would continue to generate new arcs even after no new game instances were seen as the fixpoint iteration continues to resolve the graph arcs and intervals. Figure 73 on page 181 depicts the cumulative phase time (in seconds) experienced to produce the game results. The Dynamakr game included both a productive *work* phase (analysis time) and a non-productive *play* phase (dynamo time), the latter intended to reward the help to retain the player.

Finally, Figure 74 on page 182 shows one measure of Dynamakr player retention reported during the course of the program. For each registered player ID we counted the number of different days that player contributed game results. The player may have contributed many results on a given day but we do not distinguish single-day contributions. These data reveal that the Dynamakr game had a few “whale” players that returned several days and contributed most of the productive work, although not as many as the CircuitBot game (moreover Dynamakr had not been running as long as CircuitBot).

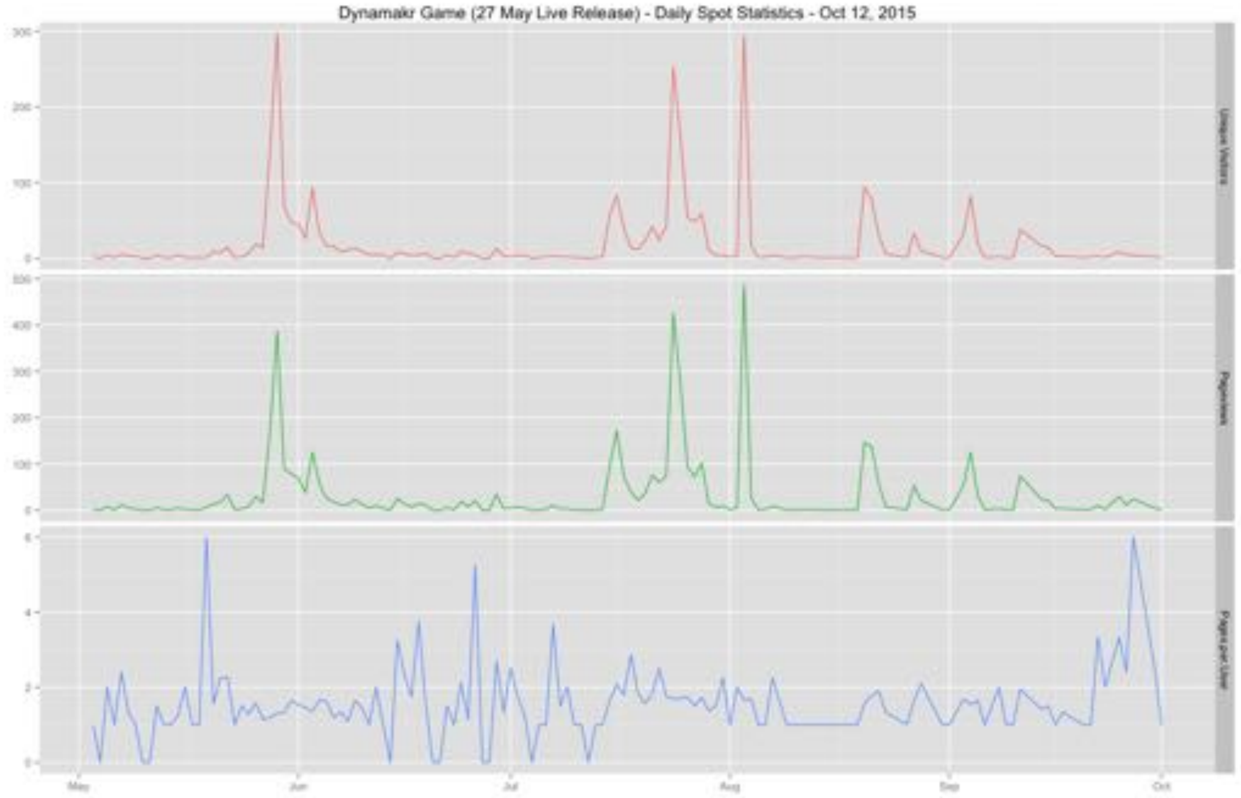


Figure 63: Dynamakr game daily visitor and page view counts over time. Samples from 27 May 2015 live release through 12 October 2015.

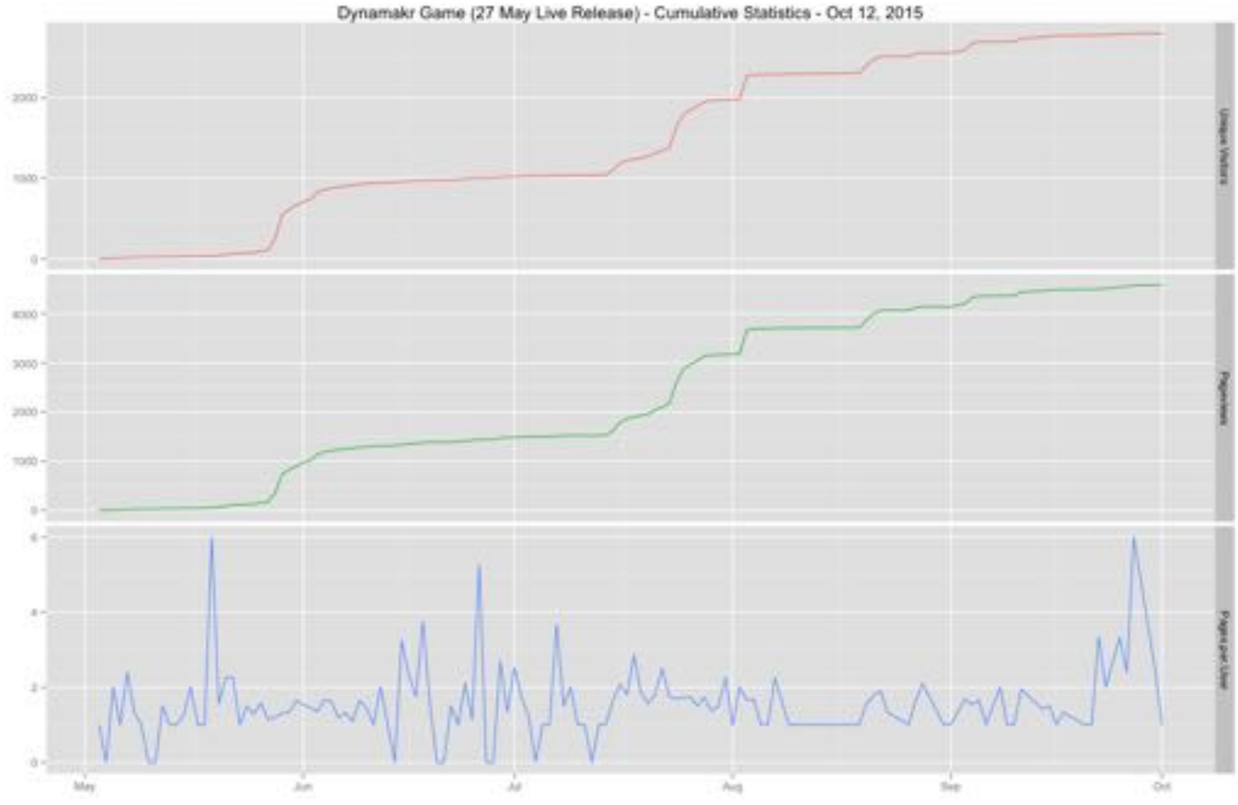


Figure 64: Dynamakr game daily visitor and page view counts accumulated over time. Samples from 27 May 2015 live release through 12 October 2015.

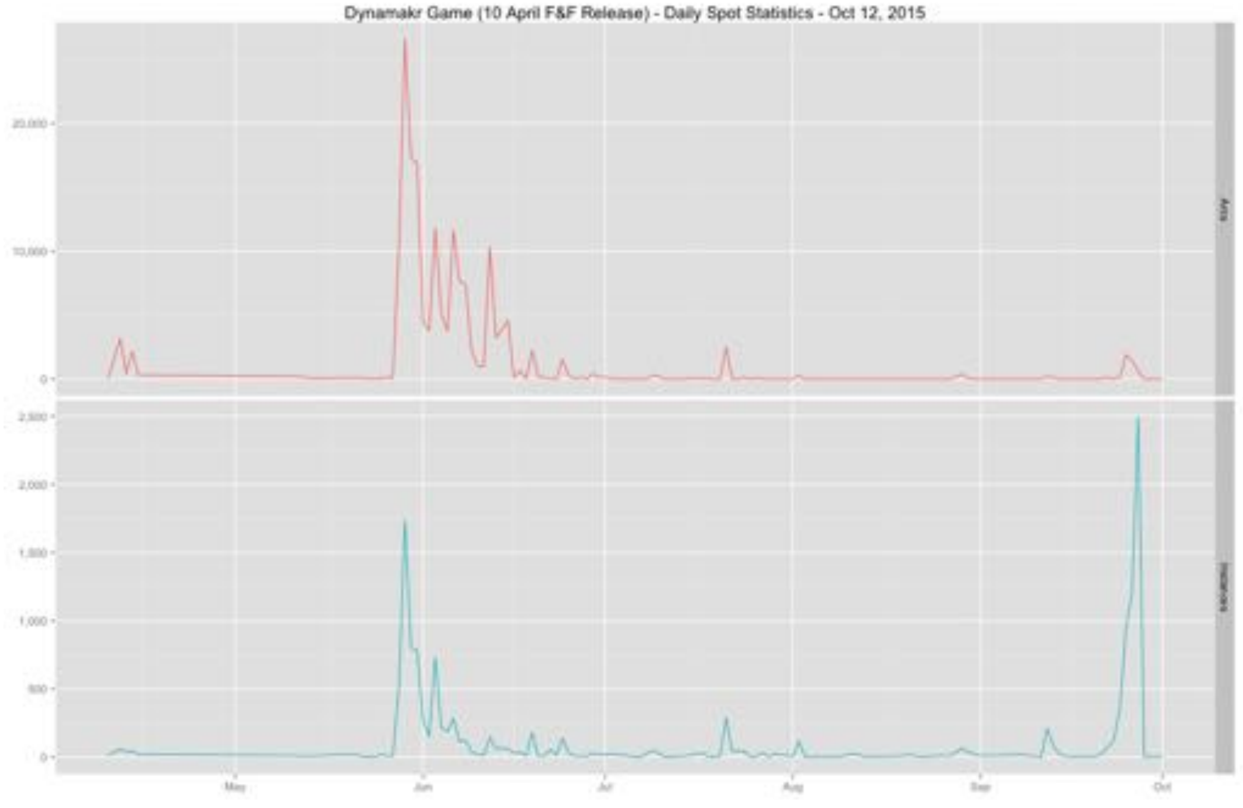


Figure 65: Dynamakr arc generation and game instance consumption daily counts over time. Samples from 10 April 2015 Friends and Family release through 12 October 2015.

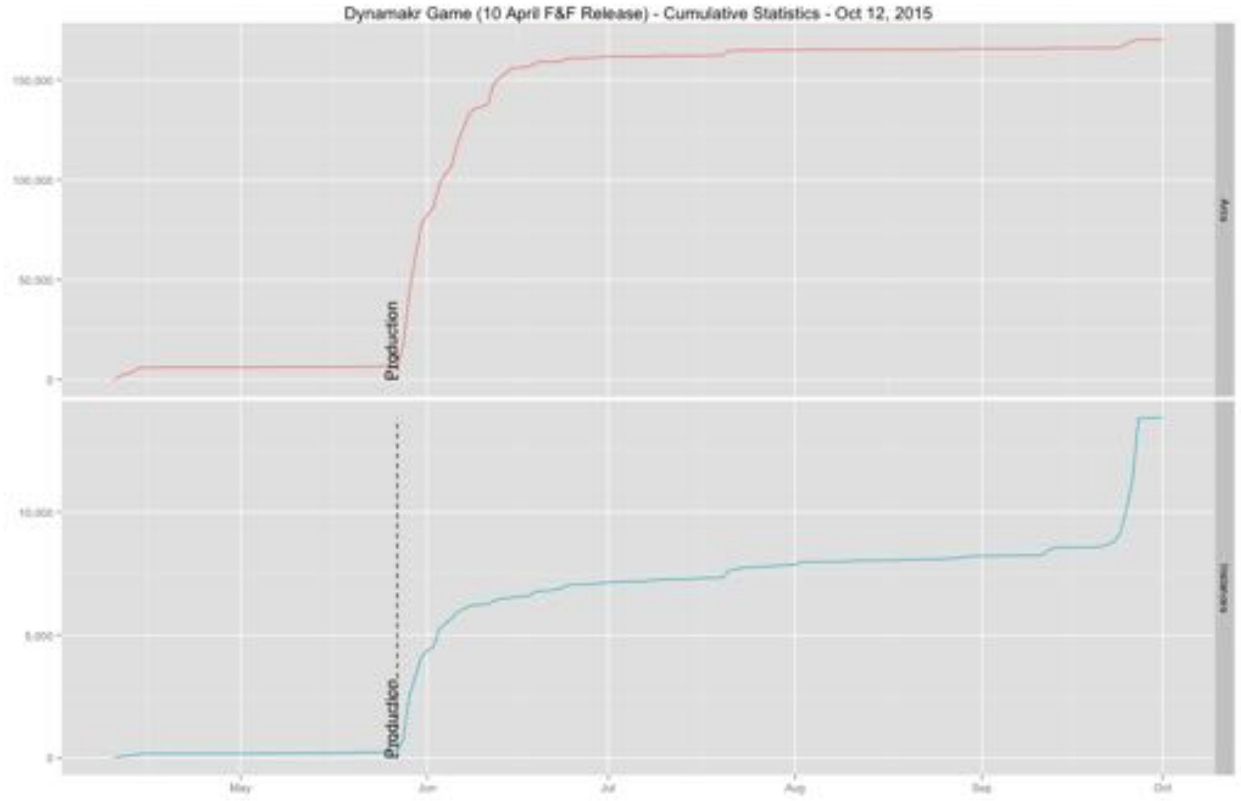


Figure 66: Dynamakr arc generation and game instance consumption daily counts accumulated over time. Samples from 10 April 2015 Friends and Family release through 12 October 2015.

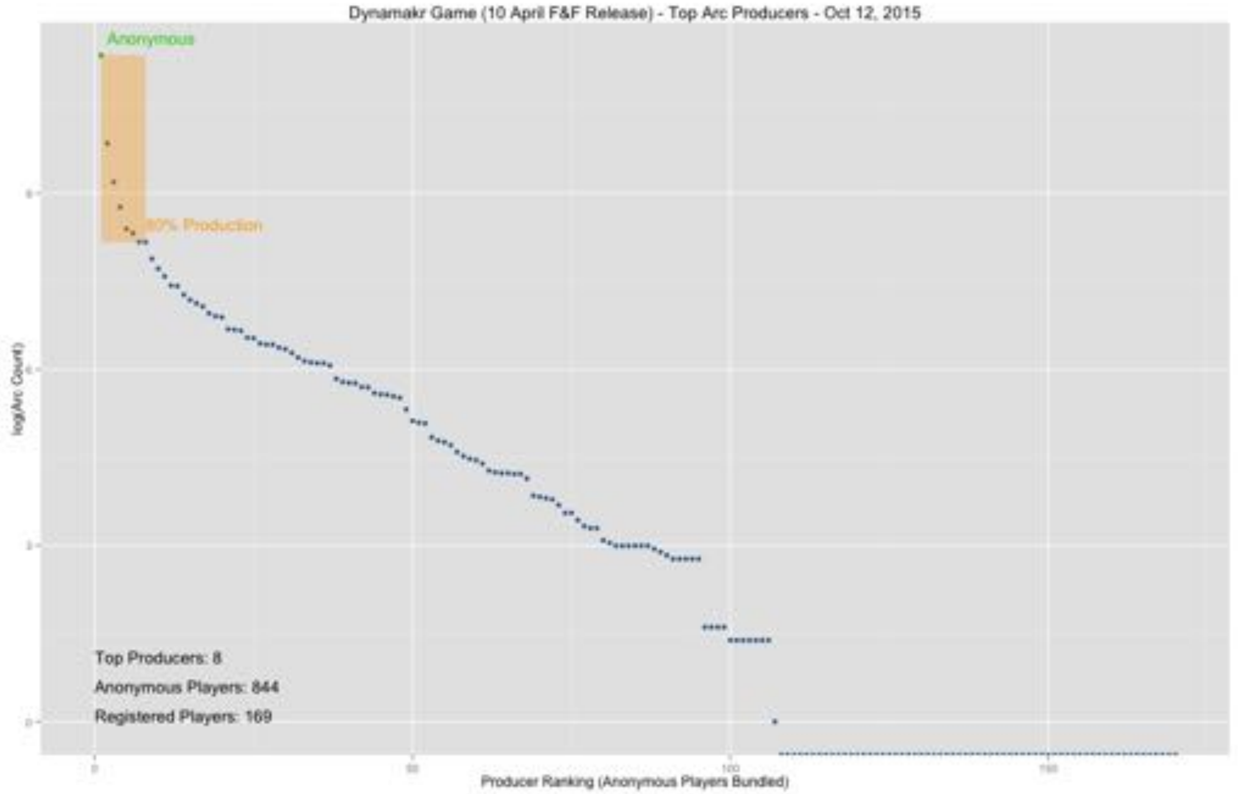


Figure 67: Dynamakr game top arc producers. Ranked high to low from left to right, the top few players produced 80% of all the productive results (points-to graph arcs) for the game. This plot bundles all the anonymous players into one point. Samples from 27 May 2015 live release through 12 October 2015.

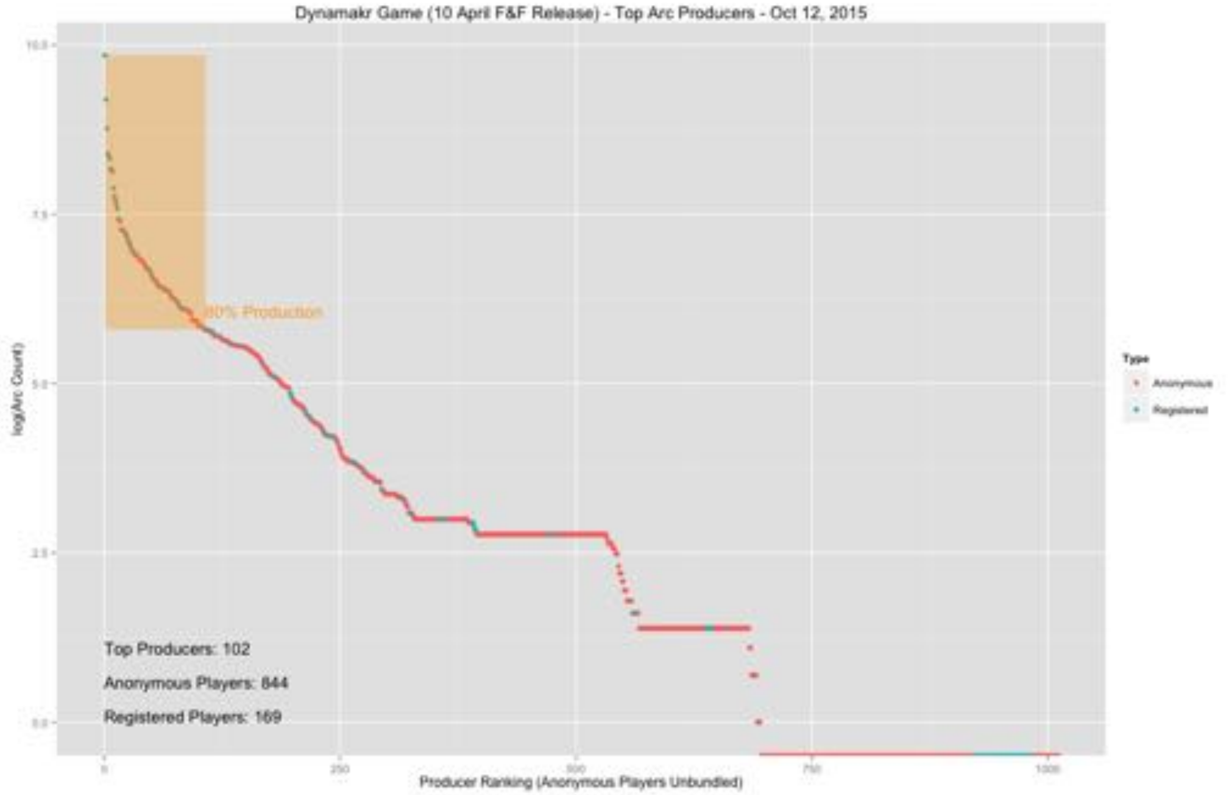


Figure 68: Dynamakr game top arc producers. Ranked high to low from left to right, the top few players produced 80% of all the productive results (points-to graph arcs) for the game. This plot unbundles all the anonymous players into separate points in order to see their contribution in relation to the registered players. Samples from 27 May 2015 live release through 12 October 2015.

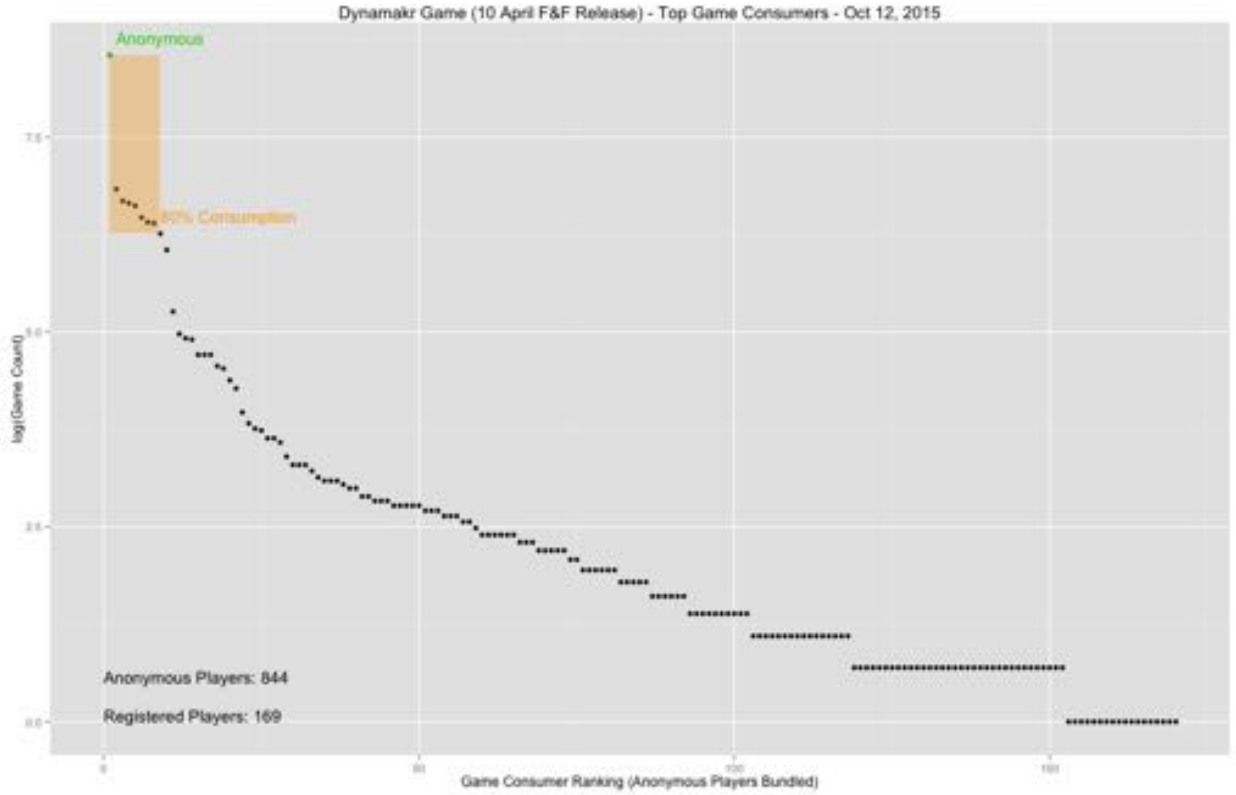


Figure 69: Dynamakr game top game consumers. Ranked high to low from left to right, the top few players consumed 80% of all the constraints problem game instances. This plot bundles all the anonymous players into one point. Samples from 27 May 2015 live release through 12 October 2015.

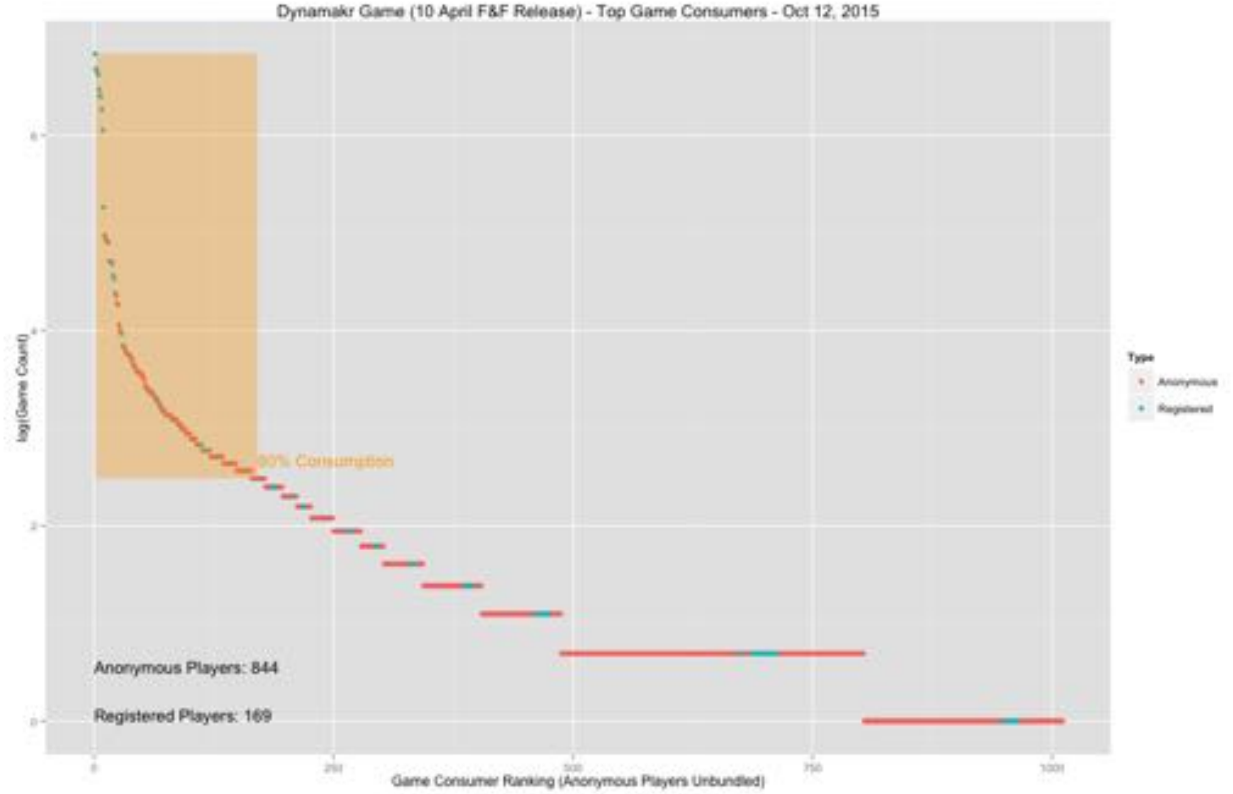


Figure 70: Dynamakr game top game consumers. Ranked high to low from left to right, the top few players consumed 80% of all the constraints problem game instances. This plot unbundles all the anonymous players into separate points in order to see their contribution in relation to the registered players. Samples from 27 May 2015 live release through 12 October 2015.

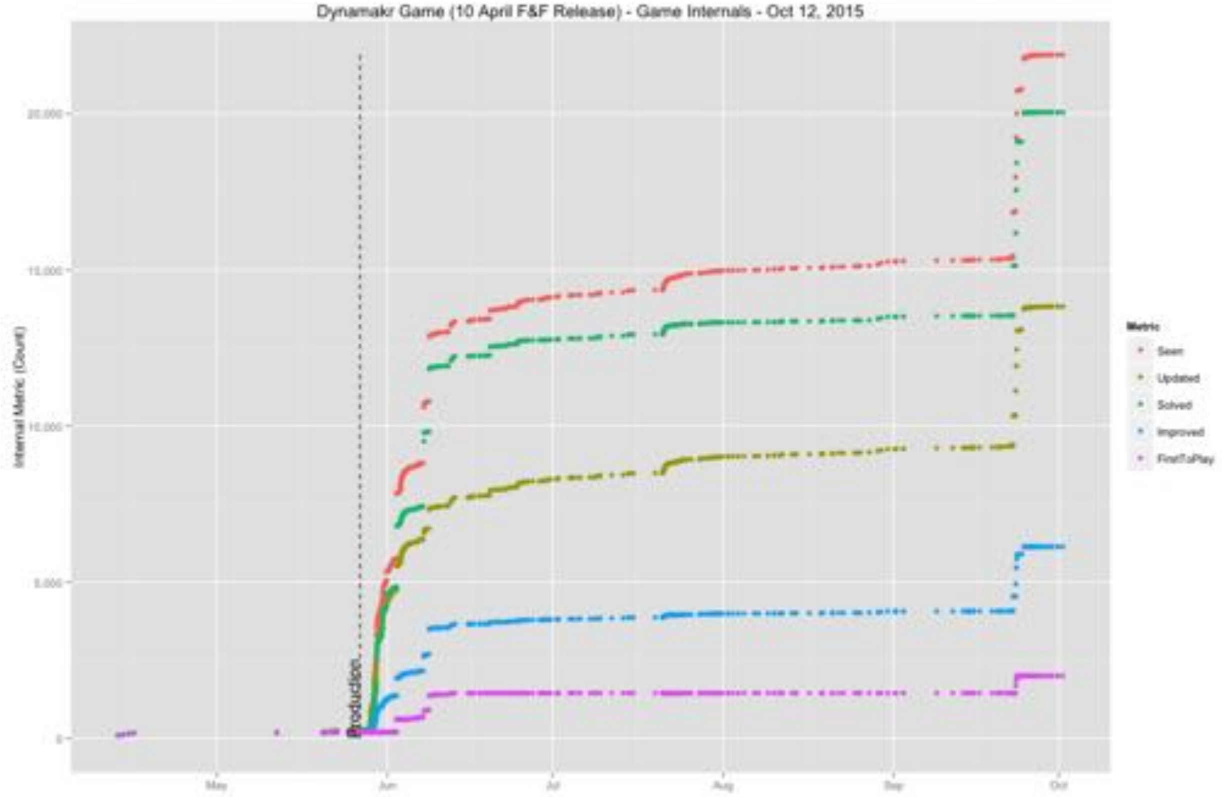


Figure 71: Dynamakr game internal metrics over time. Internal metrics tracked regarding game instance activity included game instances seen, updated, solved, improved, and first-played counts. Samples from 10 April 2015 Friends and Family release through 12 October 2015.

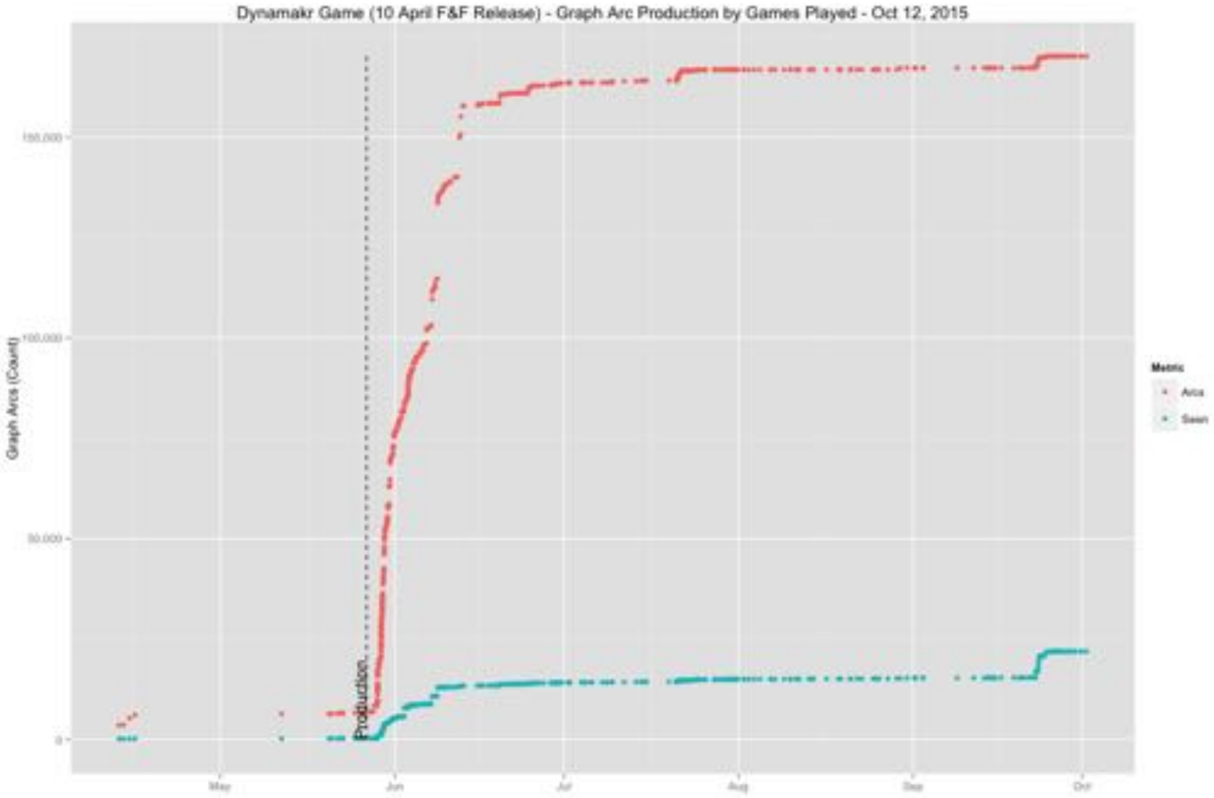


Figure 72: Dynamakr game graph arc production (arc count) and player games seen over time. Samples from 10 April 2015 Friends and Family release through 12 October 2015.

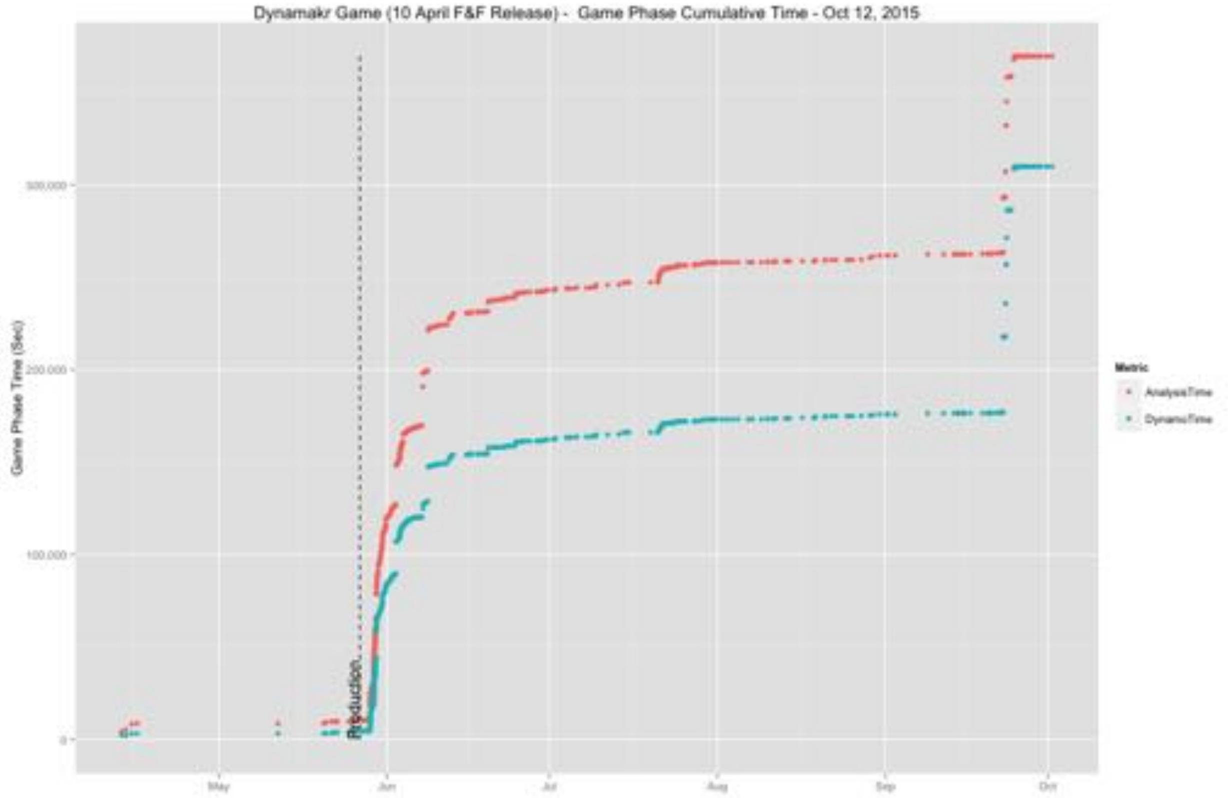


Figure 73: Dynamakr game cumulative analysis and dynamo time (seconds) over time. In the Dynamakr game there were two phases to the game: the analysis time is the work or productive phase time, while the dynamo time is the play or reward phase time. Samples from 10 April 2015 Friends and Family release through 12 October 2015.

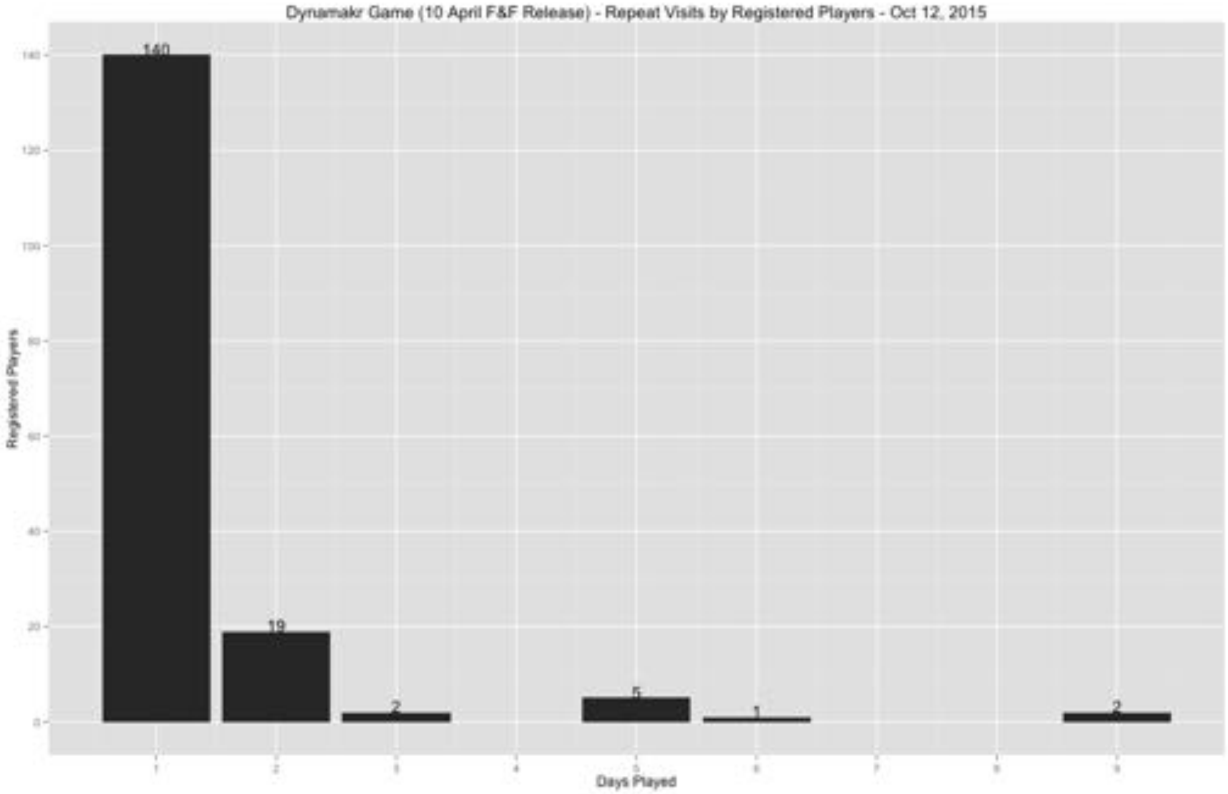


Figure 74: Dynamakr game player retention represented as number of days played (repeat visits) per unique player identifier. Within each day a player may have played the game one or more times. Samples from 27 May 2015 live release through 12 October 2015.

4.4 VIPER Results

VIPER was our Amazon Mechanical Turk (AMT) game. This section presents a number of statistical results collected from game play through AMT, human intelligence task (HIT) reward value (dollar value) assignment, and our backend. These data were collected starting with the live release date of 4 June 2015. We dispatched the games through a succession of HITs, each with an objective and payout budget in mind. After each HIT objective was achieved we prepared and dispatched another HIT; this explains the somewhat discontinuous appearance of these data over time.

Figure 75 on page 185 shows the spot graph arc generation and game instance consumption over time owing to HIT activity from the AMT workers. Figure 76 on page 186 provides the cumulative count representation of these statistics. We collected a variety of internal metrics, shown in Figure 77 on page 187, for game instance counts that had been seen, updated, solved, improved, and first-played by the AMT workers over time, clearly influenced by HIT release events. The productive work time we refer to as *analysis time* during which points-to arc generation occurs for the results graph, and as Figure 78 on page 188 shows we accumulated those seconds over time and these too were influenced by game instance delivery and HIT releases. Figure 79 on page 189 shows the individual game play sessions for analysis time (in seconds) and graph arc production (arcs produced). While most of the samples appear in the wandering and non-productive area of the bottom left-hand corner (low production rate), many workers produced a large number of arcs in a relatively small period of time (high production rate), while a few workers played for a long time and produced almost no arcs. Spreading the data out for better examination requires a log-log scale as shown in Figure 80 on page 190. In this figure we also have shown a conditional fitted mean to 95% confidence level with standard error bars. Figure 81 on page 191 depicts the arc production over time together with the cumulative number of game instances seen by the players. Initially a few key game instances drive the arc production, but later as the graph fills out the other instances arrive to complete the periphery.

Ranking the players by contributions we can learn whether a few players contributed most of the results. Figure 82 on page 192 ranks the individual AMT workers by their total arc production during a session. It highlights the region in orange to indicate that a small proportion of the total players produced 80% of the total volume of arcs for the game's graph iteration. Moreover, it highlights that a large proportion of players contributed no arcs to the solution, most of these being the players that returned the HIT unsolved. Similarly, because we know we have a finite number of game levels contributing constraints to the constraint problem the workers are solving, we can compute how many of these game levels have been consumed. Figure 83 on page 193 ranks the individual AMT workers by their total game instance consumption during a session. It highlights the region in orange to indicate that a larger proportion of players is required to consume 80% of the game instances.

Relating the arc production effort to money, we can introduce the HIT reward or *payout* value. If the worker successfully completes the HIT by our criteria given at the start of the task, then he earns the payout. We were given a budget to manage and experiment with for the project. Figure 84 on page 194 shows the VIPER game total arc production by HIT payout value; there were four payout values of 15 cents, 50 cents, $1 and $2. At the time of the data collection we had run many more HITs at the 50 cent payout. Interestingly, the medians of the 50 cent HITs are higher and the upper quartile is much higher than the 15 cents and $1 HITs. These data include the non-productive sessions in which the workers look at the task and return it unsolved. If we look instead at the arc production rate (arcs per second), then the medians and means are much closer together for the total sessions, shown in Figure 85 on page 195, including productive and non-productive sessions regardless of the number of HITs released at each payout. Interestingly, when we examine the time spent by workers in each session, omitting just a few outliers as shown in Figure 86 on page 196, we see that at all four payout values the workers spend about the same amount of time in a session: about 1,200 seconds on average including productive and non-productive time, regardless of payout value. But once we eliminate the non-productive sessions, shown in Figure 87 on page 197, we see an interesting difference among the payout values. Workers seem to be indifferent to the 15 cent and 50 cent payouts regarding production time: the medians and means are rather close. The productive workers at the $1 and $2 HIT payouts, however, are willing to spend much more time solving the problem. The median AMT worker spent over 2,200 seconds (about 36 minutes) on the $1 payout problems.

We prepared each HIT with an expected difficulty level. The number of simultaneous game instances to be reasoned about comprised the difficulty level, and it also stressed the game engine plugin and player's browser. We started with smaller difficulties and worked our way up to higher difficulties to learn how much work we might push out in a HIT before the plugins, browsers, and players complained. We expected player performance to drop with higher difficulties at lower payouts, as the time investment for the smaller reward likely was not worthwhile. Some players nevertheless pressed through the difficulties to earn the rewards. Figure 88 on page 198 shows the AMT worker graph arc production by HIT value and difficulty. Our difficulty thresholds were 15, 20, 25 and 50 simultaneous game instances. One can see in this figure for instance at a difficulty of 15 that for each HIT payout value the median and quartile player times spent completing the HIT increases with the HIT payout value. Figure 89 on page 199 shows the game instances the players managed to solve during the HIT sessions, by HIT value and difficulty. The game would stream in more game instances as the player worked through the constraints problems, and the auto-solvers would solve constraints by applying the game model rules where it could. Figure 92 on page 202 plots the graph arc production by games solved during the HIT session, by HIT difficulty and payout value, while Figure 93 on page 203 reduces the content to the graph arc production by games *improved* during the HIT session, again by HIT difficulty and payout value. In the latter case the AMT worker has taken a previously-solved game instance and by his arc contributions made the game score higher for that game instance. Figure 94 on page 204 rearranges these results by difficulty rather than payout.

Figure 90 on page 200 ignores the payouts and combines the difficulties onto a graph arc production by games solved plot, using log scales, together with conditional mean fits on each difficulty level in order to identify whether there may be a varying correspondence between production and game solutions by difficulty. Ignoring the payout in this case seems to destroy such a relationship because the incentive factor that increases productivity is lost. Moreover in these data we have an issue that the fixpoint iteration is near stable such that the arc production tails off while game solutions continue; adding new game content in future HITs will restart arc production. Figure 91 on page 201 makes a similar statement by ignoring difficulty and fitting payouts.

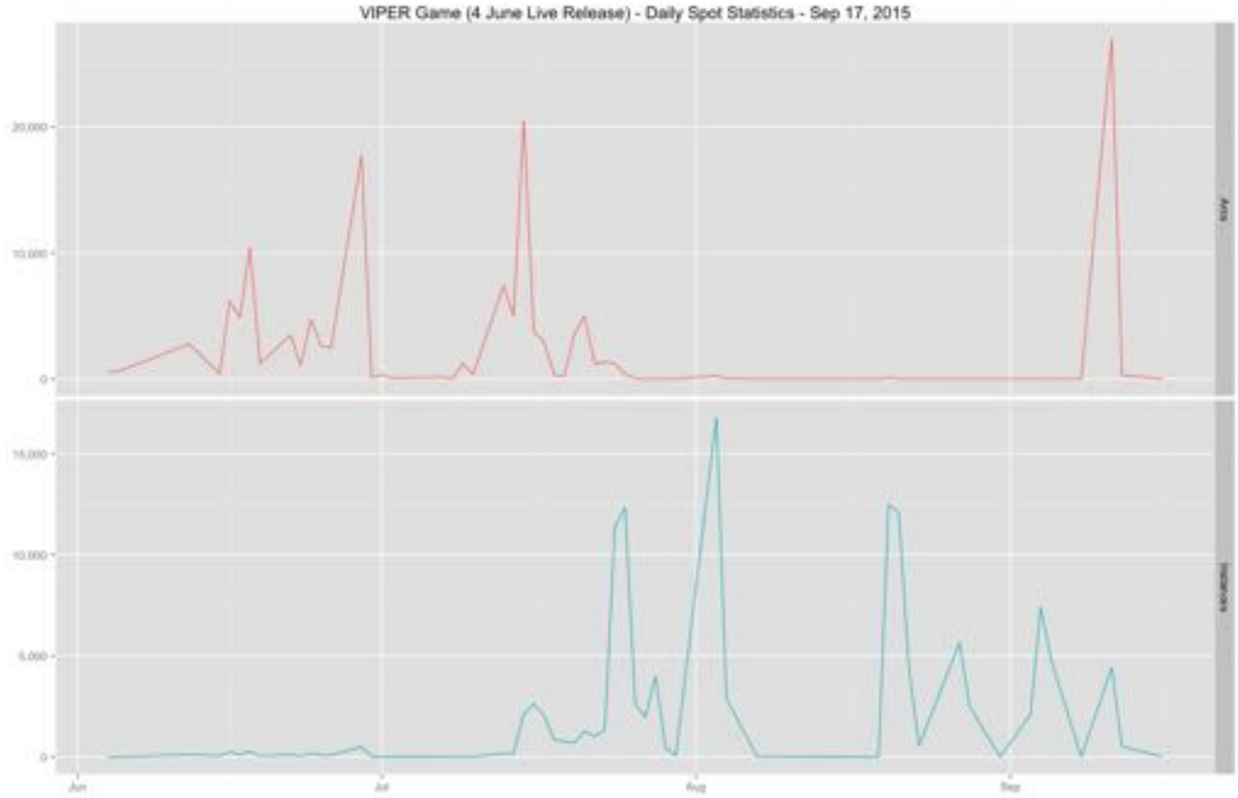


Figure 75: VIPER game spot arc production and game levels (instances) played over time. Samples from 4 June 2015 live release through 17 September 2015.

Figure 95 on page 205 sums the total play time for each level of solved game instances. The level of zero solved game instances is the *balking level* at which the AMT workers played the game but did not come up with any solutions, returning the game unsolved and therefore not earning a payout. The sum total time at that level is the balking time for the AMT workers which exceeded 63,000 seconds.

Finally, Figure 96 on page 206 shows AMT worker retention. Workers may revisit the AMT HIT list and select to perform another VIPER HIT. The plot indicates the number of times known worker IDs played only one HIT as well as the number of times known worker IDs returned to play on two occasions, three occasions, and so on. These data include visits at all HIT payout values.

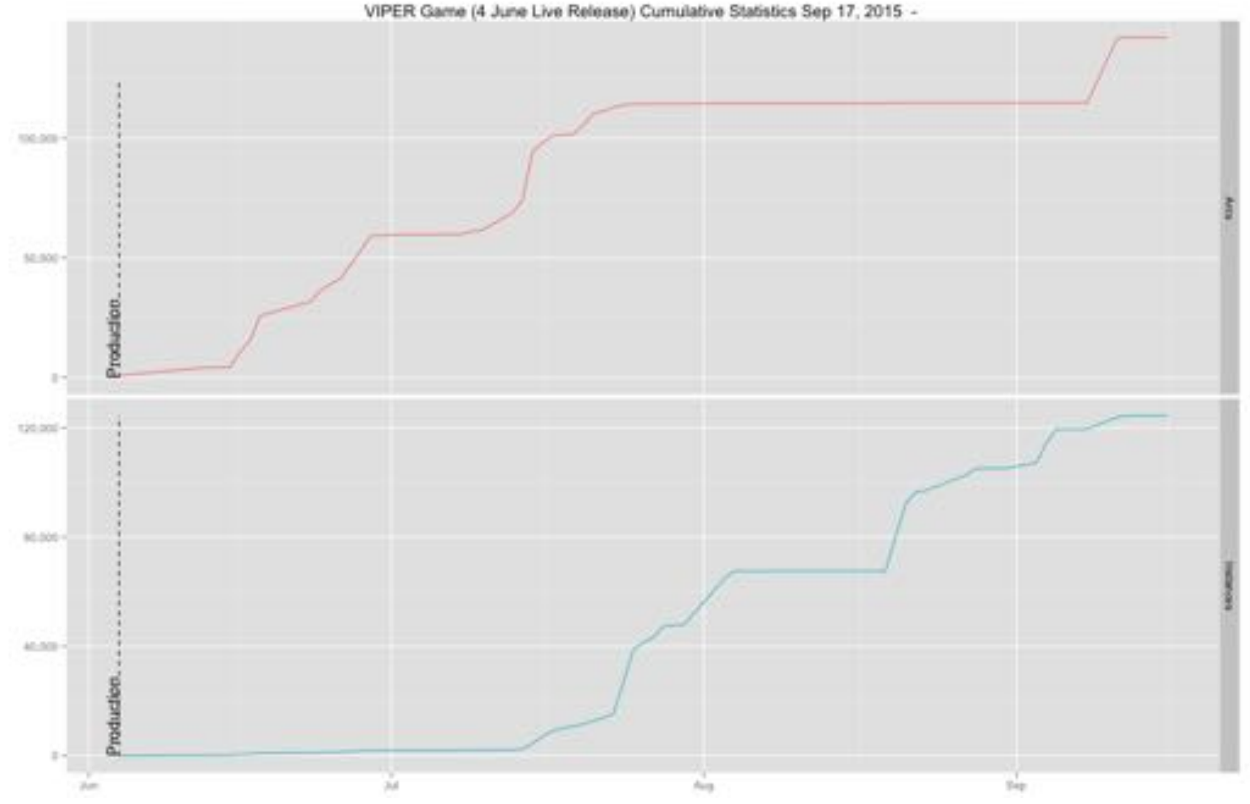


Figure 76: VIPER game cumulative arc production and game levels (instances) played over time. Samples from 4 June 2015 live release through 17 September 2015.

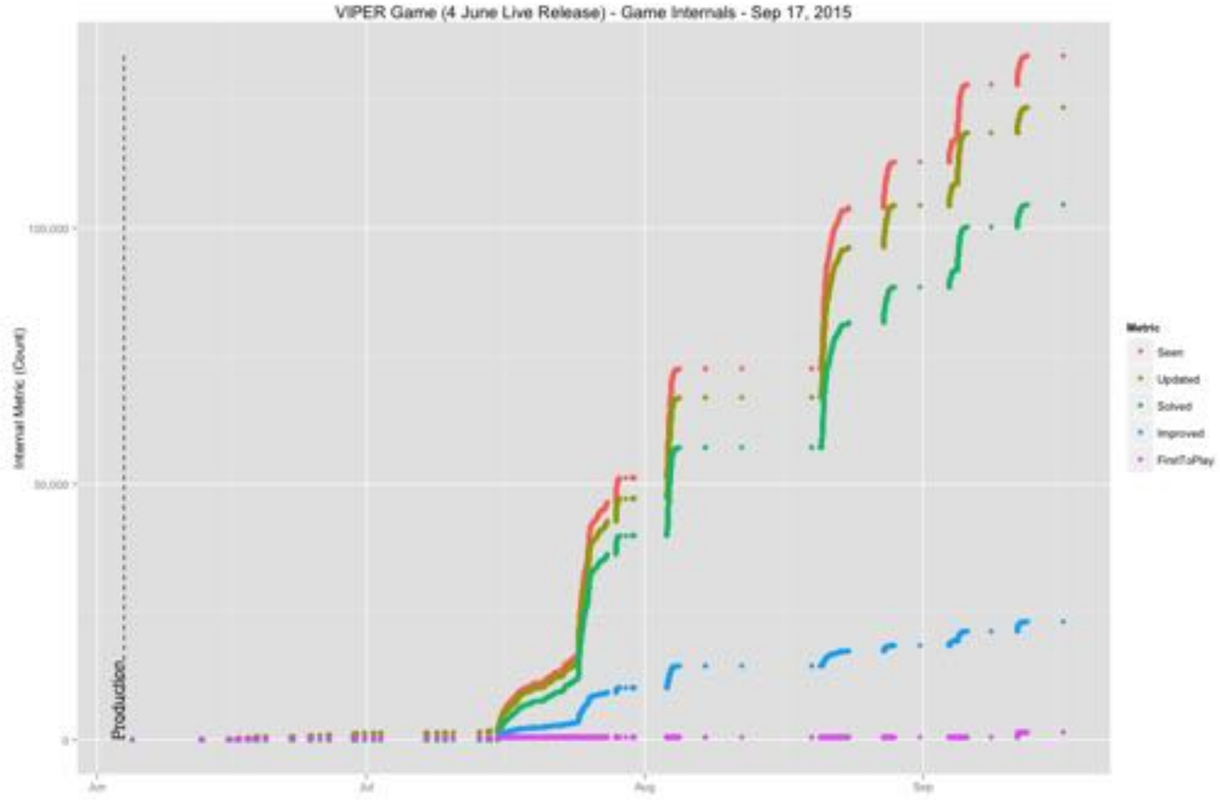


Figure 77: VIPER game internal metrics over time. These internals accumulate counters for game instances by whether they have been seen, updated, solved, improved, or first-played by the player community. Samples from 4 June 2015 live release through 17 September 2015.

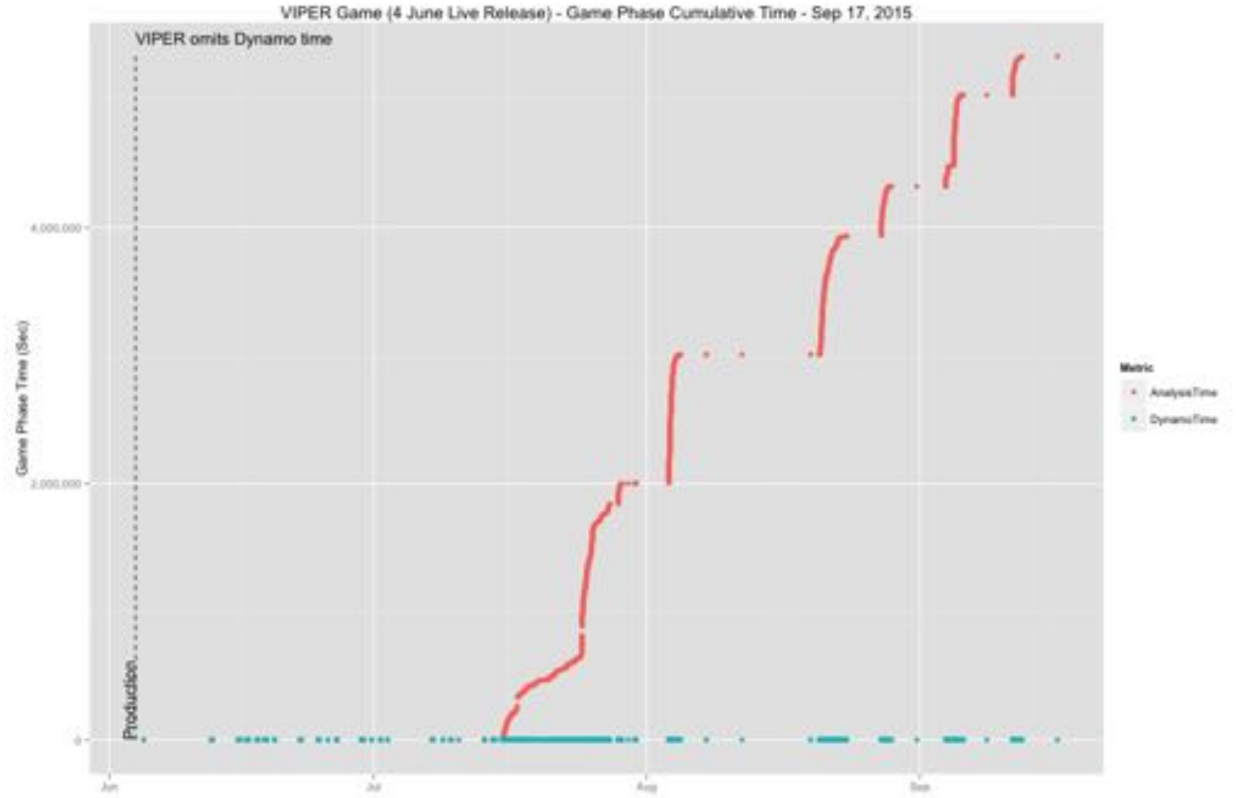


Figure 78: VIPER cumulative analysis time by date. Because the VIPER game uses Amazon Mechanical Turk delivery it strips away the game elements. Consequently we tracked only the productive analysis time for the solution and there was no non-productive play time (labeled Dynamo phase time as in the Dynamakr game) for the solution. Samples from 4 June 2015 live release through 17 September 2015.

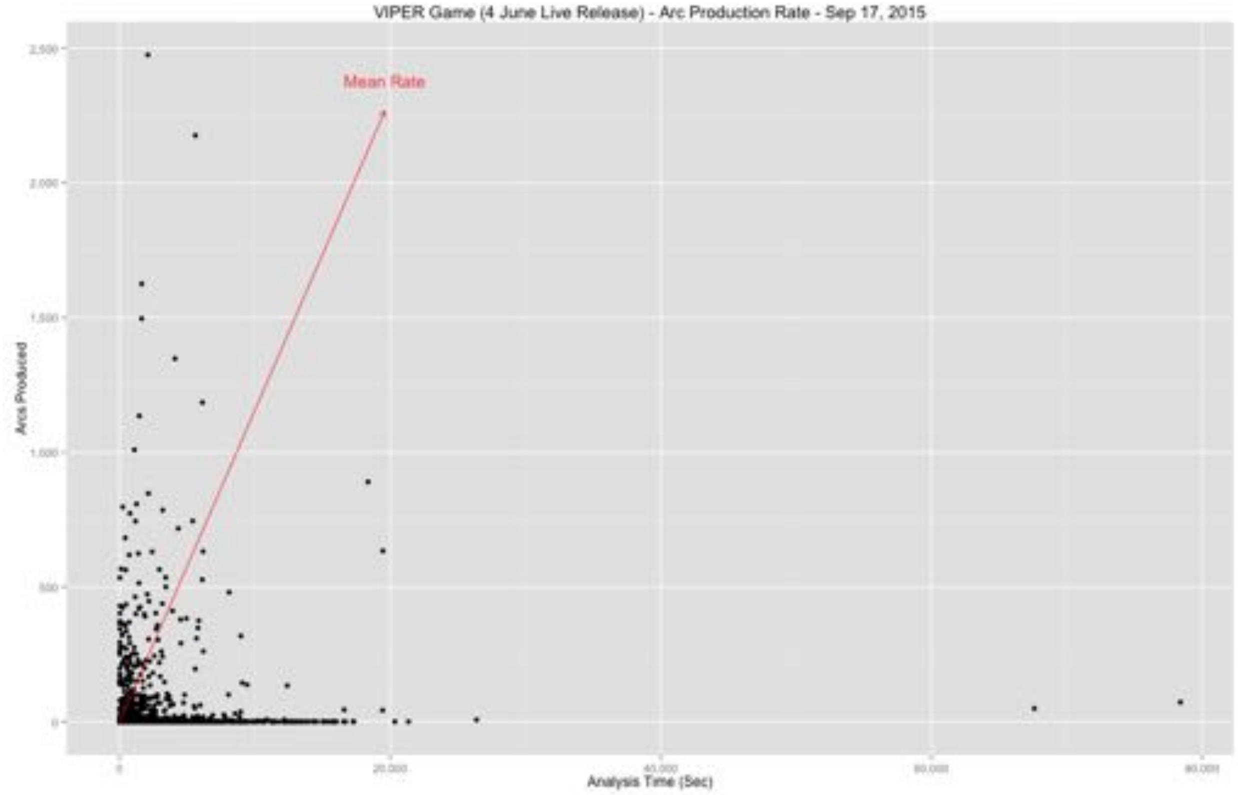


Figure 79: VIPER game analysis time versus arcs produced. The mean line represents the mean arc production rate (arcs per unit time) with an intercept of zero. Samples from 4 June 2015 live release through 17 September 2015.

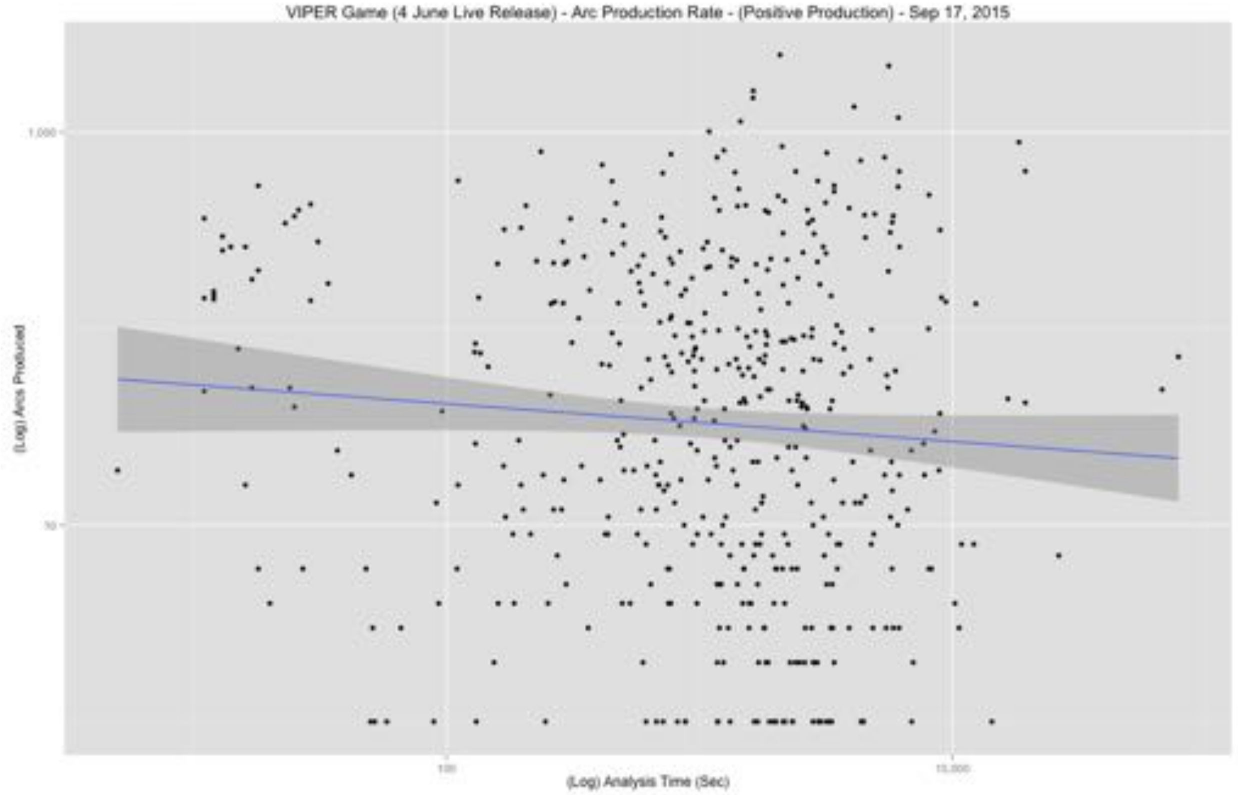


Figure 80: VIPER game analysis time versus arcs produced, log-log scale. The conditional mean shown is fitted to 95% confidence and shown with standard error range. Samples from 4 June 2015 live release through 17 September 2015.

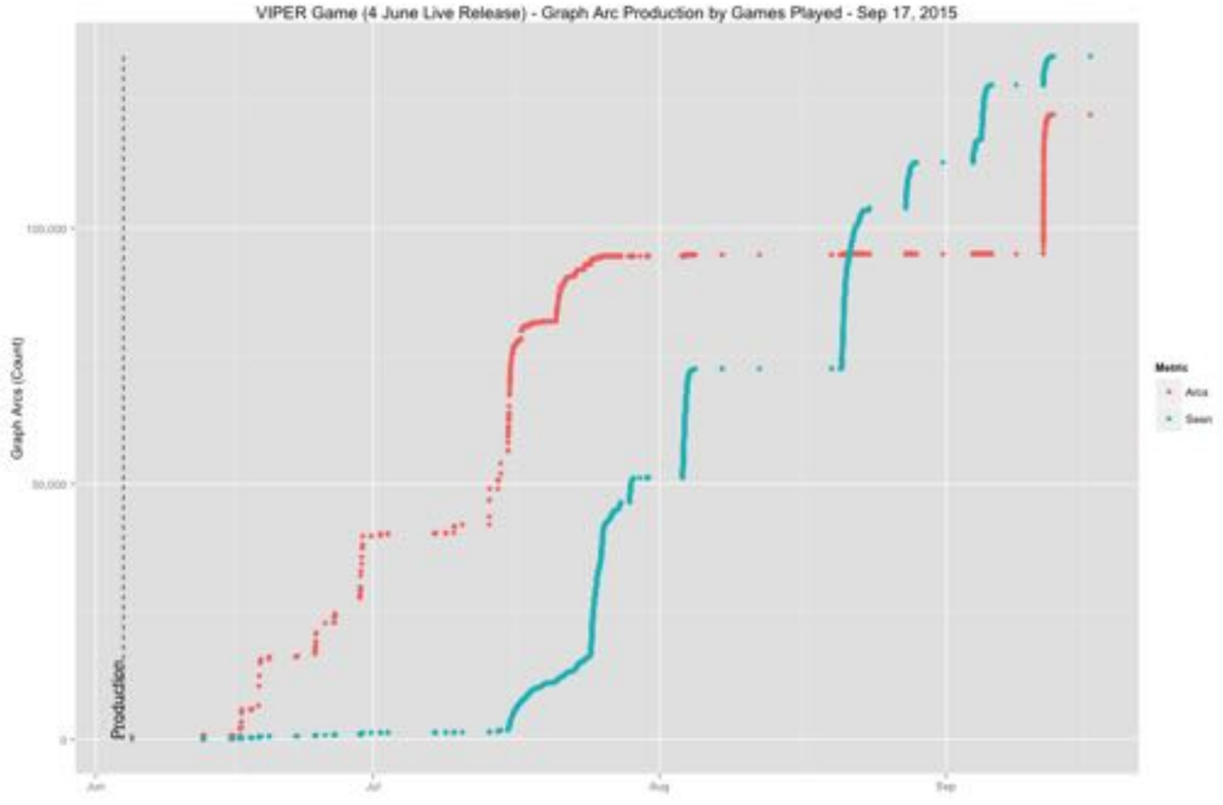


Figure 81: VIPER game arcs produced and seen by all players over time. Samples from 4 June 2015 live release through 17 September 2015.

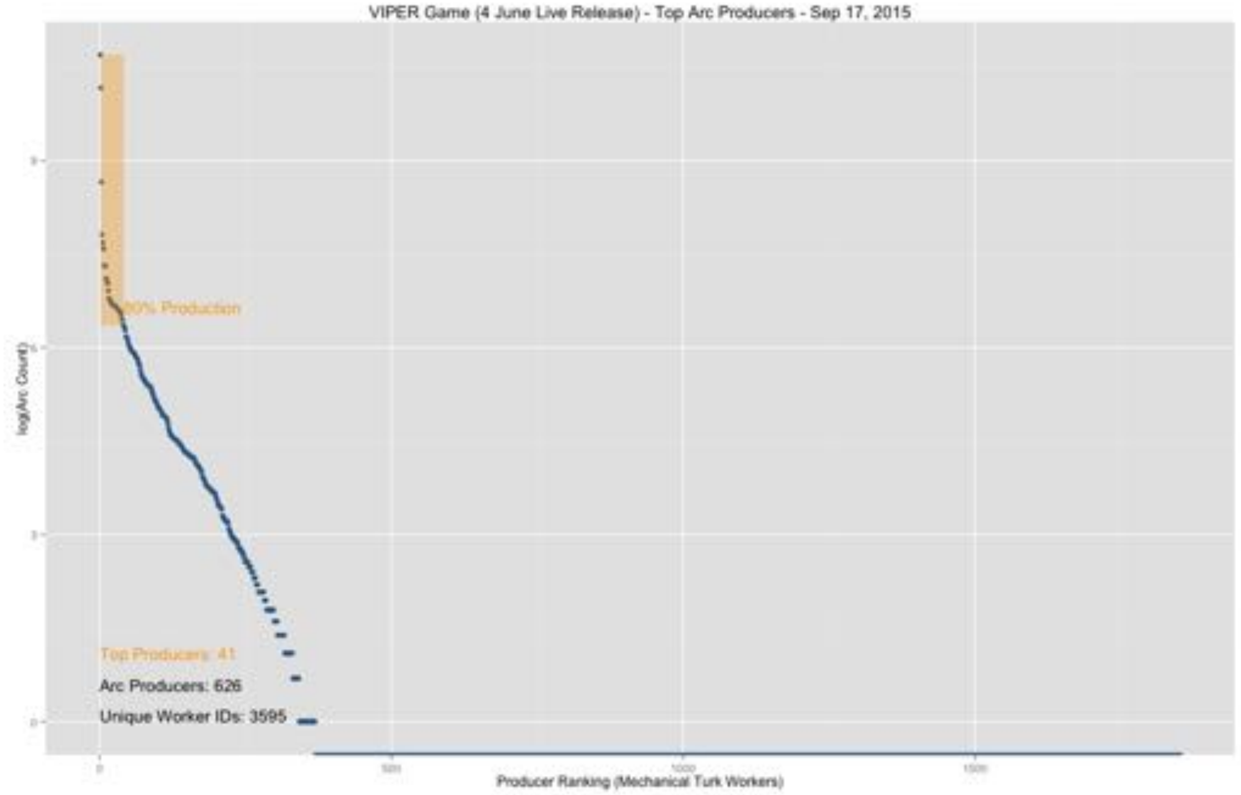


Figure 82: VIPER player rank by arc production. Players ranked left-to-right by arc production count. The highlighted region indicates the players contributing 80% of the total arc production. Samples from 4 June 2015 live release through 17 September 2015.

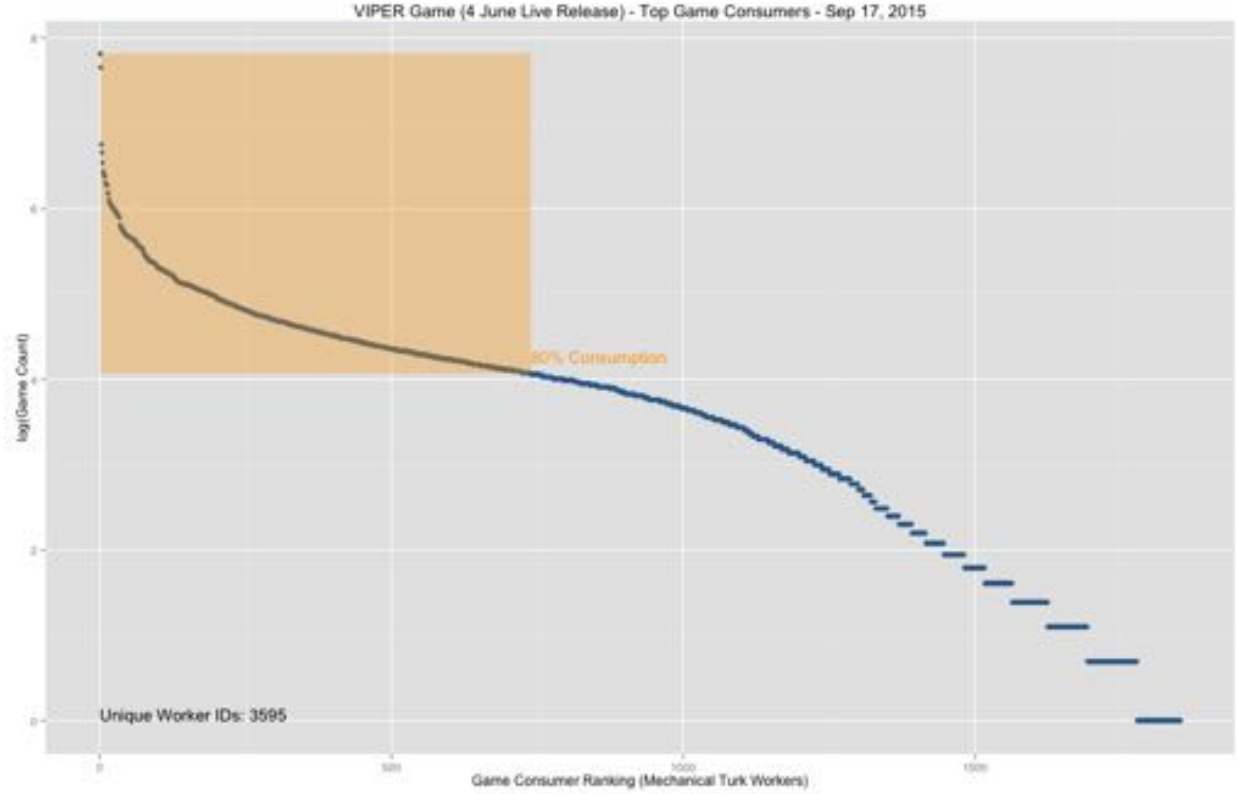


Figure 83: VIPER player rank by game consumption. Players ranked left-to-right by game instance consumption (play) count. The highlighted region indicates the players consuming 80% of the played game count. Samples from 4 June 2015 live release through 17 September 2015.

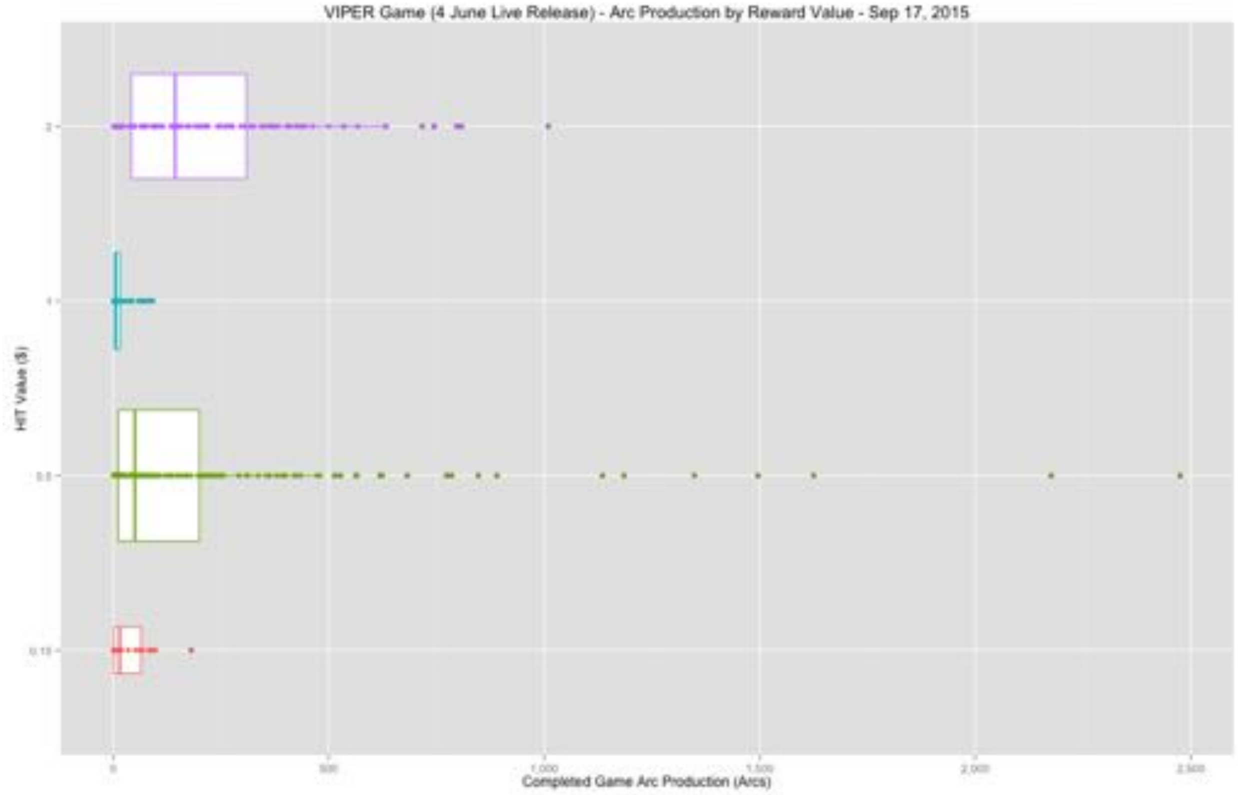


Figure 84: VIPER game arc production statistics by HIT reward value. Box-and-whisker representation of population statistics with median and quartiles. Samples from 4 June 2015 live release through 17 September 2015.

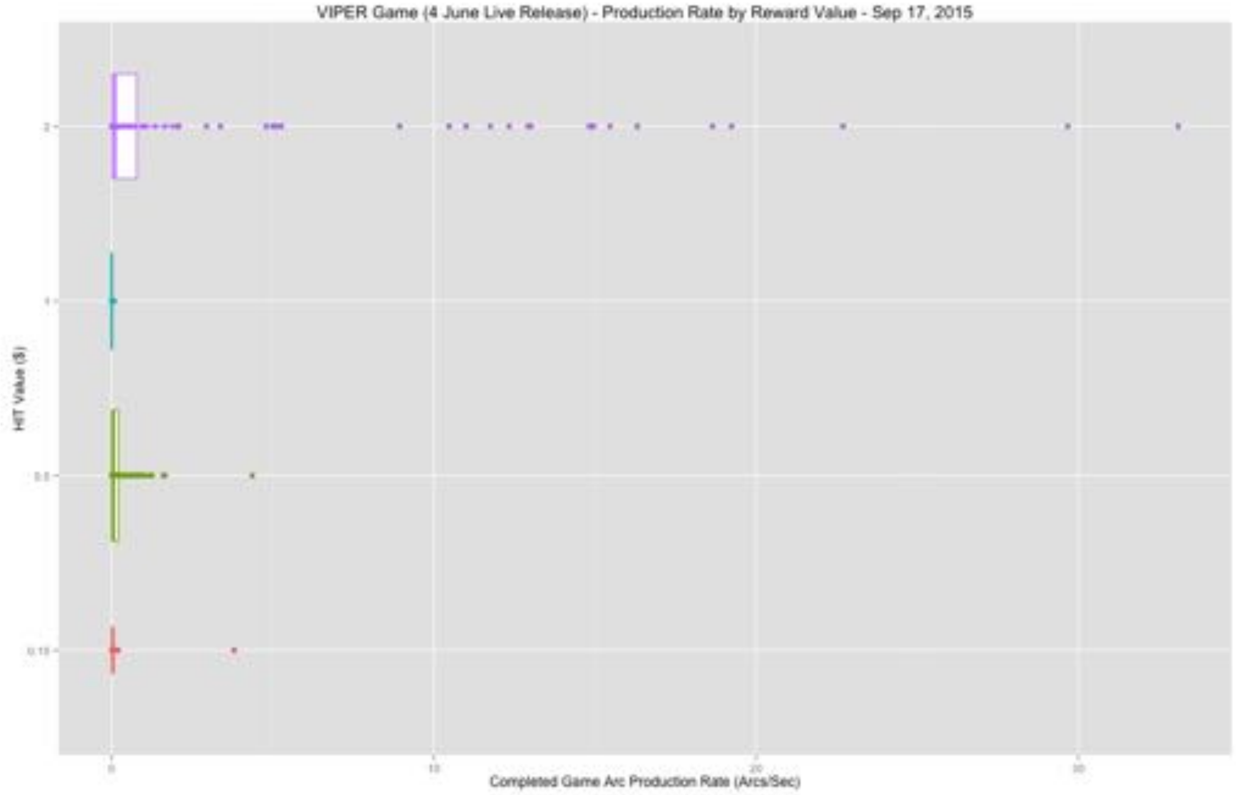


Figure 85: VIPER game arc production rate (arcs per second) by HIT reward value. Box-and-whisker representation of population statistics with median and quartiles. Samples from 4 June 2015 live release through 17 September 2015.

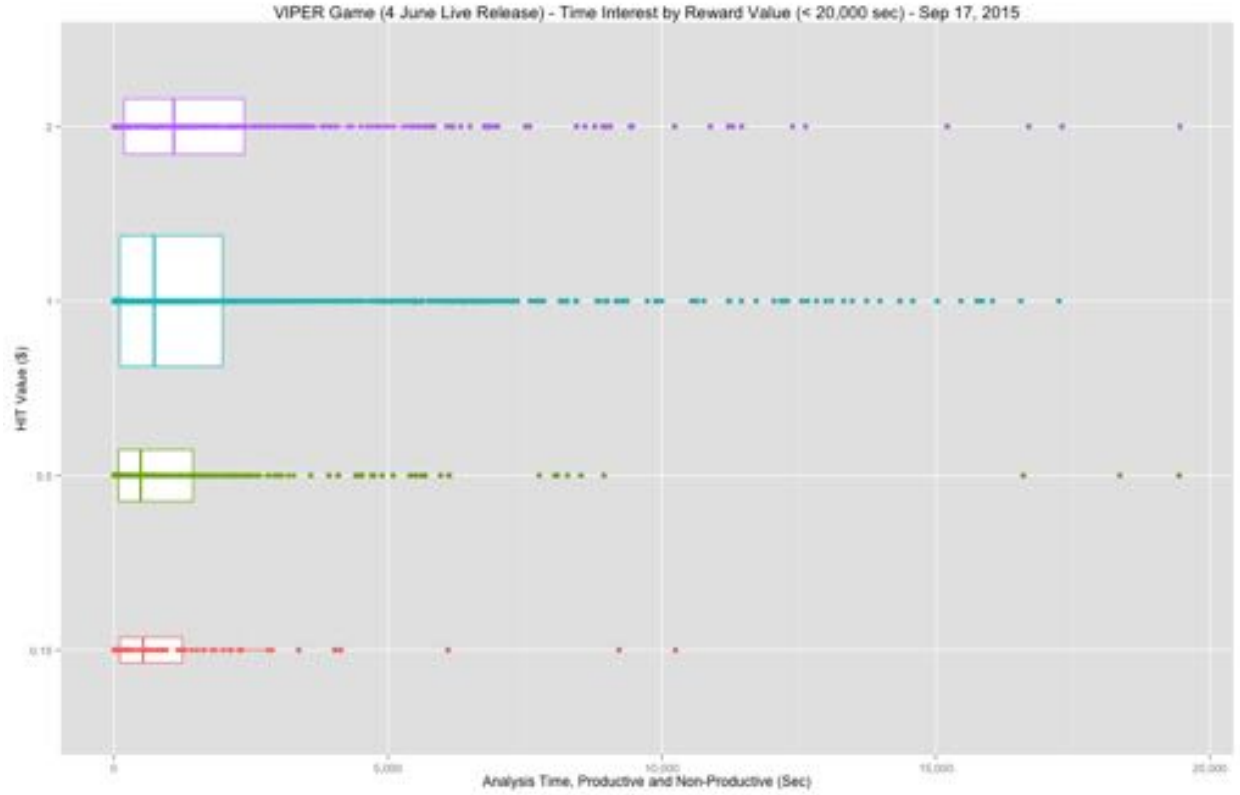


Figure 86: VIPER player time interest by HIT reward value, including non-productive cases. Box-and-whisker representation of population statistics with median and quartiles. Two cases above 20,000 samples removed for presentation. Samples from 4 June 2015 live release through 17 September 2015.

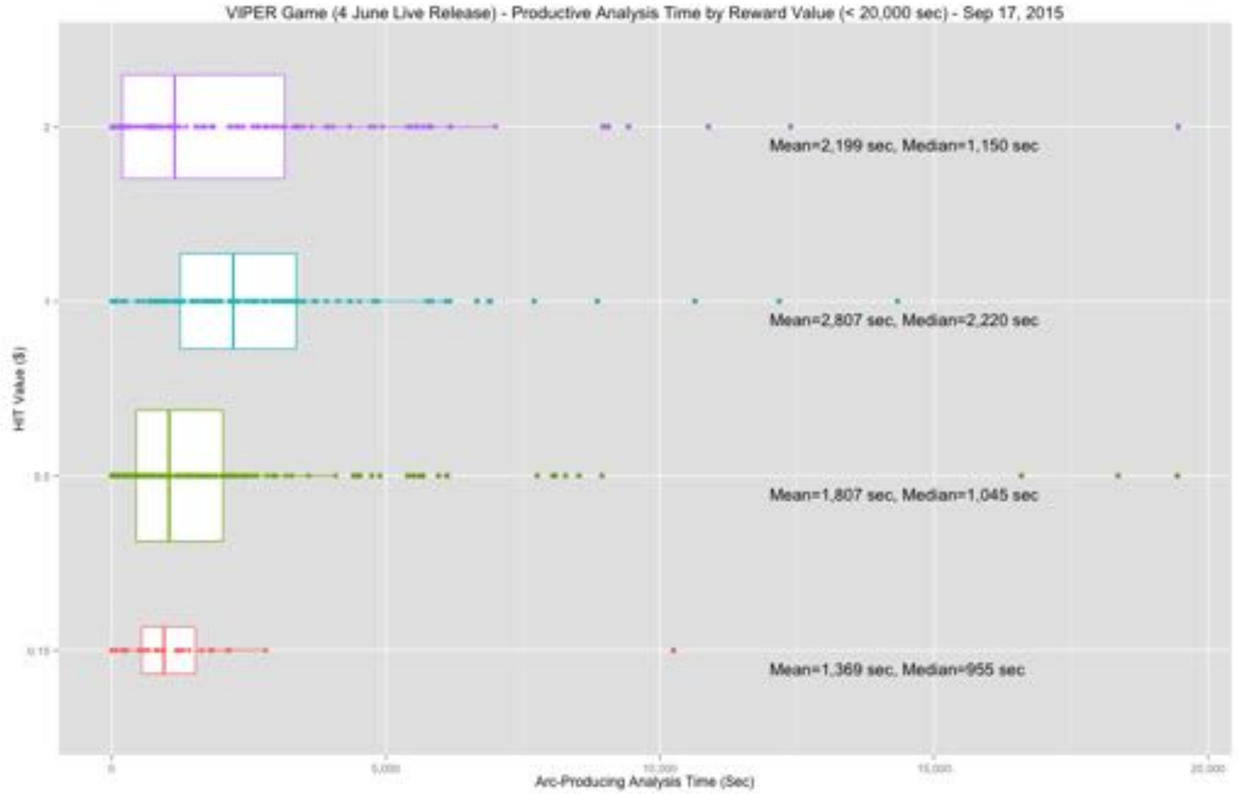


Figure 87: VIPER player productive time by HIT reward value, including only productive cases. Box-and-whisker representation of population statistics with median and quartiles. Two cases above 20,000 samples removed for presentation. Samples from 4 June 2015 live release through 17 September 2015.

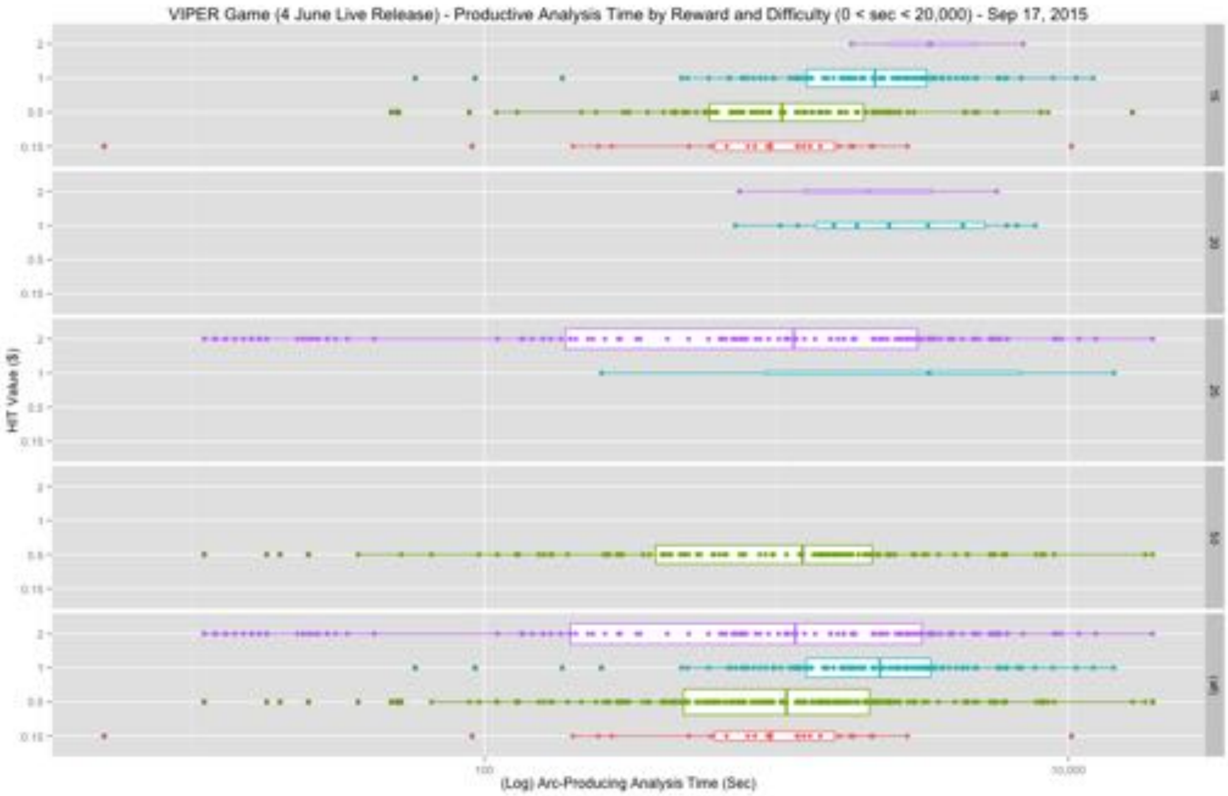


Figure 88: VIPER productive analysis time by reward and difficulty. Human intelligence task (HIT) payout value in dollars by the log of arc-producing analysis time in seconds, allocated by HIT difficulty level panels of values 15, 20, 25 and 50 simultaneous game instances. The statistics are presented as box-and-whisker plots with median bars and the upper and lower quartiles defining the boxes, and whiskers extending to 1.5 times the interquartile range. Outliers are shown as points beyond the whiskers. The bottom panel combines all difficulty levels from the panels above. Samples from 4 June 2015 live release through 17 September 2015.

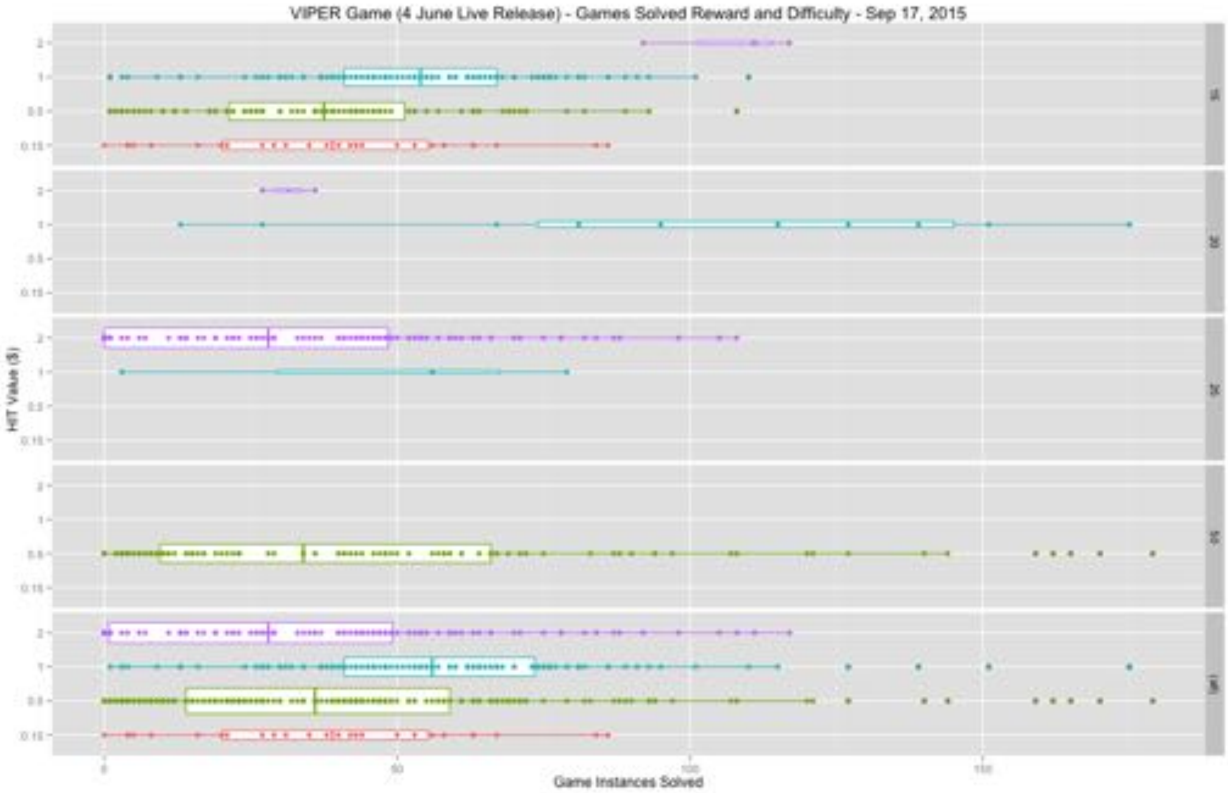


Figure 89: VIPER games solved by reward and difficulty. Human intelligence task (HIT) payout value in dollars by the number of game instances solved during HIT completion, allocated by HIT difficulty level panels of values 15, 20, 25 and 50 simultaneous game instances. The statistics are presented as box-and-whisker plots with median bars and the upper and lower quartiles defining the boxes, and whiskers extending to 1.5 times the interquartile range. Outliers are shown as points beyond the whiskers. The bottom panel combines all difficulty levels from the panels above. Samples from 4 June 2015 live release through 17 September 2015.

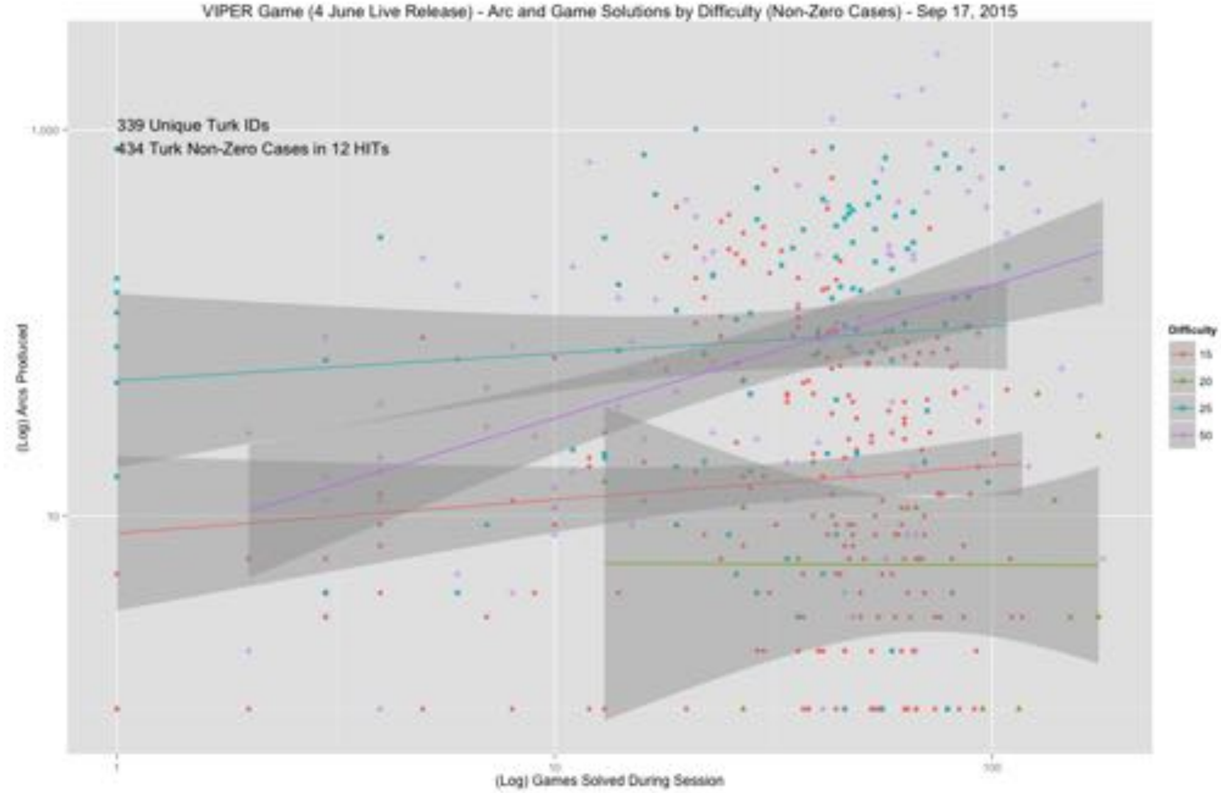


Figure 90: VIPER arc production by game solutions and difficulty. Human intelligence task (HIT) game solutions and graph arc production samples annotated and fitted by HIT difficulty rating (15, 20, 25 or 50 simultaneous game instances). Fits are linear fits of conditional means to each difficulty on the log-log data to 95% confidence. The difficulty 20 data were hampered by the near-completion of a fixpoint solution for the program under analysis and therefore few arcs available for production; future HITs at this difficulty with a fresh fixpoint iteration would yield a positive-sloping fit. Samples from 4 June 2015 live release through 17 September 2015.

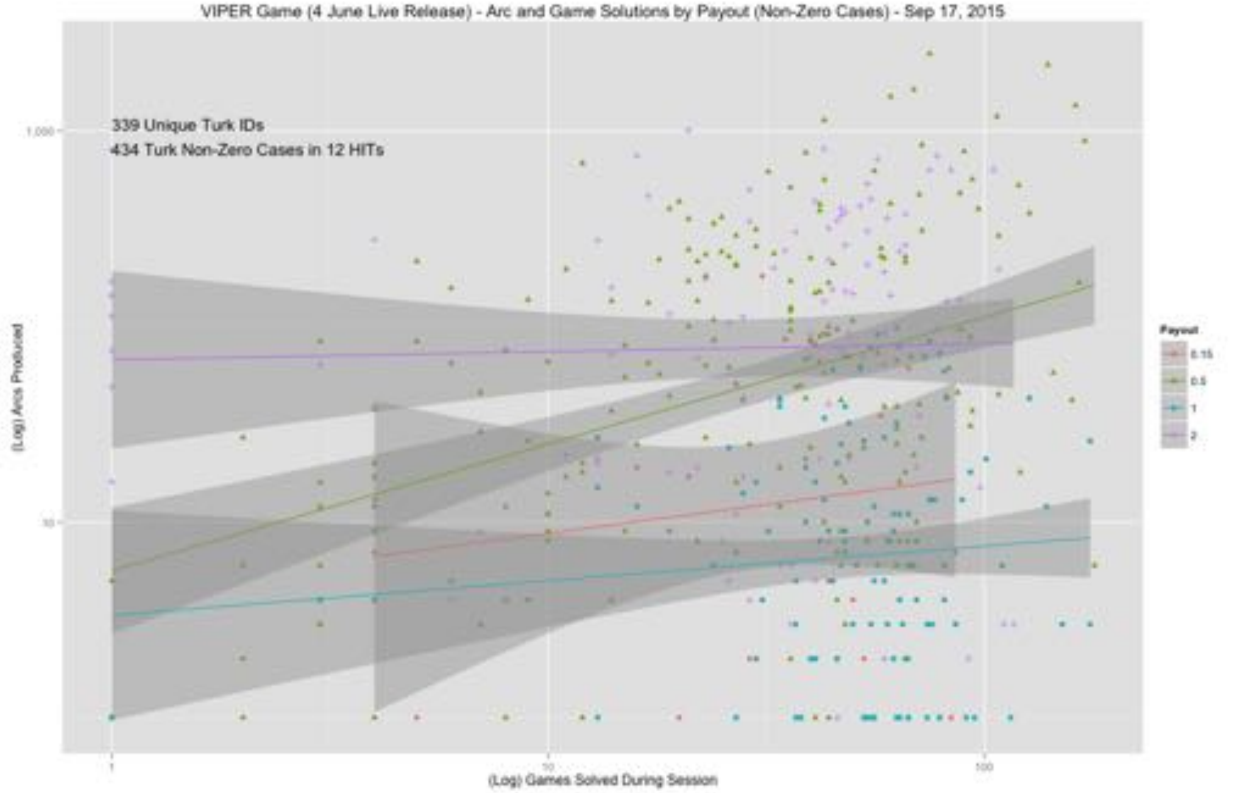


Figure 91: VIPER arc production by game solutions and payout. Human intelligence task (HIT) game solutions and graph arc production samples annotated and fitted by HIT payout value (0.15, 0.50, 1.00 and 2.00 dollars). Fits are linear fits of conditional means to each payout value on the log-log data to 95% confidence. The payout 2 data were hampered by the near-completion of a fixpoint solution for the program under analysis and therefore few arcs available for production; future HITs at this payout with a fresh fixpoint iteration would yield a positive-sloping fit. Samples from 4 June 2015 live release through 17 September 2015.

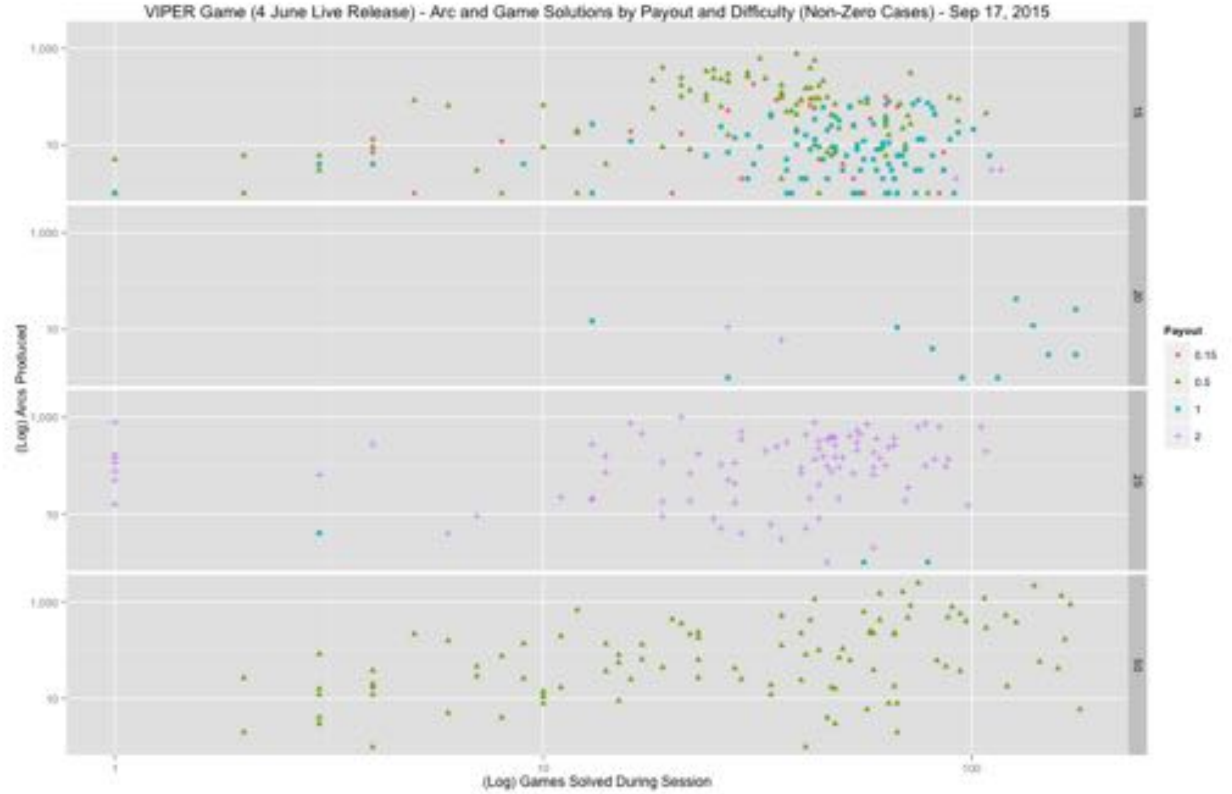


Figure 92: VIPER arc production by game solutions and payout grid. Human intelligence task (HIT) game solutions and graph arc production samples by HIT difficulty (15, 20, 25 and 50 simultaneous game instances) and annotated by HIT payout value (0.15, 0.50, 1.00 and 2.00 dollars). Samples from 4 June 2015 live release through 17 September 2015.

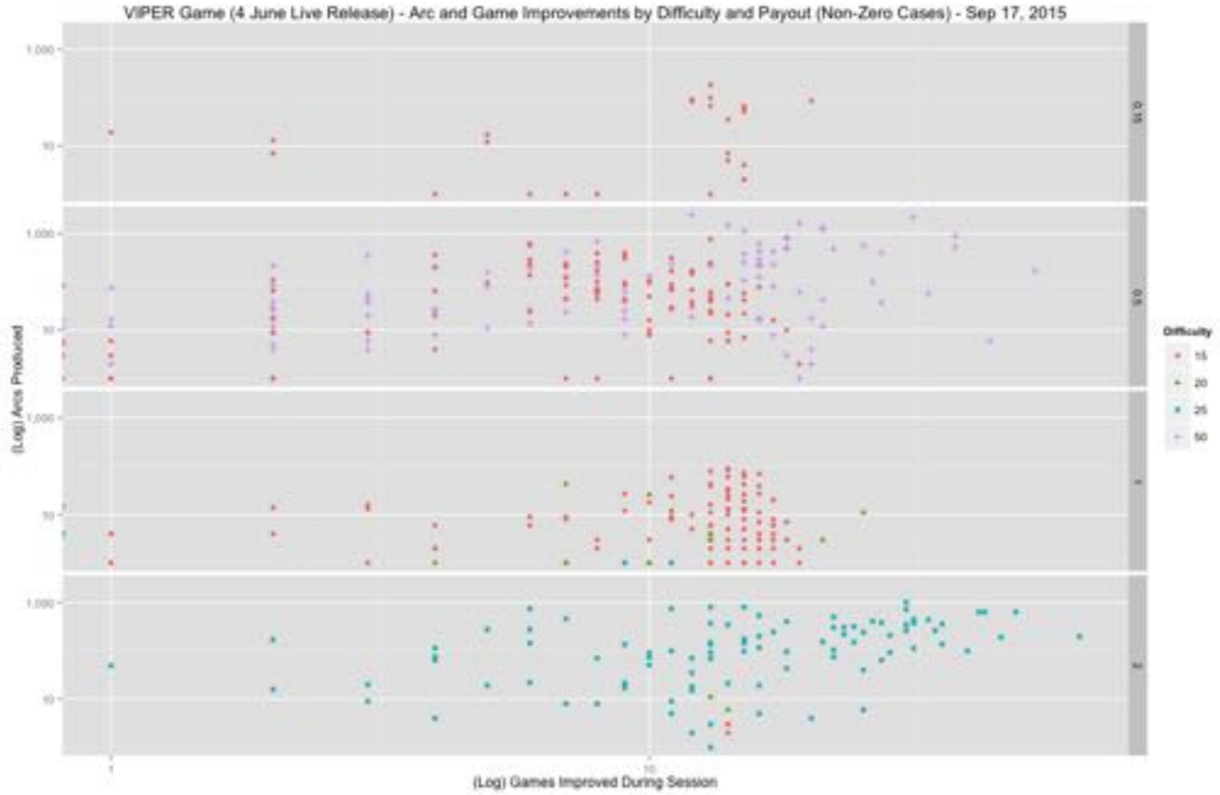


Figure 93: VIPER arc production and game solution improvement by human intelligence task (HIT) payout value and difficulty rating. The HIT payout values are 0.15, 0.50, 1.00 and 2.00 dollars. The HIT difficulty ratings are 15, 20, 25, and 50 simultaneous game instances. All series exclude samples in which no arcs were generated or no solutions were improved. Samples from 4 June 2015 live release through 17 September 2015.

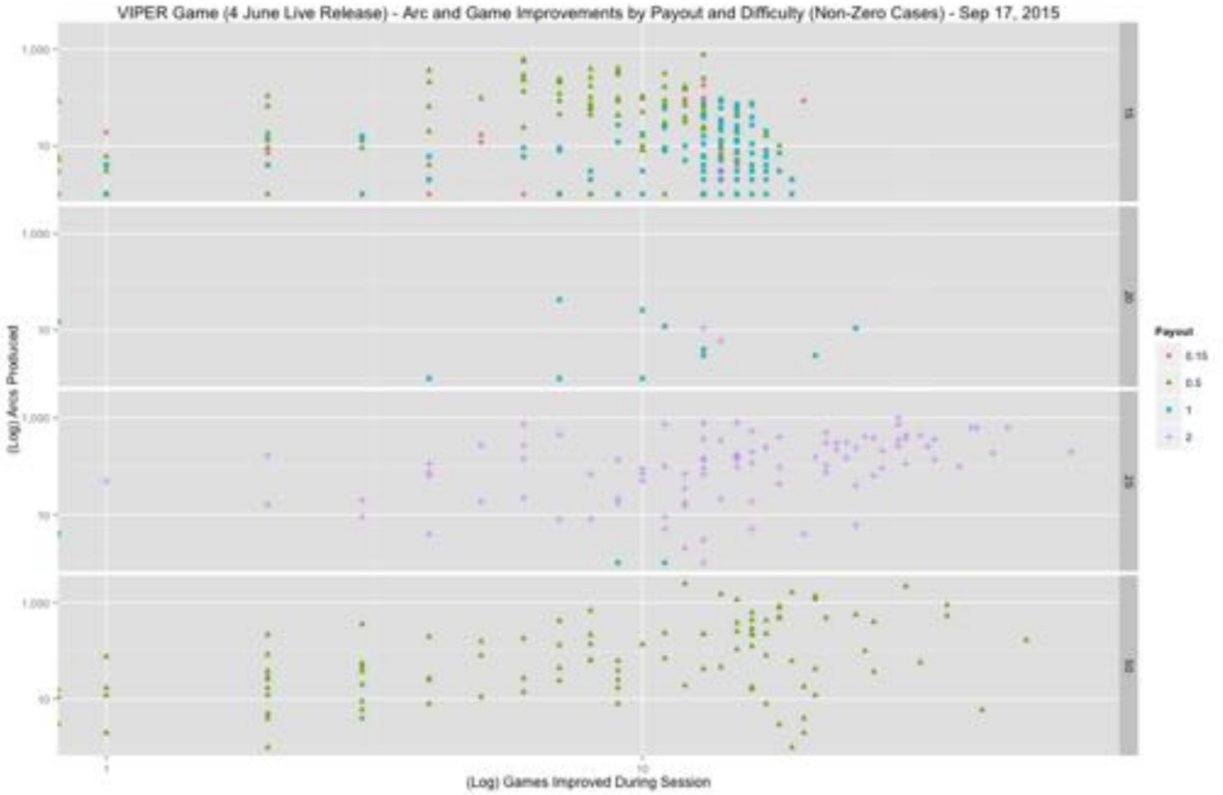


Figure 94: VIPER arc production and game solution improvement by human intelligence task (HIT) difficulty rating and payout value. The HIT payout values are 0.15, 0.50, 1.00 and 2.00 dollars. The HIT difficulty ratings are 15, 20, 25, and 50 simultaneous game instances. All series exclude samples in which no arcs were generated or no solutions were improved. Samples from 4 June 2015 live release through 17 September 2015.

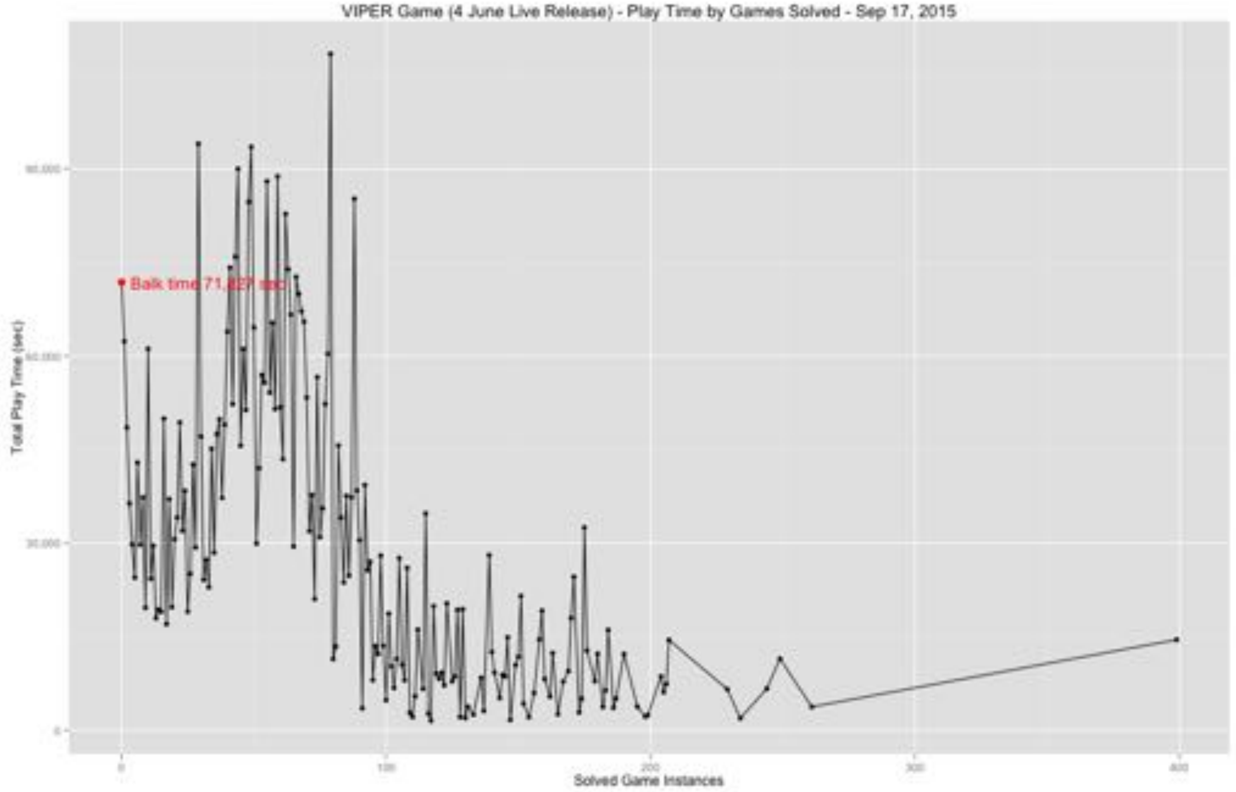


Figure 95: VIPER play time by games solved. Total play time seconds for each number of game instances solved. Total play time at zero game solutions is considered the *balking time* in which players returned games unsolved. Samples from 4 June 2015 live release through 17 September 2015.

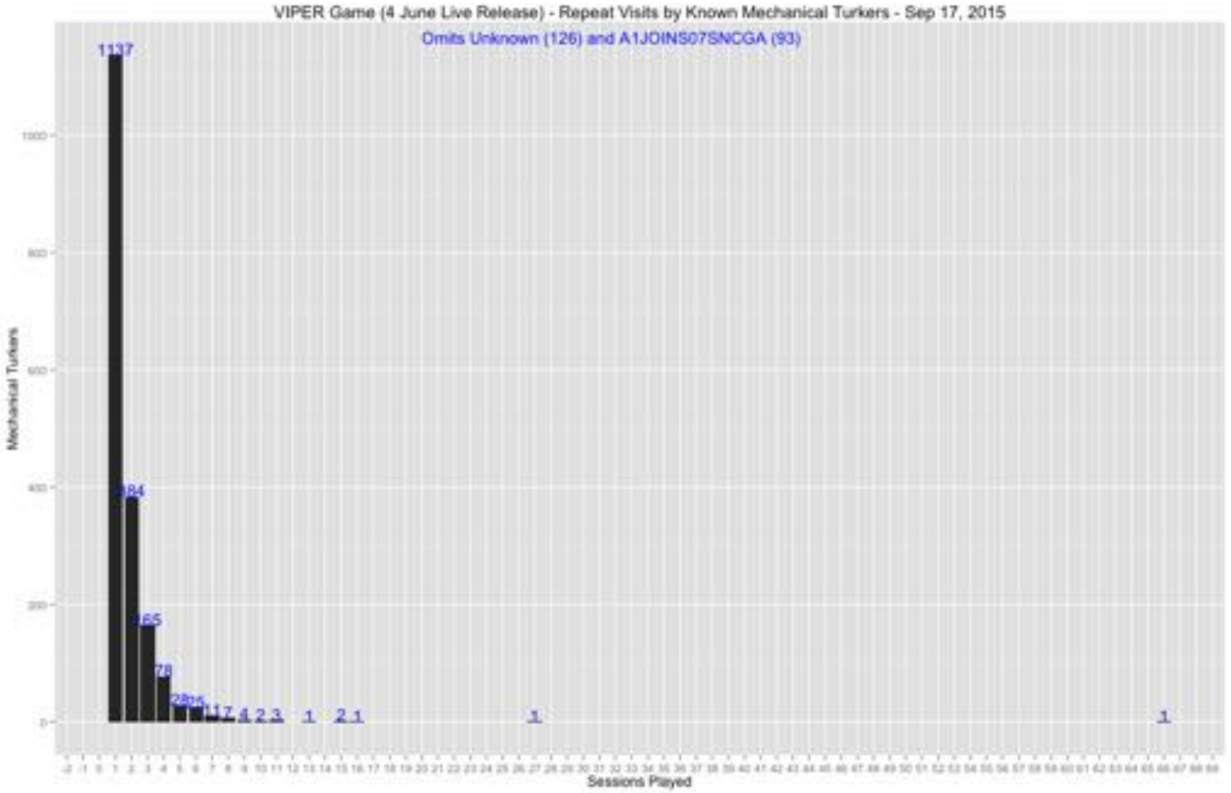


Figure 96: VIPER worker retention by return visit count. Indicates the number of times workers revisited to performed another HIT. Samples from 4 June 2015 live release through 17 September 2015.

4.5 Combined Results

We combined the Dynamakr and VIPER results onto the same plots in order to facilitate comparison. Although the games started at different times, played different game loads at different times, saw a different number and likely different population of participants (we were not able to identify anonymous players), the games use the same game model and underlying analysis targets so we are interested in comparing certain productivity measures. Figure 97 on the following page shows the cumulative contribution of graph arc production by player ranking for the three player types in the two games; Dynamakr had *registered* and *anonymous* players while VIPER had paid players here labeled *turks*. As explained in the individual game data, the top contributors for each contributed most of the productive work, so the traces here climb quickly indicating the top players – left to right – contribute most of the work. Because there are fewer registered players than either turks or anonymous players, that curve reaches its maximum value 100% quickly. Zooming in for more details of the top performers, Figure 98 on page 209 shows the top-100 performers in each player category. Here we see the top-performing turks actually out-perform the registered players initially, but because there are more of them many turks contribute to the results. Figure 99 on page 210 zooms in further to examine to top-50 performers.

Examining the individual contributions directly, we plot the total number of graph arcs produced by ranked player type. Figure 100 on page 211 plots the log of the total arcs produced by the top-100 players, ranked left-to-right and colored by player type. Figure 101 on page 212 shows the same for the top-50 players. These plots reconfirm the finding that the handful of key contributors generate most of the productive work. Interestingly, while the registered players contribute the most productive work, there are fewer of them. The turk players contribute nearly as much productive work and there are many more of them so overall they can be much more productive to a large objective. Figure 102 on page 213 shows the sample statistics for each player type. The VIPER data show some lower productive capacity metrics because of the dwell time the game delivery mechanism uses allowing the player to evaluate the game before returning, ultimately yielding lower arc production per play although with higher volume overall; these capacity metrics would be much higher if we could filter those cases in which the game ultimately was returned without payout, but the data collection did not identify such cases. Finally, Figure 103 on page 214 shows the time-production samples together with model fits for these data by player type. Here again the VIPER data show the consequence of a wider range of analysis time on the productivity rate results.

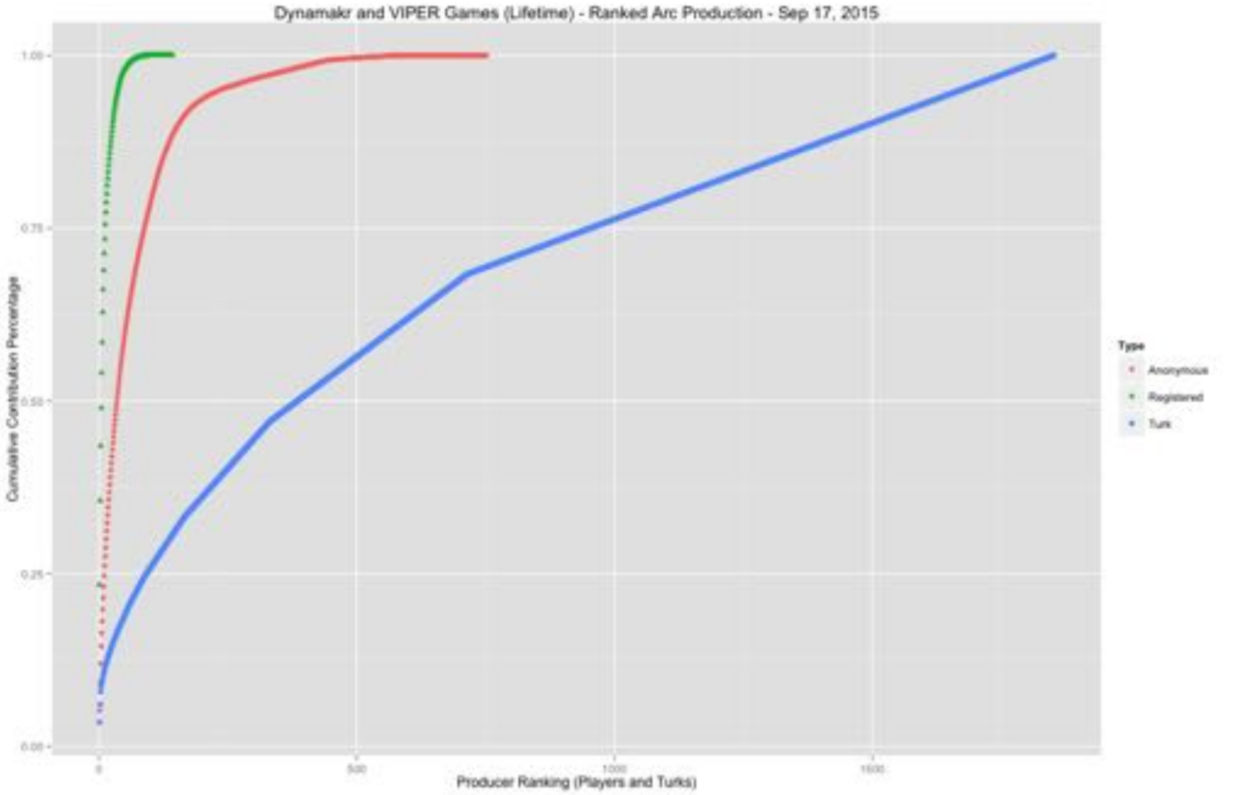


Figure 97: Cumulative production percentage for all of the Dynamakr and VIPER game players, ranked in descending order of contribution. The *registered* and *anonymous* player statistics are for the Dynamakr game while the *turk* player statistics are for the VIPER game. The population sizes differ so the trends reach their 100% contribution levels at different points. Samples from 4 June 2015 live release through 17 September 2015.

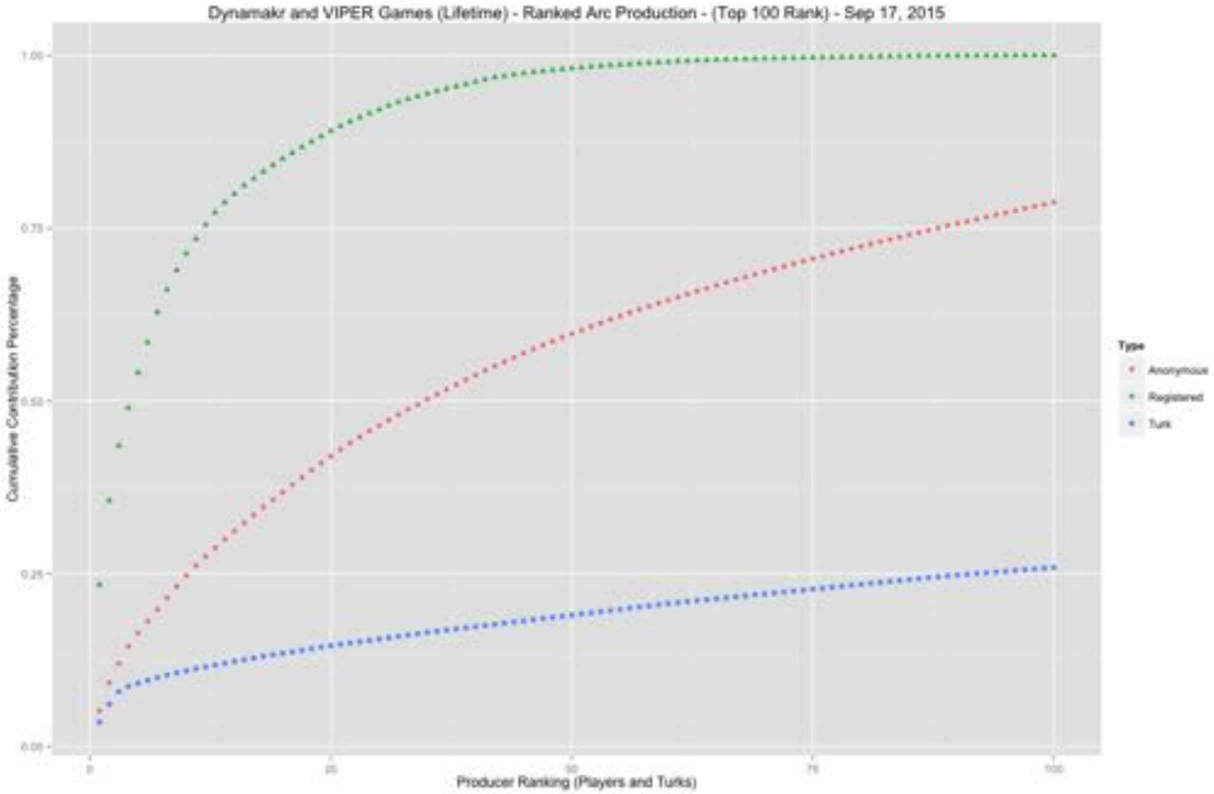


Figure 98: Cumulative arc contributions for the Dynamakr and VIPER game top 100 players, ranked in descending order of contribution. The *registered* and *anonymous* player statistics are for the Dynamakr game while the *turk* player statistics are for the VIPER game. Samples from 4 June 2015 live release through 17 September 2015.

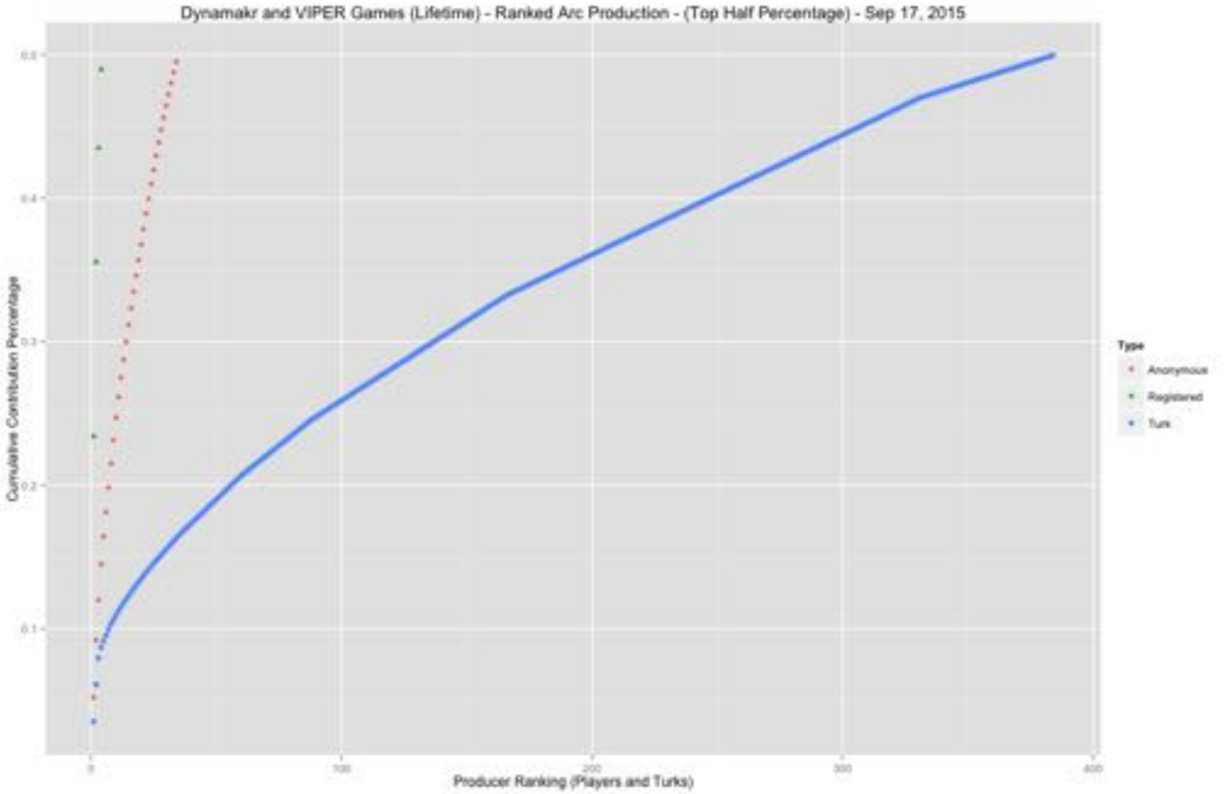


Figure 99: Cumulative arc contributions for the Dynamakr and VIPER game top 50 players, ranked in descending order of contribution. The *registered* and *anonymous* player statistics are for the Dynamakr game while the *turk* player statistics are for the VIPER game. Samples from 4 June 2015 live release through 17 September 2015.

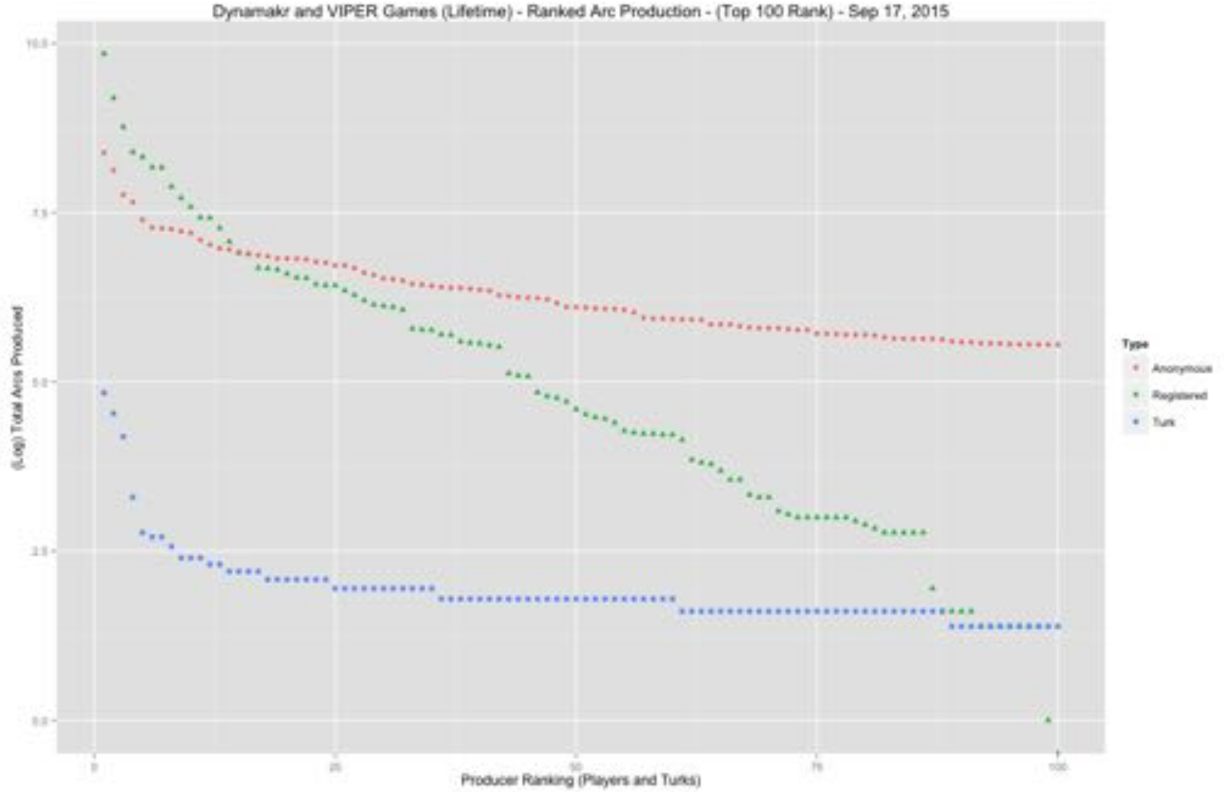


Figure 100: Arc production rank ordering for the Dynamakr and VIPER game top 100 players. The *registered* and *anonymous* player statistics are for the Dynamakr game while the *turk* player statistics are for the VIPER game. Samples from 4 June 2015 live release through 17 September 2015.

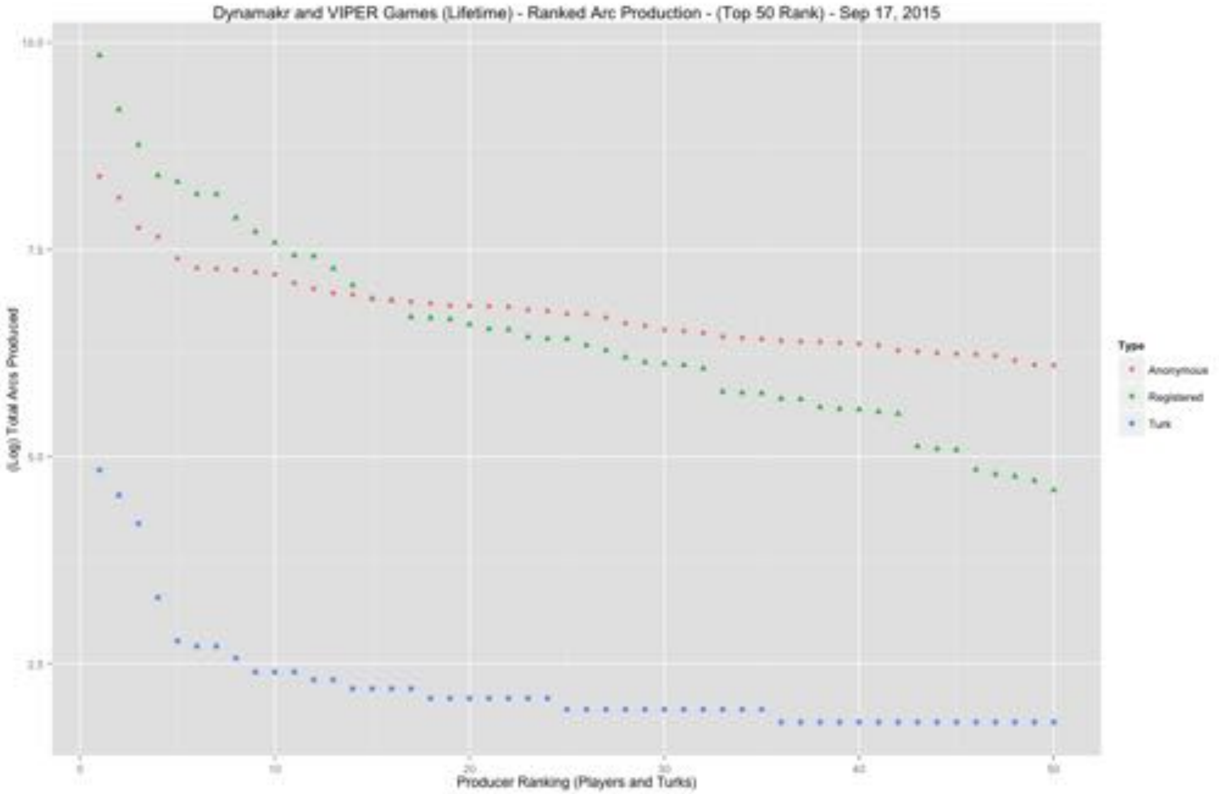


Figure 101: Arc production rank ordering for the Dynamakr and VIPER game top 50 players. The *registered* and *anonymous* player statistics are for the Dynamakr game while the *turk* player statistics are for the VIPER game. Samples from 4 June 2015 live release through 17 September 2015.

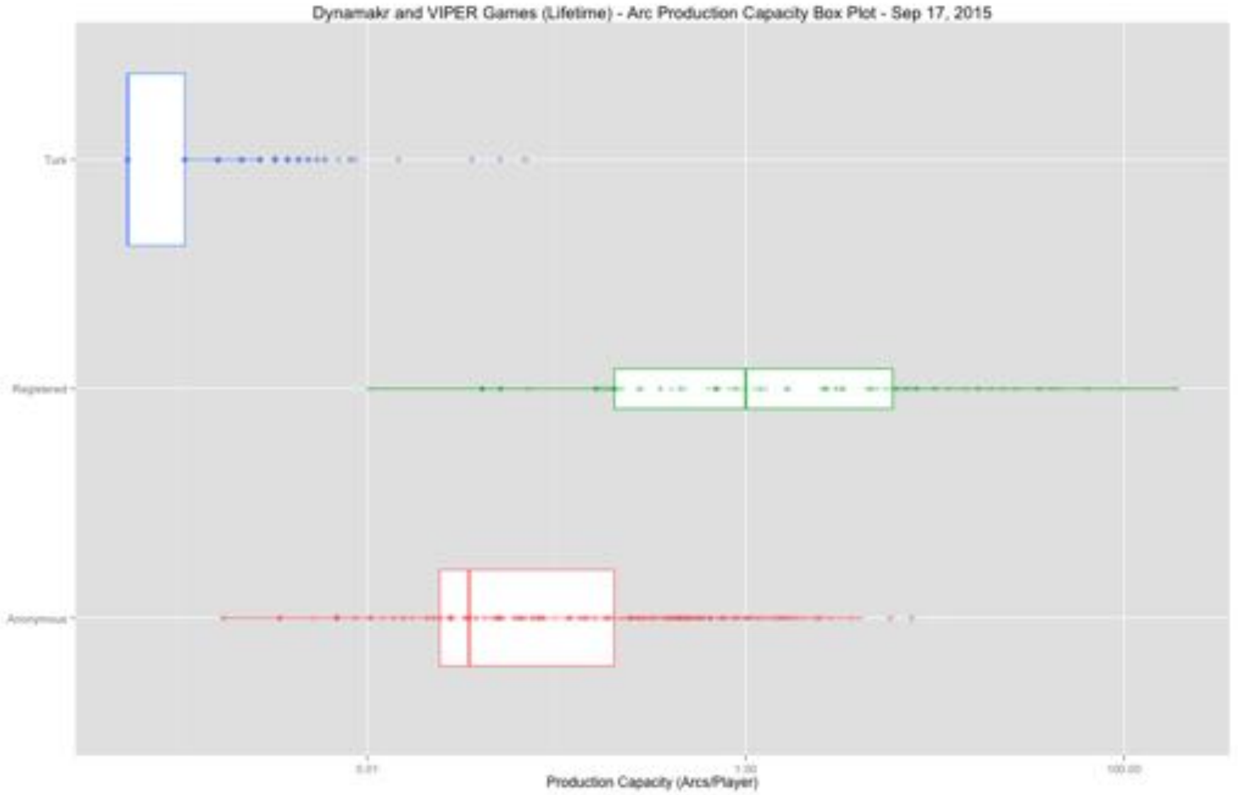


Figure 102: Arc production capacity statistics for the Dynamakr and VIPER game players, box-and-whisker statistic representation. The *registered* and *anonymous* player statistics are for the Dynamakr game while the *turk* player statistics are for the VIPER game. The boxes show the median and the first and third quartiles. The whiskers extend to 1.5 times the interquartile range. The samples of production capacity are the number of valid points-to graph arcs generated by game play per player, and omit cases of non-producing samples. Samples from 4 June 2015 live release through 17 September 2015.

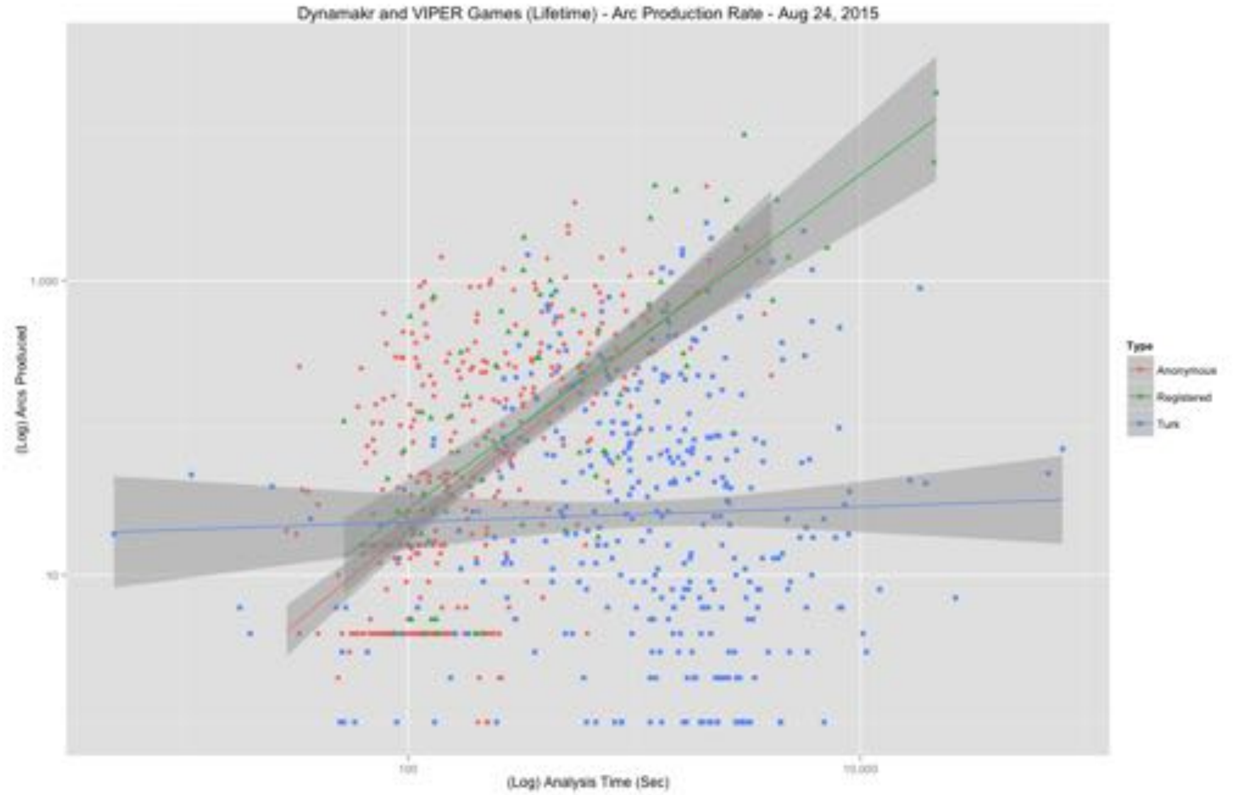


Figure 103: Arc production by analysis time (log scale, non-zero arc production cases only) for the combined Dynamakr and VIPER games. The data distinguish between the contributions of registered and anonymous Dynamakr players, and the qualified Mechanical Turk VIPER players. Fitted means with standard errors to 95% confidence are shown for each type. The production rates for the two game methods are not strictly comparable because the same game instance modules were not loaded in these cases, the same players did not play each case, and the relative contributions of auto-solvers have not been taken into account. Samples from 4 June 2015 live release through 17 September 2015.

4.6 Arcweaver Results

Arcweaver was our cloud-hosted auto-solver. This auto-solver was a C# source language solver compiled and linked with the *mono* libraries and built into a standalone executable for Linux. The solver algorithms applied our game model rules using a fixpoint search algorithm in brute force fashion, adding graph arcs per the rules until the arcs and intervals stabilized. We installed this solver onto an Amazon EC2 Linux instance and from that prepared an Amazon Machine Instance (AMI) that we could deploy from a script to process a given problem (see Figure 104 on page 219). We prepared a few dozen candidate target problems with CodeHawk which we loaded into a GitHub repository for retrieval by Arcweaver, then ran up to 20 Arcweaver solvers simultaneously to process the work queue. Some of these solvers ran for over 13 days of compute time. This section presents the results.

Figure 105 on page 220 presents the verification benefit sample statistics by program module type for the case of zero inlining depth. Here we see that the bulk of the benefit we obtain from the *program* module types, which are by far the largest modules and therefore contain the most verification conditions by count. Solving an additional 11% of all verification conditions across all module types, one would expect the largest benefit to be earned by the program modules. The Arcweaver solver, however, was unable to solve such large programs (owing to memory constraints of the virtual machine host operating systems) for deeper levels of inlining where the problems are say up to 100-times larger at three levels of inlining depth. Figure 106 on page 221 drops the program module types to example the remaining types in more detail at depth zero, where we observe there is little verification benefit for the library module types. This too is to be expected because it would be unusual to perform whole-program analysis on just a library module where the analyzer can resolve very little without specific entry points.

Figure 107 on page 222 and Figure 108 on page 223 present the verification marginal benefit picture as verification conditions improved by verification effort benefit. The figures distinguish the samples by depth of inlining and module type. The second figure uses a log scale for better separation along the improvement axis. The benefit includes the economic value of the verification condition minus the computing cost of obtaining that verification result from the solver. Figure 109 on page 224 then applies the log to the benefit axis and fits models to 95% standard error on depth of inlining samples, unsurprisingly finding a nice discrimination of points with this benefit model. As we increase the constraint generator's depth of inlining we increase substantially the number of verification conditions required to prove for each module (e.g. 6 times for level 1, 28 times for level 2, 100 times for level 3, and so on) which in turn decreases the economic benefit of an individual verification condition by our economic value model. Our economic model, and the constraint generator, are independent of the module type. Figure 110 on page 225 nevertheless shows that reasonable fits might be made in the log-log space along the type characteristic, regardless of depth, for these data. In some cases, such as the program types, there are not enough Arcweaver result samples to make reasonable fits.

Figure 111 on page 226 presents the Arcweaver results by module identifier sorted by verification condition improvements. Because the program types on the whole are larger modules and have more verification conditions required to prove, these yielded larger total improvements. Similarly, the auto-solver results for the deeper inlining cases that also had more verification conditions required to prove yielded larger total improvements. If instead of total verification conditions we examine the percent of verifications improved (of the total required for the particular module) we obtain the result of Figure 112 on page 227. Using percentages shows the contributions are a little more evenly distributed but the large program modules still benefit the most, which is good news for verification overall. The figure shows the mean improvement for these results at over 11% improvement in safe verification condition count per module. The unusually high counts for the modules on the right-hand side of the figure, for library modules, might be discounted because it would be unusual to analyze library modules in isolation.

Sorting by module identifier, we present the before-and-after safe percentage improvement values by depth of inlining in Figures 113 on page 228, 114 on page 229, 115 on page 230, and 116 on page 231. The Arcweaver auto-solver was unable to complete fixpoint iterations for the larger problems at deeper levels of inlining so there are few samples there for analysis. Because depth of inlining turns out to be so important for the results we can filter the data along this value. Figure 117 on page 232 presents the before-and-after safe verification condition count sorted by module, then Figure 118 on page 233 presents the running total in cumulative safe count improvement, which is a way of showing the total improvement for all of BIND at depth zero; that is, nearly 200,000 new safe verification conditions proved correct automatically (about 11% of the total) as a result of this non-relational pointer analysis, never done before CSFV. At depth of inlining one, Figures 119 on page 234 and 120 on page 235 present the spot and cumulative values, again amounting to about 11% of the 12 modules that were solvable by Arcweaver. Figures 121 on page 236 and 122 on page 237 present the results for inlining depth 2, and Figures 123 on page 238 and 124 on page 239 present the results for inlining depth 3.

Figure 125 on page 240 presents solution benefit samples (verification improvement ratio by the log of verification benefit dollars) in a panel grid of inlining depth and module type. The library modules types are not shown for reasons given earlier. At the deeper levels of inlining we have fewer successful Arcweaver fixpoint solutions so fewer — or no – samples shown in the panels. One can see the verification benefit of deeper levels of inlining that we expect with static analysis, and this effect is enhanced with our game model and solvers. Figure 126 on page 241 presents a similar panel grid for the Arcweaver solution rates data (log of arcs generated per hour by log of verification conditions improved per hour). The deeper levels of inlining create much larger constraint problems that require more computing time to solve, so the arc generation rates are somewhat slower, yet their improvement rates are nevertheless reasonable and, as shown in the previous grid, their verification improvement ratios and economic benefits are valuable. One can also see in the rates data that the larger P-type modules, the full programs which are the largest of the modules, require longer solution times even at zero levels of inlining and therefore lower production rates than the smaller T- and U-type modules.

Combining the solution rate samples onto the same grid we might explore type and depth relationships. Figure 127 on page 242 presents the samples on linear scales of arcs generated per hour by verification conditions improved (safe counts improved) per hour by the Arcweaver solver. The samples are distinguished by the constraint generator depth of inlining and the module type L for library, P for program, T for test, and U for utility per the BIND manual. On this plot the L-type modules stand out as generating many arcs and improved verification conditions per hour. In order to spread out the other samples we use log scales as shown in Figures 128 on page 243 and 129 on page 244. The latter figure includes an attempt to fit a linear model of the samples as a single group to 95% confidence, which does not fit especially well to these data in this way as we see by the standard error bands. We show a better model below. Because the L-type modules are shown in these figures to be unusual, and because we know their standalone static analysis value is dubious, we repeat the analysis without these modules included. Figure 130 on page 245 presents the samples on the linear scales of arcs generated per hour by verification conditions improved (safe counts improved) per hour. The samples are better distributed without the library samples, but the log scales will help again. Figures 131 on page 246 and 132 on page 247 present the samples with log-linear and log-log scales. The latter figure includes linear models fitted to the samples of each module type with 95% confidence, and we see much better fits in this case. There are three fits shown, two of which (the T and U type fits) overlap.

The next series of figures explore Arcweaver's economic value by way its rate of delivering verification improvements. Figures 133 on page 248, 134 on page 249 and 135 on page 250 show the Arcweaver samples of verification benefit rate (dollars per hour) by verification conditions improved, in an inlining depth grid, with combinations of linear and log scales. The samples are distinguished by BIND module type, but the plots do not include the library module types. At zero depth of inlining, the smaller modules deliver benefits at faster rates but smaller verification improvement quantities, while the program types are slower to solve but yield higher verification improvements. At higher levels of inlining, the benefit rate is slower for all modules but the verification quality is higher. Unfortunately we did not obtain fixpoint solution samples from Arcweaver for the larger modules at higher levels of inlining. Figure 136 on page 251 presents the log-log version of these samples with the different panel arrangement, this time with module type panels and the samples distinguished by depth of inlining. Here we see interesting clustering of samples by depth of inlining. This result is intuitive because similar modules are roughly the same size, have roughly the same number of total verification conditions, link the same libraries, and so on. That is, the test modules are roughly the same kind of analysis program and the utility modules are roughly kind of analysis problem. To the constraint generator and Arcweaver solver, they appear similar from a pointer-flow constraints standpoint and yield similar points-to graphs. They require about the same time to solve. They resolve many of the same open verification conditions.

Finally, we explore some of the distribution statistics of the samples using the familiar box-and-whisker representation. In the boxes the inner line is the median of the samples and the edges are the first and third quartiles. The whiskers extend to 1.5 times the inter-quartile range. Any outliners beyond the whiskers are shown as points. Figure 137 on page 252 shows the distribution statistics of the samples for verification benefit rate (dollars per hour using our verification condition to SLOC scale model) by module type and depth of inlining. The library module types are not shown. As mentioned above, many of the module type samples have similar analysis profiles at each level of inlining, so the sample statistics present fairly narrow distributions. Figure 138 on page 253 presents in a similar fashion the samples of verification conditions produced per hour, again revealing fairly narrow distributions on types owing to the design of the target program being analyzed. Figure 139 on page 254 presents the distributions for verification condition improvement rate (safe count increase per hour) by module type regardless of inlining depth. Interestingly the distributions for the utility and test modules are similar even though the program designs are quite different. The distribution for the program modules is unique, and quite narrow, although we have just a few samples at shallow inlining levels for the Arcweaver auto-solver. The module distribution similarity ends if we separate the samples by inlining depth, however, as shown in Figure 140 on page 255. Here we see the zero-inlining cases provide faster improvement rates overall (they are smaller analysis problems), while the deeper-inlining cases provide incrementally slower improvement rates at each depth (they are larger analysis problems). These results are intuitive because each level of inlining depth dramatically expands the number of verification conditions and pointer flow constraints that must be solved and analyzed. Figure 141 on page 256 provides the dot plot for these distributions, replacing the box-and-whisker summary with the stacked dots representing the samples.

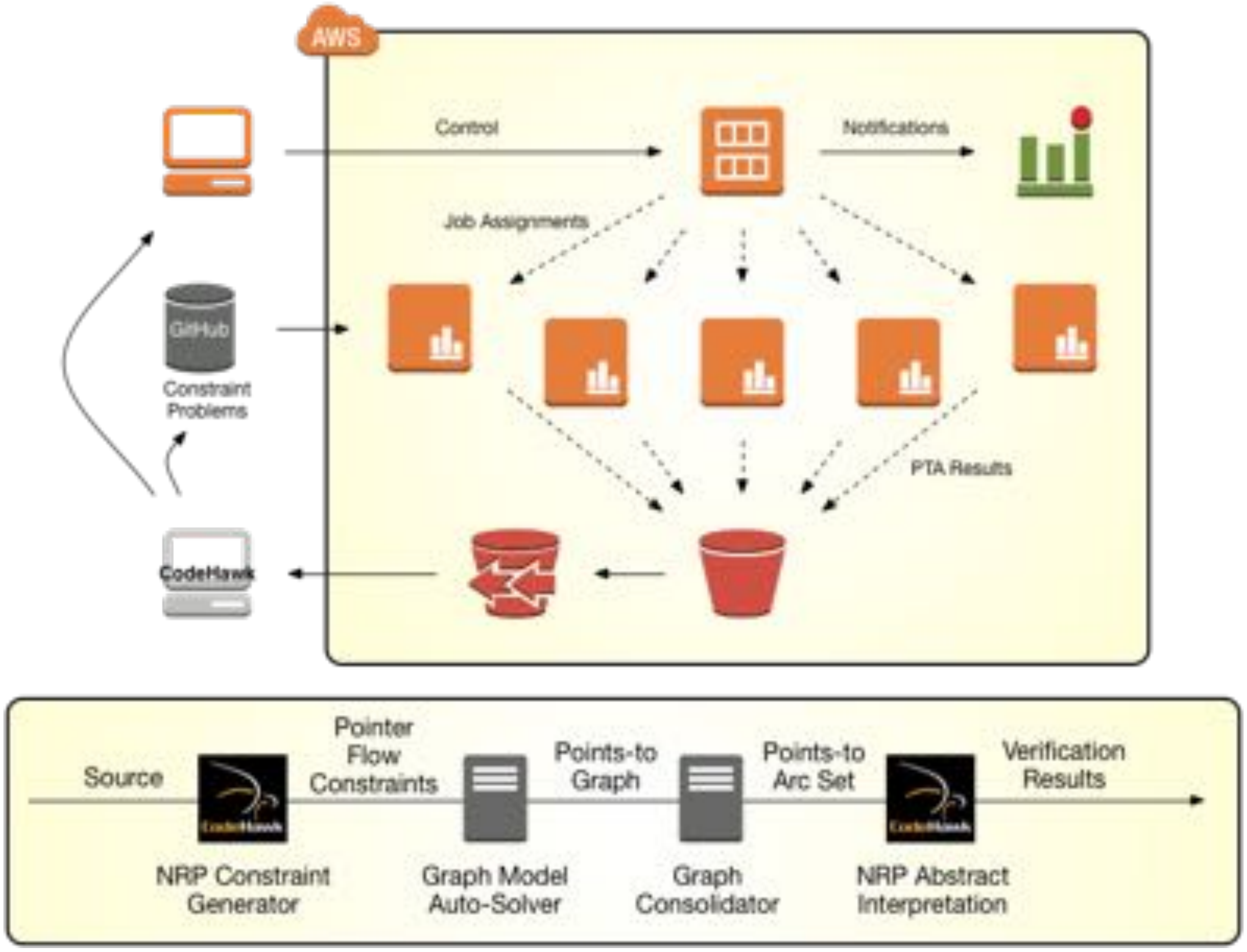


Figure 104: Arcweaver cloud-hosted auto-solver architecture overview. The solver programs run on specially-configured Amazon machine instances deployed by a manager instance running in the Amazon cloud. The manager pulls constraint problem candidates prepared by CodeHawk from a GitHub repository pool and assigns them to the AMI workers, which process the problems to completion (or failure) then provide notifications through the simple notification service (SNS) topics and subscriptions, then terminate the instance. The instances store any logs and results to an Amazon S3 bucket for further processing or retrieval. The results are points-to arc sets for the constraints problem. We then feed the results to the CodeHawk analyzer, together with the original constraints, dictionary, and anchors file for the problem, to obtain the after-game-play verification condition counts.

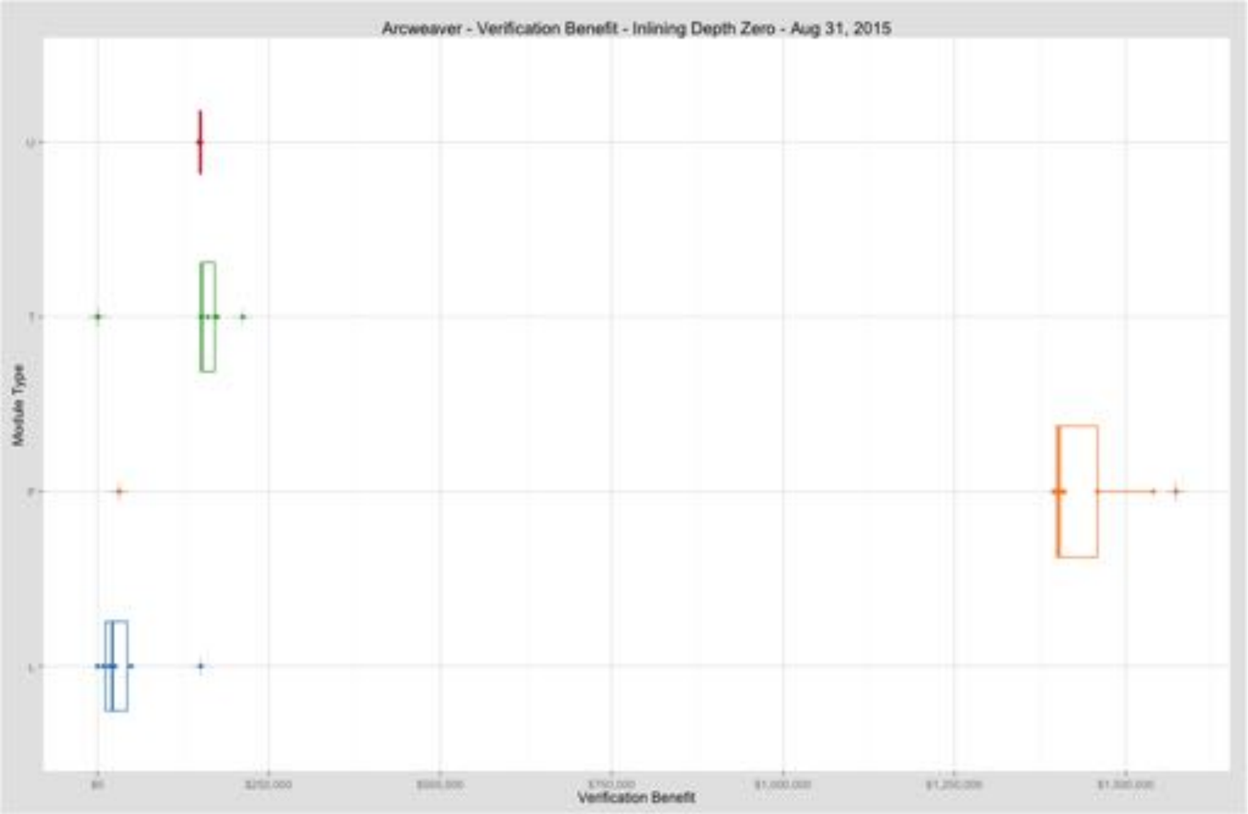


Figure 105: Arcweaver auto-solver estimated verification benefit for the constraint generator using zero inlining, by module type (L for library, P for program, T for test, and U for utility per the BIND manual). Box-and-whisker representation of sample statistics. The boxes bound the median and the first and third quartiles. The whiskers extend to 1.5 times the interquartile range. Overall, the verification benefit for a typical P type module is higher because it has substantially more verification conditions improved by the game model and analysis, with each such improvement earning a constant dollar benefit. At the same time, the total benefit requires proportionally more time to be accomplished. Data and analysis are current through 31 August 2015.

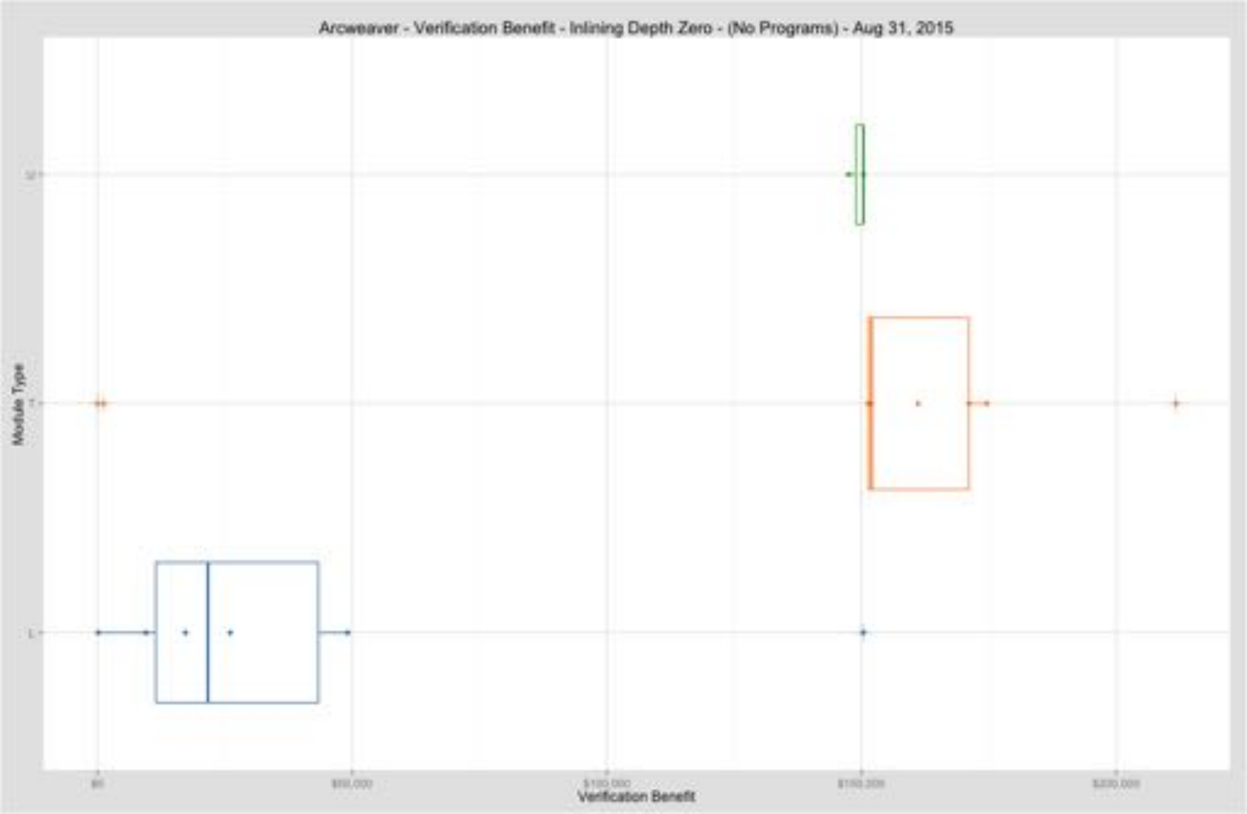


Figure 106: Arcweaver auto-solver estimated verification benefit for the constraint generator using zero inlining, by module type (*L* for library, *T* for test, and *U* for utility per the BIND manual). The type *P* for program samples have been hidden in order to expand the details for the other types. Box-and-whisker representation of sample statistics. The boxes bound the median and the first and third quartiles. The whiskers extend to 1.5 times the interquartile range. Overall, the verification benefit for library modules is lower because there is lower expected value in performing static analysis on a library as a standalone artifact; the analysis is better performed when the library is linked with a main program as with the test, utility, and program type modules. Data and analysis are current through 31 August 2015.

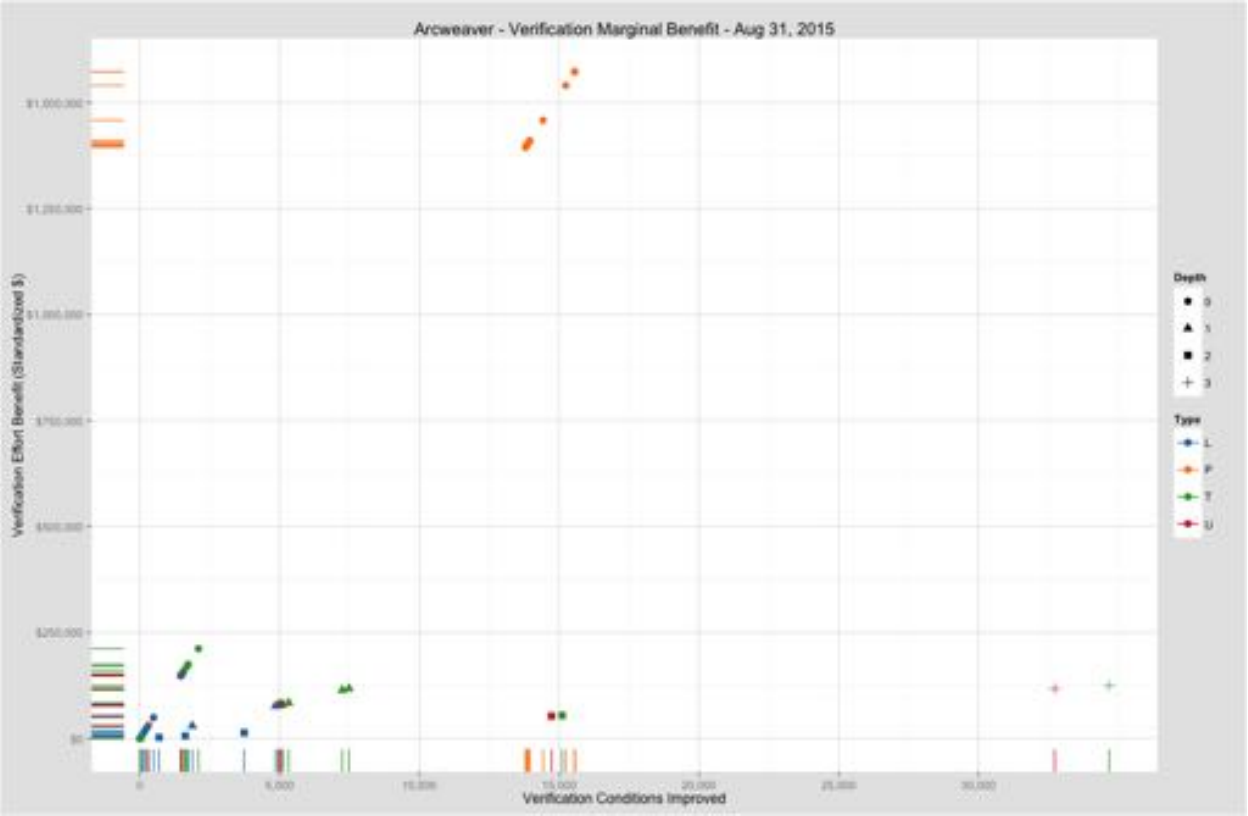


Figure 107: Arcweaver auto-solver verification marginal benefit, represented as verification conditions improved (safe conditions increased as a result of game play) versus verification effort benefit in dollars. For the effort benefit we use our verification conditions and source lines of code proportional model described in the text. The samples are shown distinguished by module type and depth of inlining. The few samples of type P near the top of the plot indicate the relatively high value of verification condition improvements for programs, owing to the relatively high quantity of verification conditions even at low levels of inlining. By contrast, we see in the bottom right-hand corner with three levels of inlining there are many more verification conditions to improve (over 27 times the quantity at depth zero) so their individual value and overall contribution to benefit is substantially lower. Data and analysis through 31 August 2015.

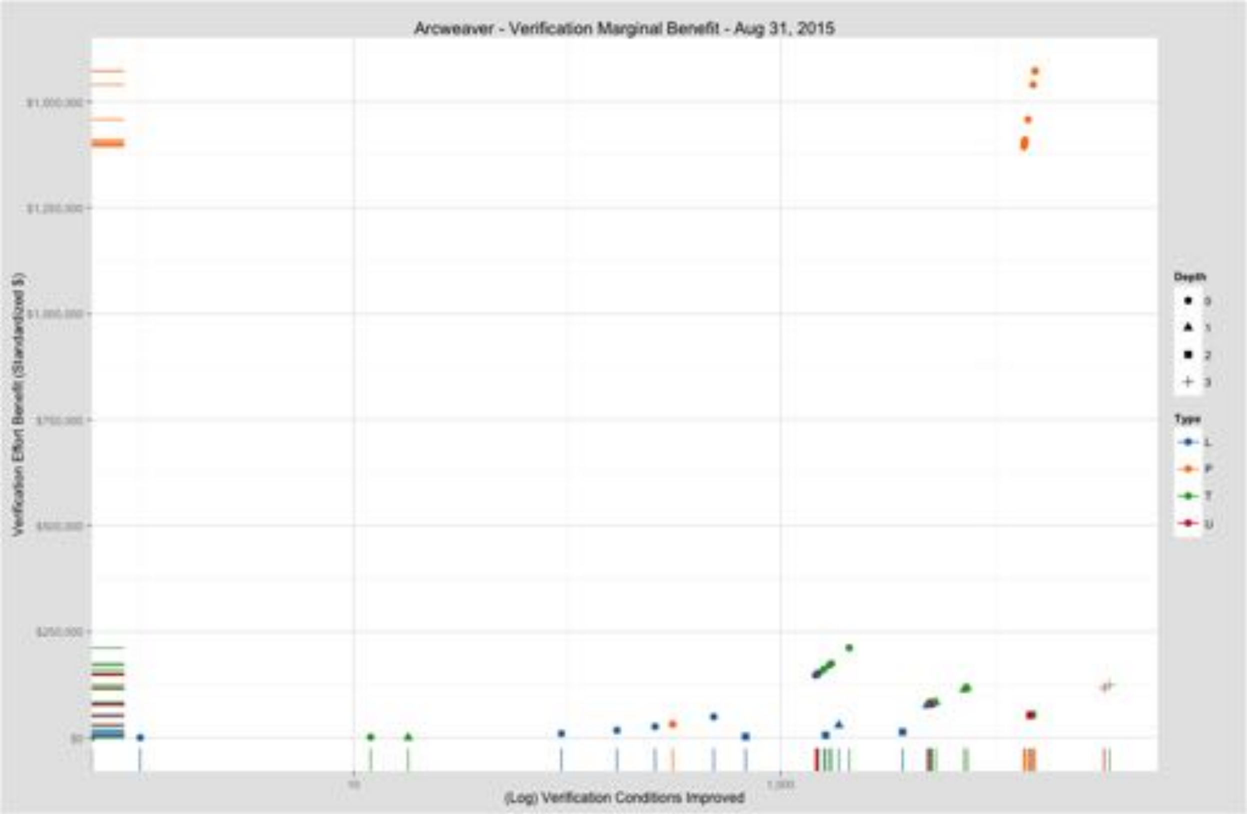


Figure 108: Arcweaver auto-solver verification marginal benefit, represented as the log of verification conditions improved (safe conditions increased as a result of game play) versus verification effort benefit in dollars. For the effort benefit we use our verification conditions and source lines of code proportional model described in the text. The samples are shown distinguished by module type and depth of inlining. The log scale is used to expand the detail of the verification conditions count. The few samples of type P near the top of the plot indicate the relatively high value of verification condition improvements for programs, owing to the relatively high quantity of verification conditions even at low levels of inlining. Data and analysis through 31 August 2015.

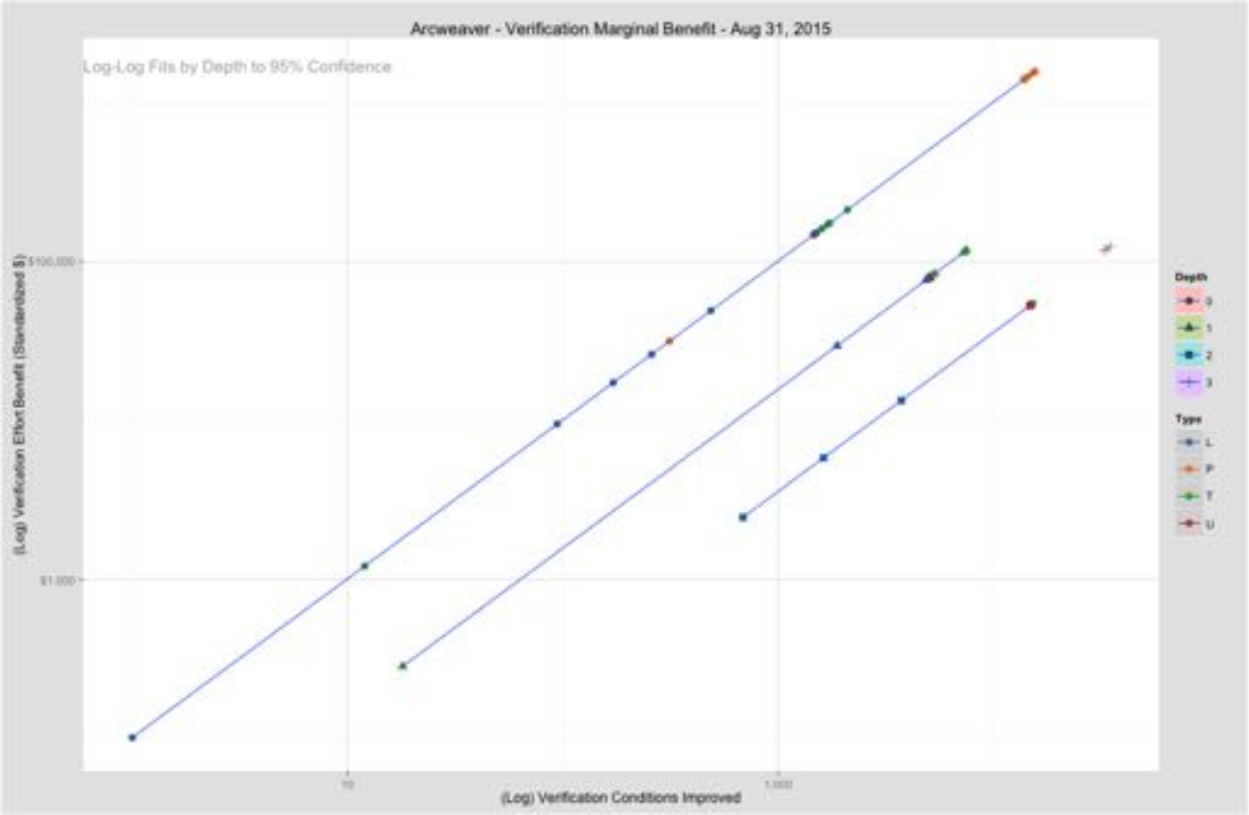


Figure 109: Arcweaver auto-solver verification marginal benefit, represented as the log of verification conditions improved (safe conditions increased as a result of game play) versus the log of verification effort benefit in dollars. For the effort benefit we use our verification conditions and source lines of code proportional model described in the text. The samples are shown distinguished by module type and depth of inlining. The log-log scale is used to expand the detail on both scales, and the log fit on depth serves to highlight the effect of inlining on the number of verification conditions and benefit. Because inlining increases the verification conditions count for each module, it also decreases in proportion the relative value of each safe result and its contribution to verification benefit, which appears as a nice separation by inlining level in such a plot. Data and analysis through 31 August 2015.

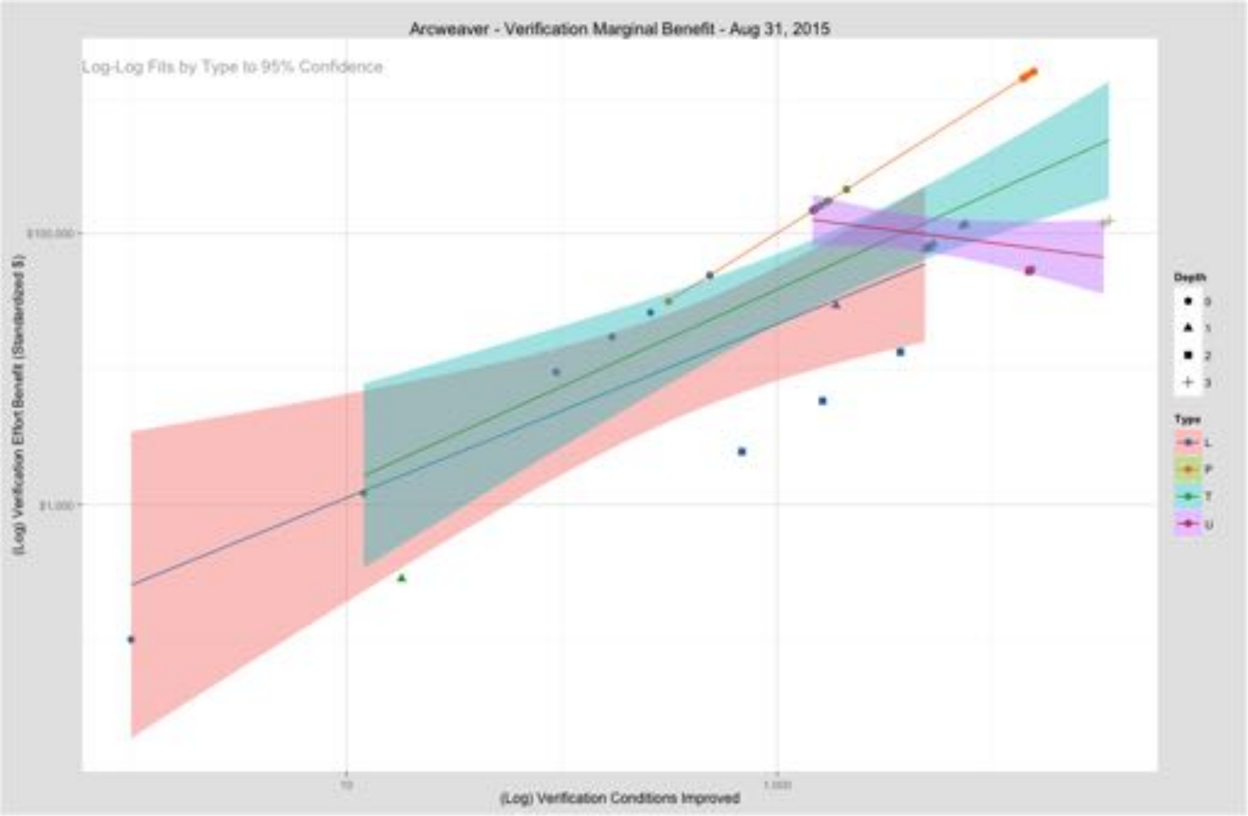


Figure 110: Arcweaver auto-solver verification marginal benefit, represented as the log of verification conditions improved (safe conditions increased as a result of game play) versus the log of verification effort benefit in dollars. For the effort benefit we use our verification conditions and source lines of code proportional model described in the text. The samples are shown distinguished by module type and depth of inlining. The log-log scale is used to expand the detail on both scales. The fit on module type does not discriminate the data as much as depth does, other than perhaps to repeat our refrain that the library modules ought not be analyzed in isolation. Data and analysis through 31 August 2015.

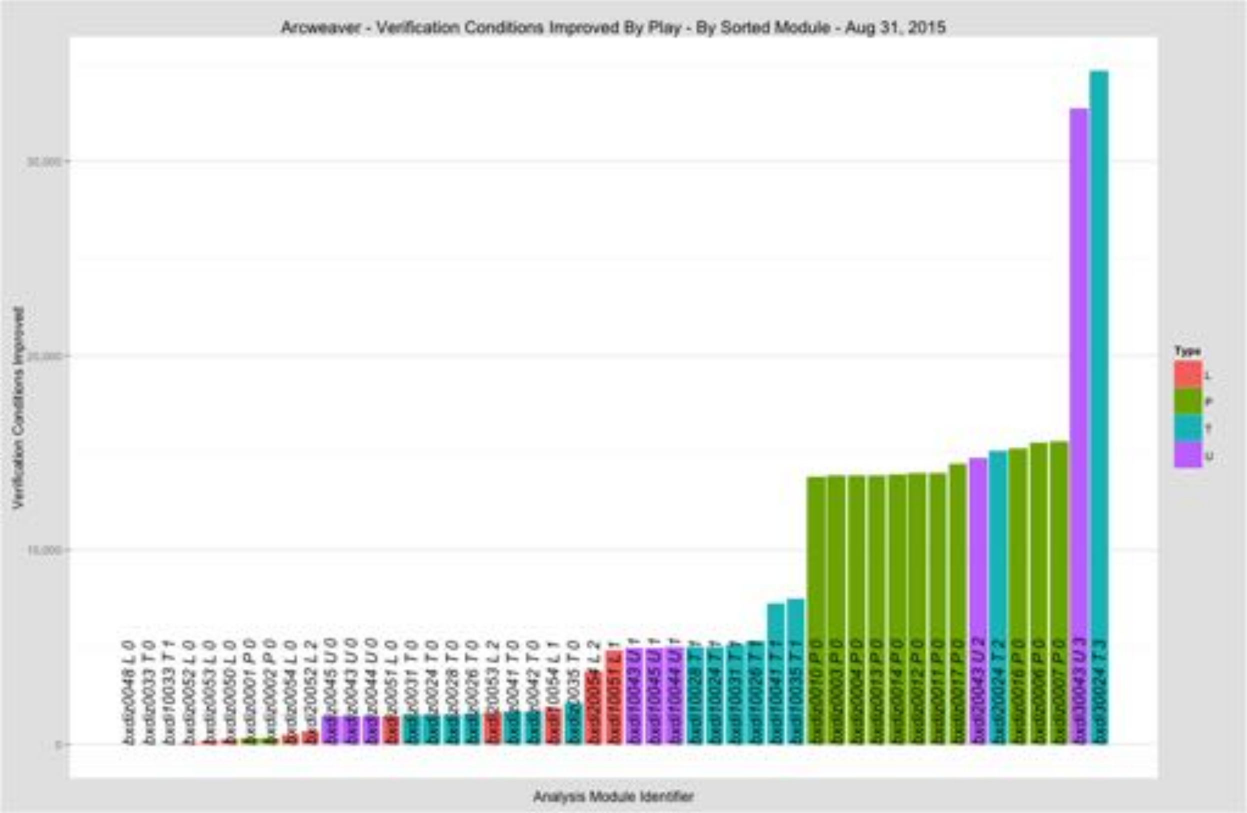


Figure 111: Arcweaver auto-solver results represented by BIND module total safe verification condition improvements as a result of applying our game model. The module identifiers are sorted by increasing verification condition improvement count, and color-coded by module type. The module identifier label also indicates its type L for library, P for program, T for test, and U for utility per the BIND manual, and by the depth of inlining 0 through 3 applied at the time of constraint generation. In general, the larger program types P benefitted the most from game play, as expected, as did some utilities and tests with high levels of inlining. Increasing the levels of inlining increases the number of verification conditions for the module. Data and analysis through 31 August 2015.

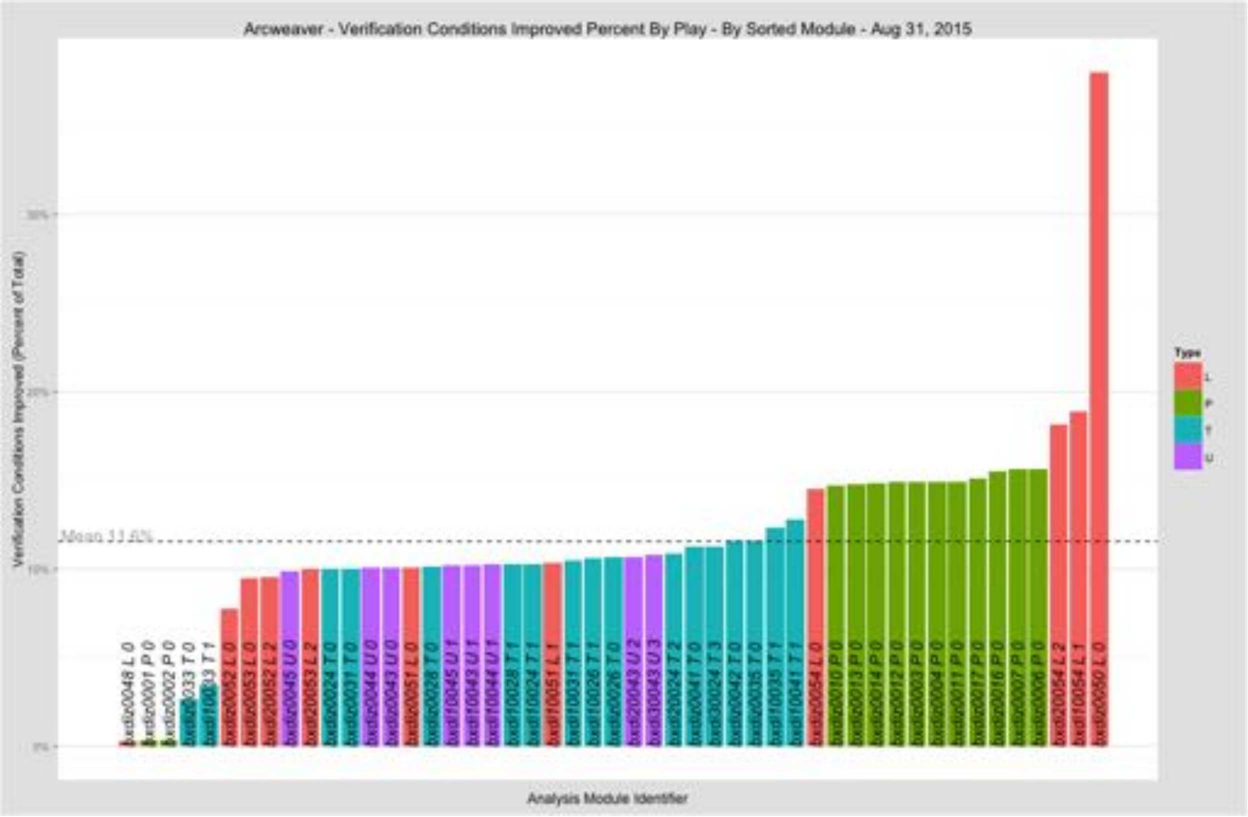


Figure 112: Arcweaver auto-solver results represented by BIND module percent of total verification condition improvements as a result of applying our game model. The improvements represent an increase in the safe verification condition count as a fraction of the total required. The module identifiers are sorted by increasing verification condition improvement percent, and color-coded by module type. The module identifier label also indicates its type L for library, P for program, T for test, and U for utility per the BIND manual, and by the depth of inlining 0 through 3 applied at the time of constraint generation. In general, the larger program types P benefitted the most from game play, as expected, while the library modules might be treated cautiously for their analysis in isolation. The mean percent improvement – considering only the non-relational pointer analysis performed – is shown for this data set at over 11%. Data and analysis through 31 August 2015

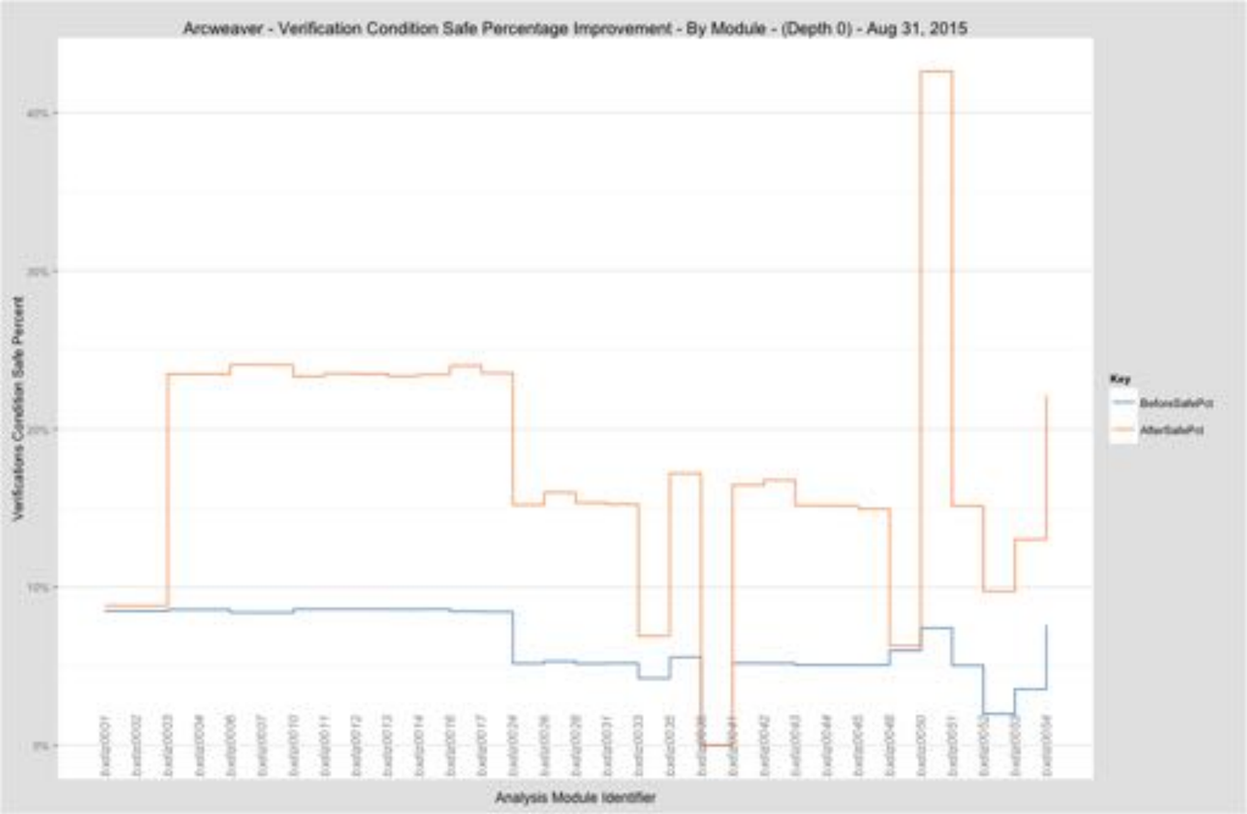


Figure 113: Arcweaver auto-solver results presented as before-and-after safe verification condition percentages by module. The modules are listed in numerical order and results given without inlining. The blue line represents the percentages proven safe before application of our techniques. The orange line represents the percentages proven safe afterward. On the whole, the program modules listed on the left-hand side benefit the most from the application of the game model techniques, increasing from less than 10% proven safe to nearly 25% proven safe. Considering these also are the largest programs with the most verification conditions overall, this is where most of the verification effort occurs. Data and analysis through 31 August 2015.

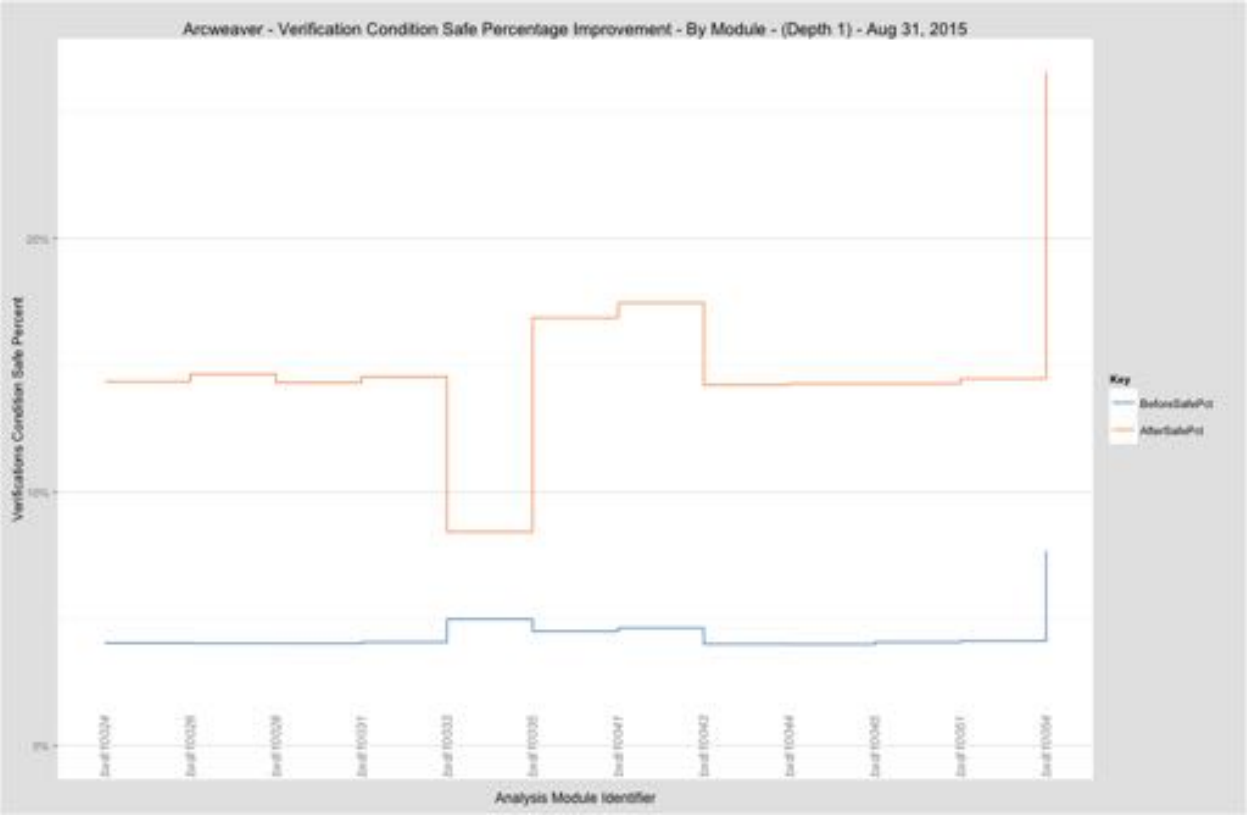


Figure 114: Arcweaver auto-solver results presented as before-and-after safe verification condition percentages by module. The modules are listed in numerical order and results given with one level of inlining. The blue line represents the percentages proven safe before application of our techniques. The orange line represents the percentages proven safe afterward. Because these problems are at least six times larger than the zero-inlining cases, the solver was unable to complete fixpoint iteration for any of the full programs of type P. Consequently we are unable to compare the impressive safe percentage improvements for type P for zero inlining with deeper inlining cases. Data and analysis through 31 August 2015.

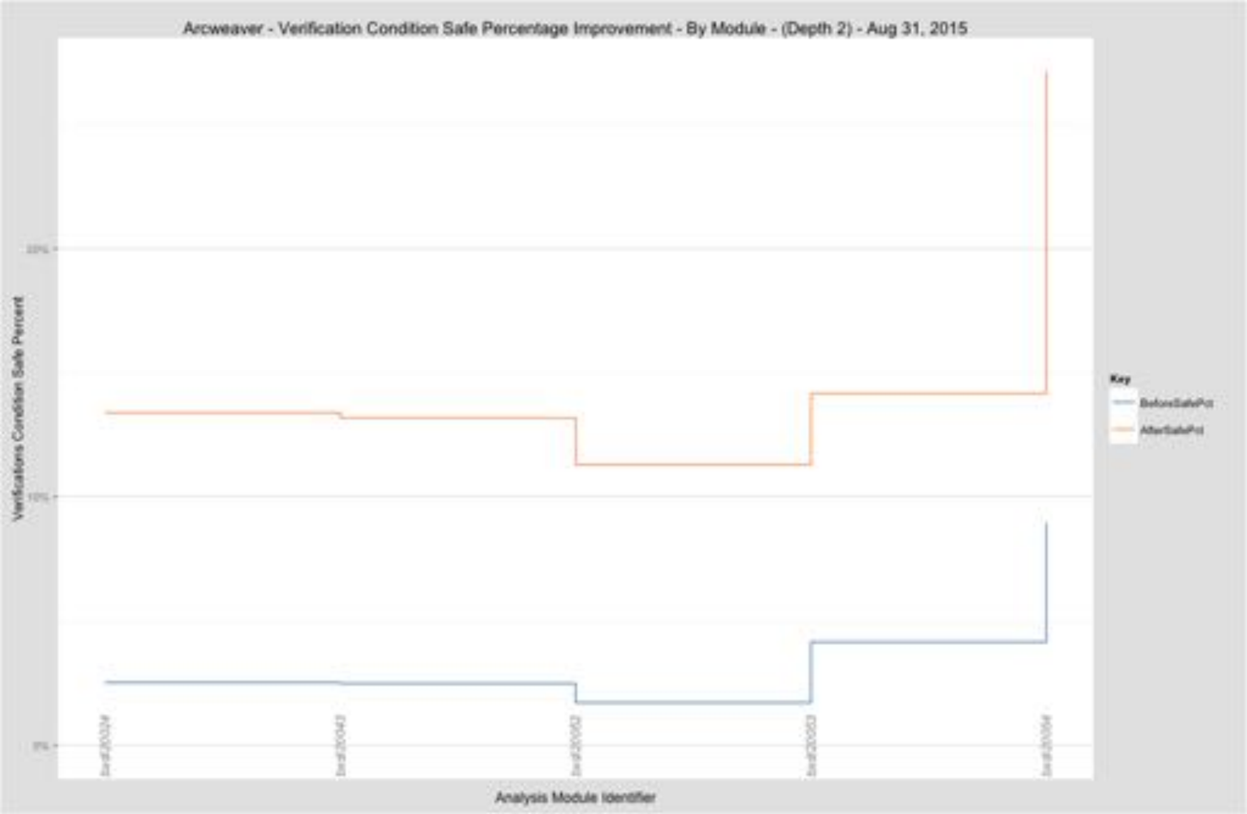


Figure 115: Arcweaver auto-solver results presented as before-and-after safe verification condition percentages by module. The modules are listed in numerical order and results given with two levels of inlining. The blue line represents the percentages proven safe before application of our techniques. The orange line represents the percentages proven safe afterward. Because these problems are at least 28 times larger than the zero-inlining cases, the solver was unable to complete fixpoint iteration for any of the full programs of type P and all but one of the utility programs of type U. Consequently we are unable to compare many of the improvements for these types found for zero inlining with deeper inlining cases. Data and analysis through 31 August 2015.

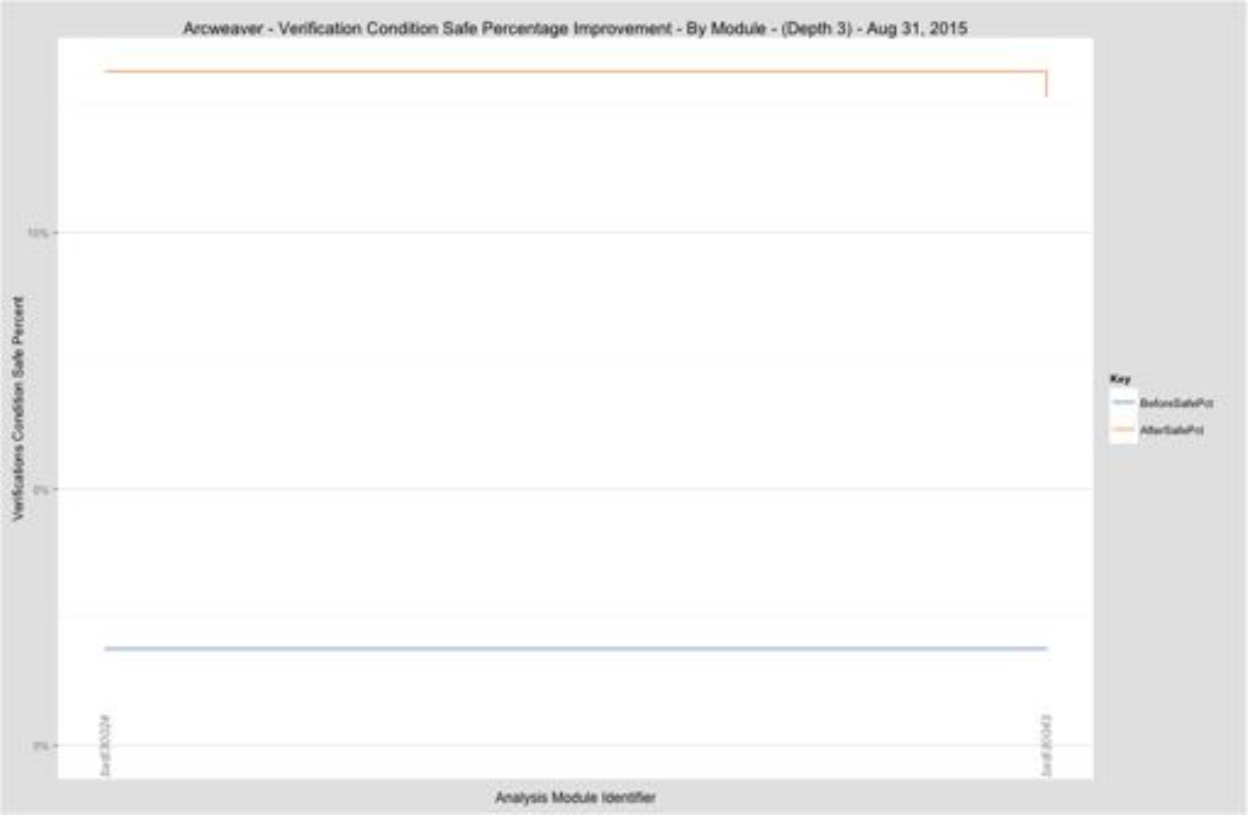


Figure 116: Arcweaver auto-solver results presented as before-and-after safe verification condition percentages by module. The modules are listed in numerical order and results given with three levels of inlining. The blue line represents the percentages proven safe before application of our techniques. The orange line represents the percentages proven safe afterward. Because these problems are at least 100 times larger than the zero-inlining cases, the solver was able to complete fixpoint iteration for only two of the smaller modules: one test and one utility. Consequently we are unable to compare many of the improvements for these types found for zero inlining with deeper inlining cases. Data and analysis through 31 August 2015.

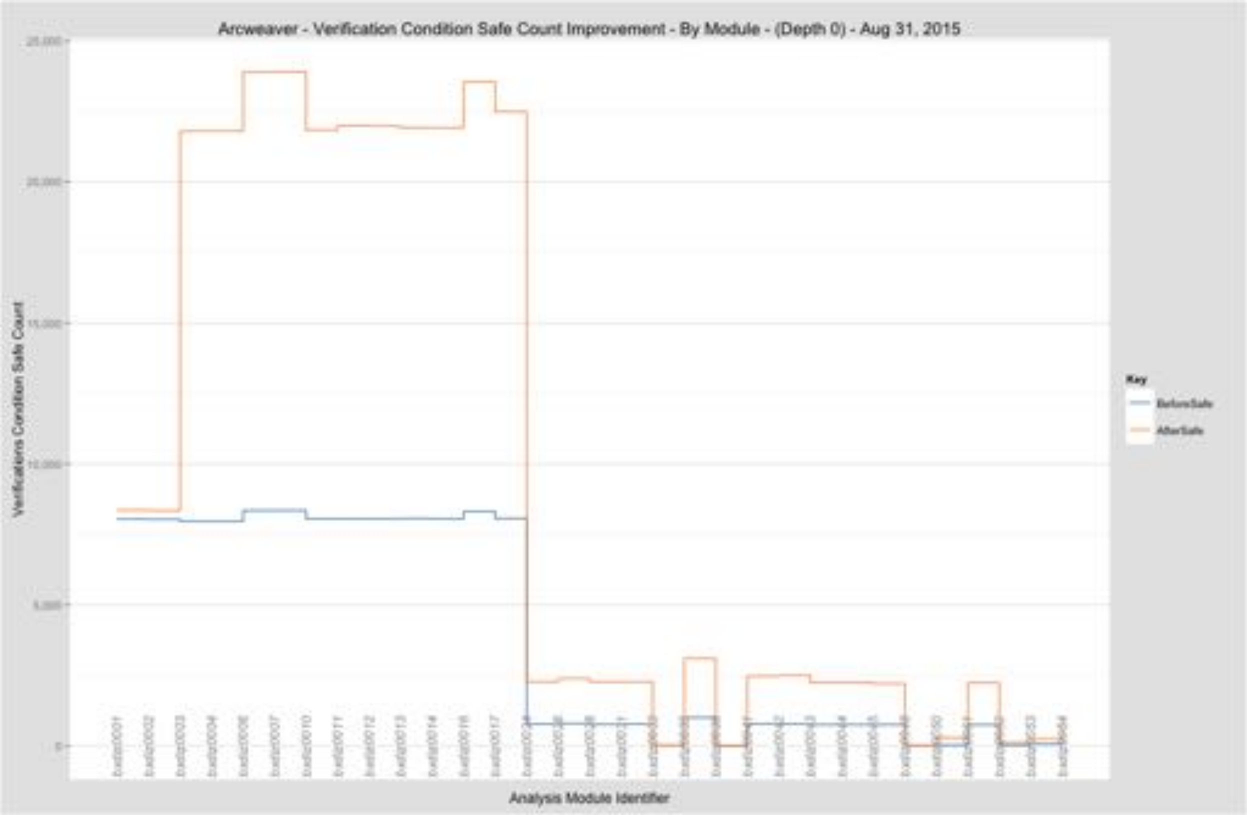


Figure 117: Arcweaver auto-solver results presented as before-and-after safe verification condition count improvements by module with zero levels of inlining. The module arrangement is by identifier. The blue line represents the count of safe verification conditions before game play. The orange line represents the count of safe verification conditions after game play. Data and analysis through 31 August 2015.

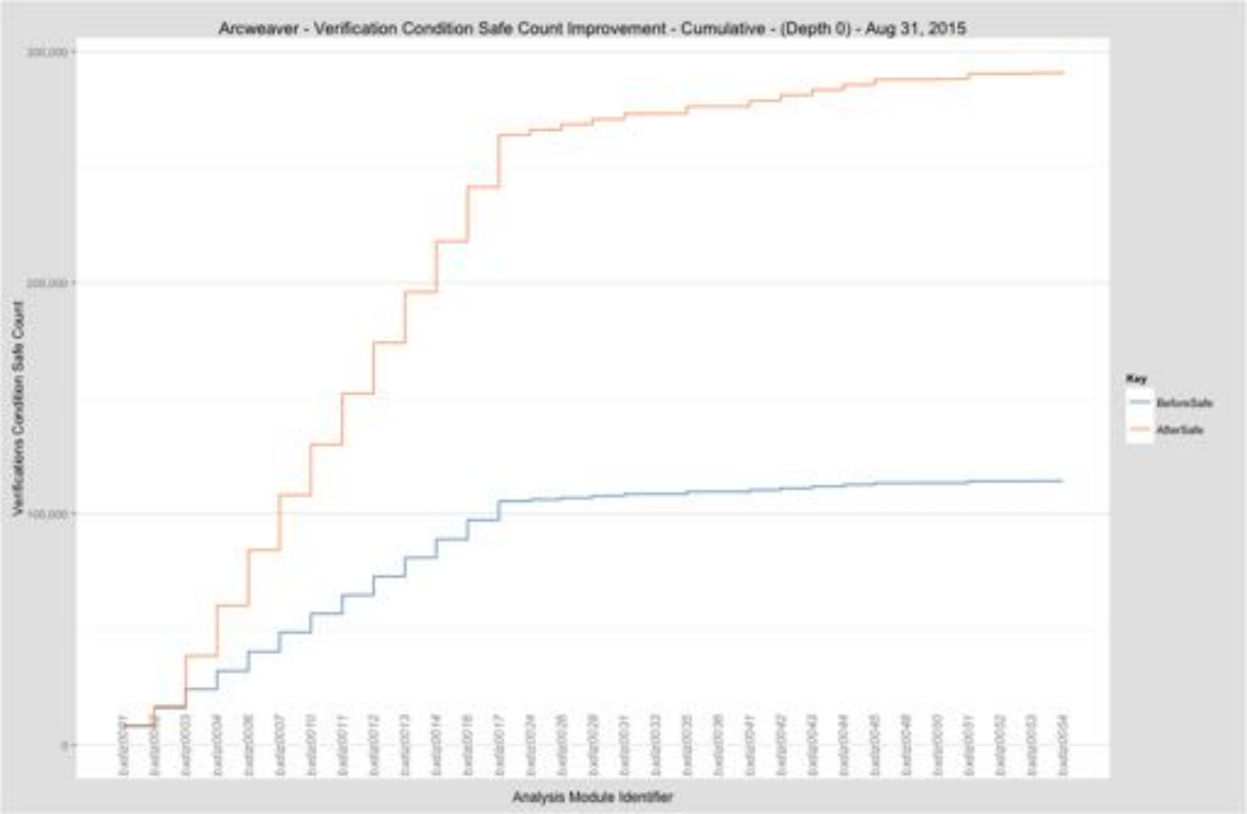


Figure 118: Arcweaver auto-solver results presented as before-and-after safe verification condition count cumulative improvements by module with zero levels of inlining. The module arrangement is by identifier. The blue line represents the cumulative count of safe verification conditions before game play. The orange line represents the cumulative count of safe verification conditions after game play. The overall result is roughly 11% improvement in total verification closure. Data and analysis through 31 August 2015.

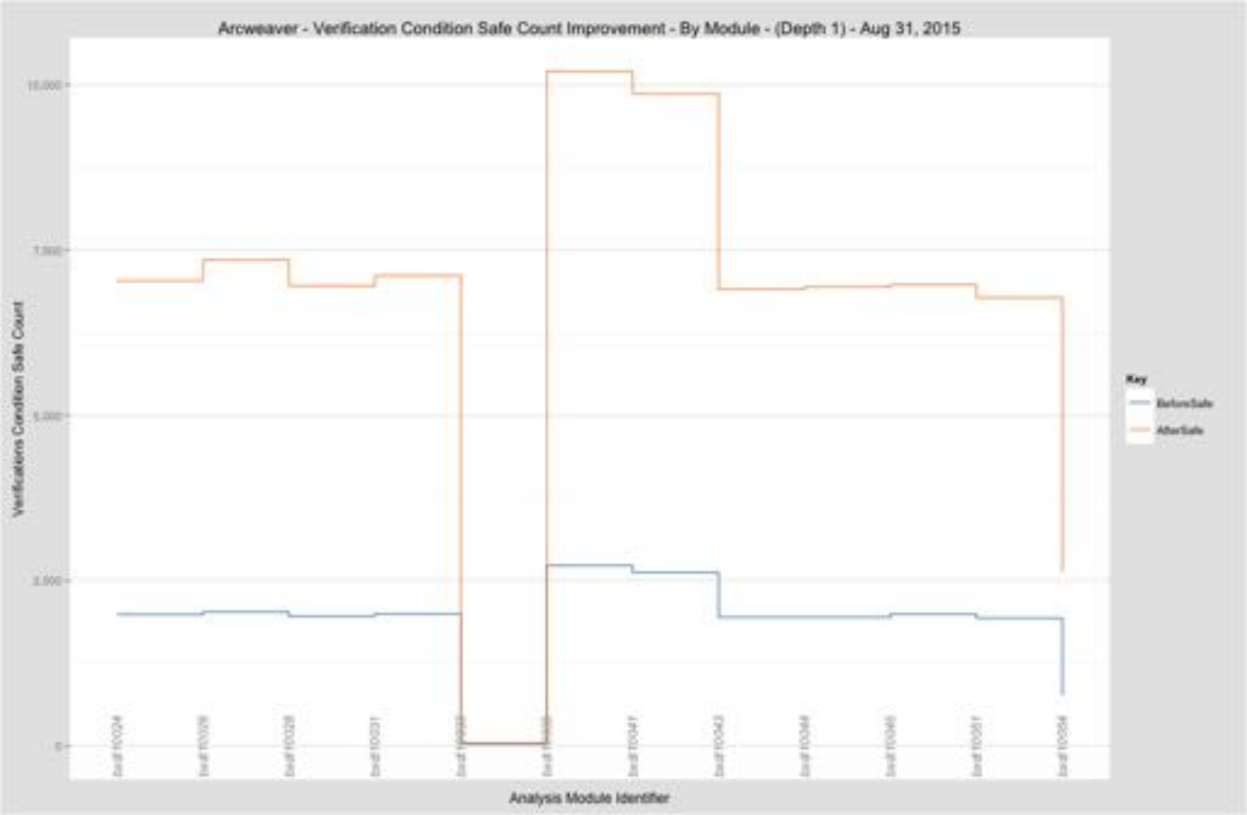


Figure 119: Arcweaver auto-solver results presented as before-and-after safe verification condition count improvements by module with one level of inlining. The module arrangement is by identifier. The blue line represents the count of safe verification conditions before game play. The orange line represents the count of safe verification conditions after game play. Data and analysis through 31 August 2015.

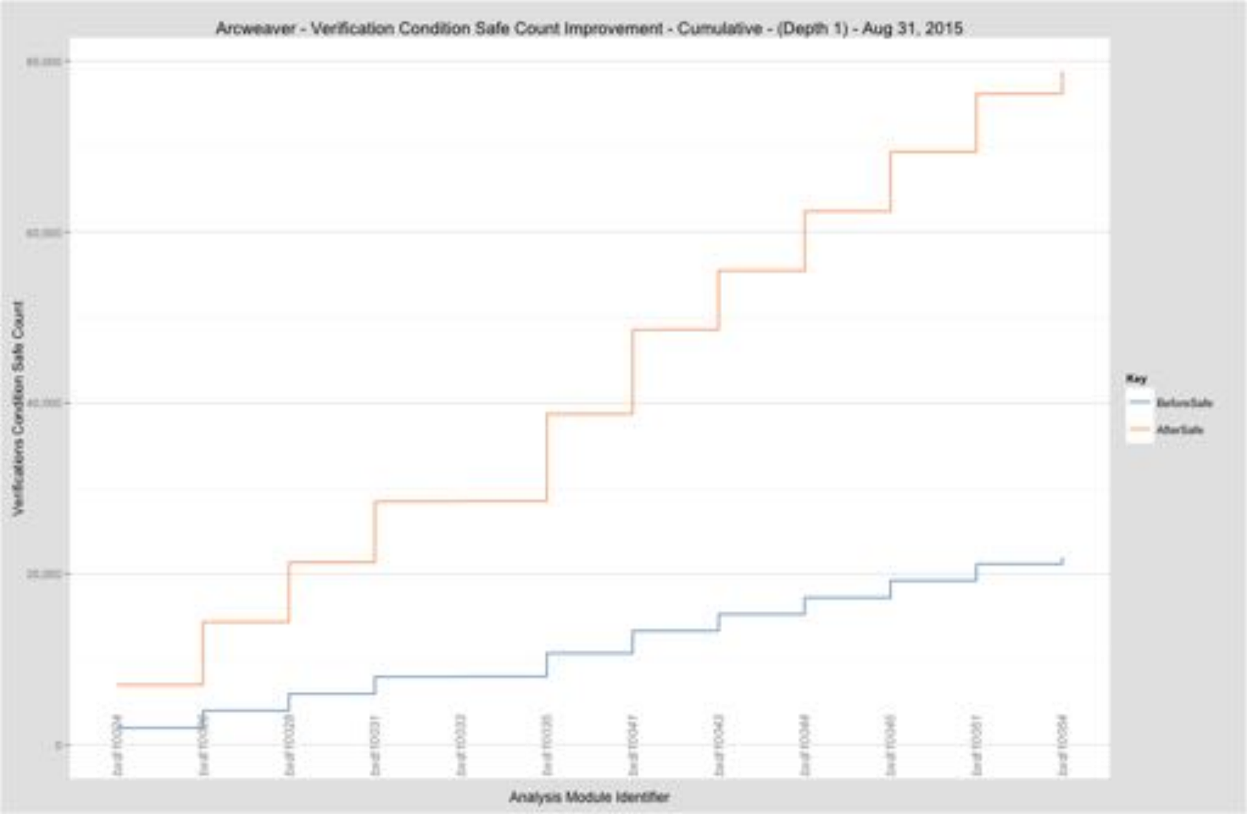


Figure 120: Arcweaver auto-solver results presented as before-and-after safe verification condition count cumulative improvements by module with one level of inlining. The module arrangement is by identifier. The blue line represents the cumulative count of safe verification conditions before game play. The orange line represents the cumulative count of safe verification conditions after game play. The overall result is roughly 11% improvement in total verification closure. Data and analysis through 31 August 2015.

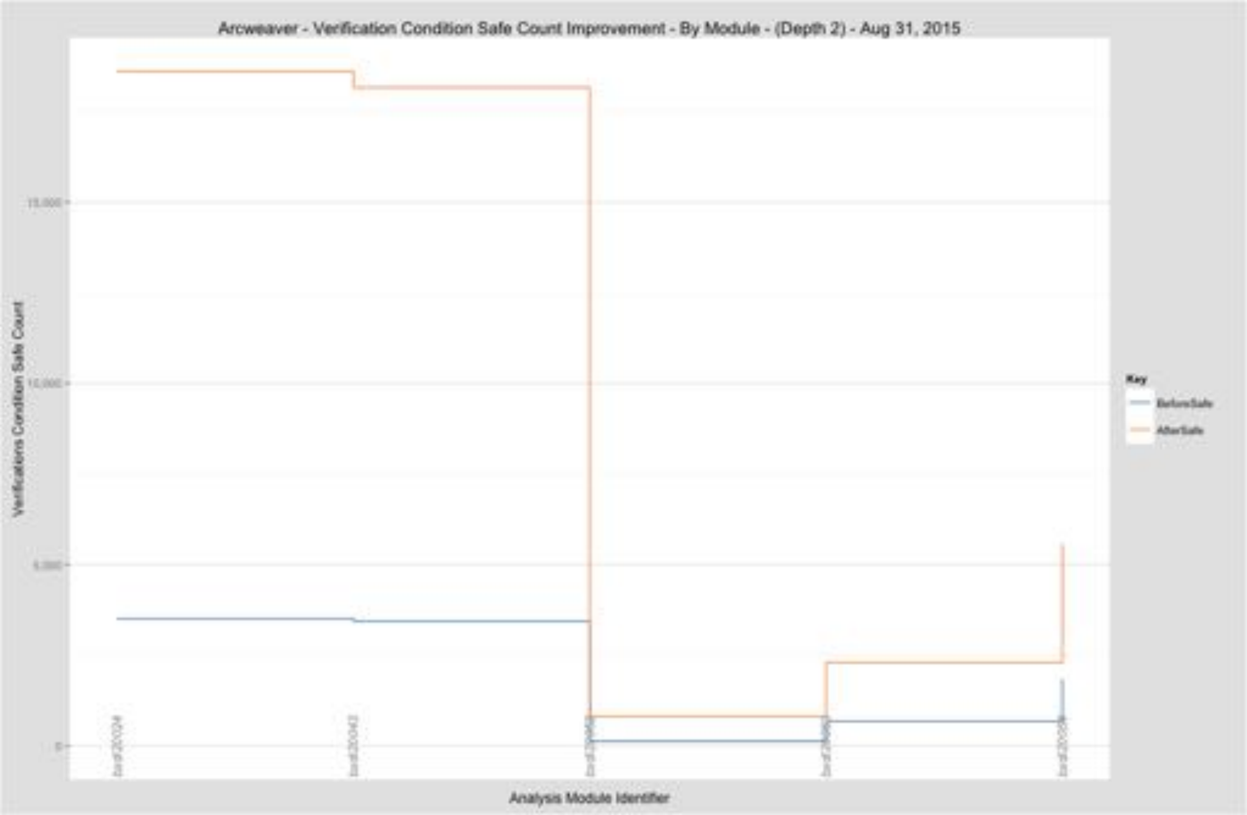


Figure 121: Arcweaver auto-solver results presented as before-and-after safe verification condition count improvements by module with two levels of inlining. The module arrangement is by identifier. The blue line represents the count of safe verification conditions before game play. The orange line represents the count of safe verification conditions after game play. Data and analysis through 31 August 2015.

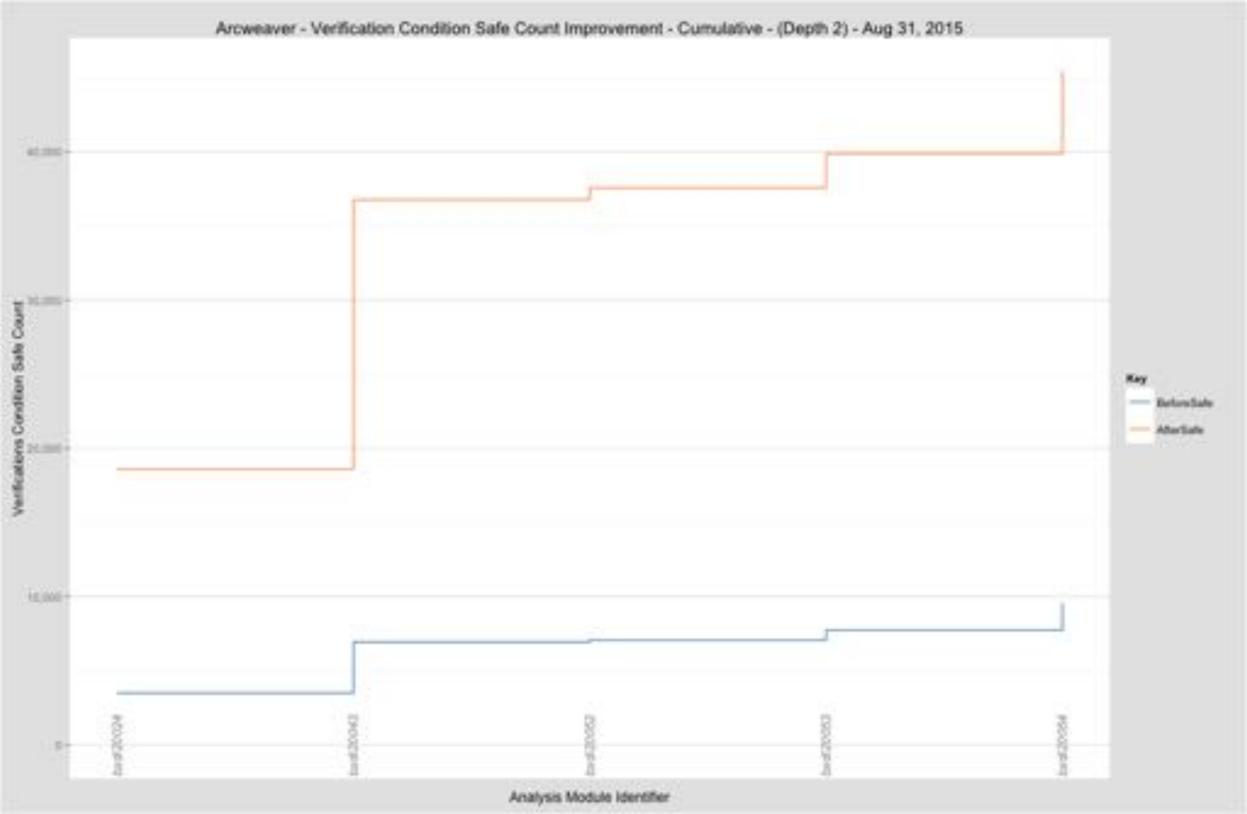


Figure 122: Arcweaver auto-solver results presented as before-and-after safe verification condition count cumulative improvements by module with two levels of inlining. The module arrangement is by identifier. The blue line represents the cumulative count of safe verification conditions before game play. The orange line represents the cumulative count of safe verification conditions after game play. The overall result is roughly 11% improvement in total verification closure. Data and analysis through 31 August 2015.

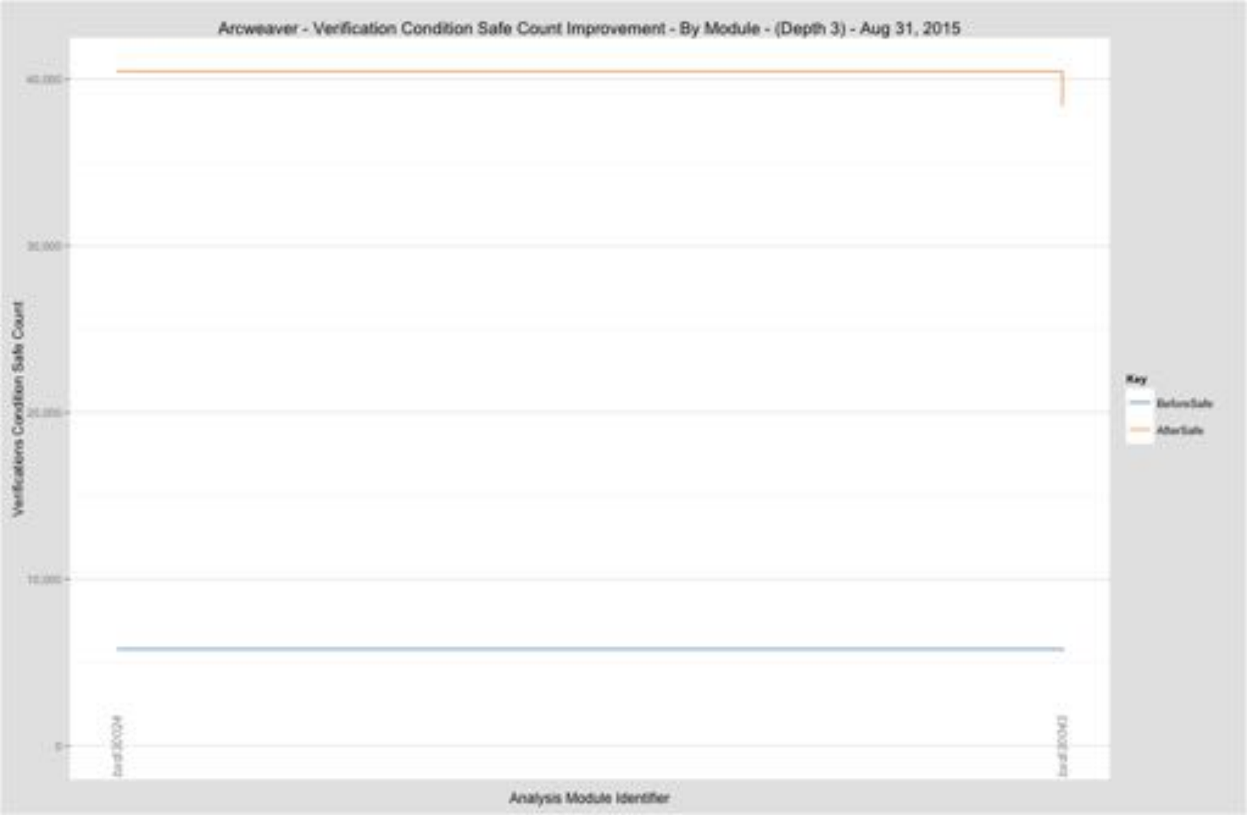


Figure 123: Arcweaver auto-solver results presented as before-and-after safe verification condition count improvements by module with three levels of inlining. The module arrangement is by identifier. The blue line represents the count of safe verification conditions before game play. The orange line represents the count of safe verification conditions after game play. Data and analysis through 31 August 2015.

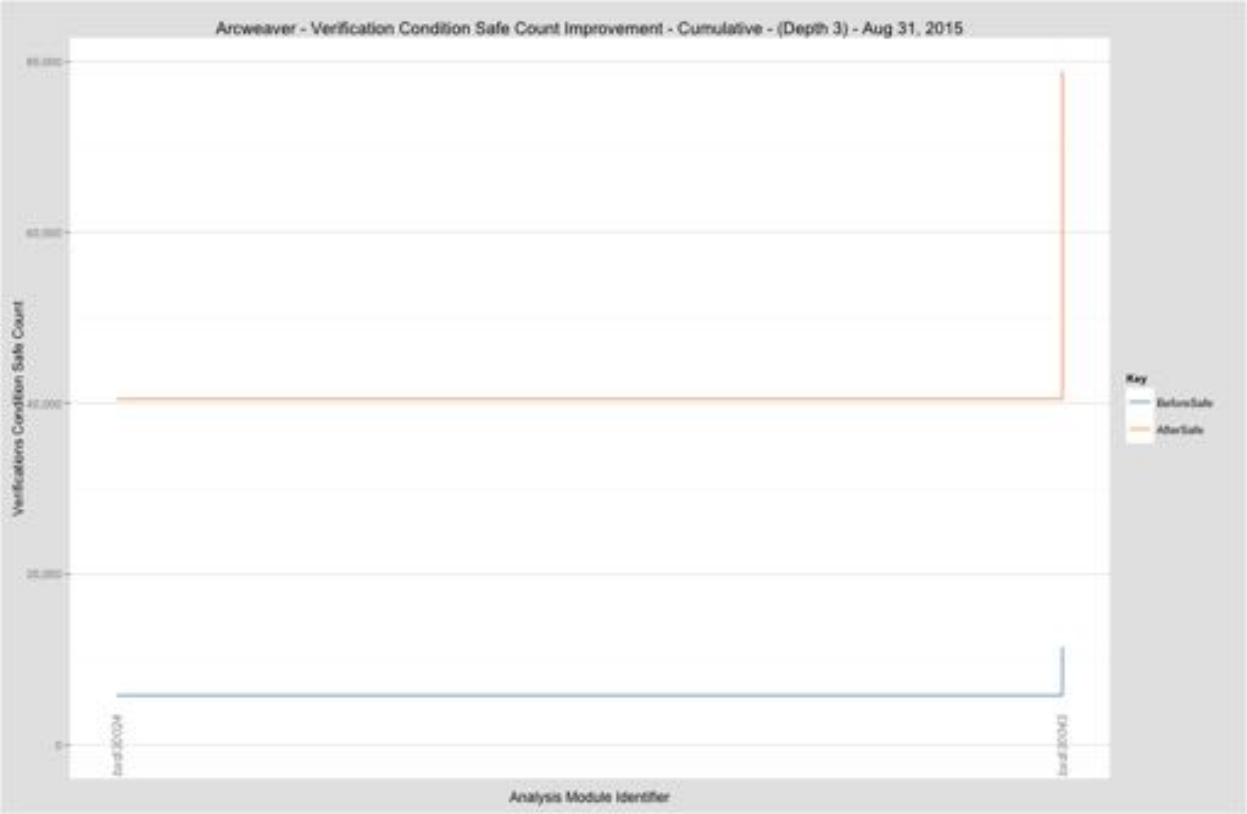


Figure 124: Arcweaver auto-solver results presented as before-and-after safe verification condition count cumulative improvements by module with three levels of inlining. The module arrangement is by identifier. The blue line represents the cumulative count of safe verification conditions before game play. The orange line represents the cumulative count of safe verification conditions after game play. The overall result is roughly 11% improvement in total verification closure. Data and analysis through 31 August 2015.

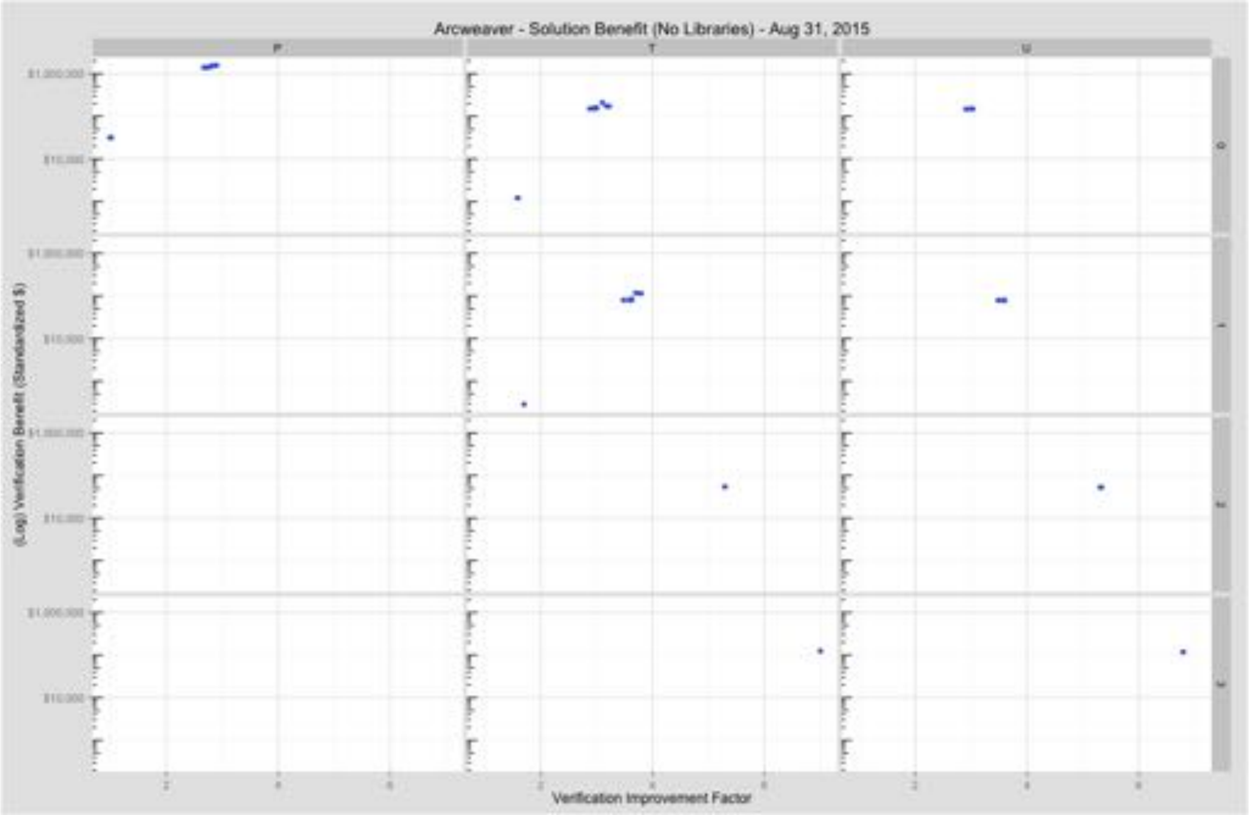


Figure 125: Arcweaver auto-solver result samples for verification improvement factor by the log of verification benefit dollars. The verification improvement factor is the ratio of the safe verification condition counts after and before applying the game techniques. The benefit is estimated by our verification condition to SLOC scale model described in the text. The grid panel dimensions are the constraint generator depth of inlining (0-3, deeper is better for verification) and module type according to the BIND manual (*P* for program, *T* for test, *U* for utility; type *L* library modules are not included as unreasonable for standalone analysis). Data and analysis through 31 August 2015.

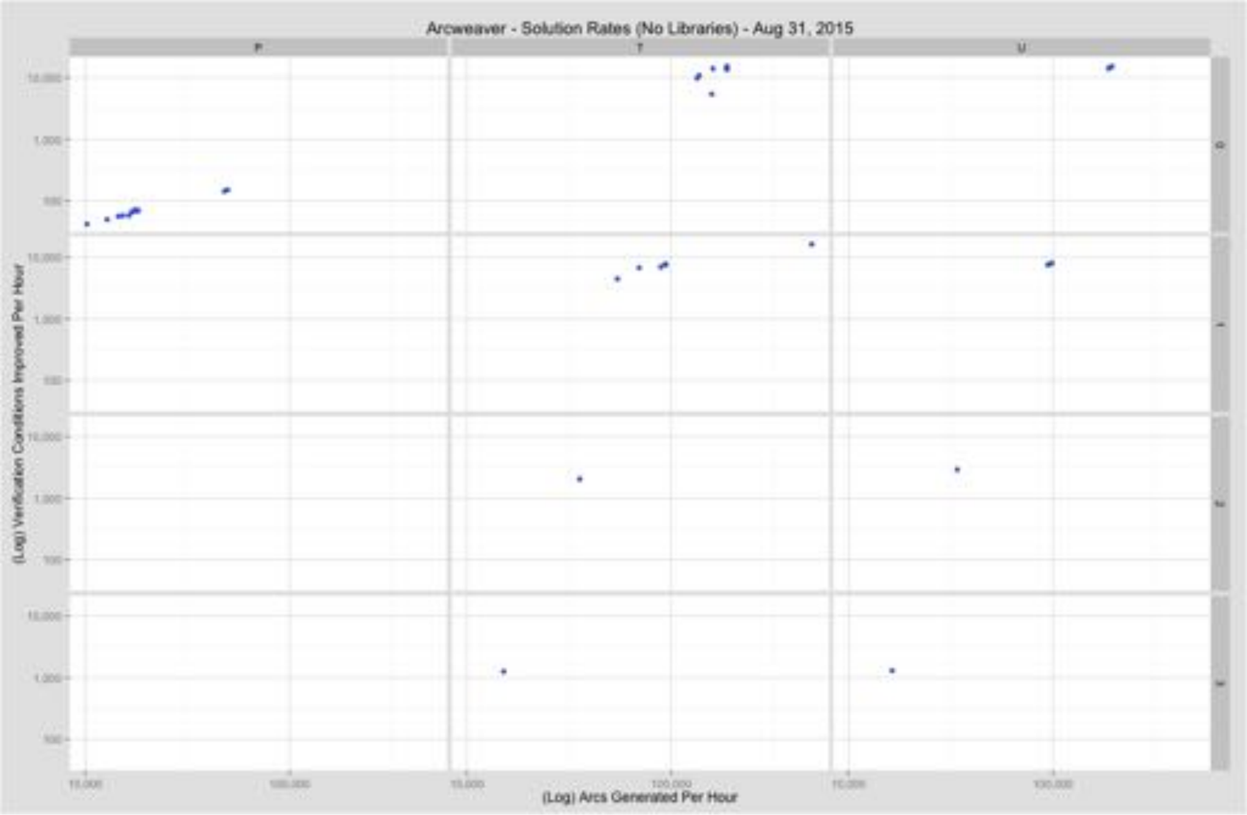


Figure 126: Arcweaver auto-solver result samples for log of arcs generated per hour by the log of verification conditions improved per hour. The grid panel dimensions are the constraint generator depth of inlining (0-3, deeper is better for verification) and module type according to the BIND manual (*P* for program, *T* for test, *U* for utility; type *L* library modules are not included as unreasonable for standalone analysis). Data and analysis through 31 August 2015.

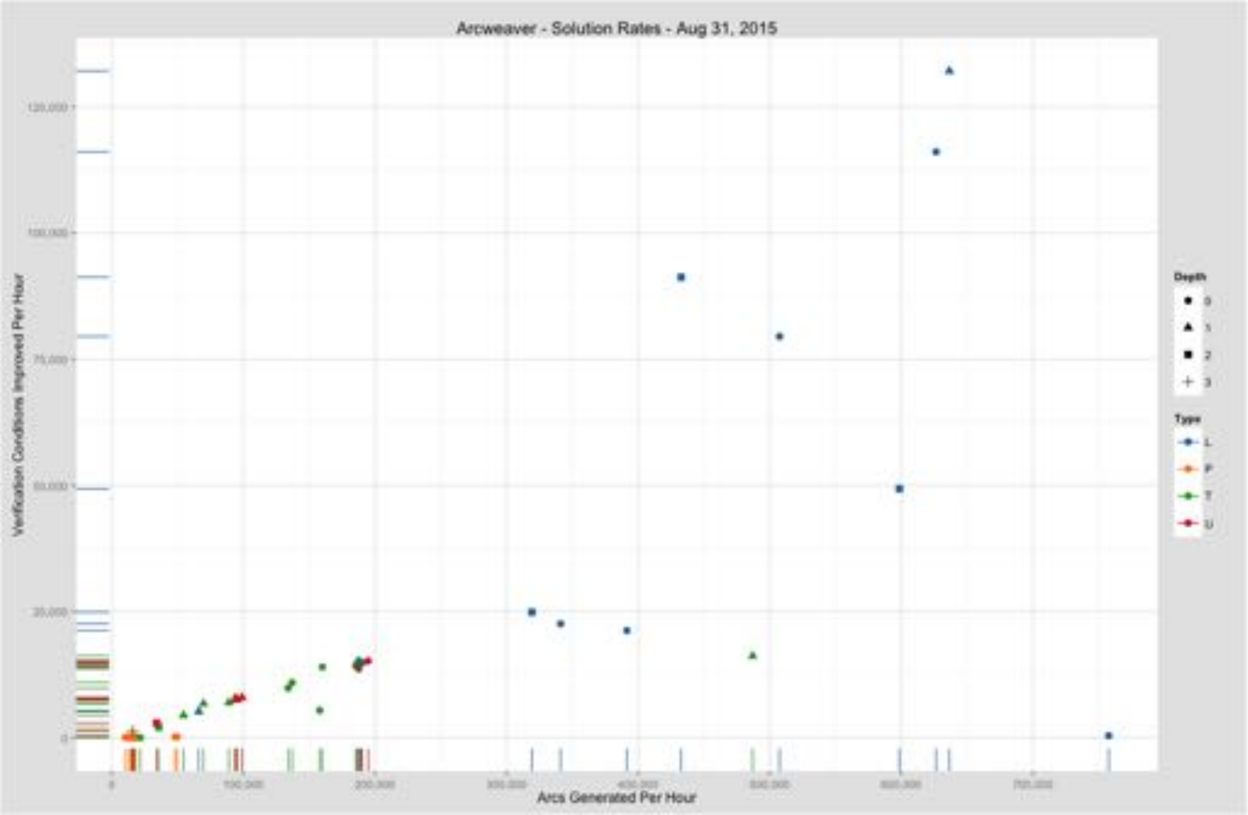


Figure 127: Arcweaver auto-solver solution rates represented as points-to graph arcs generated per hour against verification conditions improved per hour. Samples are distinguished by depth of inlining and by program type *U* for utility, *T* for test, *P* for program, and *L* for library according to the BIND manual. Data and analysis through 31 August 2015.

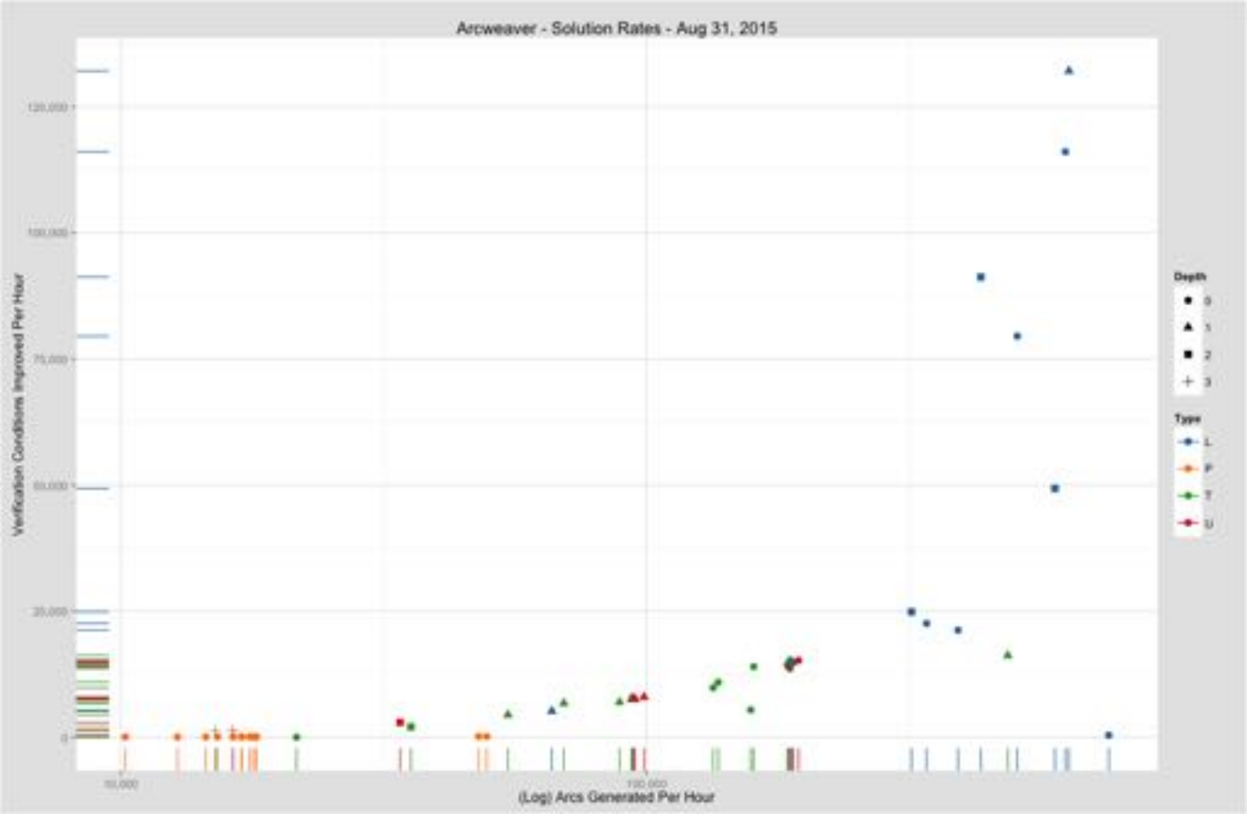


Figure 128: Arcweaver auto-solver solution rates represented as the log of points-to graph arcs generated per hour against verification conditions improved per hour. Samples are distinguished by depth of inlining and by program type *U* for utility, *T* for test, *P* for program, and *L* for library according to the BIND manual. The log scale does a better job than the linear scale of Figure 127 on the preceding page in separating the samples along the arc generation rate axis. Data and analysis through 31 August 2015.

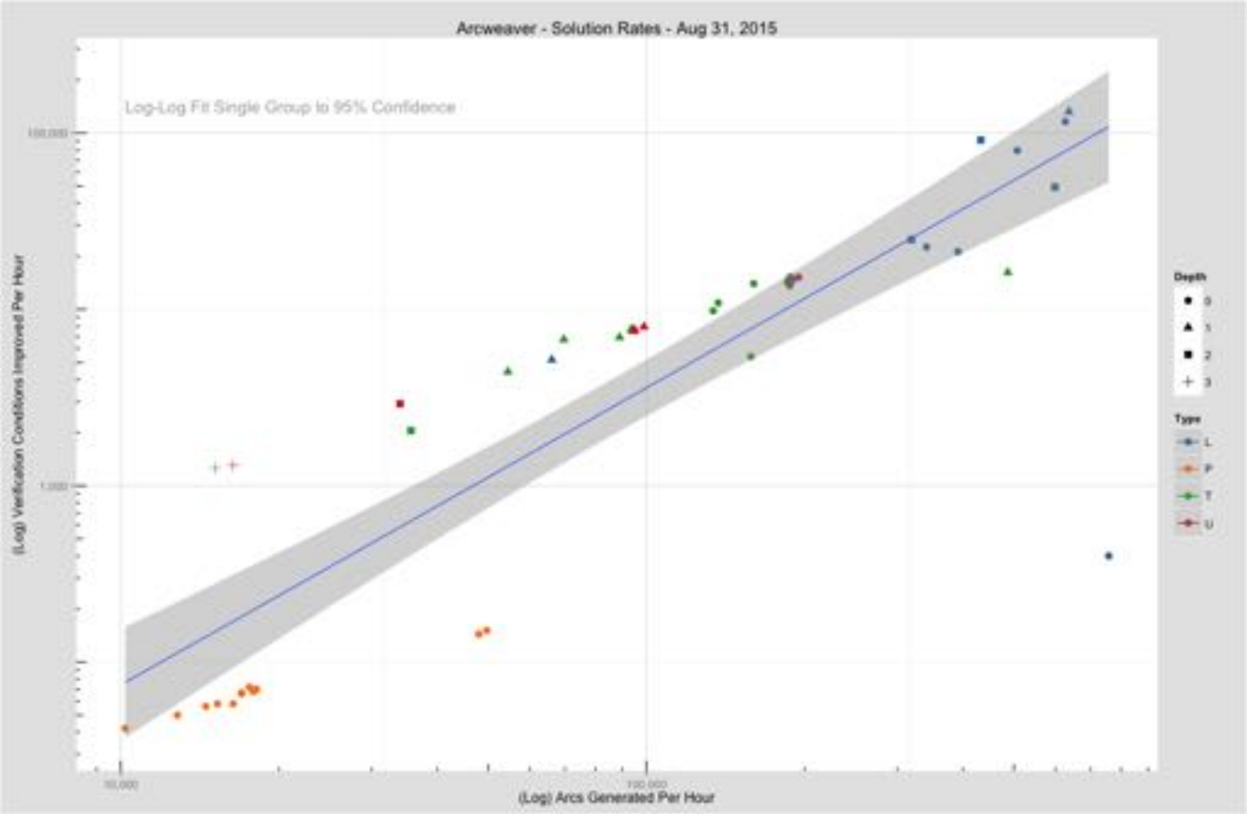


Figure 129: Arcweaver auto-solver solution rates represented as the log of points-to graph arcs generated per hour against the log of verification conditions improved per hour. Samples are distinguished by depth of inlining and by program type U for utility, T for test, P for program, and L for library according to the BIND manual. The double log scales do a better job than the single-log scale of Figure 128 on the previous page of separating the samples. The blue line fit shown is a linear model with standard error to 95% confidence encompassing *all program types as one group*. Owing to apparent clustering by types other model fits may be better; a fit by type is shown in Figure 132 on page 247. Data and analysis through 31 August 2015.

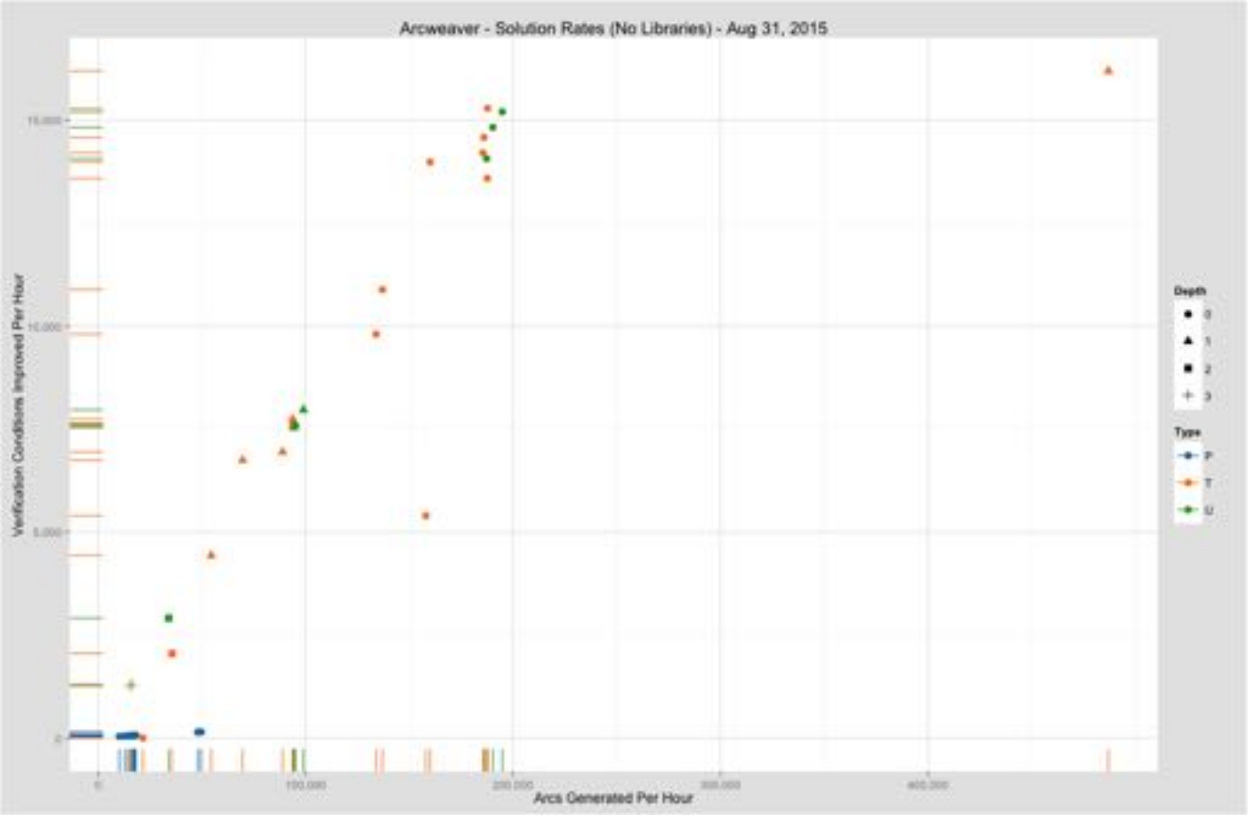


Figure 130: Arcweaver auto-solver solution rates represented as points-to graph arcs generated per hour against verification conditions improved per hour. Samples are distinguished by depth of inlining and by program type U for utility, T for test, and P for program according to the BIND manual. Library types are not included. Data and analysis through 31 August 2015.

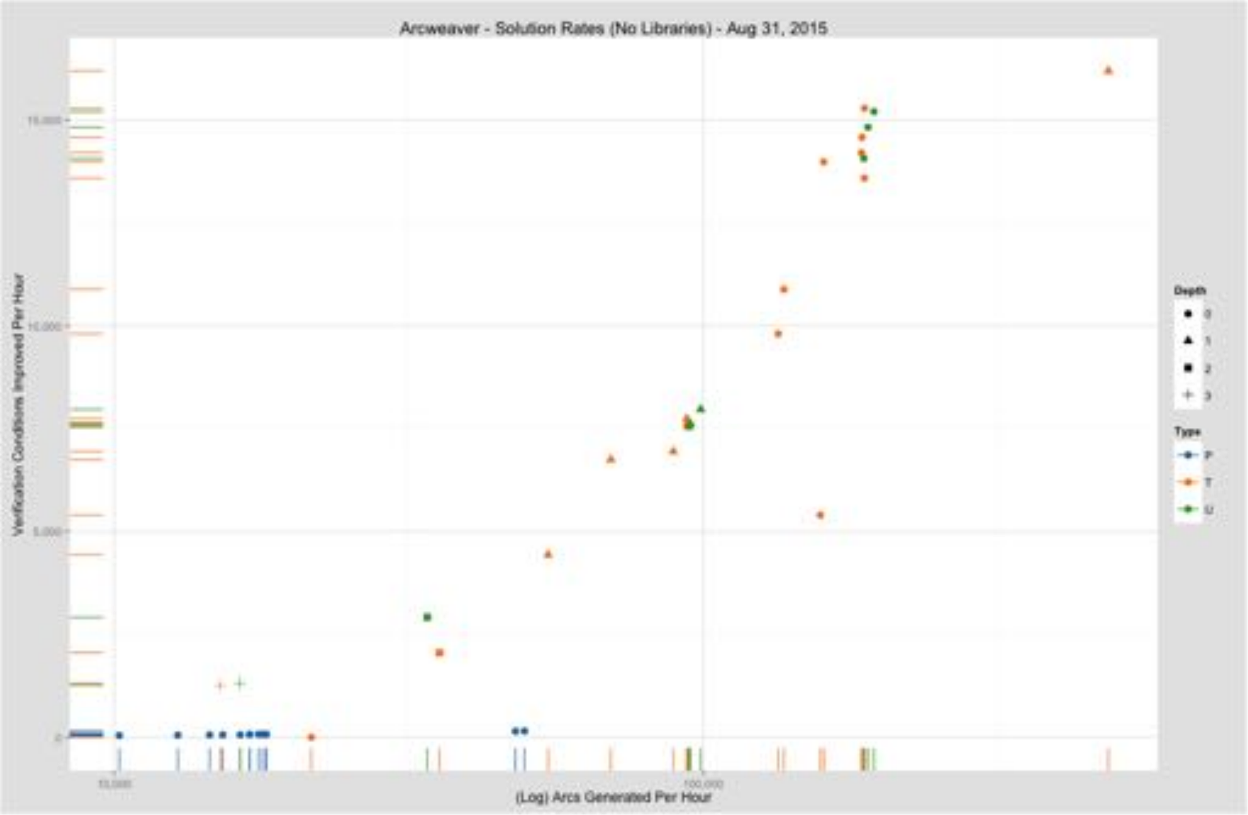


Figure 131: Arcweaver auto-solver solution rates represented as the log of points-to graph arcs generated per hour against verification conditions improved per hour. Samples are distinguished by depth of inlining and by program type U for utility, T for test, and P for program according to the BIND manual. Library types are not included. The log scale does a better job of separating samples than the linear scale of Figure 130 on the preceding page. Data and analysis through 31 August 2015.

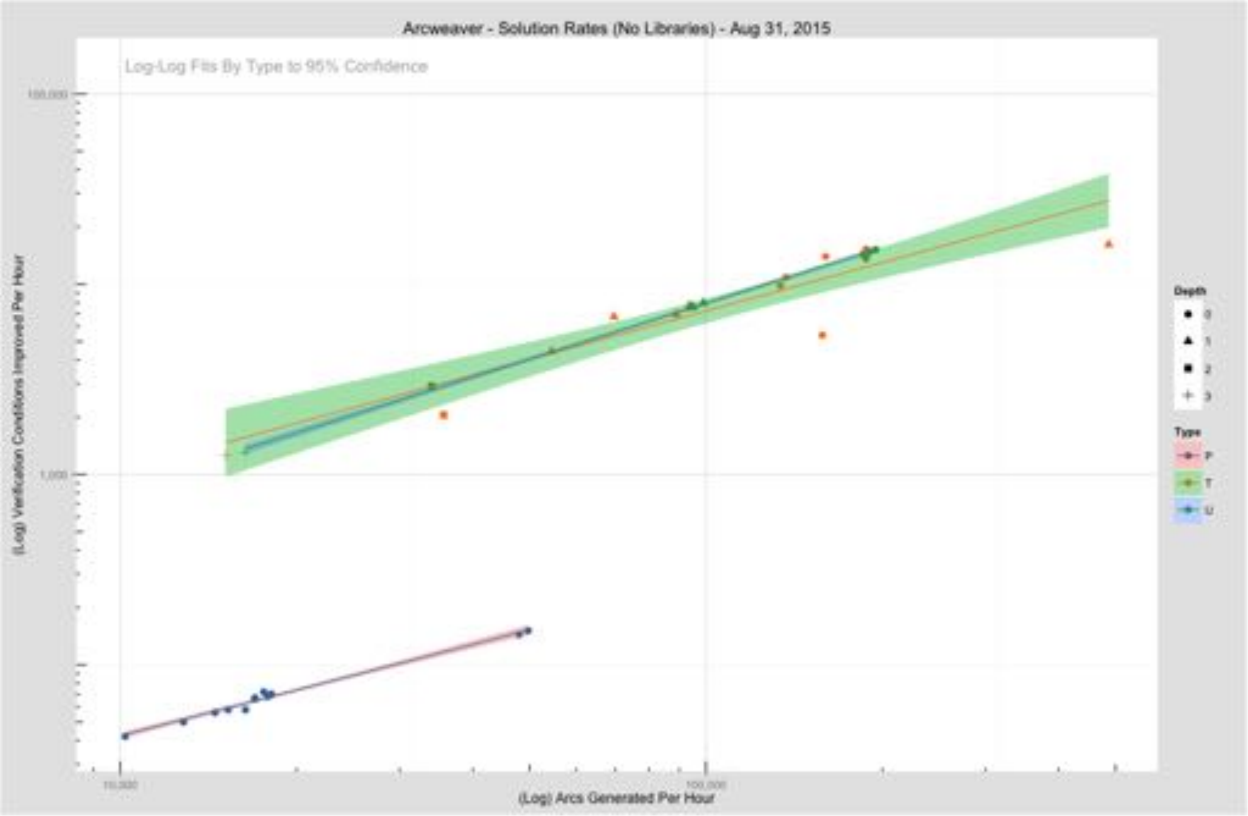


Figure 132: Arcweaver auto-solver solution rates represented as the log of points-to graph arcs generated per hour against the log of verification conditions improved per hour. Samples are distinguished by depth of inlining and by program type U for utility, T for test, and P for program according to the BIND manual. Library types are not included. The double log scale does an even better job of separating samples than the log-linear scale of Figure 131 on the preceding page. The three color-coded fits shown are linear models with standard error to 95% by program type. Data and analysis through 31 August 2015.

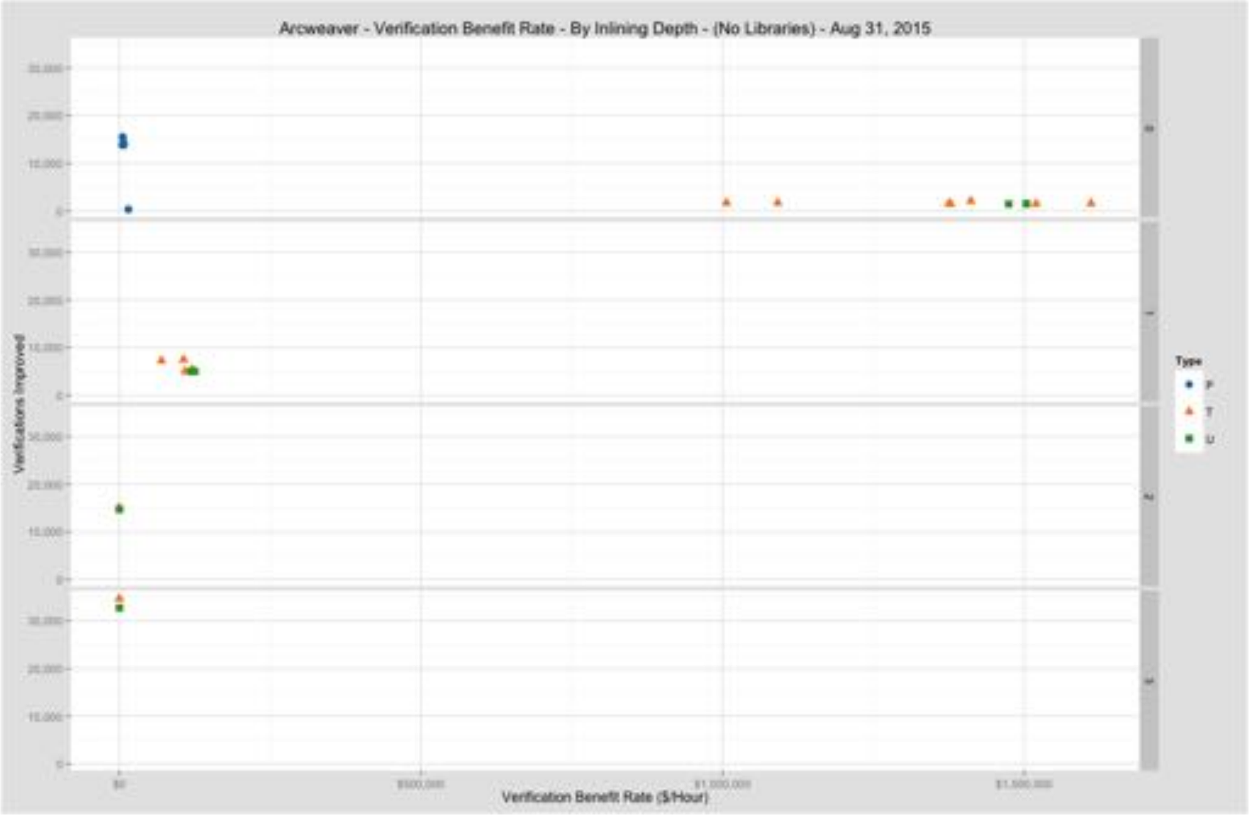


Figure 133: Arcweaver auto-solver verification benefit rates represented as verification benefit dollars per hour versus verification conditions improved, by inlining depth panel. Samples are distinguished by program type U for utility, T for test, and P for program according to the BIND manual. Library module types are not included. The benefit dollars per hour are computed by our verification condition to SLOC scale model described in the text. Data and analysis through 31 August 2015.

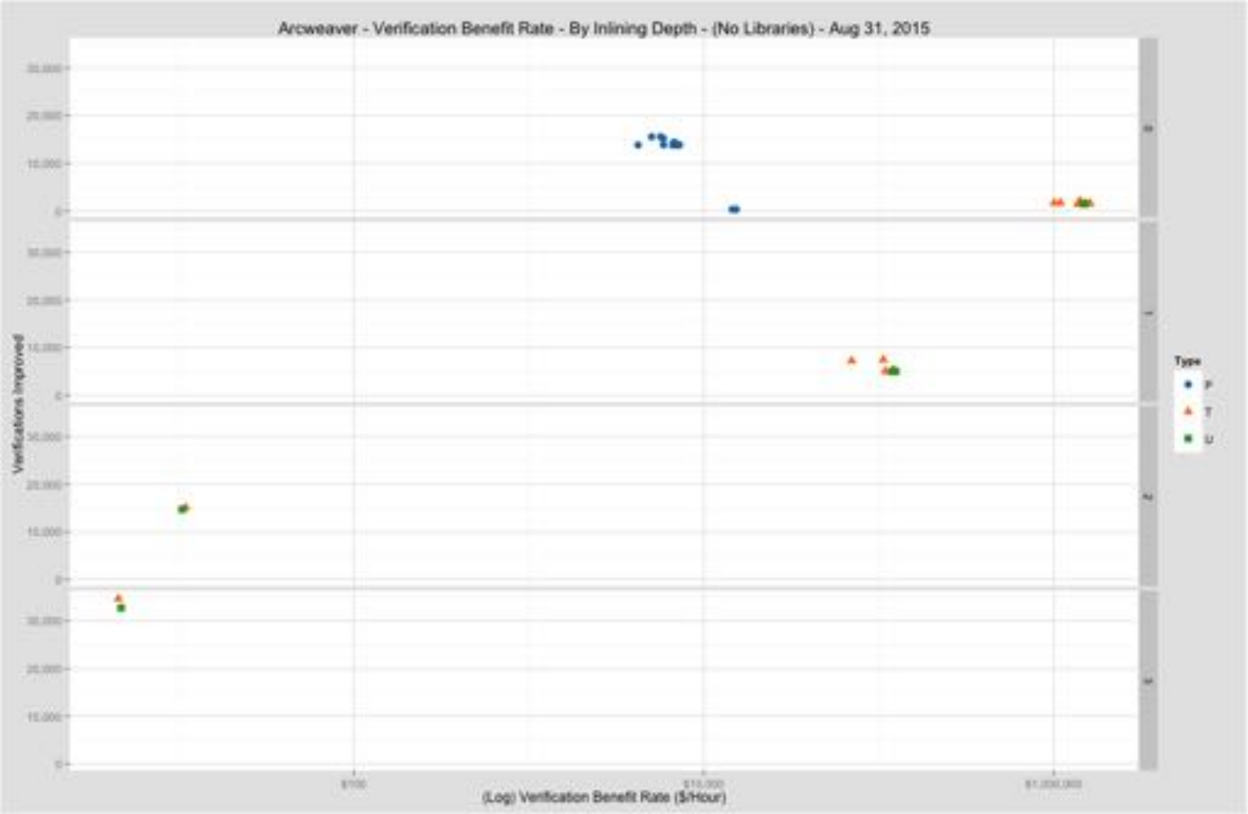


Figure 134: Arcweaver auto-solver verification benefit rates represented as the log of verification benefit dollars per hour versus verification conditions improved, by inlining depth panel. Samples are distinguished by program type U for utility, T for test, and P for program according to the BIND manual. Library module types are not included. The benefit dollars per hour are computed by our verification condition to SLOC scale model described in the text. Data and analysis through 31 August 2015.

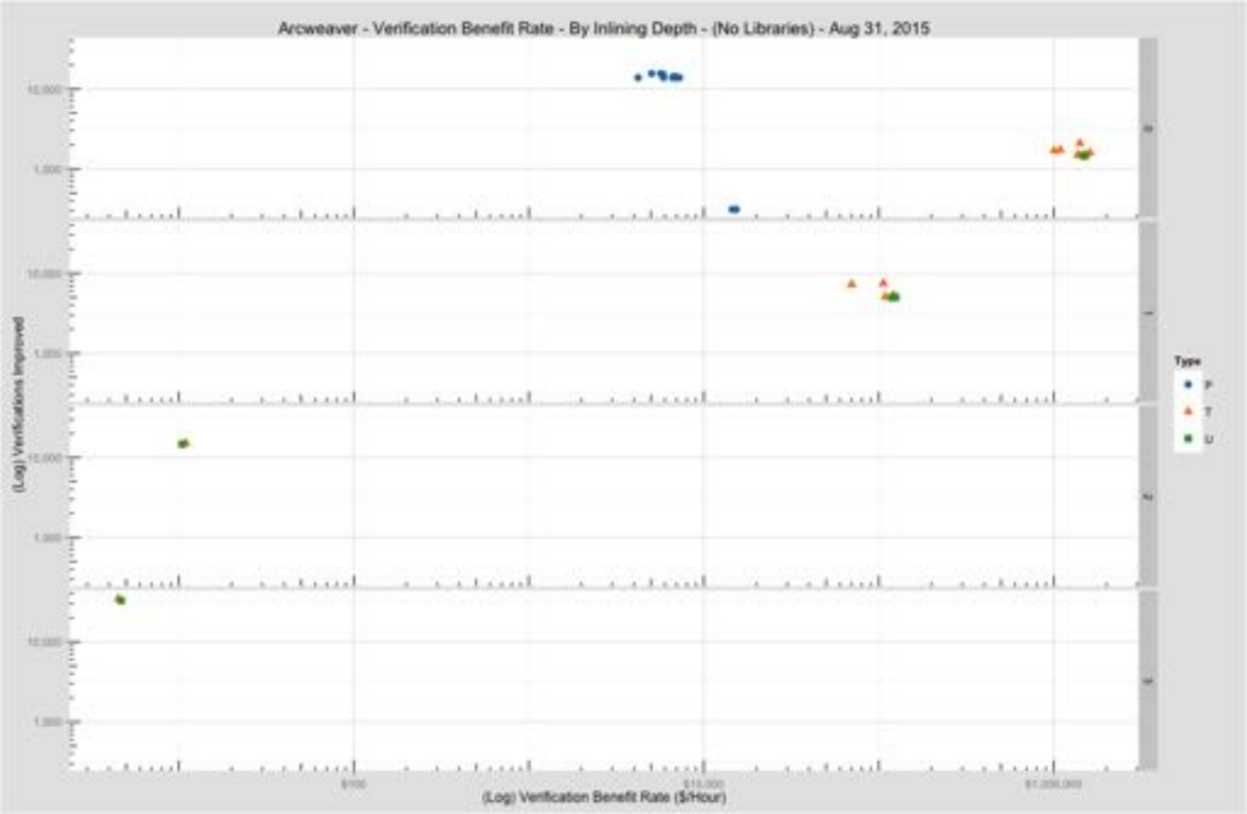


Figure 135: Arcweaver auto-solver verification benefit rates represented as the log of verification benefit dollars per hour versus the log of verification conditions improved, by inlining depth panel. Samples are distinguished by program type *U* for utility, *T* for test, and *P* for program according to the BIND manual. Library module types are not included. The benefit dollars per hour are computed by our verification condition to SLOC scale model described in the text. Data and analysis through 31 August 2015.

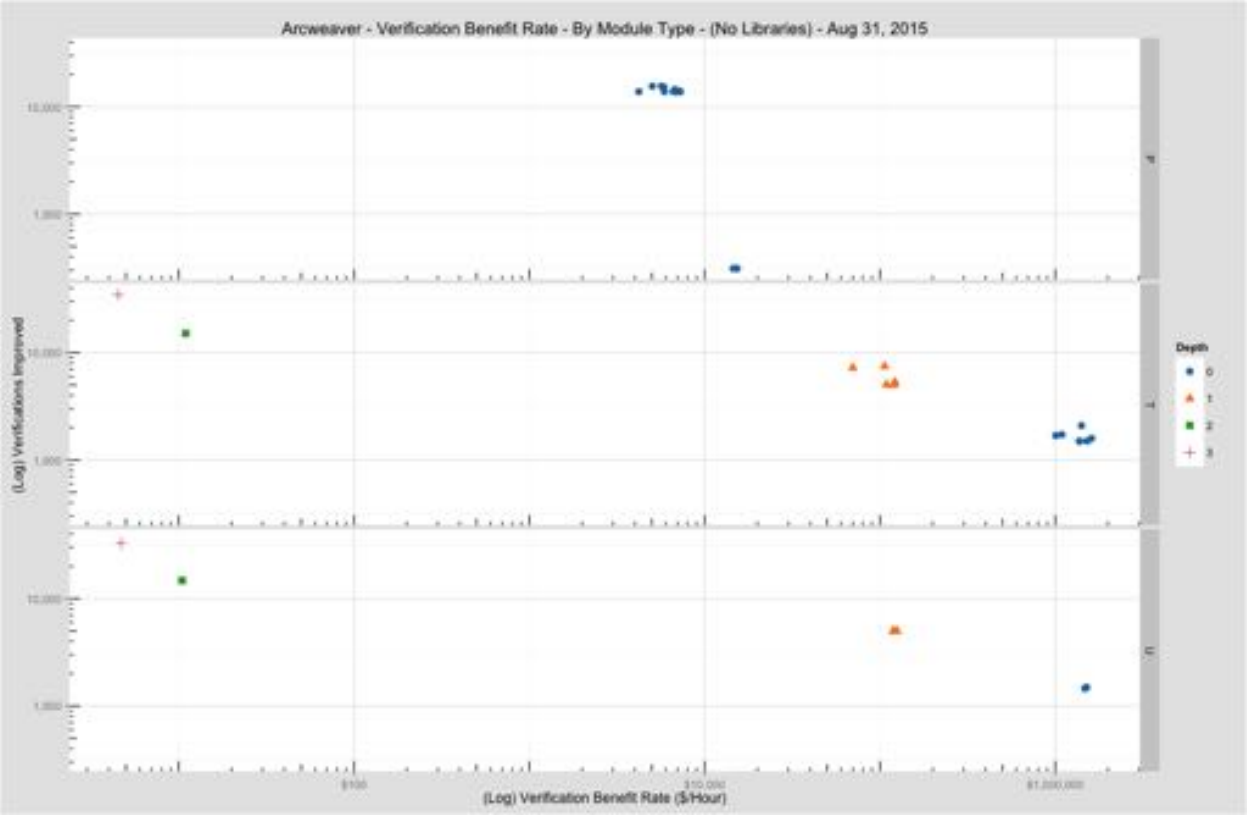


Figure 136: Arcweaver auto-solver verification benefit rates represented as the log of verification benefit dollars per hour versus the log of verification conditions improved, by program type panel. Samples are distinguished by constraint generator depth of inlining 0-3. Program types are U for utility, T for test, and P for program according to the BIND manual. Library module types are not included. The benefit dollars per hour are computed by our verification condition to SLOC scale model described in the text. Data and analysis through 31 August 2015.

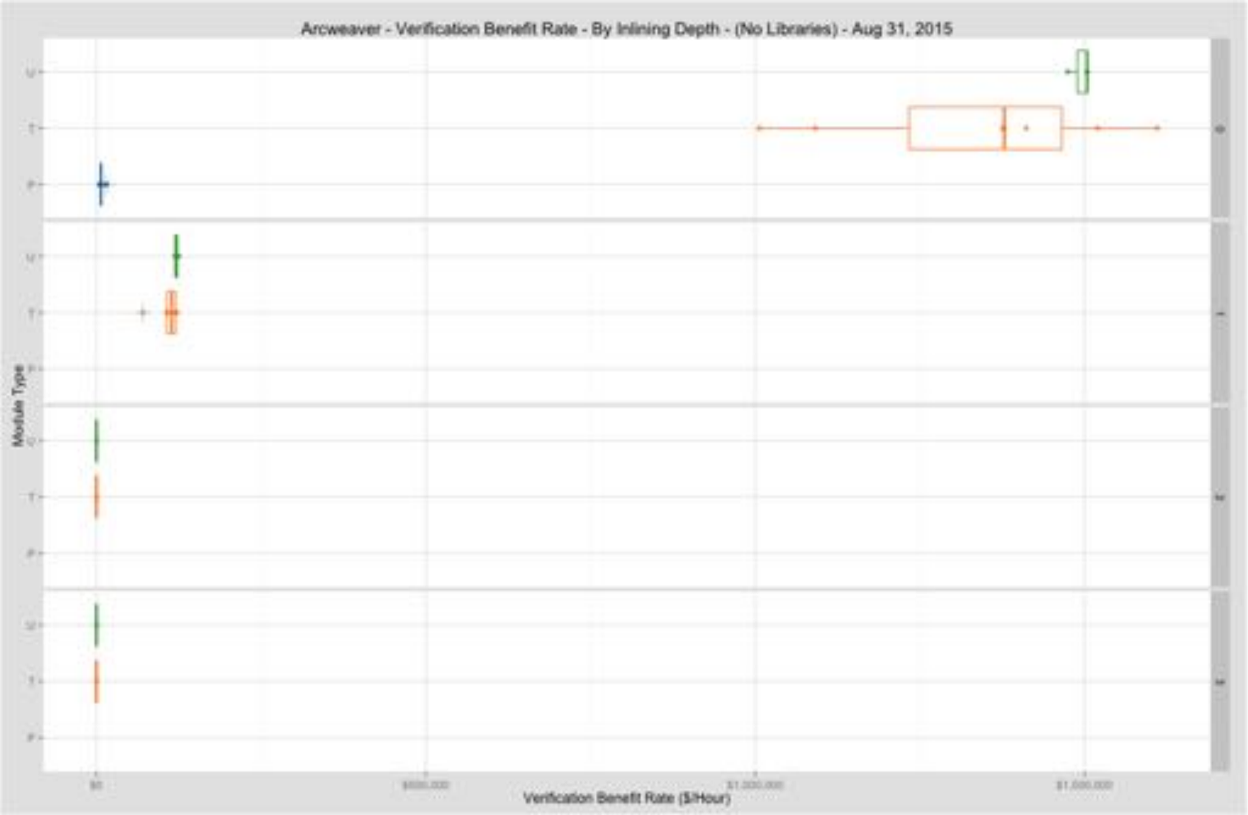


Figure 137: Arcweaver verification benefit rate sample statistics in box-and-whisker plot representation, verification benefit per hour by program module type and constraint generator depth of inlining (0-3). Program types are *U* for utility, *T* for test, and *P* for program according to the BIND manual. Library module types are not included. The benefit dollars per hour are computed by our verification condition to SLOC scale model described in the text. The box elements represent the median and the first and third quartiles of the samples. The whiskers extend to 1.5 times the inter-quartile range. Outliers are shown as points beyond the whiskers. Data and analysis through 31 August 2015.

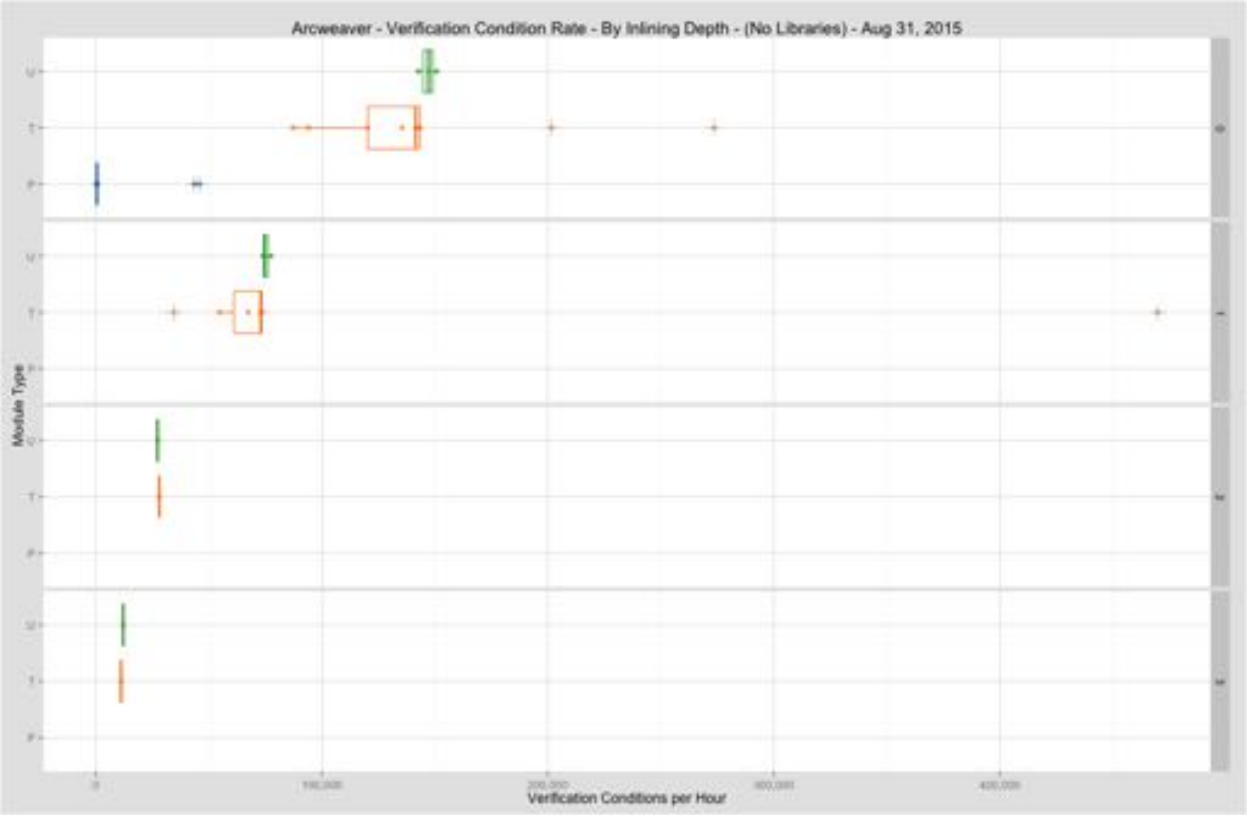


Figure 138: Arcweaver verification condition production rate sample statistics in box-and-whisker plot representation, verification condition production per hour by program module type and constraint generator depth of inlining (0-3). Program types are *U* for utility, *T* for test, and *P* for program according to the BIND manual. Library module types are not included. The box elements represent the median and the first and third quartiles of the samples. The whiskers extend to 1.5 times the inter-quartile range. Outliers are shown as points beyond the whiskers. Data and analysis through 31 August 2015.

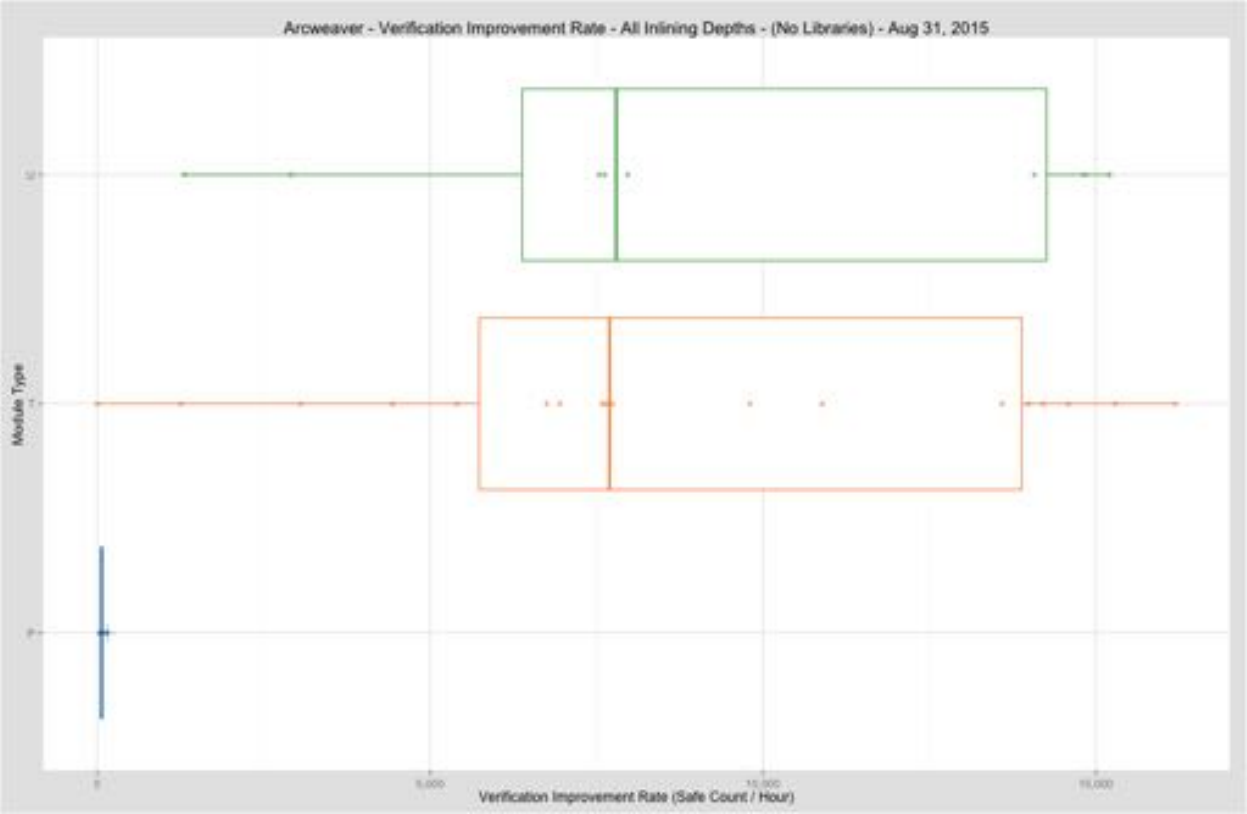


Figure 139: Arcweaver verification condition improvement rate sample statistics (safe count improved per hour) in box-and-whisker plot representation, verification improvement rate by program module type for all combined levels of constraint generator inlining. Program types are *U* for utility, *T* for test, and *P* for program according to the BIND manual. Library module types are not included. The box elements represent the median and the first and third quartiles of the samples. The whiskers extend to 1.5 times the inter-quartile range. Outliers are shown as points beyond the whiskers. Data and analysis through 31 August 2015.

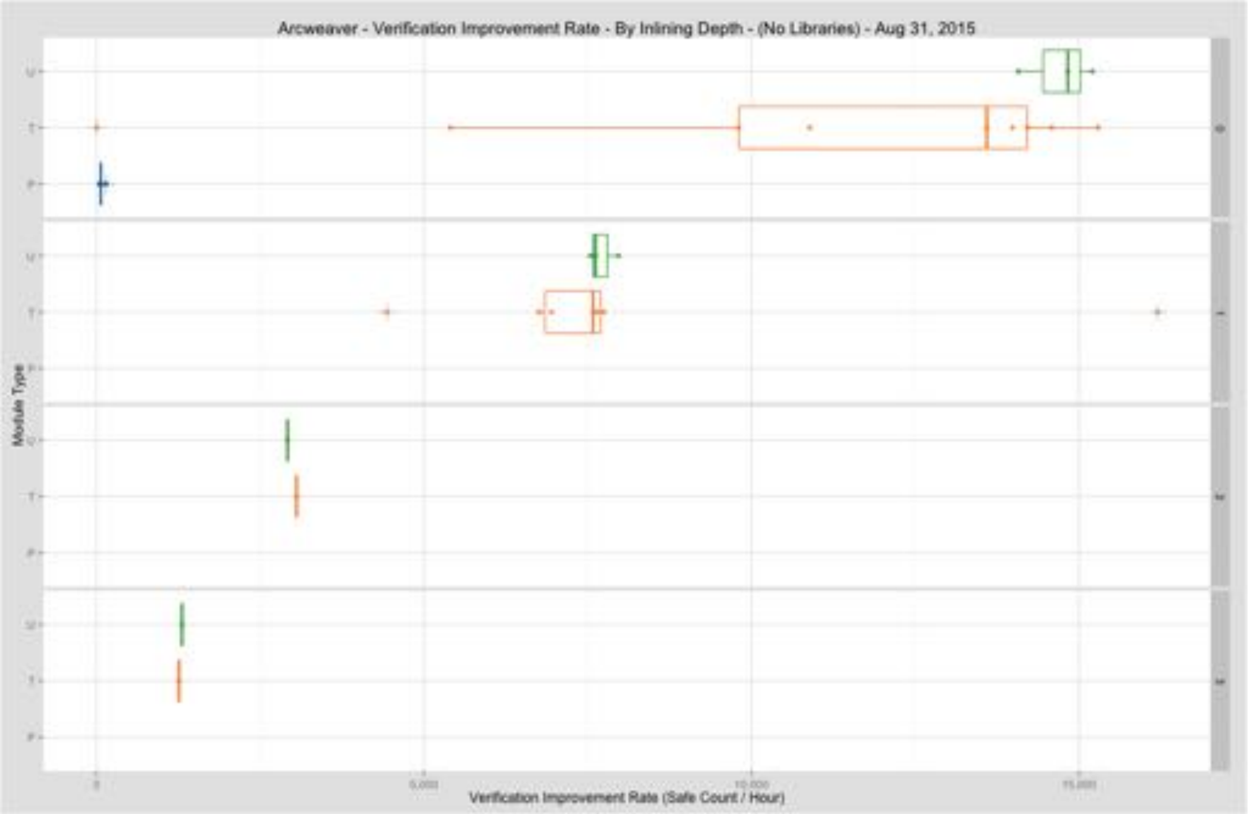


Figure 140: Arcweaver verification condition improvement rate sample statistics (safe count improved per hour) in box-and-whisker plot representation, verification improvement rate by program module type by constraint generator inlining depth panel (depth 0-3). Program types are U for utility, T for test, and P for program according to the BIND manual. Library module types are not included. The box elements represent the median and the first and third quartiles of the samples. The whiskers extend to 1.5 times the inter-quartile range. Outliers are shown as points beyond the whiskers. Owing to the significantly larger size of the constraint problem, the auto-solver was not able to produce P-type results for higher levels of inlining. Data and analysis through 31 August 2015.

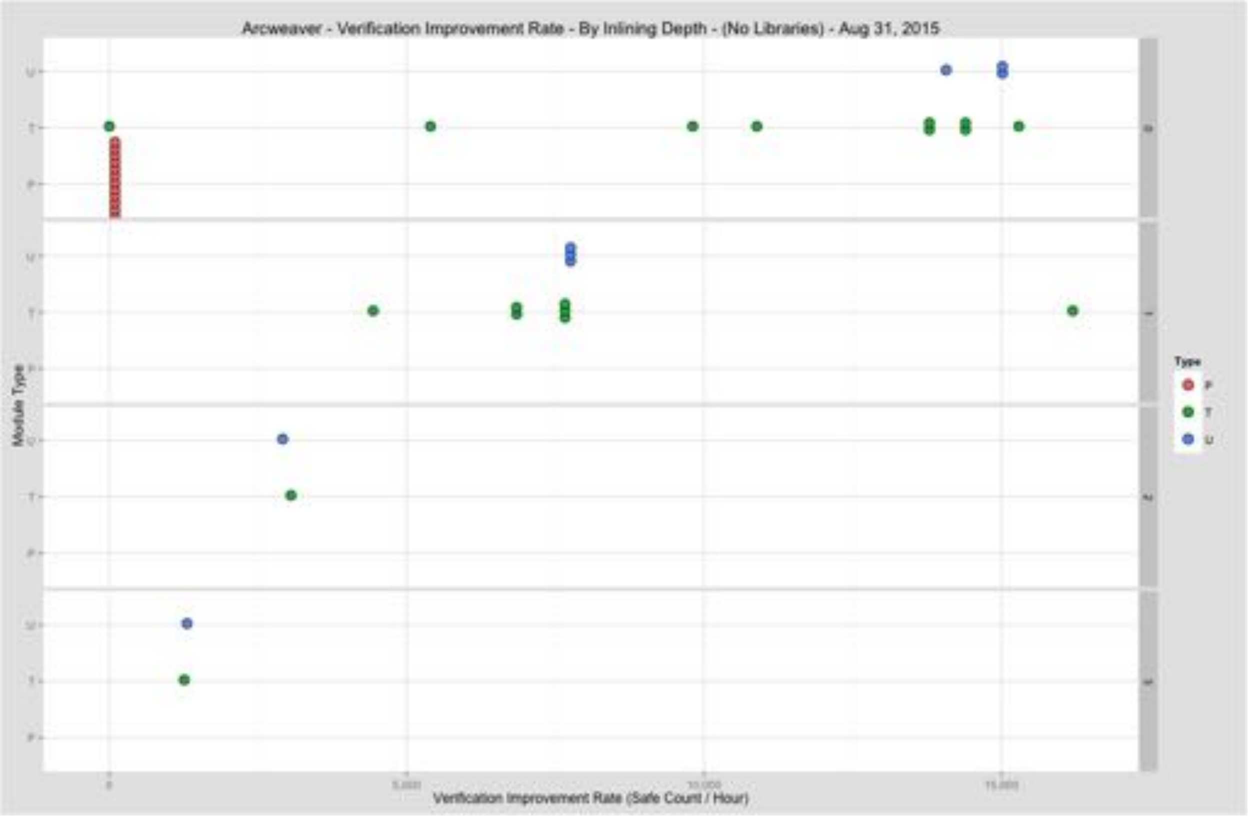


Figure 141: Arcweaver verification condition improvement rate samples (safe count improved per hour) in stacked dot plot representation, verification improvement rate by program module type by constraint generator inlining depth panel (depth 0-3). Program types are U for utility, T for test, and P for program according to the BIND manual. Library module types are not included. Owing to the significantly larger size of the constraint problem, the auto-solver was not able to produce P-type results for higher levels of inlining. Data and analysis through 31 August 2015.

4.7 Solution Data

This section presents the raw solution data for the game play and auto-solver results. It includes the verification results before and after applying the game model and solution iteration. We present several tables each collecting a different category of data. In the tables the major column headers are defined as follows:

Case A study case name, usually a code name for a test case or a solution case for the BIND program under analysis. The coding for the BIND modules usually followed a naming template for example `bxdi10024` meant *BIND executable depth of inlining 1 module 24*. The numbering of the various build modules of BIND referred to the sequence in which the `make` programs would build the load modules.

Module The case module type and depth, referring to the module type and inlining depth of the target program.

Before Game Play CodeHawk verification results before game play results have been applied.

Game Non-Fixed Point Solution CodeHawk results after game play results have been applied, but with the understanding these results have not reached a fixpoint solution and are not completely trustworthy.

Arcweaver Fixed Point Solution CodeHawk results after auto-solver results have been applied.

Backend Statistics for the backend performance of the auto-solver including run time and estimated cost and economic model benefit.

EC2 `c3.large` The Amazon Web Services EC2 virtual machine size used for the auto-solver host and its performance statistics; usually `c3.large` size for the single-threaded solvers.

Rate Production rate and improvement rate measures for the solutions.

Below the major column headers are minor column headers defined as follows:

Type The module type identifier. The types were `P` for program, `T` for test, `U` for utility, and `L` for library, as characterized by the BIND manual.

Depth The inlining depth used by CodeHawk during constraint generation.

Safe The number of CodeHawk verification conditions producing safe results.

Warning The number of CodeHawk verification conditions producing warning results.

Error The number of CodeHawk verification conditions producing error results.

Total The total verification conditions, sum of `Safe`, `Warning`, and `Error`. This total is the same before and after applying the game solution because the game does not change the obligations.

ID The game identifier. The identifiers were C for CircuitBot, D for Dynamakr, and V for VIPER.

Iter The number of auto-solver iterations required to reach a fixpoint solution.

Arcs The number of graph arcs generated for the solution.

Improved The number of verification conditions improved; that is, the increase in Safe verification count.

Factor The ratio of after Safe verification count to before Safe verification count.

Duration The run time for the auto-solver or backend to obtain the solution, in hours.

Cost The AWS infrastructure cost (including fractional hour round-up for CPU time) to obtain the solution, in dollars.

Benefit The economic value, in dollars, of the Improved verification conditions according to our SLOC/VC proportional economic model.

Arc/hr The solution production rate, arcs per hour, for the full solution.

Impr/hr The verification count improvement rate, improvements per hour, for the full solution.

Table 1 on page 260 provides the raw data for our production game play results. Because the games did not see sufficient player traffic to reach complete fixpoint solutions for these games, the results are useful but not as impressive as the auto-solver results. The data shown in the table are useful as *results in progress* because while they are sound results, they can nevertheless be improved with further game play.

Table 2 on page 261 provides the raw data for our Arcweaver auto-solver results for modules analyzed to inlining depth zero. We obtained more case solutions at depth zero because – although quite large – these cases were the smallest we could produce and solve with our single-threaded solvers and condensers. These are all thought to be fixpoint solutions, although we believe they point to a quasi-fixpoint solution because the reported found errors should be reported as warnings, and would be reported as warnings if the iteration continued a bit further. We were unable to pinpoint the reason for this apparent early termination.

Tables 3 on page 262, 4 on page 263, and 5 on page 264 provide the raw data for Arcweaver auto-solver results for modules analyzed to inlining depth one, two and three. Each level of depth results in fewer successful solutions owing to the vast increase in size of the analysis products and computational resource demand for solving the constraint problems. However, note that with each increase in inlining depth the verification *improvement factor* increases substantially owing to the increased precision of analysis. At inlining depth level zero the typical improvement factor owing to the pointer analysis is about 2.5 times. At level one the improvement factor is about 3.5 times; at level two the improvement factor is 5.3; at level three the improvement factor is 7.0. Typically we run CodeHawk the highest level of inlining possible – usually starting by default with around five or six levels. We were not able to run the pointer-flow analysis that deeply (yet) without exhausting memory. Nevertheless, the pointer-flow model and "game play" results contributed substantially to the improvement of the verification results, as shown by improvement factor, improvement rate, and economic benefit.

Regarding the initial safe verification counts, we offer an aside. Typically we run CodeHawk with many more static analysis domains enabled. Were we to run CodeHawk with more analysis domains enabled the *before game play* verification safe counts would be much higher relative to the total, say 70-80% of the total, much higher than the 10% or so of the total shown in these tables. CodeHawk is able to prove correct, automatically, soundly, and without annotations, a very high proportion of all memory safety properties in C source code. The reason our tables show a low proportion of the total is that the version of CodeHawk we used in these analyses was built with only the analysis domains of interest to the pointer analysis and none of the others enabled.

5.0 CONCLUSIONS

Section 5.1 summarizes the important contribution regarding our verification improvement for memory safety properties under sound static analysis. Section 5.2 on page 265 reflects on our experience attracting and retaining members of this crowd of players we want to contribute to solving verification problems. Section 5.3 on page 265 offers a conclusion that, despite all the data we collected with auto-solvers, there remains an important role for humans in this verification process.

5.1 Verification Improvement and Valuation

The C program pointer analysis problem we tackled was already known to be undecidable. When performing verification manually, human experts make progress by asserting summaries of the points-to graph without perhaps complete knowledge of every variable value and memory location. We tackled specifically the pointer analysis problem taking into account pointers with offsets, abstracting away the program control flow information. A points-to graph for the program supports static analysis by describing buffer arrangements and sizes. When performing memory safety property verification with our CodeHawk analyzer we can use the graph to support the production of invariants and discharge of proof obligations to make substantially more progress resolving buffer accesses than without the points-to graph. Our results show a mean of 11% improvement at even the lowest levels of precision. At higher levels of precision, where the constraint problem size grows exponentially (6 times larger for one level of inlining, 28 times larger for two levels, 100 times for three levels, and so on) and the corresponding verification contribution improves even more. At inlining depth level zero the typical improvement factor owing to the pointer analysis is about 2.5 times. At level one the improvement factor is about 3.5 times; at level two the improvement factor is 5.3; at level three the improvement factor is 7.0. Typically we run CodeHawk for memory safety properties at the highest level of inlining possible ? usually starting by default with around five or six levels. We were not able to run the pointer-flow analysis that deeply without exhausting memory. We remain quite eager to find ways to achieve this capability.

Case	**Module**		**Before Game Play**				**Game Non-Fixed Point Solution**							**Backend**			**Rate**	
Module	Type	Depth	Safe	Warning	Error	Total	ID	Arcs	Safe	Warning	Error	Improved	Factor	Duration	Cost	Benefit	Arc/hr	Impr/hr
bxdi10024	T	1	1,985	47,011	0	48,996	D	39,736	4,970	43,335	691	2,985	2.5	39.622	$12.600	$47,045	1,003	75
bxdi10035	T	1	2,726	57,746	0	60,472	D	10,473	3,594	56,740	138	868	1.3	39.622	$12.600	$13,680	264	22
bxdi10035	T	1	2,726	57,746	0	60,472	V	68,013	9,139	50,784	549	6,413	3.4	190.962	$211.400	$101,072	356	34
Total/Avg														270.206	$236,600	$161,796	541	44

Table 1: Production game play data approaching fixpoint solutions. The ID column identifies the source game, D for Dynamakr and V for VIPER. The module type column identifies the loaded game modules per the BIND manual, in this case the module types are T for test modules. The inlining depth of 1 refers to the constraint generator applying inlining during the generation of pointer flow constraints for the game instances, where deeper inlining increases verification resolution but also increases the volume of the game problem. The solution columns are the verification results after applying the *arcs* from game play through the CodeHawk analyzer together with the constraints, dictionary and anchors from the original constraint generation. The improved count is the number of safe verification conditions increased by game play, and the factor is the ratio of the after to before safe counts. The backend duration is the cumulative analysis time (play time) in hours, the cost is the estimated cloud infrastructure cost using Amazon's cloud service rates, and the benefit is the economic benefit of the *improved* count multiplied by the value of a verification condition per our economic model (using $15.75 per verification condition for the inlining depth of one). The cost for the VIPER games also includes the cumulative payouts of the human intelligence task (HIT) paid to the mechanical turk workers. The rates are backend arcs per hour and improvements per hour, Data and analysis current through 31 August 2015.

Case	Module		Before Game Play				Arcweaver Fixed Point Solution							EC2 c3.large			Rate	
Module	Type	Depth	Safe	Warning	Error	Total	Iter	Arcs	Safe	Warning	Error	Improved	Factor	Duration	Cost	Benefit	Arc/hr	Impr/hr
bxdiz0001	P	0	8,046	86,524	0	94,570	3	104,163	8,358	86,212	0	312	1.0	2.171	$0.315	$31,549	47,983	144
bxdiz0002	P	0	8,032	86,430	0	94,462	3	102,369	8,344	86,118	0	312	1.0	2.060	$0.315	$31,549	49,694	151
bxdiz0003	P	0	7,964	84,848	0	92,812	13	3,386,480	21,796	71,012	4	13,832	2.7	332.000	$34.860	$1,398,692	10,200	42
bxdiz0004	P	0	7,964	84,860	0	92,824	12	3,386,658	21,802	71,018	4	13,838	2.7	192.724	$20.265	$1,399,299	17,573	72
bxdiz0006	P	0	8,348	90,938	0	99,286	12	4,028,313	23,900	75,382	4	15,552	2.9	277.478	$29.190	$1,572,618	14,518	56
bxdiz0007	P	0	8,349	91,033	0	99,382	12	4,006,341	23,905	75,473	4	15,556	2.9	312.543	$32.865	$1,573,023	12,819	50
bxdiz0010	P	0	8,051	85,501	0	93,552	12	3,613,170	21,839	71,709	4	13,788	2.7	202.262	$21.315	$1,394,243	17,864	68
bxdiz0011	P	0	8,051	85,477	0	93,528	12	3,596,956	21,995	71,529	4	13,944	2.7	198.791	$20.895	$1,410,017	18,094	70
bxdiz0012	P	0	8,051	85,507	0	93,558	12	3,590,746	21,979	71,575	4	13,928	2.7	198.166	$20.895	$1,408,399	18,120	70
bxdiz0013	P	0	8,064	85,798	0	93,862	12	3,895,439	21,925	71,933	4	13,861	2.7	238.044	$25.095	$1,401,624	16,364	58
bxdiz0014	P	0	8,051	85,475	0	93,526	12	3,592,679	21,922	71,600	4	13,871	2.7	211.694	$22.260	$1,402,636	16,971	66
bxdiz0016	P	0	8,313	89,775	0	98,088	12	4,016,366	23,548	74,536	4	15,235	2.8	262.814	$27.615	$1,540,563	15,282	58
bxdiz0017	P	0	8,064	87,468	0	95,532	12	3,657,880	22,485	73,043	4	14,421	2.8	215.394	$22.680	$1,458,252	16,982	67
bxdiz0024	T	0	774	14,170	0	14,944	6	20,682	2,273	12,665	6	1,499	2.9	0.110	$0.105	$151,579	187,545	13,593
bxdiz0024	T	0	774	14,170	0	14,944	6	20,682	2,340	12,598	6	1,566	3.0	0.110	$0.105	$158,354	187,545	14,201
bxdiz0026	T	0	792	14,124	0	14,916	6	19,553	2,385	12,525	6	1,593	3.0	0.104	$0.105	$161,084	187,709	15,293
bxdiz0028	T	0	764	14,012	0	14,776	6	19,164	2,267	12,503	6	1,503	3.0	0.103	$0.105	$151,983	185,958	14,584
bxdiz0031	T	0	774	14,124	0	14,898	6	19,522	2,270	12,622	6	1,496	2.9	0.105	$0.105	$151,276	185,433	14,210
bxdiz0033	T	0	19	429	0	448	3	351	31	417	0	12	1.6	0.002	$0.105	$1,213	157,950	5,400
bxdiz0035	T	0	1,004	17,012	0	18,016	6	23,955	3,098	14,912	6	2,094	3.1	0.150	$0.105	$211,745	59,996	13,986
bxdiz0036	T	0	0	228	0	228	2	18	0	228	0	0	1.0	0.001	$0.105	$0	21,600	0
bxdiz0041	T	0	781	14,243	0	15,024	8	23,110	2,473	12,545	6	1,692	3.2	0.173	$0.105	$171,095	133,971	9,809
bxdiz0041	T	0	781	14,243	0	15,024	8	23,110	2,503	12,515	6	1,722	3.2	0.173	$0.105	$174,129	133,971	9,983
bxdiz0042	T	0	774	14,144	0	14,918	8	21,727	2,501	12,411	6	1,727	3.2	0.159	$0.105	$174,634	136,983	10,888
bxdiz0043	U	0	750	14,000	0	14,750	6	19,043	2,237	12,507	6	1,487	3.0	0.098	$0.105	$150,365	194,758	15,208
bxdiz0044	U	0	750	14,014	0	14,764	6	19,075	2,237	12,521	6	1,487	3.0	0.100	$0.105	$150,365	190,222	14,829
bxdiz0045	U	0	750	14,026	0	14,776	6	19,398	2,208	12,562	6	1,458	2.9	0.104	$0.105	$147,433	187,219	14,072
bxdiz0048	L	0	20	314	0	334	3	1,894	21	313	0	1	1.1	0.003	$0.105	$101	757,600	400
bxdiz0050	L	0	31	387	0	418	3	1,392	289	389	0	258	9.3	0.002	$0.105	$26,089	626,400	116,100
bxdiz0051	L	0	745	14,001	0	14,746	6	19,022	2,232	12,508	6	1,487	3.0	0.101	$0.105	$150,365	189,169	14,788
bxdiz0052	L	0	24	1,188	0	1,212	6	1,421	118	1,094	0	94	4.9	0.004	$0.105	$9,505	341,040	22,560
bxdiz0053	L	0	64	1,740	0	1,804	5	3,153	235	1,569	0	171	3.7	0.008	$0.105	$17,292	391,407	21,228
bxdiz0054	L	0	254	3,088	0	3,342	5	3,101	740	2,602	0	486	2.9	0.006	$0.105	$49,144	507,436	79,527
Total/Avg														2,647.756	$280.665	$18,230,217	161,708	12,777

Table 2: Arcweaver solution data for inlining depth zero. Amazon `c3.large` instance type virtual machines at unit cost of $0.105/hour. Benefit estimate using verification condition ratio model at $101.12 per improved verification condition. Data and analysis current through 31 August 2015.

Case	Module		Before Game Play				Arcweaver Fixed Point Solution							EC2 c3.large			Rate	
Module	Type	Depth	Safe	Warning	Error	Total	Iter	Arcs	Safe	Warning	Error	Improved	Factor	Duration	Cost	Benefit	Arc/hr	Impr/hr
bxdi10024	T	1	1,985	47,011	0	48,996	5	64,631	7,035	41,663	298	5,050	3.5	0.727	$0.105	$79,590	88,908	6,947
bxdi10026	T	1	2,024	48,122	0	50,146	5	64,397	7,348	42,182	616	5,324	3.6	0.688	$0.105	$83,908	93,668	7,744
bxdi10028	T	1	1,960	46,668	0	48,628	5	62,050	6,960	41,050	618	5,000	3.6	0.659	$0.105	$78,802	94,094	7,582
bxdi10031	T	1	1,996	46,892	0	48,888	5	62,911	7,114	41,152	622	5,118	3.6	0.670	$0.105	$80,662	93,936	7,642
bxdi10033	T	1	26	496	0	522	2	541	44	476	2	18	1.7	0.001	$0.105	$284	486,900	16,200
bxdi10035	T	1	2,726	57,746	0	60,472	5	77,125	10,203	49,644	625	7,477	3.7	1.108	$0.210	$117,841	69,639	6,751
bxdi10041	T	1	2,621	53,861	0	56,482	7	88,961	9,861	45,995	626	7,240	3.8	1.632	$0.210	$114,105	54,503	4,436
bxdi10043	U	1	1,944	46,596	0	48,540	5	61,880	6,904	41,018	618	4,960	3.6	0.651	$0.105	$78,172	95,078	7,621
bxdi10044	U	1	1,944	46,660	0	48,604	5	62,041	6,940	41,042	622	4,996	3.6	0.627	$0.105	$78,739	98,958	7,969
bxdi10045	U	1	1,993	46,855	0	48,848	5	62,434	6,976	41,254	618	4,983	3.5	0.661	$0.105	$78,534	94,478	7,540
bxdi10051	L	1	1,933	44,905	0	46,838	5	61,859	6,779	39,441	618	4,846	3.5	0.936	$0.105	$76,375	66,081	5,177
bxdi10054	L	1	763	9,155	0	9,918	3	9,017	2,635	7,279	4	1,872	3.5	0.014	$0.105	$29,504	636,494	132,141
Total/Avg														8.373	$1.470	$896,516	164,395	18,146

Table 3: Arcweaver solution data for inlining depth one. Amazon `c3.large` instance type virtual machines at unit cost of $0.105/hour. Benefit estimate using verification condition ratio model at $15.76 per improved verification condition. Data and analysis current through 31 August 2015.

Case	**Module**		**Before Game Play**				**Arcweaver Fixed Point Solution**							**EC2 c3.large**			**Rate**	
Module	Type	Depth	Safe	Warning	Error	Total	Iter	Arcs	Safe	Warning	Error	Improved	Factor	Duration	Cost	Benefit	Arc/hr	Impr/hr
bxdi20024	T	2	3,506	135,446	0	138,952	5	176,331	18,611	118,210	2,131	15,105	5.3	4.946	$0.525	$54,203	35,648	3,054
bxdi20043	U	2	3,437	134,369	0	137,806	5	171,713	18,163	116,896	2,747	14,726	5.3	5.051	$0.630	$52,843	33,995	2,915
bxdi20052	L	2	123	7,029	0	7,152	4	8,313	808	6,344	0	685	6.6	0.014	$0.105	$2,458	598,536	49,320
bxdi20053	L	2	675	15,545	0	16,220	4	20,840	2,298	13,561	361	1,623	3.4	0.065	$0.105	$5,824	319,251	24,863
bxdi20054	L	2	1,837	18,663	0	20,500	3	17,666	5,563	14,921	16	3,726	3.0	0.041	$0.105	$13,370	432,637	91,249
Total/Avg														10.118	$1.470	$128,699	284,013	34,280

Table 4: Arcweaver solution data for inlining depth two. Amazon `c3.large` instance type virtual machines at unit cost of $0.105/hour. Benefit estimate using verification condition ratio model at $3.59 per improved verification condition. Data and analysis current through 31 August 2015.

Case	**Module**		**Before Game Play**				**Arcweaver Fixed Point Solution**							**EC2 c3.large**			**Rate**	
Module	Type	Depth	Safe	Warning	Error	Total	Iter	Arcs	Safe	Warning	Error	Improved	Factor	Duration	Cost	Benefit	Arc/hr	Impr/hr
bxdi30024	T	3	5,800	301,892	0	307,692	4	416,944	40,476	262,212	5,004	34,676	7.0	27.546	$2.940	$124,432	15,136	1,259
bxdi30043	U	3	5,670	297,574	0	303,244	4	409,838	38,398	258,886	5,960	32,728	6.8	25.074	$2.730	$117,442	16,345	1,305
Total/Avg														52.620	$5.670	$241,874	15,741	1,282

Table 5: Arcweaver solution data for inlining depth three. Amazon `c3.large` instance type virtual machines at unit cost of $0.105/hour. Benefit estimate using verification condition ratio model at $4.91 per improved verification condition (see narrative for explanation of scaling difference for verification condition value between depth three and two). Data and analysis current through 31 August 2015.

We continue at this writing to experiment with the auto-solver iterators. We believe the Arcweaver iteration for the largest models terminates too early at a quasi-fixpoint having to do not with arc stability but with interval stability. We know the resulting verification shows buffers too narrow because the graph resolution appears incomplete; we summarize these as verification warnings. This does not affect the results for the other games.

We developed economic models to assess the value of these results. Our main approach was parametric and based on the cost of obtaining a result for a single verification condition. A program has a fixed number of verification conditions for the properties of interest. If we know the dollar cost of verifying the program manually for the same properties, then we can compute an average dollar cost per verification condition. Our crowd-sourced solution obtains results for verification conditions for very little cost each, so the difference is an economic benefit. We show these cost and benefit values in our results. Static analysis and abstract interpretation in particular, because it offers sound results, offer tremendous economic and verification value.

5.2 Attracting and Retaining Players

Our ability to attract and retain players with the CircuitBot game clearly did not meet our expectations. Where we envisioned thousands of players engaged in the exploration backtheme and contributing verification results, only a few dozen turned up. Of the few dozen who turned up, only a few played seriously for a long time. We later learned these were software engineering professionals interested in the problem and wanting to make a contribution. Part of engagement miss might have been the somewhat late and unexpected limitation to the 18-and older age group we did not anticipate, but we understood the rationale. Other factors in the engagement miss likely were the somewhat repetitive play experience, browser delivery, and murky purpose. We did not capture anonymous player results during CircuitBot; we did capture anonymous player results during Dynamakr so obtained more verification data this way. Our Dynamakr game improved the delivery and experience and brought-up the engagement numbers a little, but not to the levels we needed them to reach to achieve fixpoint iteration on real verification problems. The Dynamakr game dispensed with the exploration game backstory and chose a more immediate, phase-swapping, game switching back and forth between work time and play time. The VIPER game clearly was the best of all, attracting productive workers seemingly regardless of the delivery and play experience, and it did reach the level of potential fixpoint solutions. The VIPER game stripped way most of the pretense of the game and revealed the puzzle constraint problem underneath, asking the worker in a way to marshal the efforts of auto-solvers working on the servers. The VIPER workers seemed not to be motived by the verification or game challenge but by the monetary reward for solving puzzles.

5.3 Role for Humans

For each of the CircuitBot, Dynamakr, and VIPER games the human reasoning challenge was essentially the same. From the game player's perspective, the challenge to game play is that adding an arc 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 of 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 the solution nears a fixpoint some sections of graph and subset of the constraints become idle. Humans notice when the solution gets stuck at quasi-fixpoints that appear to be near solutions but are in fact fragile and upset by a judicious arc insertion. The expert move is one that abstracts away a small bit of information (our synergy game move) in exchange for breaking free from a quasi-fixpoint to make dramatic progress toward the real fixpoint. Although a solution that abstracts away all of the information (all nodes are connected to all other nodes) is sound according to our game model, it does not help improve verification at all. The best solution is one that abstracts away the least amount of information. Humans are able to do this using intuition, whereas auto-solvers are not yet clever enough to do this on real programs.

6.0 RECOMMENDATIONS

This section offers a few recommendations to ourselves and for future projects. Section 6.1 is the recommendation to ourselves to integrate our new verification techniques into mainstream analyzers. Sections 6.3 on the following page and 6.2 suggest there is more to explore in the forums of orchestrated verification competitions and the paid crowd source channel. Finally, Section 6.4 on the following page leaves behind a concept for a new exploration game that builds upon our lessons learned from three generations of crowd-source verification game building.

6.1 Verification Integration

Even with no inlining in place our new pointer analysis technique was able to improve the safe verification condition count by about 11% in programs of typical complexity. With inlining in place the size of the problem to solve expands accordingly but dramatic improvements in safe verification condition accounts will be achieved. Whereas CodeHawk already can attain memory safety proof levels over 80% without pointer analysis capability, another 10-20% possibly accrued with the support of pointer analysis will be significant.

For the CircuitBot project we forked a branch of the main CodeHawk analyzer software development tree to create our specialized analyzer. We will re-integrate the project's new CodeHawk pointer analysis software branch into the main CodeHawk analyzer software tree as a mainstream feature of the analyzer. This integration involves adding the constraint generator, brute-force auto-solver, and pointer analysis domain into the core analysis tool chain. The project's auto-solver is written in C# so consequently is less platform agnostic than other parts of the tool chain. We will investigate various methods for rendering a different auto-solver using other more conventional (off-the-shelf) solver engines.

6.2 Paid Iterations

As can be seen in the results data, we attracted many more contributors through the paid crowd source channel than through the unpaid channels. Within the limitations of time and money we experimented along the two dimensions of game size and solution reward to discover how much work we could accomplish per dollar. The platform appears to be a potentially bountiful and cost-effective way to produce results. Much more experimentation could be done here. When players are willing to work a puzzle for nearly an hour to earn a dollar, and there are plenty of puzzle-solvers available, this offers a potentially rewarding verification-as-a-service opportunity. The size of the puzzle presents a challenge that suggests a teamwork-like approach would be interesting. One can imagine a vertical teamwork model in which each contributor builds upon the work of the previous solver, earning rewards only for improving the result. One can also imagine a horizontal teamwork model in which multiple concurrent players cooperate on sections of the problem in order to solve a larger problem than any one individual could solve either because the problem is too large to deliver to his desktop or because the problem exceeds human performance capabilities.

6.3 Verification Tournaments

One of the authors had several discussions with a member of management of the Program's TA3 performer regarding the potential viability of holding *verification tournaments* with its crowd source platform. TopCoder already holds design, development, and data science challenges. The problem of verification is not far removed from either development or data science problem-solving. Whether packaged as game levels, or unpackaged as constraint problems, or merely pushed out as proof obligations, it may be more cost- and time-effective to host a tournament of willing participants to produce verification results. These challenge problems are more easily explained than design problems. These results are easily measured and easily checked. Participants can compute when they're finished, or compute when they have achieved a *good enough* or *best so far* solution. Competing solutions might earn partial credit for unique results. As we showed with CircuitBot, it is not necessary to reveal the program under analysis to the verification contributors, so if the target is sensitive it need not be disclosed.

6.4 Follow-On Exploration Game

We describe here an idea for a follow-on game development project for future consideration. The game builds upon all the lessons learned stated above, and builds upon the infrastructure already developed and the theoretical verification problems already under exploration. This explains where we would take the game were there to be a hypothetical Phase III activity, unpaid players without AMT support.

The current artifacts of CodeHawk pointer constraint analysis, which we used for all prior games, could also be used as the source for a Minecraft–style explorable world.[3] In this game concept, the 3-D world uses the developing points-to graph properties to create a dynamic world the player explores using his keyboard or touch controls. A map builder algorithm would process related game instances to develop a structural world map, locating adjacent instances onto a 2-D grid according to properties available in the function call graph and weightings of shared anchors and function parameters. When complete, a map texturizer

[3] Minecraft ®/TM & © 2009-2015 Mojang / Notch, http://minecraft.net

function would decorate the structural map into a fantasy world, with forests and mountains and other places to explore. When exploring this world each game instance behaves as a random seed for a section of the world, using constraints and nodes to define terrain features and the locations for trees, rocks, shrines, and other elements to be determined. The player's location in the world defines a starting point for a simple auto-solver.

In this "Minecraft" world the board sections would be defined by the pointer-flow constraints populating the game instances. As the player travels the board, the game generator will use the analysis artifact's call graph to identify the relevant game instances for the direction that they player is heading. If two players visit the same section of a world, the layout should be consistent, so that even though all players may arrive at the same area of the map in different ways, because the terrain is always generated from the same game instance, all players should be able to navigate through the same environment. Hills might represent many constraints referring to the same node. Valleys might reflect function parameter node types leading to neighboring instances. As the player travels forward, the most distance instances would be unloaded from memory to make room for new ones on the horizon. Individual nodes referred to in other instances can represent portals which teleport the player, possibly randomly, to other instances in the world. They might appear as magical items, or even just trees and rocks, depending on the number of graph vertices. An additional piece of magic available to players would be to allow the player to create magical creatures which they can set loose in the world. These would be different mini auto-solvers which the player can engage, aimed at a node, and they will teleport away following the node's unique identifier to any referring nodes, solving the related constraints, and then returning eventually with energy or gold for the player.

The entire milieu would need some overarching game play elements to tie the entire fantasy role playing package together. Enemies could be spawned in a similar way to the handling of the Dynamakr arcade game, where constraints which are missing information would spawn hostile creatures. A player can try to shut down the enemy spawner by creating their own creature which searches through the game instances for the missing arcs to activate the specific constraint and shut down the enemy spawner.

A secondary goal in the design is to manage some lingering resource management issues. By using a horizon-based game system we maintain in memory only the information which is *near enough* to the player's location to be relevant for game rendering purposes. We would use the same system to reduce the number of solution graph arcs and nodes in memory concurrently. We would add a new query algorithm for our game content. The new algorithm would use a time-decay feature to enable recency queries on both graph arcs and pointer flow constraints, enabling stale content to be presented to players to check the fixpoint iteration stability. We would also add subsetting tools to enable deeper levels of content inlining, which dramatically expands the quantity of graph nodes and vertices and imposes orders of magnitude increasing demand for memory and network traffic. We want increased inlining in order to increase verification accuracy. Another resource management tool might include *transient synergy*, wherein we apply the synergy move of our earlier games (collapsing nodes) until achieving an interim fixpoint solution, and then unwind that synergy bundle and attempt to work the problem locally with a subset of the graph and determine which points-to arc set results may change as a consequence. A player action might be to define these synergies then witness the 3-D world changing as a result this mysterious effect. There are some technical issues to manage with this approach, such as merging multiple synergy bundles, de-conflicting adjacent synergy bundles, or propagating synergy constraint and node list modifications. Again for resource management reasons we would move much of this work to the backend, where it was not done prior, and so our game model updating and fixpoint iteration algorithms would require modifications and event-driven triggers.

7.0 REFERENCES

[1] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[2] Bachman, F., Bass, L., and Klein, M. Deriving architectural tactics: A step toward methodical architectural design. Tech. Rep. CMU/SEI-2007- TR-002, Software Engineering Institute, 2003.

[3] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, second edition ed. SEI Series in Software Engineering. Addison-Wesley, 2003.

[4] Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

Appendix A Compute Platform Experiment

We chose to use cloud resources to deploy an array of auto-solvers to run against our game content. Many of our game content constraint problems are huge from a constraint count and expected graph size standpoint, and our solver requires that this data be kept in memory. Consequently we were interested in which virtual machine type would be most cost-effective for our solver – the *compute-optimized* type or the *memory-optimized* type. We ran an experiment which solved the same challenge problem (module `bxdiz0024`) on several types to collect the solution duration. The table below shows the results.

Family	Type	CPU	ECU	RAM	Store	$/hr	Time
Compute optimized	`c3.large`	2	7	3.75	2 x 16 SSD	$0.105	0:43:37
Compute optimized	`c3.xlarge`	4	14	7.5	2 x 40 SSD	$0.210	
Compute optimized	`c3.2xlarge`	8	28	15	2 x 80 SSD	$0.420	0:43:09
Compute optimized	`c3.4xlarge`	16	55	30	2 x 160 SSD	$0.840	0:43:24
Compute optimized	`c3.8xlarge`	32	108	60	2 x 320 SSD	$1.680	
Compute optimized PG	`c1.medium`	2	5	1.7	1 x 350	$0.130	1:18:30
Compute optimized PG	`c1.xlarge`	8	20	7	4 x 420	$0.520	
General purpose	`m3.medium`	1	3	3.75	1 x 4 SSD	$0.067	1:30:14
General purpose	`m3.large`	2	6.5	7.5	1 x 32 SSD	$0.133	
General purpose	`m3.xlarge`	4	13	15	2 x 40 SSD	$0.266	
General purpose	`m3.2xlarge`	8	26	30	2 x 80 SSD	$0.532	
General purpose	`m1.small`	1	1	1.7	1 x 160 SSD	$0.044	2:45:02
General purpose	`m1.medium`	1	1	3.7	1 x 410 SSD	$0.087	
General purpose	`m1.large`	2	2	7.5	2 x 420 SSD	$0.175	
General purpose	`m1.xlarge`	4	4	15	4 x 420 SSD	$0.350	
Memory optimized	`m2.xlarge`	2	6.5	17.1	1 x 420 SSD	$0.245	
Memory optimized	`m2.2xlarge`	4	13	34.2	1 x 840 SSD	$0.490	1:00:20
Memory optimized	`m2.4xlarge`	8	26	68.4	2 x 840 SSD	$0.980	0:57:59
Apple	iMac						1:04:33
DARPA	PROCEED						1:07:19

In the table the column Type refers to the Amazon Web Services (AWS) label for the EC2 instance type, CPU refers to the AWS notion of a virtual CPU (labeled by AWS as vCPU), ECU is the equivalent computing units, RAM is random access memory in gigabytes, Store is instance storage in gigabytes and either fixed (default) or solid state drive (SSD), Price is the run-time price in dollars per hour, and Time is the solution time for the test problem in HH:MM:SS. In the type annotations the PG annotation refers to the *previous generation* family. Where there is not a time value given we did not run the sample for that particular configuration.

The table provides two additional reference computer samples. We include an Apple iMac computer having 3.2 GHz Intel Core i5, and the DARPA PROCEED *Death Star* computer which is a special computer containing 64 Intel Xeon cores each at 2.13 GHz, non-hyper-threaded, sharing 1 TB physical RAM.

Because our auto-solver program is single-threaded we found unsurprisingly that it could not take advantage of multicore or multiprocessor computers. Consequently our test showed that the lowest solution cost was achieved on the compute-optimized virtual machine with the lowest CPU count and thus lowest lease rate. The `c3.large` type turned out to be both the fastest and least expensive choice. We then used this type for our future auto-solver work in the AWS cloud.

The wide availability of multiprocessor and multicore machines however compelled us to pursue development of a multithreaded solver. The key to our design is to enable threads to share the global points-to graph in memory while it is being updated by other threads.

Appendix B Databases

The following subsections identify the design of each database table for the backend. We used the same backend for all three editions of our games, and used largely the same tables for data management. We designed the data tables in Ruby using Sinatra, and included sufficient annotations in the source code such that the usual documentation tools can produce HTML or PDF documentation from the source code if desired. Moreover, launching the Ruby-language *models* creates the associated *collections* in the MongoDB server. Each subsection below corresponds to one such model and collection.

B.1 Awards

Property	Type	Index	Req'd	Description	Default
`award_id`	String	true	true	identifier key for the award	N/A
`description`	String			award description	“”
`award_type`	String			type of award: resource, badge, etc	“”
`item_id`	String			key identifier for award type lookup	“”

B.2 Badge

Property	Type	Index	Req'd	Description	Default
`badge_id`	String	true	true	identifier key for the badge	N/A
`description`	String			description of the badge	“”
`icon`	String			image file name, icon displayed when badge is displayed in social interface	“”

B.3 Call Graph

Property	Type	Index	Req'd	Description	Default
`function_caller`	String	true		game instance identifier source for this function UID	nil
`function_callee`	String	true	true	game instance identifier target for this function UID	nil
`xid`	String	true	true	an identifier for program under analysis, non-identifying	“”

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`function_uid`	String	true	true	the analyzer's function UID for the relationship	“”

B.4 Dictionary

Property	Type	Index	Req'd	Description	Default
`uid`	String		true	an identifier for the dictionary element	“”
`xid`	String		true	an identifier for the program under analysis, non-identifying	“”
`entry`	String			the dictionary support in XML	nil
`references`	Collection[String]			references to game instance identifiers (unused)	

B.5 Factory

Property	Type	Index	Req'd	Description	Default
`factory_instance_id`	String	true	true	factory identifier key	N/A
`player_id`	String	true	true	reference to player identifier	“”
`game_mission_id`	String	true	true	reference to game mission identifier	“”
`factory_type`	String			factory type identifier from factory definition collection	“”
`game_instance_id`	String			game instance identifier	“”
`location`	String			location for the model	“”
`creation_date`	Integer			date of factory creation in seconds since Jan 1, 1970	0
`production_progress`	Integer			last time factory produced resources in seconds since Jan 1, 1970	0
`satisfied`	Boolean			whether game has been satisfied	false
`condition`	String			factory condition: efficient, damaged, disabled, etc	“”
`status`	String			factory status: building, producing, idle	“”

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
auto	Integer			production cycle state, 1 at end of cycle	1
efficiency	Float			efficiency measure affecting production quantity, range 0 to 1	1.0
quantity	Float			total quantity of goods, used for transporting goods to and from Earth	0

B.6 Factory Definition

Property	Type	Index	Req'd	Description	Default
factory_definition_id	String	true	true	factory definition key	N/A
planet_id	String			planet identifier	“”
display	String			whether to display or hide this factory from this player	“”
name	String			display name	“”
icon	String			image file name, used in popups	“”
image	String			image file name, image displayed when resources are displayed or goal achieved	“”
summary	String			short description of the factory	“”
description	String			long description of the factory	“”
location_requirements	String			some factories need to be built in specific planet locations based on geography	“”
build_requirement_id	String			player must have already built this type of factory before this will become available to build	“”
build_req_quantity	Float			how much is needed in order to build the factory	0
build_robot_req	Float			how many robots needed to begin building	0
build_robot_release	Float			how many robots will be released after building	0

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`build_time`	Integer			how long will the building take	0
`production_type_id`	String			resource that will be produced by the factory, including non-resources like planetary defense or units for moving planet	“”
`production_quantity`	Float			how much will be produced when production time elapses	0
`production_time`	Integer			how long factory takes to complete production	0
`production_robot_quantity`	Float			how many robots needed for production	0
`consumes`	String			how many factories consume a primary resource each time it produces one production cycle	“”
`consume_quantity`	Float			how much of X is required	0
`consume_robot_quantity`	Float			how many robots are consumed during production	0
`level_up_consumes`	String			to level up a building requires consuming this resource	“”
`level_up_quantity`	Float			this much of this resource is consumed	0
`level_up_robot_consume_qty`	Float			how many robots will be consumed to level up	0
`level_up_robot_produce_qty`	Float			how many robots will be produced after level up	0
`level_up_id`	String			after level up cycle is complete, the factory will be transformed into this new identifier	“”

B.7 Game Instance

Property	Type	Index	Req'd	Description	Default
`instance_id`	String	true	true	game instance identifier	N/A
`level_id`	String		true	resource allocator level identifier, same as instance for our implementation	N/A

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`xid`	String	true	true	identifier for program under analysis, non-identifying	""
`model_data`	String			game specification in XML	nil
`dictionary`	String			dictionary support in XML	nil
`points_to`	String			points-to array pairs result of game play	nil
`points_to_graph`	String			points-to graph with arcs as written by game model	nil
`creation_date`	Date			date of creation for this document	nil
`played_count`	Integer	true		number of times the game instance has produced results	0
`variable_count`	Integer			number of variables in the game instance, used for scoring potential	0
`constraint_count`	Integer	true		number of constraints in the game instance, used for scoring potential	0
`complexity`	Integer	true		complexity measure, usually function of variable and constraint values	0
`performance`	Integer			a player performance measure, used to determine whether solution improved	-100
`priority`	Integer	true		resource allocator play priority, determined by complexity of all program under analysis functions	0
`function_uid`	String	true		the analyzer's function UID for this instance's model data	nil
`fixed_count`	Integer			number of times instance played without generating arcs	0

B.8 Goal

Property	Type	Index	Req'd	Description	Default
`goal_id`	String	true	true	goal identifier key	N/A

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`planet_id`	String		true	associated planets model identifier	N/A
`display`	String			some goals may be hidden as extra credit goals; which interface will display each goal	"Resource"
`sort_order`	String			used to manage order for display	"alphabetical"
`goal_title`	String			used to display quick list of tasks player needs to complete to win game	""
`goal_summary`	String			longer description, why does player need to do it?	""
`resource_def_id`	String			key to the resource definitions model identifier, what is this resource?	""
`quantity`	Float			how much of it needs to be generated?	0
`award_id`	String			key to an awards model identifier, award or badge will receive if goal accomplished	""

B.9 Graph

Property	Type	Index	Req'd	Description	Default
`aid`	String		true	an identifier for the graph arc	N/A
`uid_lhs`	String	true	true	identifier referenced on the left-hand side	N/A
`uid_rhs`	String	true	true	identifier referenced on the right-hand side	N/A
`xid`	String	true	true	an identifier for the program under analysis, non-identifying	N/A
`lhs`	String			the points-to graph in XML	""
`rhs`	String			the points-to graph in XML	""

B.10 History

Property	Type	Index	Req'd	Description	Default

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
snapshot	String			name of the snapshot, if any	""
error_count	Integer			number of errors proven	0
safe_count	Integer			number of safe accesses proven	0
warning_count	Integer			number of warning verification conditions to resolve	0
unreachable_count	Integer			number of unreachable verification conditions to resolve	0
player_count	Integer			number of players in player model	0
games_solved	Integer			number of games solved	0
games_unsolved	Integer			number of games unsolved	0
arc_count	Integer			number of graph arcs	0
worker_count	Integer			number of workers in worker model	0

B.11 Mission

Property	Type	Index	Req'd	Description	Default
mission_id	String	true	true	game mission identifier key	N/A
player_id	String	true	true	player identifier for the mission	N/A
planet_id	String			planet identifier key	""
completion	String			completion percentage	""
arrival	Integer			arrival date to start mission	0
is_complete	Integer			completion state, 0=not complete, 1=complete	0
name	String			descriptive name given to this mission by the game	""
summary	String			descriptive text to be used to describe mission on game mini-site	""

B.12 Planet

Property	Type	Index	Req'd	Description	Default
planet_id	String	true	true	planet identifier key	N/A
level_id	String			which internal level to load to display planet	""

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`name`	String			name of the planet	“”
`description`	String			description of the planet	“”
`active_state`	String			“Available” for planets that are available to the player, or “Deprecated” for planets that are in use but should no longer be offered as a choice	“Available”
`image`	String			image file name, larger view of planet for selection screens, planet mission info, completion	“”
`icon`	String			image file name, smaller view of planet for menus, awards	“”
`star_info`	String			additional information about the star and space around planet	“”
`completion-_message`	String			message to be displayed when all goal have been achieved	“”
`start_message`	String			message to display at beginning of game	“”

B.13 Player

Property	Type	Index	Req'd	Description	Default
`player_id`	String	true	true	game player identifier key	N/A
`total_points`	Integer			accumulated points awarded	0
`level`	String			player level associated with total points	“”
`world_clock`	Integer			number of game-world days the player has been playing	0
`games_attempted`	Integer			number of games attempted	0
`accumulated_time`	Integer			accumulated play time in minutes	0
`solutions_found`	Integer			number of solutions found	0
`first_to_play`	Integer			number of times the player was first to play an instance	0
`preferences`	String			specification of player preferences in JSON	“”

B.14 Player History

Property	Type	Index	Req'd	Description	Default
player_id	String	true	true	player identifier	N/A
instance_id	String		true	game instance identifier	N/A
when_started	Date			when did the player play this instance?	Today
play_duration	Integer			play duration in seconds	0
edges_found	Integer			number of points-to edges found	0
level_when_played	Integer			the player's performance level when he played this instance	0
play_options	String			snapshot of game options or settings when played in JSON	“”
play_metrics	String			game play performance metrics in JSON	“”

B.15 Resource

Property	Type	Index	Req'd	Description	Default
resource_instance_id	String		true	resource identifier key	N/A
player_id	String	true	true	player identifier	N/A
game_mission_id	String	true	true	game mission identifier	N/A
resource_definition_id	String	true	true	unique identifier for the resource	N/A
quantity	Float			total quantity of this resource based on production and use	0
maxquantity	Float			max quantity is total produced	0

B.16 Resource Definition

Property	Type	Index	Req'd	Description	Default
resource_definition_id	String	true	true	resource definition identifier key	N/A
name	String			resource name	“”
icon	String			icon file name, icon displayed when resource is displayed in interface	“”
image	String			image file name, image is display when information about resources is displayed or goal achieved	“”

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`displayed`	Integer			whether resource is displayed or just tracked invisibly	1
`description`	String			longer description of the resource	“”
`unit_display`	String			what types of units does this resource come in?	“”
`price`	Float			price of this resource when sold	1

B.17 Statistics

Property	Type	Index	Req'd	Description	Default
`function_id`	String		true	function identifier from pre-analysis	N/A
`xid`	String	true	true	executable identifier of program under analysis, non-identifying	N/A
`error_count`	Integer	true		number of errors proven	0
`safe_count`	Integer	true		number of safe accesses proven	0
`warning_count`	Integer	true		number of warning verification conditions to resolve	0
`unreachable_count`	Integer	true		number of unreachable conditions proven	0
`total_count`	Integer	true		number of total verification conditions, sum of above four	0

B.18 UID

Property	Type	Index	Req'd	Description	Default
`gid`	String	true	true	game instance identifier for this UID	N/A
`xid`	String	true	true	an identifier for the program under analysis, non-identifying	N/A
`uid`	String	true	true	the analyzer's UID for any term in the model data	

B.19 Worker

Property	Type	Index	Req'd	Description	Default
`worker_id`	String	true	true	worker identifier key	N/A

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
total_points	Integer			total points awarded, sum of analysis and dynamo	0
analysis_points	Integer			analysis points awarded	0
dynamo_points	Integer			dynamo points awarded	0
level	String			worker level associated with total points	“”
gi_seen	Integer			instances loaded into the analysis stage of the game	0
gi_solved	Integer			instances passed between analysis and dynamo stages	0
gi_processed	Integer			game instances solved by the game independently of worker	0
gi_improved	Integer			number of instances where player improved his score	0
first_to_play	Integer			number of times the worker was first to play an instance	0
arcs_added	Integer			cumulative number of arcs added by player's dynamo	0
analysis_time	Integer			cumulative seconds in the analysis stage	0
dynamo_time	Integer			cumulative seconds in the dynamo stage	0
tool_configuration	String			tool settings based on last game setup	“”
preferences	String			specification of worker preferences in JSON	“”

B.20 Worker History

Property	Type	Index	Req'd	Description	Default
worker_id	String	true	true	worker identifier	“”
instance_id	String		true	game instance identifier	“”
xid	String		true	executable identifier for the game instance	“”
session_id	Integer			session counter	0
tool_setup	String			tool setup when played	“”

continued on following page . . .

. . . continued from previous page

Property	Type	Index	Req'd	Description	Default
`played_count`	Integer			played count of instance when read	0
`priority_start`	Integer			instance priority at start of play	0
`priority_end`	Integer			instance priority at end of play	0
`performance_start`	Integer			instance performance at start of play	0
`performance_end`	Integer			instance performance at end of play	0
`arcs`	Integer			arcs added based on worker solving instance	0
`extra`	String			space for additional data (unused)	“”

Appendix C Run Book

The CircuitBot Project aims to improve the throughput of program analysis through crowdsourcing. The backend project provides a collection of Sinatra-based services to support the game and program analyzer.

C.1 Technologies

- The CircuitBot project uses a Thin, Unicorn or Rainbows! application server for scalable multiprocessing on Linux CentOS, Mac OS X, Windows, or Raspbian Wheezy.
- The application server runs a Rack web server and a variety of Sinatra-supported Ruby classes.
- Several application servers can run in parallel behind an Nginx proxy server for load allocation.
- The architecture is meant to support both public network and private (no network) use.
- The project uses the Unity game development and deployment technologies.
- The project uses the CodeHawk static analysis technology.
- The project backend can be deployed to Heroku, but will not support analyzer updates in that deployment.

C.2 Dependencies

- These instructions presume the target is CentOS 6. Similar instructions are known to work on other versions of Linux as well as Mac OS X and Windows. The backend installation package does not currently include analyzers built for the latter targets, although the analyzer is not needed to run the game and services; the analyzer could be applied off-line after results gathering.
- The backend services require association with a MongoDB NoSQL document store service.
- The backend analyzer requires installation of OCaml and CIL packages.

C.3 Preparing the Backend Target Platform

These instructions are for building the `CircuitBot` backend from source on CentOS 6 64-bit (including Amazon AWS EC2 virtual machines). The `CircuitBot` backend is written in Ruby so we install several Ruby packages from repositories. We also install the Ruby Virtual Machine (RVM) to enable swapping versions and we install `bundle` to manage Ruby gems. The prepared installation script installs development tools, Ruby, MongoDB, OCaml, CIL and some supporting packages all on this single host. These instructions presume access to a network for reaching Linux installation packages and GitHub repositories. To configure the backend to use different hosts (e.g. a separate host for MongoDB) simply edit the installation script.

1. `yum -y install git` installs `git` for reaching GitHub.

2. Change to desired installation directory, e.g. `cd circuitbot`.
3. `git clone --depth 1 https://github.com/mrbkt/circuitbot-dependencies.git` copies install packages and installation script. This is a public repository.
4. `git clone --depth 1 https://[user]@github.com/mrbkt/circuitbot-backend.git` copies install packages and installation script. This is a private repository, so replace `user` with your user ID and provide the password when prompted.
5. `cd circuitbot-dependencies`
6. `./install_dependencies.sh` presuming root privileges; otherwise use `sudo`. This will install many Linux packages using `yum` and will build OCaml and CIL directly from the dependencies repo. This script checks whether certain files were already installed and will skip their installation on subsequent runs.

Alternative Manual Methods Several wiki pages are available to guide installations using more manually-intensive methods. The wiki is alongside the `circuitbot` repository at GitHub. Clone the wiki repo as

```
git clone --depth 1 https://github.com/mrbkt/circuitbot.wiki.git
```

then find markdown files of interest for manual installations.

C.4 Running the CircuitBot Backend

First Time Configuration

1. `cd ../circuitbot-backend` continuing from target preparation steps.
2. Edit the backend configuration file `config/config.yml` to point to the desired hosts and ports for `backend_address` and `backend_port`. If the analysis products are situated in the backend then the `analysis_*_folder` and `export_graph_file` terms should point to the relative path within the backend. If the products are situated outside the backend then these terms should point to the absolute path outside the backend.
3. Edit the MongoModel configuration file `config/mongomodel.yml` to point to the `host` and `port` of the MongoDB database. If you are using MongoLab services then set the shell environment variable `MONGOLAB_URI` to point to the URI for the MongoLab services.
4. Run `bundle install` to install the required gems. If your environment does not find `bundle` or complains about a missing `rvm` then run the RVM configuration script `source /usr/local/rvm/scripts/rvm` to set up the RVM environment and then repeat this step.

Run the Backend The backend can use `bundle` or `foreman` to launch. It can use several application servers including `thin`, `unicorn` and `rainbows`. The default and most tested is `thin`. These instructions show how to run with `bundle` and `thin`. `bundle` will use the `config.ru` file, used by `rackup` to configure and launch the services. If you use `foreman` then edit the `Procfile` to set the desired environment and ports.

1. `cd circuitbot-backend` run from the backend top-level directory.

2. `bundle exec thin start -e production` to start `thin` in the production environment running all CircuitBot services. By default `thin` will use port 3000 but you can change this in the `config.yml` file or on the command line with a `-p` switch. (By default `foreman` will use port 5000.)

Check Whether the Backend is Running One easy way to check whether the backend is running is to ping one of the services. Each service has a ping GET method returning a one-sentence response if alive. For example use `curl --get localhost:3000/api/v0.6/admin/ping`. If running, the ping will respond `Administration service is alive`.

Another way to check the backend is running is to hit it from a browser. Use any of the GET routes from the browser address bar. For example enter `http://localhost:3000/api/v0.6/admin` to receive a simple notification page that the service is running.

Using the Administrative Control Pages The backend provides a simple browser-based administrative control page for common tasks. From a browser launch `http://<host>:<port>/api/v0.6/admin/control` to open the page. Several links to related service calls are available, as are some small forms for entering or querying data. Status information is shown at the bottom.

Logging and Notifications The logging configuration is specified in the file `config.yml`. Secondary notification for some important services is available via HipChat. For installations on a closed network there is no HipChat support and the administrator can disable this in the `config.yml` file by deleting the `hipchat_api_token`. Contact mrb@circuitbot.net to be added to the existing HipChat room for CircuitBot notifications, or set up your own with a different API token.

Services The following services run as individual applications in the backend and cooperate to provide the full backend experience:

- `admin_msg` for administrative control use
- `admin` for administrative use
- `award` for game play awards
- `badge` for game play badges
- `disaster` for game play disasters
- `factory_definition` for game play factory definitions
- `factory` for game play factories
- `game_instance` for game model instance specifications to game
- `goal` for game play goals
- `history` for play status snapshots
- `mission` for game play missions
- `planet` for game play planets
- `player` for game play player data
- `player_history` for player game completion history
- `resource_definition` for game play resource definitions

- `resource` for game play resources
- `statistics` for analysis statistics

Clients The backend uses a browser as its only external client.

API Version Use the API version in the URLs as `/api/v0.6`

- 0.1 preliminary
- 0.3 pairs testing
- 0.5 for end-to-end integration testing
- 0.6 for load testing and non-AMQP messaging

C.5 Ruby Gem List

The following Ruby gems are cached in the vendor folder of the backend. There are very few version number restrictions so for the most part we upgrade all gems as new versions appear. The most significant version restriction at this writing is with the `activemodel` gem where we cannot use the more recent version `4.0.0` owing to return message incompatibilities.

```
activemodel-3.2.13.gem
activesupport-3.2.13.gem
backports-3.3.3.gem
bson-1.8.6.gem
bson_ext-1.8.6.gem
builder-3.0.4.gem
daemons-1.1.9.gem
diff-lcs-1.2.4.gem
eventmachine-1.0.3.gem
haml-4.0.3.gem
hipchat-0.10.0.gem
httparty-0.11.0.gem
i18n-0.6.1.gem
json-1.8.0.gem
mail-2.5.4.gem
mime-types-1.23.gem
mini_portile-0.5.0.gem
mongo-1.8.6.gem
mongomodel-0.4.9.gem
multi_json-1.7.7.gem
multi_xml-0.5.4.gem
nokogiri-1.6.0.gem
polyglot-0.3.3.gem
pony-1.5.gem
rack-1.5.2.gem
rack-cache-1.2.gem
```

```
rack-protection-1.5.0.gem
rack-test-0.6.2.gem
rake-10.1.0.gem
rdiscount-2.1.6.gem
rdoc-3.12.2.gem
rdoc-rake-1.0.1.gem
require_relative-1.0.3.gem
rest-client-1.6.7.gem
rspec-2.13.0.gem
rspec-core-2.13.1.gem
rspec-expectations-2.13.0.gem
rspec-http-0.10.0.gem
rspec-mocks-2.13.1.gem
sinatra-1.4.3.gem
sinatra-advanced-routes-0.5.3.gem
sinatra-contrib-1.4.0.gem
test-unit-2.5.5.gem
thin-1.5.1.gem
tilt-1.4.1.gem
treetop-1.4.14.gem
will_paginate-3.0.4.gem
yajl-ruby-1.1.0.gem
yard-0.8.6.2.gem
yard-rspec-0.1.gem
yard-sinatra-1.0.0.gem
```

C.6 Building the CodeHawk Analyzer

The CodeHawk analyzer executables are found in the `circuitbot-backend` repository. These are already built for the CentOS target platform. If different platforms are needed please contact Matt Barry at (832)356-8211.

C.7 Game and Web Server Files

The CircuitBot game file is a web browser game and requires the Unity3D browser plugin. The player is available for the following web browsers: Internet Explorer, FireFox, Chrome, Safari, Opera - on Windows and Mac OS X. The plugin is available from the Unity web site.

The game and the supporting files will be copied to a web server which supports PHP 5.3 or later (with cURL library 7.10.5 or later). The game will run in a web browser and expects to communicate with the PHP scripts in the same location that the game launches from. The PHP scripts will communicate with our backend services as well as services provided by TopCoder (Resource Allocator API and Game API). The host and port address for each of the services is stored in the file `cbconfig.txt`. Although this file is included in the git repository, it will need to be edited to set the proper host and port values. We've created a browser page for doing this, `admin_inputconfigvalues.php`. This script will attempt to set the file permissions of the file (0666), but depending on the server configuration may not succeed. Set the file permissions to `rw-rw-rw-`.

Another admin tool is `admin_main.php`, used primarily as a development aid, provides access to the `admin_inputconfigvalues.php` page, displays achievement information, and provides access to a page for purging player information.

Appendix D Source Lines of Code

The following table identifies our count of the lines of analyzable C-language source code for the various directories of the BIND 9.9.3-P2 distribution.

Directory	SLOC
`bin`	79,331
`contrib`	55,498
`doc`	0
`docutil`	0
`lib/bind`	92,878
`lib/export`	2,563
`lib/isc`	42,644
`lib/isccfg`	5,219
`lib/tests`	669
`lib/dns`	99,405
`lib/dns/rdata`	16,742
`lib/dns/tests`	3,041
`lib/dns/win`	3270
`lib/irs`	2,552
`lib/isccc`	1,936
`lib/lwres`	7,353
`lib/win32`	25
`unit`	12,233
`util`	0
`win32utils`	0
Total	332,159

Appendix E Game Statistics: BIND Inlining Level 0

The statistics in Table 26 pertain to the game level collection CodeHawk generated for the BIND modules using zero levels of inlining. During the CircuitBot game development our team initially was concerned that game levels that were too “complex” by some measure would be too dense to display in the player’s browser and too cumbersome to play and to distribute well across the network. Initially we determined that about 36 constraints was a playable number, so these statistics revealed that nearly 85% of the generated game levels would be playable under that threshold. Unfortunately these also turned out to be tedious for the player. Later we determined that a much larger value could be tolerated. During the Dynamakr game development we in fact required to auto-solvers to play these smaller (less interesting) games and we presented the larger games in different ways to the humans.

	Complexity	Variables	Constraints	Score
Mean	245.0	20.3	21.7	50.5
Median	116.0	10.0	11.0	50.0
Standard Deviation	602.4	67.8	47.6	28.9
Minimum	3.0	0.0	0.0	0.0
Maximum	30348.0	3886.0	2190.0	100.0
Percentile 10%	32.0	2.0	3.0	11.0
Percentile 25%	52.0	4.0	5.0	26.0
Percentile 50%	116.0	10.0	11.0	50.7
Percentile 75%	207.0	16.0	19.0	70.7
Percentile 95%	834.0	67.0	75.0	95.0
Percentile 100%	30358.0	3886.0	2190.0	100.0
Percent Rank 36	15.0%	87.8%	84.6%	
Percent Rank 100	44.5%	97.3%	96.7%	
Games				154,424

Table 26: Game level statistics for BIND with zero inlining. Complexity is the number of nodes in the constraint definition. Variables is the number of game model variables in the constraint. Constraints is the number of flow constraints in the function. Score is the play priority value assigned to the game level according to its complexity.

Appendix F Backend Transactions

The following table illustrates the use of APIs across the TA3, game, and backend services. For each phase of game play it shows which service is performing a function and using a particular API.

Phase	N	Game	TA3 Host	Game API	Achievement API	Backend	Status
Prelude	P1					Read XML and extract game instances. Write game instance to document store. Performed in game model client.	Done in backend.
	P2					For each instance, extract its variable count, constraint count, and complexity. Compute a priority value, write these as priority and parameters array and parameters array to the game API using declare level metadata API 6.1. Performed in game model client.	Done in backend.

continued on following page . . .

... continued from previous page

Phase	N	Game	TA3 Host	Game API	Achievement API	Backend	Status
	P3			Stores level metadata description. Creates level if not defined.			
	P4					Import task knows when level generation is complete. Activate all levels API 5.10. Performed in game model client.	Done in back-end.
	P5			Activates all levels. Levels should be ready to be assigned upon player request.			
Initialize	I1		Delivers game to player's browser. Game includes configuration for address and port of Game API services, and address and port of CircuitBot backend.				

continued on following page ...

. . . continued from previous page

Phase	N	Game	TA3 Host	Game API	Achievement API	Backend	Status
	I2	Launches in browser					
	I3	Requests authentication of player from TC services.					
	I4		Authenticates with Oauth, passes token to game.				
	I5	Receives token for player					
	I6	Create player if necessary. Passes same player ID from TA3 host to CircuitBot backend as player ID.					
	I7					Creates new player using given ID. Create new blank player API 7.3. Performed in the player services.	Done in backend.
	I8			Creates a new player with given ID.			
	I9	Activate player for the allocator, calling game API 5.4.					

continued on following page . . .

. . . continued from previous page

Phase	N	Game	TA3 Host	Game API	Achievement API	Backend	Status
	I10			Activates player API 5.4			
Startup	S1	Request player's score, awards, by player ID using achievement API					
	S2				Returns player's score, awards		
	S3	Request game instance from backend game instance service.					
	S4					Requests match for player given player ID calling API 5.2. Performed in game instance service for player ID.	Done in backend. Changed route for earlier matcher to use *incomplete for player-id* segment.

continued on following page . . .

. . . continued from previous page

Phase	N	Game	TA3 Host	Game API	Achievement API	Backend	Status
	S5			Returns match for player, which provides a level ID.			
	S6					Find instance by level ID. Return instance to game. Performed in game instance service.	Not needed. The instance ID and level ID are the same value.
	S7	Read game instance ID response from backend. Request game instance data from backend.					
	S8					Returns game instance data from document store given game instance ID.	Not changed.
	S9	Retrieve game data from backend.					
	S10					Return instance and play support data from document stores.	Not changed.

continued on following page . . .

. . . continued from previous page

Phase	N	Game	TA3 Host	Game API	Achievement API	Backend	Status
Play	P1	Play game. Notify level started API 5.8.					
	P2			Receive level started.			
	P3	Write result back to game instance and increase play count.					
	P4					Store game instance data. Decrease the priority by 10% down to 1 minimum. Implemented in game instance *update and increment play count* route only.	Done in back-end.
	P5	Notify level stopped API 5.9.					
	P6			Receive level stopped.			
Wrap-Up	W1					De-activate all levels API 5.12.	Provided in admin control service.
	W2					De-activate all players API 5.13.	

Appendix G Backend Service Routes

The following table identifies the backend API's HTTP service routes and the application providing the corresponding service. Each service is a subclass of the `CircuitbotService` Sinatra service.

Verb	Route	Application
GET	/admin/ping	AdminService
GET	/help	AdminService
GET	/admin/help	AdminService
GET	/admin/?	AdminService
GET	/admin/routes	AdminService
GET	/	AdminService
POST	/admin/control/email	AdminService
GET	/admin/analyzer/analysis	AdminMessageService
GET	/admin/analyzer/preanalysis	AdminMessageService
GET	/admin/control	AdminMessageService
GET	/admin/control/notify/games/hipchat	AdminMessageService
GET	/admin/control/notify/players/hipchat	AdminMessageService
GET	/admin/control/notify/statistics/hipchat	AdminMessageService
GET	/admin/control/notify/workers/hipchat	AdminMessageService
GET	/admin/control/test/populate	AdminMessageService
GET	/admin/environment	AdminMessageService
GET	/admin/generate	AdminMessageService
GET	/admin/production/start	AdminMessageService
GET	/admin/production/stop	AdminMessageService
GET	/admin/reset/played_count	AdminMessageService
GET	/admin/snapshot	AdminMessageService
GET	/admin/snapshot/start	AdminMessageService
GET	/admin/snapshot/stop	AdminMessageService
GET	/admin/statistics/gather	AdminMessageService
GET	/admin/statistics/stop	AdminMessageService
POST	/admin/control/notify	AdminMessageService
PUT	/award/:award_id	AwardService
GET	/award/help	AwardService
GET	/award/peek	AwardService
GET	/award/view	AwardService
GET	/award/ping	AwardService
GET	/award/awards	AwardService
GET	/award/award_ids	AwardService
GET	/award/:award_id	AwardService
GET	/award	AwardService

continued on following page ...

... continued from previous page

Verb	Route	Application
GET	/award/	AwardService
POST	/award/:award_id	AwardService
DELETE	/award/all	AwardService
DELETE	/award/:award_id	AwardService
PUT	/badge/:badge_id	AwardService
GET	/badge/help	BadgeService
GET	/badge/peek	BadgeService
GET	/badge/view	BadgeService
GET	/badge/ping	BadgeService
GET	/badge/badges	BadgeService
GET	/badge/badge_ids	BadgeService
GET	/badge/:badge_id	BadgeService
GET	/badge	BadgeService
GET	/badge/	BadgeService
POST	/badge/:badge_id	BadgeService
DELETE	/badge/all	BadgeService
DELETE	/badge/:badge_id	BadgeService
GET	/call_graph/arc/count	CallGraphService
GET	/call_graph/dot/xid/:xid	CallGraphService
GET	/call_graph/dot/xid/uid/:xid/:uid	CallGraphService
GET	/call_graph/peek	CallGraphService
GET	/call_graph/ping	CallGraphService
GET	/call_graph/view	CallGraphService
DELETE	/call_graph/all	CallGraphService
PUT	/dictionary/:uid/:xid	DictionaryService
GET	/dictionary/:uid/:xid	DictionaryService
GET	/dictionary/field/:part_id/:xid/:uid	DictionaryService
GET	/dictionary/peek	DictionaryService
GET	/dictionary/ping	DictionaryService
GET	/dictionary/view	DictionaryService
DELETE	/dictionary/all	DictionaryService
DELETE	/dictionary/:uid/:xid	DictionaryService
POST	/dictionary/:uid/:xid	DictionaryService
PUT	/disaster/:disaster_id	DisasterService
GET	/disaster/help	DisasterService
GET	/disaster/peek	DisasterService
GET	/disaster/view	DisasterService
GET	/disaster/ping	DisasterService
GET	/disaster/disasters	DisasterService
GET	/disaster/disaster_ids	DisasterService

continued on following page ...

... continued from previous page

Verb	Route	Application
GET	/disaster/:disaster_id	DisasterService
GET	/disaster	DisasterService
GET	/disaster/	DisasterService
POST	/disaster/:disaster_id	DisasterService
DELETE	/disaster/all	DisasterService
DELETE	/disaster/:disaster_id	DisasterService
PUT	/game_instance/:instance_id	GameInstance
PUT	/game_instance/log/error	GameInstance
PUT	/game_instance/log/warn	GameInstance
PUT	/game_instance/log/info	GameInstance
GET	/game_instance/help	GameInstance
GET	/game_instance/view	GameInstance
GET	/game_instance/peek	GameInstance
GET	/game_instance/statistics/complexity	GameInstance
GET	/game_instance/statistics/constraints	GameInstance
GET	/game_instance/ping	GameInstance
GET	/game_instance/instances	GameInstance
GET	/game_instance/played/:instance_id	GameInstance
GET	/game_instance/played_count	GameInstance
GET	/game_instance/unplayed_count	GameInstance
GET	/game_instance/all_played_ids	GameInstance
GET	/game_instance/all_unplayed_ids	GameInstance
GET	/game_instance/:instance_id	GameInstance
GET	/game_instance/for/:player_id	GameInstance
GET	/game_instance	GameInstance
GET	/game_instance/	GameInstance
GET	/game_instance/export/graph	GameInstance
POST	/game_instance/increment/:instance_id	GameInstance
POST	/game_instance/clear/:instance_id	GameInstance
POST	/game_instance/up_and_inc_pc/:instance_id	GameInstance
POST	/game_instance/:instance_id	GameInstance
DELETE	/game_instance/all	GameInstance
DELETE	/game_instance/:instance_id	GameInstance
PUT	/goal/:goal_id	GoalService
GET	/goal/help	GoalService
GET	/goal/peek	GoalService
GET	/goal/view	GoalService
GET	/goal/ping	GoalService
GET	/goal/goals	GoalService
GET	/goal/goal_ids	GoalService

continued on following page ...

. . . continued from previous page

Verb	Route	Application
GET	/goal/:goal_id	GoalService
GET	/goal	GoalService
GET	/goal/	GoalService
POST	/goal/:goal_id	GoalService
DELETE	/goal/all	GoalService
DELETE	/goal/:goal_id	GoalService
PUT	/graph/:uid_lhs/:uid_rhs/:xid	GraphService
GET	/graph/:uid_lhs/:uid_rhs/:xid	GraphService
GET	/graph/arc/count	GraphService
GET	/graph/arc/count/subset/:xid	GraphService
GET	/graph/export	GraphService
GET	/graph/node/:uid/:xid	GraphService
GET	/graph/peek	GraphService
GET	/graph/ping	GraphService
GET	/graph/region/both/:xid/:hops/:uid	GraphService
GET	/graph/region/left/:xid/:hops/:uid	GraphService
GET	/graph/region/right/:xid/:hops/:uid	GraphService
GET	/graph/view	GraphService
DELETE	/graph/all	GraphService
DELETE	/graph/bundle/:xid	GraphService
DELETE	/graph/:uid_lhs/:uid_rhs/:xid	GraphService
PUT	/history/compute	HistoryService
PUT	/history	HistoryService
GET	/history/help	HistoryService
GET	/history/peek	HistoryService
GET	/history/view	HistoryService
GET	/history/ping	HistoryService
GET	/history	HistoryService
GET	/history/	HistoryService
DELETE	/history/all	HistoryService
DELETE	/history	HistoryService
PUT	/mission/new	MissionService
PUT	/mission/:mission_id	MissionService
GET	/mission/help	MissionService
GET	/mission/peek	MissionService
GET	/mission/view	MissionService
GET	/mission/ping	MissionService
GET	/mission/missions	MissionService
GET	/mission/mission_ids	MissionService
GET	/mission/playermissions/:player_id	MissionService

continued on following page . . .

. . . continued from previous page

Verb	Route	Application
GET	/mission/:mission_id	MissionService
GET	/mission	MissionService
GET	/mission/	MissionService
POST	/mission/:mission_id	MissionService
DELETE	/mission/all	MissionService
DELETE	/mission/:mission_id	MissionService
DELETE	/mission/player/:player_id	MissionService
PUT	/player/:player_id	PlayerService
GET	/player/help	PlayerService
GET	/player/view	PlayerService
GET	/player/peek/attempted	PlayerService
GET	/player/peek/solved	PlayerService
GET	/player/peek/points	PlayerService
GET	/player/attempted_count	PlayerService
GET	/player/solved_count	PlayerService
GET	/player/scored_count	PlayerService
GET	/player/peek	PlayerService
GET	/player/ping	PlayerService
GET	/player/players	PlayerService
GET	/player/player_ids	PlayerService
GET	/player/:player_id	PlayerService
GET	/player	PlayerService
GET	/player/	PlayerService
POST	/player/:player_id	PlayerService
POST	/player/gameupdate/:player_id	PlayerService
POST	/player/increment/attempted/:player_id	PlayerService
POST	/player/increment/solutions/:player_id	PlayerService
POST	/player/increment/first/:player_id	PlayerService
POST	/player/add/points/:player_id	PlayerService
POST	/player/add/time/:player_id	PlayerService
POST	/player/append/history/:player_id	PlayerService
DELETE	/player/all	PlayerService
DELETE	/player/:player_id	PlayerService
PUT	/planet/:planet_id	PlanetService
GET	/planet/help	PlanetService
GET	/planet/peek	PlanetService
GET	/planet/view	PlanetService
GET	/planet/ping	PlanetService
GET	/planet/planets	PlanetService
GET	/planet/planet_ids	PlanetService

continued on following page . . .

... continued from previous page

Verb	Route	Application
GET	/planet/:planet_id	PlanetService
GET	/planet	PlanetService
GET	/planet/	PlanetService
POST	/planet/:planet_id	PlanetService
DELETE	/planet/all	PlanetService
DELETE	/planet/:planet_id	PlanetService
PUT	/resource_definition/:resource_definition_id	ResourceDefinitionService
GET	/resource_definition/help	ResourceDefinitionService
GET	/resource_definition/peek	ResourceDefinitionService
GET	/resource_definition/view	ResourceDefinitionService
GET	/resource_definition/ping	ResourceDefinitionService
GET	/resource_definition/resources	ResourceDefinitionService
GET	/resource_definition/resource_definition_ids	ResourceDefinitionService
GET	/resource_definition/:resource_definition_id	ResourceDefinitionService
GET	/resource_definition	ResourceDefinitionService
GET	/resource_definition/	ResourceDefinitionService
POST	/resource_definition/:resource_definition_id	ResourceDefinitionService
DELETE	/resource_definition/all	ResourceDefinitionService
DELETE	/resource_definition/:resource_definition_id	ResourceDefinitionService
PUT	/resource/new	ResourceService
PUT	/resource/:resource_instance_id	ResourceService
GET	/resource/help	ResourceService
GET	/resource/peek	ResourceService
GET	/resource/view	ResourceService
GET	/resource/ping	ResourceService
GET	/resource/resources	ResourceService
GET	/resource/resource_instance_ids	ResourceService
GET	/resource/instances	ResourceService
GET	/resource/player/:player_id	ResourceService
GET	/resource/game_mission/:game_mission_id	ResourceService
GET	/resource/resource_definition/:resource_definition_id	ResourceService
GET	/resource/playerresources/:player_id	ResourceService
GET	/resource/:resource_instance_id	ResourceService
GET	/resource	ResourceService
GET	/resource/	ResourceService
POST	/resource/:resource_instance_id	ResourceService
DELETE	/resource/all	ResourceService
DELETE	/resource/:resource_instance_id	ResourceService
DELETE	/resource/player/:player_id	ResourceService
PUT	/statistics/:function_id	StatisticsService

continued on following page ...

... continued from previous page

Verb	Route	Application
GET	/statistics/help	StatisticsService
GET	/statistics/peek	StatisticsService
GET	/statistics/view	StatisticsService
GET	/statistics/ping	StatisticsService
GET	/statistics/functions	StatisticsService
GET	/statistics/total/:function_id	StatisticsService
GET	/statistics/:function_id	StatisticsService
GET	/statistics	StatisticsService
GET	/statistics/	StatisticsService
POST	/statistics/add/:function_id	StatisticsService
POST	/statistics/clear/:function_id	StatisticsService
POST	/statistics/:function_id	StatisticsService
DELETE	/statistics/all	StatisticsService
DELETE	/statistics/:function_id	StatisticsService
GET	/uid/count	UidService
GET	/uid/peek	UidService
GET	/uid/view	UidService
GET	/uid/ping	UidService
DELETE	/uid/all	UidService
PUT	/worker/:worker_id	WorkerService
GET	/worker/attempted_count	WorkerService
GET	/worker/help	WorkerService
GET	/worker/peek	WorkerService
GET	/worker/peek/arcs	WorkerService
GET	/worker/peek/seen	WorkerService
GET	/worker/peek/solved	WorkerService
GET	/worker/pink	WorkerService
GET	/worker/scored_count	WorkerService
GET	/worker/solved_count	WorkerService
GET	/worker/view	WorkerService
GET	/worker/:worker_id	WorkerService
GET	/worker/worker_ids	WorkerService
GET	/worker/workers	WorkerService
POST	/worker/:worker_id	WorkerService
POST	/worker/add/analysis_points/:worker_id	WorkerService
POST	/worker/add/analysis_time:worker_id	WorkerService
POST	/worker/add/dynamo_points/:worker_id	WorkerService
POST	/worker/add/dynamo_time/:worker_id	WorkerService
POST	/worker/gameupdate/:worker_id	WorkerService
POST	/worker/increment/first/:worker_id	WorkerService

continued on following page ...

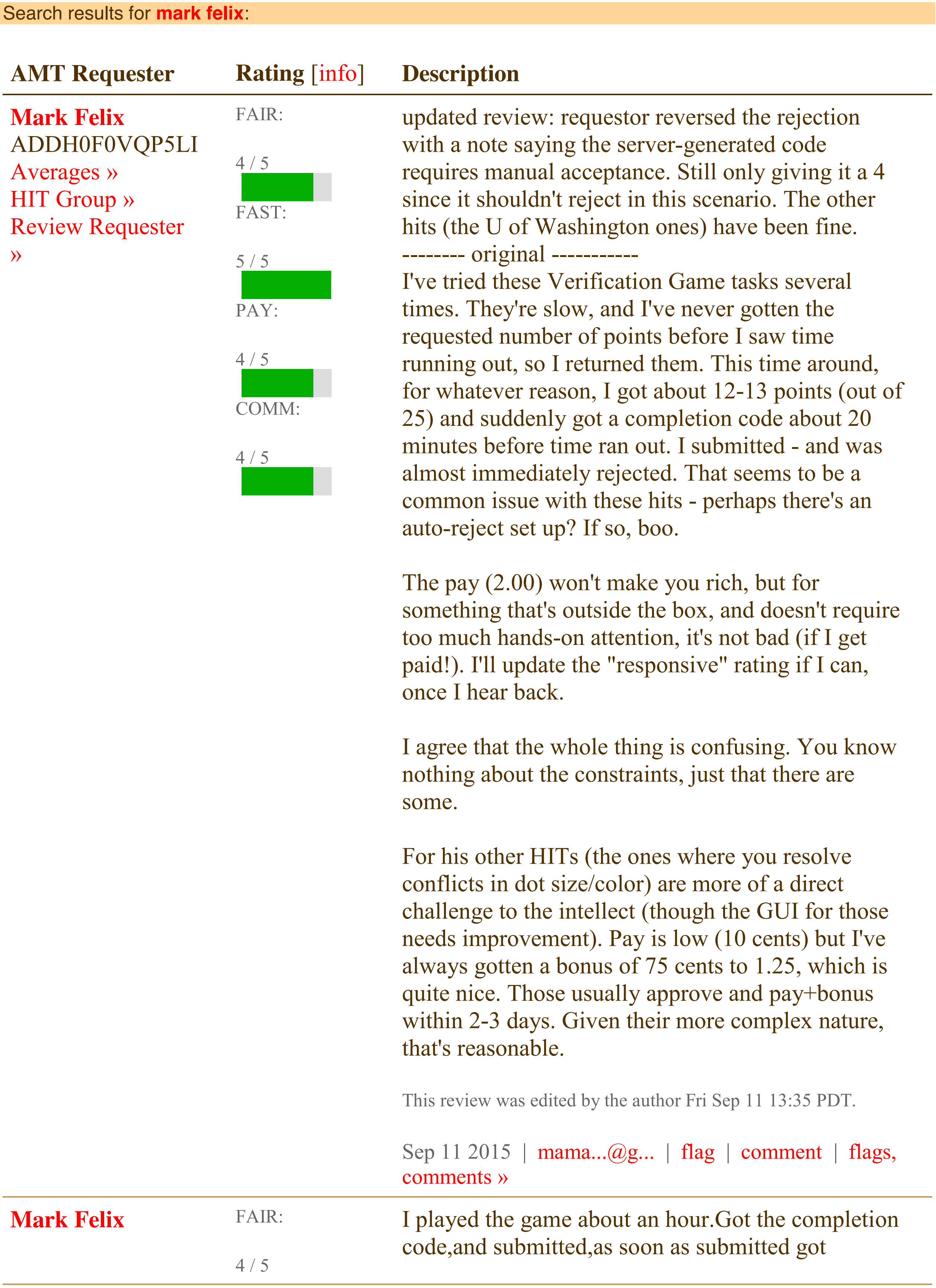
... continued from previous page

Verb	Route	Application
POST	/worker/increment/improved/:worker_id	WorkerService
POST	/worker/increment/processed/:worker_id	WorkerService
POST	/worker/increment/seen/:worker_id	WorkerService
POST	/worker/increment/solved/:worker_id	WorkerService
DELETE	/worker/all	WorkerService
DELETE	/worker/:worker_id	WorkerService
PUT	/worker_history/:worker_id	WorkerHistoryService
GET	/worker_history/:worker_id	WorkerHistoryService
GET	/worker_history/help	WorkerHistoryService
GET	/worker_history/instance/:instance_id/:xid	WorkerHistoryService
GET	/worker_history/peek	WorkerHistoryService
GET	/worker_history/ping	WorkerHistoryService
GET	/worker_history/view	WorkerHistoryService
GET	/worker_history/worker_history/:worker_id/:instance_id/:xid	WorkerHistoryService
DELETE	/worker_history/:worker_id	WorkerHistoryService
POST	/worker_history/:worker_id/:instance_id/:xid	WorkerHistoryService

Appendix H Game Feedback

The following pages are a database report representing some of the VIPER player feedback. The Amazon Turk players report good and bad experiences at a third-party forum web site Turkopticon[4]. We collected for our own information the reports for the VIPER human intelligence tasks (HITs) which TA3 published under the name "Mark Felix." We did not initiate contact with these players, but we did use the insights gained from these reports to adjust the difficulty of future HITs. The HIT feedback for this site is not exclusively for VIPER because all of the games were published under the same name, so the reports must be taken in general. However VIPER saw some of the highest traffic levels because we pushed several HITs. Throughout the AMT period we were experimenting with two dimensions of the HIT problem (payout and difficulty) to learn the size of the problem we could push out through this channel, and what payout would be required for this community to work on difficult problems. The verbal feedback and measured production rates helped with this experiment. Our measured difficulty and payout results are shown with the game results in Section 4.4 on page 183. Judging from the episodes of wavering positive and negative responses it is clear we were exploring the boundaries of the dimensions of interest. One normally would not expect many positive reviews on a site like this because it would tip off other workers to capture lucrative opportunities one might one to keep to oneself. We paid out all of our budget, so all of the games that could have been played within budget were played.

[4] http://turkopticon.ucsd.edu



Search results for **mark felix**:

AMT Requester	Rating [info]	Description
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 4 / 5 🟩 FAST: 5 / 5 🟩 PAY: 4 / 5 🟩 COMM: 4 / 5 🟩	updated review: requestor reversed the rejection with a note saying the server-generated code requires manual acceptance. Still only giving it a 4 since it shouldn't reject in this scenario. The other hits (the U of Washington ones) have been fine. -------- original ----------- I've tried these Verification Game tasks several times. They're slow, and I've never gotten the requested number of points before I saw time running out, so I returned them. This time around, for whatever reason, I got about 12-13 points (out of 25) and suddenly got a completion code about 20 minutes before time ran out. I submitted - and was almost immediately rejected. That seems to be a common issue with these hits - perhaps there's an auto-reject set up? If so, boo. The pay (2.00) won't make you rich, but for something that's outside the box, and doesn't require too much hands-on attention, it's not bad (if I get paid!). I'll update the "responsive" rating if I can, once I hear back. I agree that the whole thing is confusing. You know nothing about the constraints, just that there are some. For his other HITs (the ones where you resolve conflicts in dot size/color) are more of a direct challenge to the intellect (though the GUI for those needs improvement). Pay is low (10 cents) but I've always gotten a bonus of 75 cents to 1.25, which is quite nice. Those usually approve and pay+bonus within 2-3 days. Given their more complex nature, that's reasonable. This review was edited by the author Fri Sep 11 13:35 PDT. Sep 11 2015 | mama...@g... | flag | comment | flags, comments »
Mark Felix	FAIR: 4 / 5	I played the game about an hour.Got the completion code,and submitted,as soon as submitted got

Approved for Public Release; Distribution Unlimited.

ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	🟩 FAST: 4 / 5 🟩 PAY: 2 / 5 🟥 COMM: 4 / 5 🟩	immediate rejection.Mailed him no response yet. Updated Review -------------- He reversed rejection,with a note it's a server problem.It needs manual approval.So he approved manually. This review was edited by the author Fri Sep 11 19:56 PDT. Sep 11 2015 | sreeniv...@g... | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: NO DATA FAST: NO DATA PAY: 1 / 5 🟥 COMM: NO DATA	TERRIBLE HIT. 1. Requires UNITY. Chrome does not work, so firefox it is. Unity means that it's quite slow. No biggie. 2. Instructions are...well, I've read it in detail 3 times already and I STILL don't have a clue what I'm supposed to do. They explain how it works, but not exactly what you're supposed to do at all. They just say, "This connects to this. Ok good luck." Took me about an hour to finally get enough points to complete. Sep 11 2015 | andro...@g... | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 5 / 5 🟩 FAST: 4 / 5 🟩 PAY: 4 / 5 🟩 COMM: 5 / 5 🟩	This hit broke 3 times trying to use it, and the requester immediately rejected it. He reversed the rejection the next day. Updated scores to reflect. This review was edited by the author Sat Sep 05 09:04 PDT. This review was edited by the author Fri Sep 04 17:15 PDT. This review was edited by the author Fri Sep 04 17:14 PDT. Sep 05 2015 | Howard | flag | comment | flags, comments »

Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 5 / 5 🟩 FAST: 5 / 5 🟩 PAY: 1 / 5 🟥 COMM: NO DATA	"Crowd Source Verification Game Task Viper 1/25". $1.00, 55 min. As others have said, I played while doing other hits. Some of the bigger 'arcs' take a minute or to to load and another couple of minutes to get. My game had a goal of 25, but I won at 13, luckily. Otherwise it might have taken 2 hours! Approved as I was writing this review. Sep 04 2015 | jessema...@g... | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 5 / 5 🟩 FAST: 5 / 5 🟩 PAY: 1 / 5 🟥 COMM: NO DATA	Crowd Source Verification Game Task - Viper 2/20 on 9/3/15: 40 min, $2.00. Still have no clue how to play, but easy to set in the background during the loading parts. This review was edited by the author Fri Sep 11 00:16 PDT. Sep 03 2015 | worry | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 5 / 5 🟩 FAST: 5 / 5 🟩 PAY: 1 / 5 🟥 COMM: NO DATA	It appears the code issue is fixed for this HIT since I received a regular completion code at the end. I spent 5-minutes completing the task, which is extremely poor pay for the time invested. The bait of future higher paying work had me biting the hook. Approval was received within minutes following completion. This review was edited by the author Sun Aug 30 15:30 PDT. This review was edited by the author Sun Aug 30 14:25 PDT. Aug 30 2015 | NurseRachet | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

3 / 5

FAST:

NO DATA

PAY:

1 / 5

COMM:

1 / 5

Same as others, null code then rejected. Have contacted with no response. Avoid if possible.

Edit: Rejection reversed but no response from requester.

This review was edited by the author Mon Aug 10 08:52 PDT.

Aug 07 2015 | lrgi...@g... | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

1 / 5

FAST:

1 / 5

PAY:

1 / 5

COMM:

1 / 5

I did this hit, spent about 7 minutes on it, suddenly the test ended and I got a null code, which I entered and he rejected me soon after. I wrote him asking to reverse it but got no reversal yet.
Aug 06 2015 | melle224 | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

3 / 5

FAST:

NO DATA

PAY:

NO DATA

COMM:

1 / 5

Like others, I was rejected with no reason explained. I have a feeling it has to do with the completion code issue. I have messaged him, waiting to hear back.

Edit: Requester reversed my rejection. Did not respond to my email, however.

This review was edited by the author Mon Aug 10 08:42 PDT.

Aug 06 2015 | Vpakzu | flag | comment | flags, comments »

Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 1 / 5 FAST: 1 / 5 PAY: 1 / 5 COMM: 1 / 5	I did this qualifier and it broke, so I submitted it with what I thought was the code. I emailed him and told him and he still rejected me. Over a penny! AVOID! Aug 06 2015 | chris...@y... | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 1 / 5 FAST: NO DATA PAY: 1 / 5 COMM: 1 / 5	I completed a qualification HIT and submitted my completion code. I received a rejection quickly with no explanation. I emailed the requester and will update if they respond. This review was edited by the author Fri Aug 07 03:22 PDT. Aug 06 2015 | superhpbunny | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 3 / 5 FAST: 3 / 5 PAY: 3 / 5 COMM: 3 / 5	Spent just shy of 7 minutes on a $0.01 qualifier. Received code at the end that said "null" pasted it in - Rejected without explanation. Contacted requester, awaiting response. UPDATE: 8/10 Rejection reversed with note "undefined", however, still no response to any emails. This review was edited by the author Mon Aug 10 08:34 PDT. This review was edited by the author Thu Aug 06 09:27 PDT. Aug 06 2015 | kitten...@h... | flag | comment | flags, comments »
Mark Felix	FAIR:	Total waste of time.

ADDH0F0VQP5LI Averages » HIT Group » Review Requester » »	NO DATA FAST: NO DATA PAY: NO DATA COMM: NO DATA	Aug 03 2015 | james...@y... | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 5 / 5 🟩 FAST: 5 / 5 🟩 PAY: 3 / 5 🟨⬜ COMM: NO DATA	I will agree that the "Crowd Source Verification Game" is a waste of time, buggy, confusing, and prone to hanging completely when you're well into the task, at which point you can only start over again. Don't waste your time with that one. However, the "University of Washington Verification Game" is easy, fast, kind of fun, and pays a bonus. Those are worth the time---as long as they continue to pay a bonus (if they did not pay a bonus they would not be very worth it at 10 cents per). Fast approval, fast bonuses (on the ones that pay a bonus). EDITED: I did 15 "University of Washington Verification Game" tasks today and they were all very difficult, hard to solve, and time consuming. Even with the bonuses (smaller this time because I couldn't get to 100% on most) they are much less worth it for the time invested compared to prior versions. This review was edited by the author Mon Aug 03 17:03 PDT. This review was edited by the author Mon Aug 03 17:02 PDT. This review was edited by the author Mon Aug 03 16:14 PDT. Jul 30 2015 | VvAndromedavV | flag | comment | flags, comments »

Approved for Public Release; Distribution Unlimited.

Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: NO DATA FAST: NO DATA PAY: 1 / 5 🟥 COMM: NO DATA	Returned at 30 minutes into this buggy HIT. Game would freeze and go unresponsive. Not worth the frustration or pay. Jul 28 2015 | taylore | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: 5 / 5 🟩 FAST: 5 / 5 🟩 PAY: 2 / 5 🟥 COMM: 5 / 5 🟩	I played the game for at least 30 minutes and it still as not finished. Some parts of the game are broken and buggy. The instructions also don't make sense. Not worth the time for the pay of only $1! Update: requester reversed rejection and payed This review was edited by the author Tue Jul 28 20:49 PDT. This review was edited by the author Tue Jul 28 20:46 PDT. Jul 25 2015 | brantle...@g... | flag | comment | flags, comments »
Mark Felix ADDH0F0VQP5LI Averages » HIT Group » Review Requester »	FAIR: NO DATA FAST: NO DATA PAY: 1 / 5 🟥 COMM: NO DATA	1 hour in at 13/15 and unity crashes. waste of time and all for nothing, HIT returned. Jul 25 2015 | crappy...@y... | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

5 / 5
🟩

FAST:

5 / 5
🟩

PAY:

5 / 5
🟩

COMM:

NO DATA

I really enjoyed this HIT. I thought the games were interesting. They paid almost instantly and bonuses came very soon after. With the bonuses, the pay was good.

Jul 25 2015 | MaGra | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

5 / 5
🟩

FAST:

5 / 5
🟩

PAY:

5 / 5
🟩

COMM:

NO DATA

Interesting game. Nice bonuses. Total pay for 18 hits was almost $8 for less than an hour of work

Jul 25 2015 | michellech...@g... | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

5 / 5
🟩

FAST:

5 / 5
🟩

PAY:

5 / 5
🟩

COMM:

NO DATA

Fun to do and paid quickly.

Jul 25 2015 | kjre...@y... | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

5 / 5

FAST:

5 / 5

PAY:

4 / 5

COMM:

NO DATA

I completed all 17 of these hits today. They all involved playing a game of some sort. I never could understand the rules exactly, but I eventually fiddled with it enough to pass the levels.

All of the hits paid .10 for around a minute or less of work. The hits wouldn't let you submit until after 2 minutes had passed, but I just left them open until the time had passed and the code generated.

What really made these decent hits were the bonuses. From what I understand these were granted based on how many moves/time it took to complete the game.

Regardless the hits approved instantly, the bonuses were immediately sent out and all in all I made almost $15 in total from all of the work.

*Side note: I will mention that there were some other "game" hits from this same requester that I attempted. However the directions were unclear, the tasks were quite difficult and without knowing what kind of bonuses might arise from the work I didn't feel comfortable putting in that kind of time.

This review was edited by the author Fri Jul 24 17:55 PDT.

Jul 25 2015 | holst...@g... | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

NO DATA

FAST:

NO DATA

PAY:

NO DATA

COMM:

NO DATA

Can't even get this HIT to work regardless of browser choice. Unity is awful. Not worth the time, returned the HIT. Maybe he'll fix it at some point and I can try again.
Jul 16 2015 | Shadow | flag | comment | flags, comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

5 / 5

FAST:

5 / 5

PAY:

1 / 5

COMM:

1 / 5

One of the worst HITs I've done on mTurk. I don't even know why I decided to finish it. It took over 1 hour for 50 cents. I emailed him but he hasn't responded yet. Approved quickly though.

Jul 15 2015 | raji...@g... | flag | comment | flags, 1 comments »

Mark Felix
ADDH0F0VQP5LI
Averages »
HIT Group »
Review Requester »

FAIR:

NO DATA

FAST:

NO DATA

PAY:

1 / 5

COMM:

NO DATA

.50

Bad enough that it requires Unity.

But it's one of those hits with unclear and complicated instructions.

I would consider mastering the instructions for no less than $5.00.

But as is often the case with Unity the interface for the game itself is poor.

DON'T WASTE YOUR TIME.

This review was edited by the author Wed Jul 15 08:28 PDT.
This review was edited by the author Wed Jul 15 08:27 PDT.
This review was edited by the author Wed Jul 15 08:26 PDT.

Jul 15 2015 | mtur...@y... | flag | comment | flags, comments »

---

25 results

Query time: 0.11563

Render time: 0.00010

LIST OF ABBREVIATIONS AND ACRONYMS

Term	Description
ADD	Architecture Description Document
AMI	Amazon Machine Instance
AMT	Amazon Mechanical Turk
API	Application Program (or Programming) Interface
AWS	Amazon Web Services
BIND	Berkeley Internet Name Domain
BSON	Binary structured object notation
CSFV	Crowd-Sourced Formal Verification
CWE	Common Weakness Enumeration
DNS	Domain Name System
EC2	Elastic Compute Cloud
GNU	GNU's Not Unix
HIT	Human Intelligence Task; an AMT assignment; a VIPER game
HTTP	Hypertext Transfer Protocol
HUP	Human Use Protocol
IRB	Institutional Review Board
ISC	Internet Software Consortium
JSON	JavaScript object notation
KT	Kestrel Technology, LLC
LBG	Left Brain Games, Inc
MT	Mechanical Turk, here as Amazon's artificial artificial intelligence service
NEIRB	New England IRB
PCS	Paid crowd source, a catch-all term including AMT
PDF	Portable Document Format
PHP	Personal home page
PTA	Points-to arcs
PTG	Points-to graph
RA	Resource allocator
REST	Representational state transfer, an architectural style
S3	Simple Storage Service
SLOC	Source lines of code

continued on following page . . .

. . . continued from previous page

Term	Description
SNS	Simple Notification Service
TA1	Technical area 1, the CSFV game performers
TA2	Technical area 2, the CSFV defensive compiler team
TA3	Technical area 3, the CSFV integration performer
TTU	Texas Tech University
UID	Unique identifier, a CodeHawk identifier for model elements
URI	Uniform resource identifier
URL	Uniform resource locator
VC	Verification condition
VIPER	Verification improvement by PCS-enhanced results; our AMT game
VPC	Virtual Private Cloud
XML	Extensible markup language
YARD	Yay! A Ruby Documentation Tool