3.4.2 (Not Quite) Casual Math.

3.4.2 (Not Quite) Casual Math. *CyphrSeekr*, the original CHEKOFV concept game, was designed to allow players to describe the patterns they discovered as mathematical equations. In *Xylem*, we continued with this core mechanic as we felt this approach – despite alienating the less math-inclined of our potential audience – was productive for several reasons.

Perhaps most importantly, direct equation building allowed for a wide range of invariants to be created and submitted. Each loop may have more than one possible invariant, and each invariant collected would be useful to the backend processes that annotate the originating software. Allowing players to construct their own equations opened up the flexibility to receive multiple solutions for each puzzle while simultaneously taking advantage of the strengths of different player skill levels and play styles. This created a customized challenge for each player depending on their sophistication towards the game while taking best advantage of the strengths of crowd sourcing.

In addition to direct gameplay ramifications, focusing the main game activity on building equations had some practical aspects as well for possible future iterations. The framework is easily expandable in the future to include more tools if needed/desired. Several tools appeared in earlier versions of the game that were later removed to avoid confusing players. However, allowing more sophisticated players to unlock specialized tools (such as the mathematical logical construct, "implies") remains a possibility for future expansion. The direct equation-building approach also supports, in many situations, implementation of different data structures as game levels without having to completely redesign parts of the game. Figure 21 illustrates some of the evolving elements of styles and tools in early versions of *Xylem*.

A core tension of the game design was the desire to simultaneously have a large range of players while also having a large expressive range for their observations about loops. At times it seemed almost a one-for-one trade off between appealing to our desired audience and allowing for greater expressibility of the tool set. Early versions of the game, for example, included tools such as the mathematical symbol for the logical concept of "implies". While this would have allowed for a greater range of possible invariants that could be constructed, taking the time to explain the concept of "implies" to someone unfamiliar with it seemed like it would bog down the game flow and create a mental stumbling block. We ultimately decided that losing this bit of expressivity was less important than supporting a wider player base. Even so, *Xylem* remained sufficiently mathy, as it triggered concerns of math anxiety among some of our play testers.

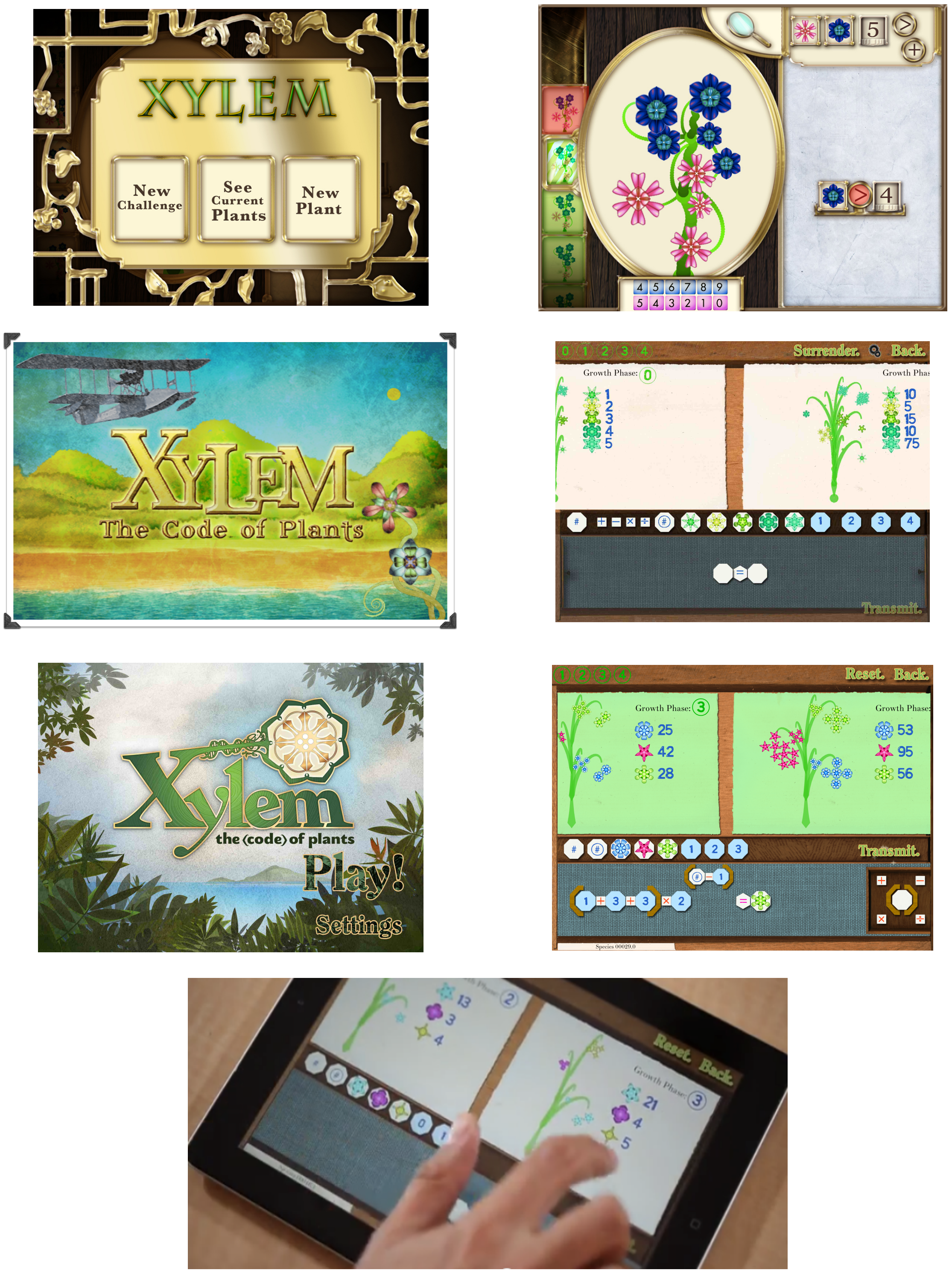
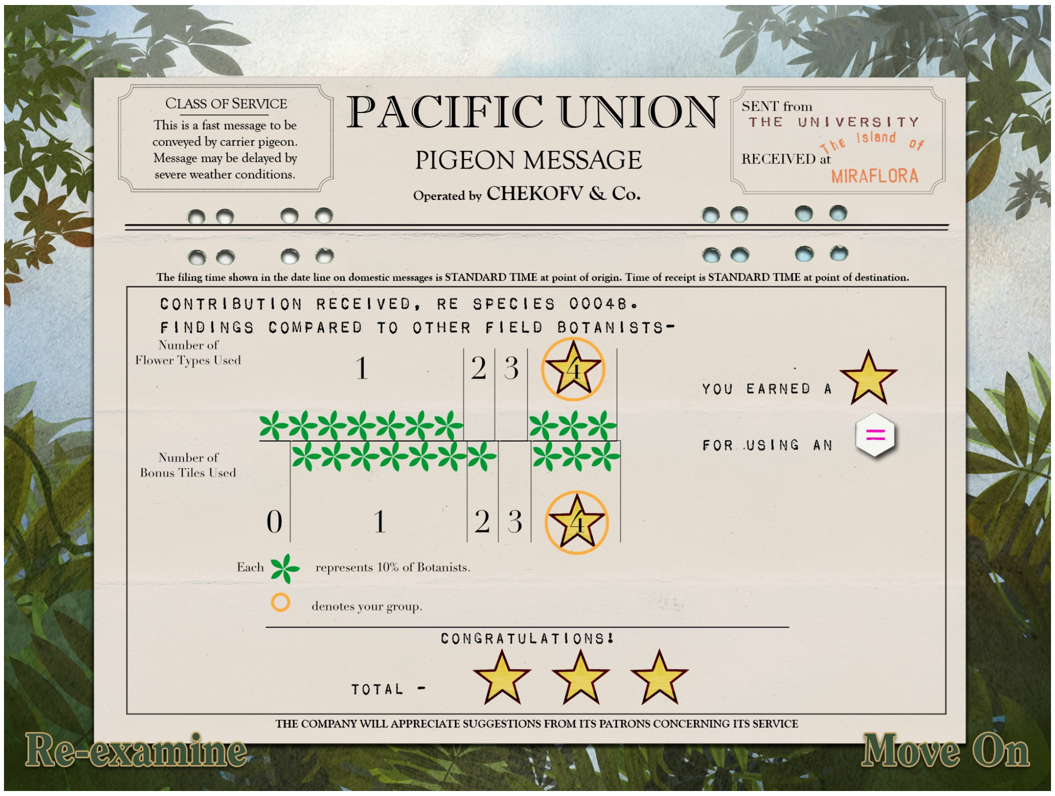


Figure 21. Evolution of introduction and play screens in Xylem

3.4.3 Rewarding the Player in *Xylem*. An important part of games is receiving feedback. Upon achieving a goal, players receive a burst of self-satisfaction. When goals are not achieved, players still see how close they came and can track their improvement in skills. Either way, this sense of achievement (or potential achievement) is an important part of what makes games enjoyable and compelling.

Xylem posed an unusual problem when it came to offering players feedback. Essentially, the game has no immediate way to gauge the utility or strength of a player-provided invariant. There are no established techniques for ranking the difficulty of an invariant-finding problem, assessing the quality of a solution, measuring incremental progress, or knowing when a problem is done.

In this situation, our task of designing game mechanics merged with the basic CS research into invariant discovery/analysis. The only way to know for certain the usefulness of a given invariant is to test it out on the original loop, in the context of a larger verification problem being explored via the use of out-of-game software verification tools. This suggested that the only way to provide feedback to players would be to have them submit their answers, wait some undefined amount of time for a remote expert to test out their solution, and then provide scoring based on this feedback. Thus, our early narrative design involved the players' solutions being transmitted to "The University" and players would continue playing other puzzles. The results would then be received by telegraph some time later; see Figure 22.


CLASS OF SERVICE
This is a fast message to be conveyed by carrier pigeon. Message may be delayed by severe weather conditions.

PACIFIC UNION
PIGEON MESSAGE
Operated by CHEKOFV & Co.

SENT from THE UNIVERSITY
RECEIVED at The Island of MIRAFLORA

The filing time shown in the date line on domestic messages is STANDARD TIME at point of origin. Time of receipt is STANDARD TIME at point of destination.

CONTRIBUTION RECEIVED, RE SPECIES 00048.
FINDINGS COMPARED TO OTHER FIELD BOTANISTS-

YOU EARNED A ★ FOR USING AN =

Each ✻ represents 10% of Botanists.
○ denotes your group.

CONGRATULATIONS!
TOTAL - ★ ★ ★

THE COMPANY WILL APPRECIATE SUGGESTIONS FROM ITS PATRONS CONCERNING ITS SERVICE

Figure 22. Presentation of results scores in *Xylem*

Although this delayed-results technique was somewhat satisfactory from a narrative point of view, traditional games don't tend to work that way. We felt that this approach alone would not be satisfying for players but, in the early design stage of the project, we did not have any useful heuristics to help us determine what a "good" invariant was.

Ultimately, we decided to use some simple parameters to provide immediate feedback, to reward the player. For example, we knew that using as many of the variables in the loop as possible is better than using fewer. We knew that utilizing the data from the time-zero iteration of the loop – initialization values, constants, etc. – generally resulted in better invariants (these became represented as the blue "bonus" tiles in the game). Furthermore, stating that one entity in the invariant is equivalent to another entity is typically stronger than stating a less-than or greater-than relationship. We also knew that we want to encourage players to come up with a variety of invariants. With a lack of solid answers for how to give players feedback, we built our scoring system on these four considerations. Figure 23 illustrates the player interaction screen.

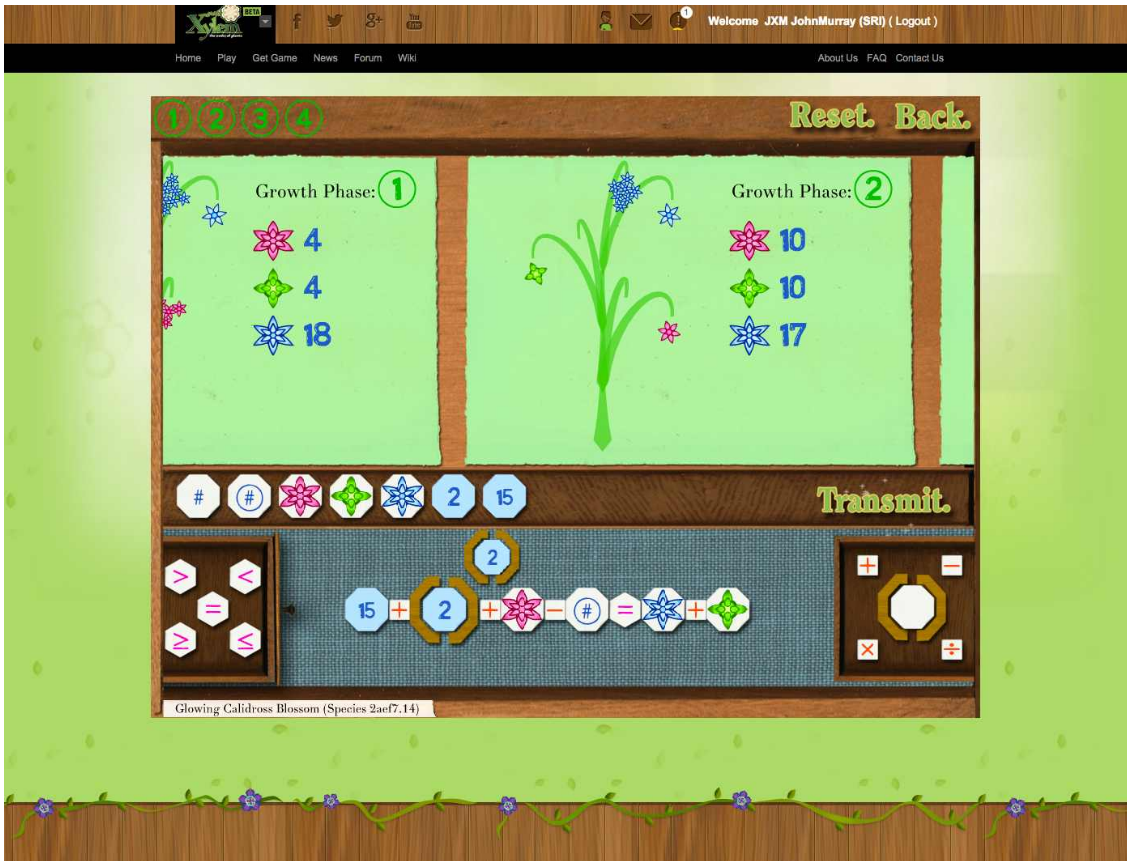


Figure 23. Example of pattern solution for game instance in Web version of *Xylem*

3.4.4 Problem Difficulty and Shaping the Player Experience. One design challenge in *Xylem* was how to craft the difficulty curve for each player. We examined several approaches to estimate puzzle difficulty. One was to link the structure of a loop (visible to the front end) to difficulty. More "guards" (conditionals in the loop) suggest more paths and potential for a more logically complex invariant *(if A then I1, if B then I2)*. However, it is unclear if this is true in practice.

A second approach was to employ social ranking that is analogous to web-page ranking in some search engines, as was noted in the CRS description in Section 3.3. If skilled players found a problem hard, it must be hard. If novice players solved a problem, it must be easy. This would be useful for serving up solved problems to new players in a sensible order, but the issue of ranking new problems remains open. Additionally, there is no firm basis for applying similarity metrics (associating loop features with loop difficulty) to leverage social analyses. A third option was to employ machine learning to classify problem difficulty from player ratings, but again, this is stymied by a lack of understanding of the underlying feature base. All of these approaches represent research into invariant-finding tasks; from the perspective of game design, it was easier, and sufficient to employ heuristic measures.

The heuristic measure of difficulty was based on the experience of team members playing pre-release versions of the game. We assume that integer-based problems are less difficult than array-based problems. We assume that working with more variables is harder than working with fewer variables. We assume that working with larger numbers is more difficult than working with smaller numbers. Based on these assumptions, difficulty profiles are assigned to each problem and problems are grouped by difficulty into different regions on the island. Unfortunately, offering players a smooth difficulty curve with this approach isn't possible. Other factors – which are hard to test for, especially on a large scale – can influence how difficult a given problem is. Therefore, a player could breeze through the first three problems in the "easy" region, then encounter a very hard problem followed by another easy one. This does not make for the type of hand-crafted difficulty curve that highly successful games can take advantage of, and occasionally causes player confusion and frustration.

3.4.5 Teaching the Game. The tutorial development for *Xylem* began after the initial theming and general concept of the game was pinned down. The task of getting new players on board proved to be a massive challenge: presenting a problem this involved as an approachable, fun experience that is easy to learn required much research, design, and iteration. The first versions of the tutorial existed as paper prototypes, with which the designers were able to test different ideas quickly and efficiently with many testers. Only through extensive testing and iteration were we able to settle on a tutorial that effectively leads the player into the game while teaching them the basic skills they need to know in order to succeed. The tutorial design and polish took place in parallel with the core game design, requiring as much design time and effort as the rest of the game combined. Early feedback from external game players indicate that the tutorial does successfully train players to play the game, but is too long, a direct consequence of the many game elements that need to be taught to the player. An early tutorial design document for the game that became *Xylem* may be found in Appendix 4.

3.4.6 Aesthetic Experience. *Xylem*'s aesthetics were developed with the dual goals of creating a pleasant place for the player to spend their time and creating consistency with the 1920's theme. Because *Xylem* was meant to be a slow, contemplative game (but with a hint of adventure), it was important that the visuals and music fostered this atmosphere. A great deal of research into map styles in the appropriate time period was done by our artist before the specific watercolor look and map orientation was decided upon. (See Figure 24)

Likewise, our sound designer researched silent adventure movies and period jazz and classical music to develop the ambient music pieces that play on the map screen and puzzle screens.

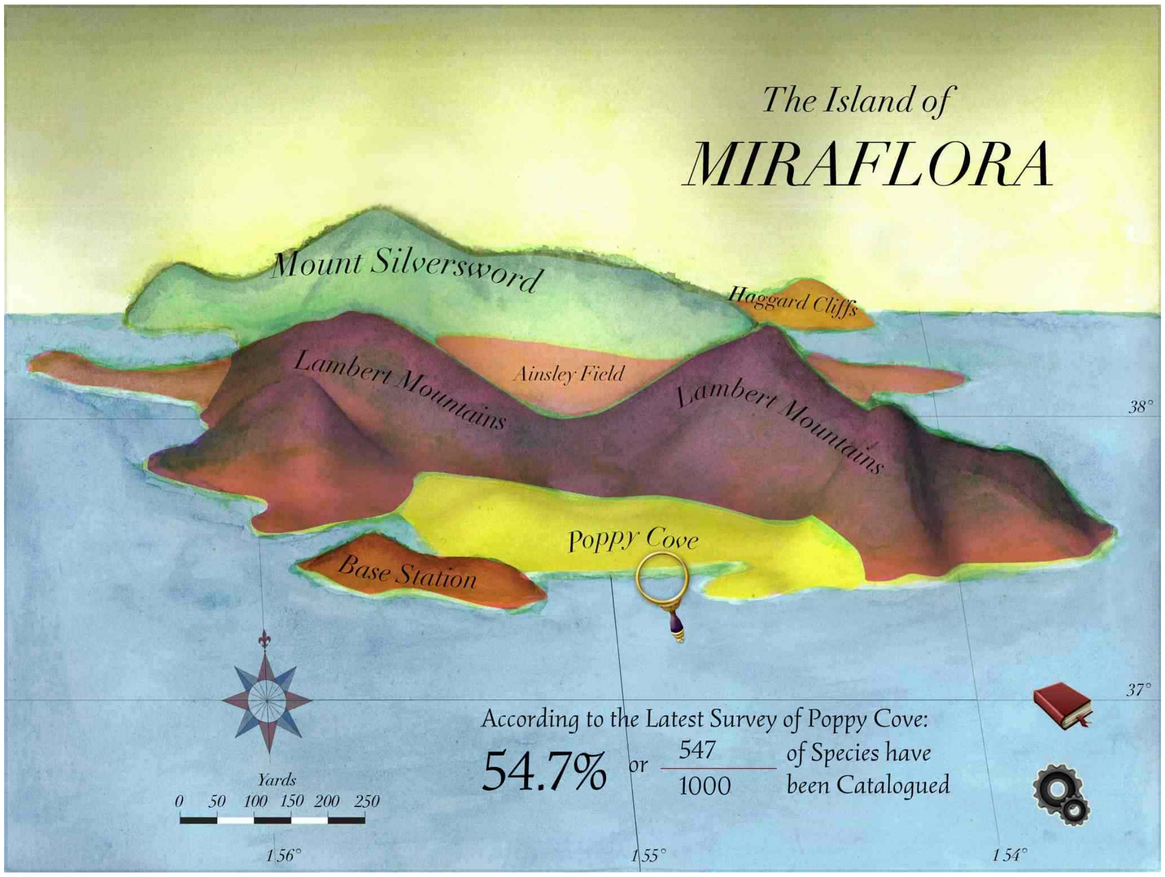


Figure 24. Miraflora – the island setting for *Xylem* player's explorations

The photorealistic look of the game interface stemmed from initial design-space ideation photography for the introductory desktop on the opening screen. This approach was further developed and used for other in-game assets. At that point, we knew that we wanted a similar look for the actual in-game UI. This direction added to the first person immersion of the game and, when combined with the corresponding period sound effects, helped to create a strong tactile feeling to the game that encourages players to manipulate the playing pieces.

3.4.7 Mobile Platform. The decision to adopt the iPad as *Xylem*'s target platform was based on considerations of the chosen audience and how the game would most likely be played. In particular, the iPad as a platform appeals to a wide variety of people – the majority of tablet owners are between the ages of 35 and 44, and are spread equally amongst the genders, whereas the smartphone user demographics skew a bit younger.

Given the contemplative nature of the gameplay, we imagined players playing the game while sitting somewhere comfortable for at least thirty minutes. More than either a smartphone or PC, the iPad has a form factor that encourages this sort of behavior. Additionally, given the number of icons that we are asking players to work with, the extra screen real estate provided by the iPad over the smartphone seemed ideal.

3.4.8 Considerations for Cooperative Play. A key vision for CHEKOFV was for players to enjoy our game together; however, sorting out exactly how to achieve this was a long and involved iterative process. The situation was further complicated by the privacy concerns associated with exposing individual player's personal information. Given the intended audience and the emotional feel we wanted for the game, we opted for a collaborative scenario, albeit a very light one. We wanted to create player investment by encouraging the feeling that all players were working together towards some greater goal. It was important that players knew other people were playing the same game at the same time. This was intended to support a sensation of being “alone together”, which was seen to be more enticing to our target audience than the more lonely feeling of interacting with a single player game.

It was also thought that the audience would be drawn in further to the fiction of the game, which would help create an impetus for players to interact on the forums (in addition to helping each other with hard problems). A complex backstory to the island was created, which was to be revealed in a semi-random manner one clue at a time. Players were to collect clues, compare them on forums, and discuss theories. The intended effect was a meta-layer of collaboration that would occur outside the game itself to reward and draw in the core player base.

Design deliberations also considered sharing player-created resources. Though initial resource constraints prevented its inclusion in the game, a feature for having players create and share “helper functions” was designed. Such helper functions would make it easier to see relationships among variables, or help subdivide larger problems into smaller pieces. Such capability would require a new layer of user tools for composing, naming, exporting, and importing these tools.

Information about the outcomes from *Xylem* gameplay may be found in Section 4 – Results.

3.5 Phase Two: Citizen Science Games

3.5.1 Transition and Reimagination. As we approached the design of another CHEKOFV game for Phase Two of the CSFV program, it became clear that we needed to rethink our assumptions about the audience to which we were appealing. In game design (as in many things) the target audience is the number one consideration that informs all design decisions, from game mechanics to color scheme. With *Xylem*, our vision was to design a casual-type puzzle game. Given the not-quite-casual math aspects of the game, we knew we were already limiting our appeal but we attempted to make up for it with narrative, story, a tactile touchscreen UI and aesthetics. However, we were obfuscating the science goals of the project, which was originally thought was a positive thing. When combined with the sudden jumps in math-difficulty, this alienating players who were otherwise drawn to it.

On the other hand, *Xylem* did attract a core group of very passionate players. We interviewed several of these players about their experiences playing the game. From this work, it became clear that our core audience was not the audience we had originally set out to capture. The most avid players were software engineers who were intrigued by the software-verification science goals of the project. They also felt that some of the work we were most proud of -- our narrative and world-building components -- were distractions that got in their way. In particular, they were keen to solve puzzles and feel like they were contributing to software security.

This player orientation was also confirmed by the marketing analysis of overall CSFV audience activity on the *Verigames* website, which further revealed that the visitors were primarily interested in the science behind the games, rather than the games themselves. In other words, the publicity articles about the CSFV project were generally drawing a science oriented audience who were interested in crowd-sourcing science projects, rather than the puzzle gamers originally hoping for. Because the most prolific players of *Xylem* were computer scientists, and because our audience was turning out to be mostly people who were intrigued in the science aspect of the project, we decided to take a very different tactic with our second game. In fact, we rethought the notion of making a "game" at all.

3.5.2 Citizen Science and Safe Passage. A *citizen scientist* may be thought of as "a member of the general public who engages in scientific work, often in collaboration with or under the direction of professional scientists and scientific institutions." In other words, an amateur scientist. (See www.en.wikipedia.org/wiki/Citizen_science).

Given our experience of the player interests and behavior with *Xylem*, we adopted a strategy of appealing to and cultivating this audience for our Phase Two offering. Specifically, we set our goals for experience as targeting a citizen scientist audience with a fun game that allowed multiple players to collaborate on a single problem. From the CSFV point of view, the goal was to create a game to compose and assemble conjunctive and disjunctive invariants, because this process is difficult for automated systems to produce.

While these goals are fairly abstract, we were able to refine them by reasoning backwards from the needs of the target audience and forwards from our experience with *Xylem*. In particular, given that the target audience was motivated by the science goal, we chose to expose more of the invariant-creation task to players than we had done in *Xylem*. To make the experience fun for this audience, we realized that the accomplishments needed to be incremental and easy to perform. That meant decomposing invariant creation into smaller steps than in *Xylem*, where each puzzle required significant concentration and time. Nevertheless, our work on *Xylem* suggested a way forward - if we could phrase invariant creation as data-driven task, we could define incremental accomplishments in terms of the data examined or explained.

This line of reasoning produced a key insight; we could model invariant composition as the task of separating good program states from bad. Here, a program state is a snapshot of variable values describing some moment during program execution. For example, if the task is finding loop invariants (as in *Xylem*), any state produced by the loop at a given iteration is good, while a bad state is a vector of variable values that could never be produced by the loop at that time. If the task is finding program preconditions, good states are inputs that satisfy program post-conditions on execution, while bad states violate those post-conditions. This perspective opened up a variety of game-design concepts; we could cast good/bad states as friends/enemies or desirable/undesirable objects, etc., and design game mechanics around the task of selecting one class, while defeating/rejecting the other.

The initial concept design for the Phase Two CHEKOFV game was called *Safe Passage*, and employed the metaphor of distinguishing friends from enemies. Figure 25 provides early sketches. The player is charged with preventing an influx of invasive species while encouraging the preservation of native species. Triangles are traps that catch some creatures and not others. Game play proceeds in two steps. In the first stage, the player places traps in sequence along corridors, and introduces branch points into the maze. Then, when the player presses the "Run" button, the fish and shells traverse the maze. They choose a direction at branch points, and pass through, or are caught by traps en route.

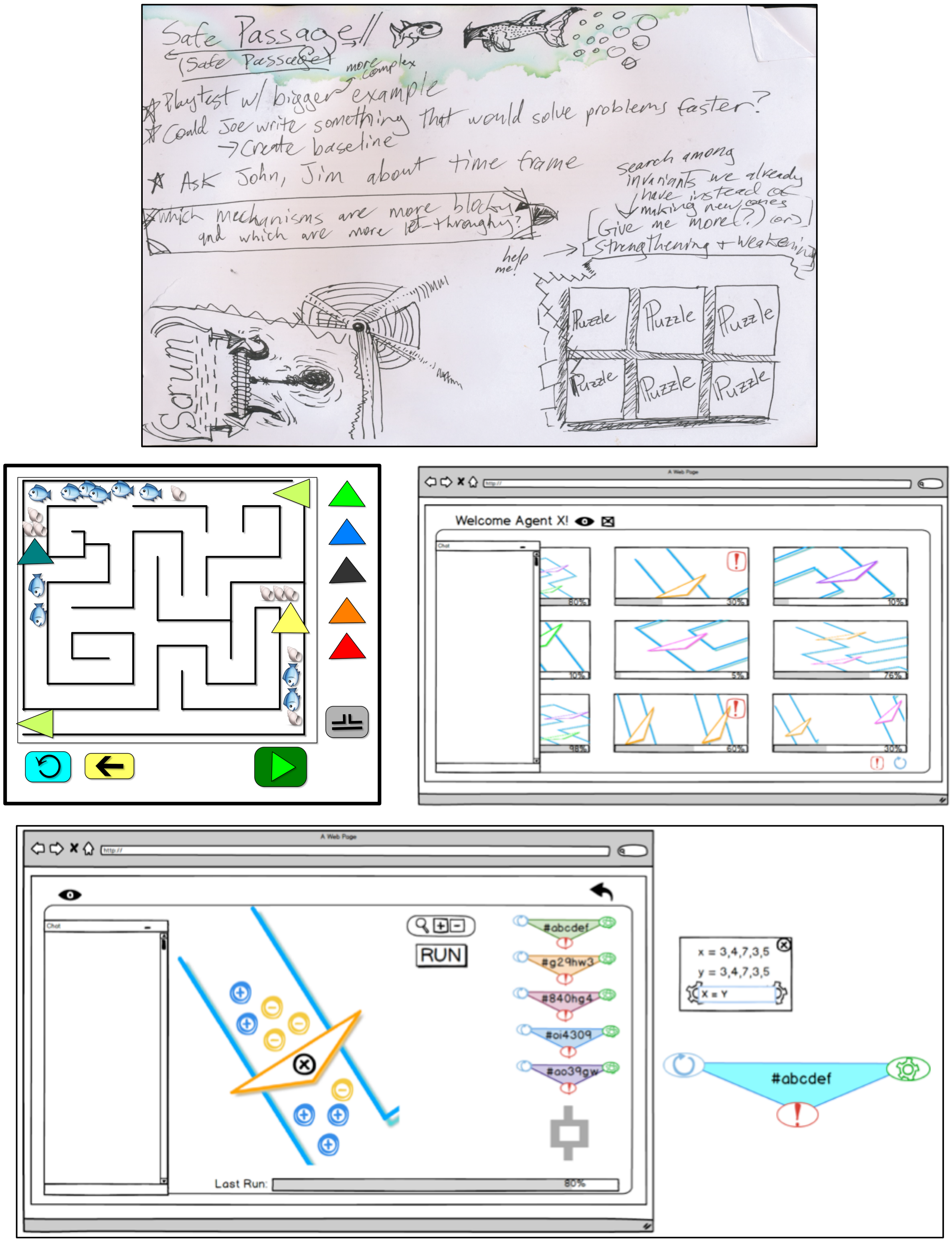


Figure 25. Early exploration of *Safe Passage* game concepts

With respect to the underlying science task, each fish is an instance of a native species and represents a good program state. Each shell represents an invasive species and a bad program state. Each trap corresponds to a boolean predicate that acts on program states, and passes any state/creature that evaluates to true. The set of possible traps are defined in advance. Sequential traps create conjunctive predicates, while branch points in the maze create disjunctive predicates. After the species run the maze, the game software reads out the expressions that characterize the individuals caught in each trap, or that exit the maze. In a partial solution, some fish and some shells are caught in certain traps, and some of each exit the maze. In the ideal situation, all fish (all good states) make it through the maze, while all shells (all bad states) are trapped en route. The very best mazes have fewer traps, suggesting simpler, more general solutions.

Safe Passage included a social mode of game play, where multiple users could collaborate on the same maze or share sequences of traps. We anticipated running synchronous and asynchronous team-based tournaments. We also envisioned an interplay between *Safe Passage* and a software package (like *Daikon*) that produces the vocabulary of predicates/traps; it let the player request new traps that would better separate the fish and shells already caught.

Our Phase Two design work substantially advanced our understanding of the constraints for building software verification games. Table 6 summarizes the comparisons and lessons learned during Phase One, and their implications for Phase Two. The most notable advances are in the increased clarity on the intended audience, and in the style of the game that should appeal to them.

It was also noted that our original game would benefit from affordances for a “wizard” or expert human player, and also a software-based robot to remove puzzles that are evidently intractable and hand them over to the experts for further study. The addition of such affordances was taken into consideration for follow-on games, where they could also be used to introduce new puzzles that were targeted to particularly successful players.

Safe Passage itself was never built out to the level of playable game, as we discovered new issues and more elegant solutions during the design process. These new insights provided a deeper appreciation of the citizen scientist audience, and a cleaner conceptualization of the data segmentation task.

3.5.3 From Maze To Trees. As part of designing *Safe Passage*, we examined other games that cultivate a citizen scientist audience and draw upon the "wisdom of the crowd" by transforming hard scientific problems into entertaining experiences [23] [24] [[25].

For example, *FoldIt* (https://fold.it/portal/), *EyeWire* (https://wiki.eyewire.org/en/Main_Page), and *Zooniverse* (https://www.zooniverse.org/projects/) each provide clear tasks that let players make small, but meaningful contributions. *FoldIt* calls on players to fold individual proteins, while *EyeWire* asks players to identify neuron types, and trace interconnections from imagery. The collection of projects in *Zooniverse* are at the boundary between Mechanical Turk tasks and scientific discovery; players identify animals in photographs, classify galaxies, characterize bat calls, annotate war diaries, and find kelp forests in satellite imagery, among other tasks.

Table 6. Insights from original deployment of *Xylem* and their application in updating *Xylem* and designing the new Phase Two game

Original Version of *Xylem*	*Xylem* Update Plans	New Phase Two Game
The audience must be a good match for the level of difficulty of the puzzles (and vice versa).	Make changes to attract/retain more math and puzzle players	Hypothesis that audience can be increased or diversified by taking out the "mathyness" of the representation and replacing it with other visual representations.
Xylem is not a "casual game" due to the complexity of the puzzles.	Make changes to attract/retain more math and puzzle players	Still not a casual game, but takes the burden off of players to come up with complete solutions de novo and instead asks players to tweak proposed solutions.
The rate at which the complexity of puzzles increases is problematic for players.	Players rate the difficulty of puzzles, which will in turn be evaluated by the team. Pass very difficult or intractable puzzles to human experts	Intractable puzzles can be detected. Players can also rate difficulty of puzzles.
Initial human-centered design research during the early phases of design was used to help to tune original targeted audience to activity and context.	Midterm design research helped remove focus from "casual" players and switch focus to math and puzzle players.	Since we are at the beginning of this new game process, we can do some human-centered design research and also create and test mockups and prototypes.
A Web version of the game is preferable for serious puzzle players.	New version of *Xylem* to be ported to the web.	Web-only game.
The narrative must be compelling and motivational with a plausible connection to the player's activity.	Remove original narrative since engaged audience found it irrelevant and annoying.	The game is forthrightly in the "puzzle game" genre. It is not a narrative game.
Peer review of solutions can function as a motivator for play as well as an encouragement for social interaction among players.	Peer review system to be introduced in *Xylem*	Individual puzzle scores can introduce competition. Filters created for particular puzzles may be traded among players.
For social interaction, the game would benefit from inducements and affordances for collaboration as well as visible indicators of competition.	The Peer Review, Prestige System, and Bona Fides introduced to address this challenge.	Puzzle scores will serve as indicators that can encourage competition. Players can collaborate on puzzles. Team competitions will also be designed.

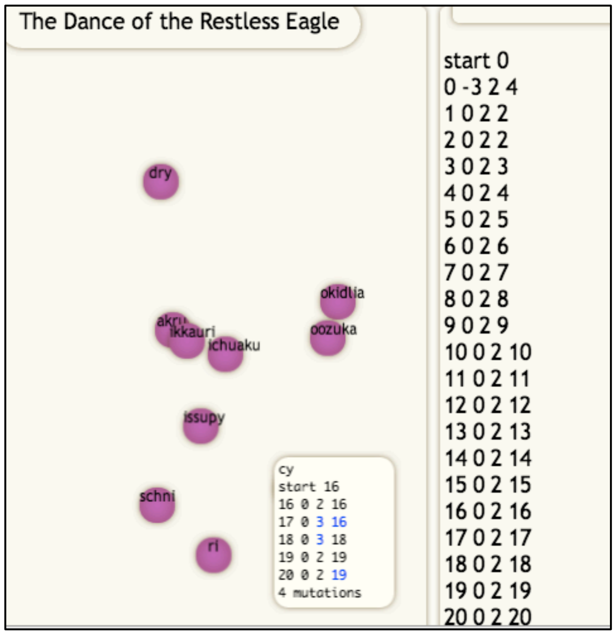
On closer examination, *FoldIt*, *EyeWire*, and *Zooniverse* all share multiple features:

- they solicit players through their interest in the science task,
- they present the science task without any narrative elements or story skin,
- they provide players with a large number of simple tasks, and
- the players can solve each task quickly, and feel good with each result.

We corroborated these observations by interviewing *Xylem* players, who requested simpler problems and more visibility into the science task (and complained about the absence of these features in *Xylem*). In addition, players indicated a strong, de-motivating influence from the need for a lengthy *Xylem* tutorial.

These interviews, and analyses of citizen-science games caused us to rethink the design of *Safe Passage*. In particular, *Safe Passage* had none of the features in the above list: it disguised the science task, it added an ecological narrative, each maze construction task appeared to be relatively heavy weight, and the two-stage model of game play imposed a delay between player action and reward. Moreover, the maze-construction task was sufficiently complex to require a lengthy tutorial. As a result, we concluded that *Safe Passage*, as designed, would badly misfire, and that we needed a simpler and more direct approach to a data-segmentation game.

Our core idea was to doff the narrative elements and phrase the game purely as a classification task with the goal of distinguishing good program states from bad. In principle, this approach would offer a cleaner design, and a simpler game mechanic that would enable rapid game play. That said, we began to explore visual concept models for the classification task, as a means of refining the game design.



```
cy
start 16
16 0 2 16
17 0 3 16
18 0 3 18
19 0 2 19
20 0 2 19
4 mutations
```

```
start 0
0 -3 2 4
1 0 2 2
2 0 2 2
3 0 2 3
4 0 2 4
5 0 2 5
6 0 2 6
7 0 2 7
8 0 2 8
9 0 2 9
10 0 2 10
11 0 2 11
12 0 2 12
13 0 2 13
14 0 2 14
15 0 2 15
16 0 2 16
17 0 2 17
18 0 2 18
19 0 2 19
20 0 2 20
```

Figure 26a. Dance of the Restless Eagle

The following sequence of sketches (Figures 26a/b/c) chronicles the design evolution during this period. Each sketch carried a code name, e.g., *The Dance of the Restless Eagle*.

Figure 26a expresses program states as labeled circles (vs fish or shells in *Safe Passage*), and displays the vector of variable values corresponding to each program state on the right.

In this example, each program state contains values for four numeric variables collected at successive iterations of a loop. These states only represent good data that is produced by the loop, as the depiction of bad states was not yet clear.

Figure 26b, a month later, adds visual elements for good and bad program states, and reduces the display of variable values to selected program states (the white box). This sketch also groups states into sets characterized by a given filter, where each filter corresponds to a (possibly complex) Boolean predicate. We imagined that players would generate these sets by applying filters from a predefined list (shown in blue), but the mechanic for applying filters was not quite clear at this stage of the design.

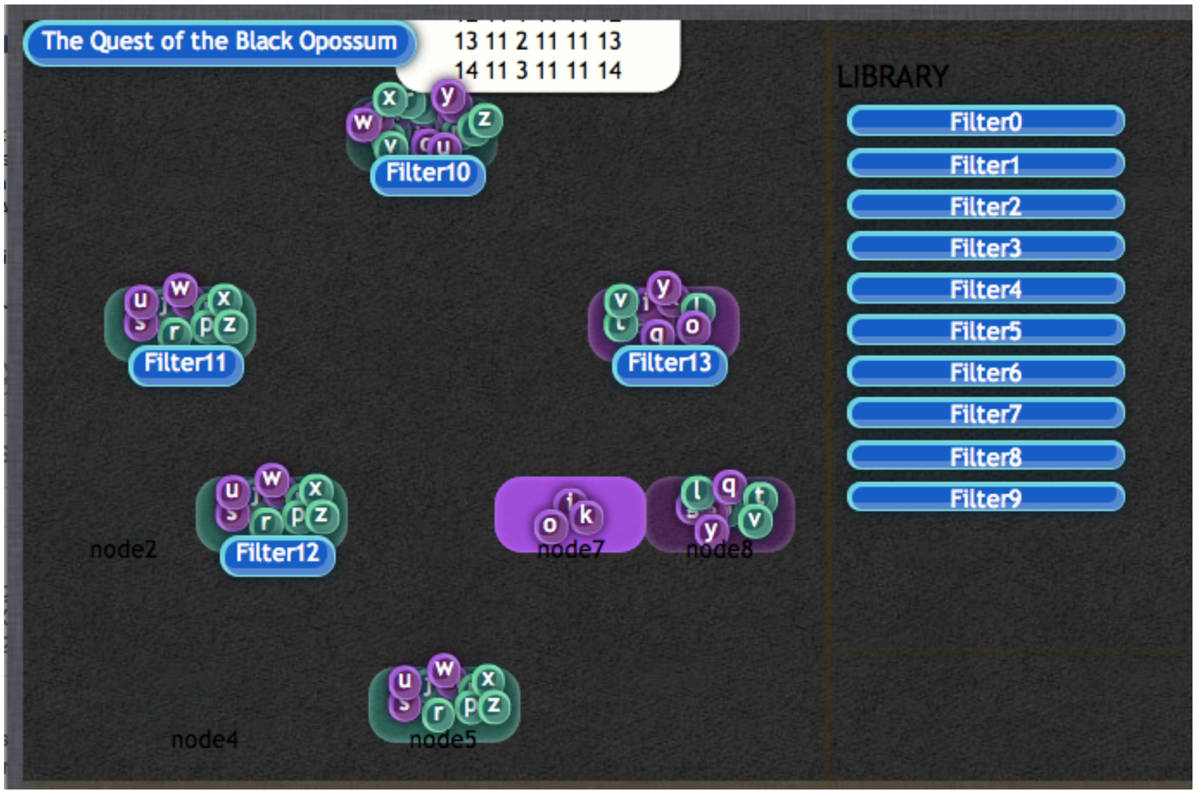


Figure 26b. The Quest of the Black Opossum

Figure 26c, another month later, clarifies the remaining elements of the classification model. It depicts a binary tree, where each branch represents the application of a filter that segments the program states present in the parent node into two sets, one containing states that pass the filter and one for states that do not. The values of program variables are largely absent in this sketch, while the filters are organized into a matrix of roll-over buttons that provide some indication of their score (larger and brighter is locally better).

In this design concept, players act by selecting a node from the tree, and clicking on a filter. They perform the science task by repeating that sequence.

As a whole, this mechanic is quite simple, resulting in rapid game play, and a minimal requirement for any tutorial. This design sketch contains most of the technical elements that were finally present in the next CHEKOFV game. While many art and user-interface elements continued to evolve, the key remaining technical features concerned reward structure, and constraints for managing screen real estate. Both of those proved important to the scientific value of the results produced through game play.

Additional features, such as the reward screen shown in Figure 27, were subsequently added in the evolving game design that became *Binary Fission.*

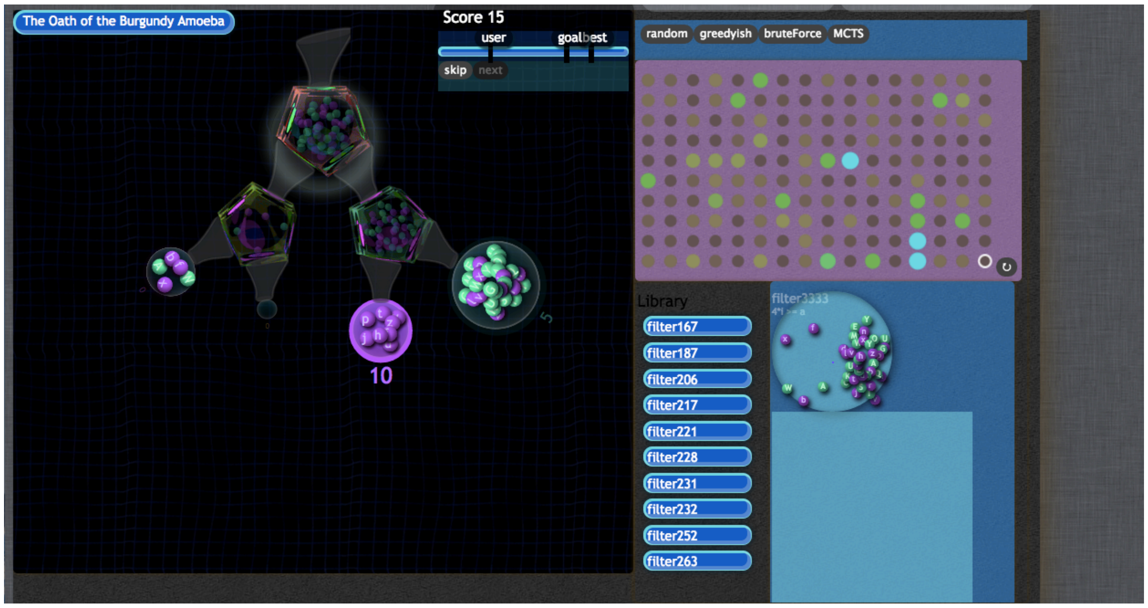
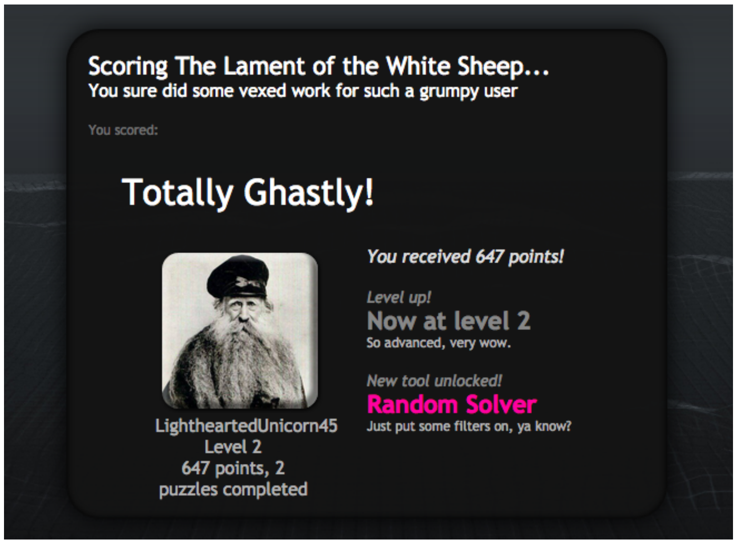


Figure 26c. The Oath of the Burgundy Amoeba



Scoring The Lament of the White Sheep...
You sure did some vexed work for such a grumpy user

You scored:

Totally Ghastly!

You received 647 points!

Level up!
Now at level 2
So advanced, very wow.

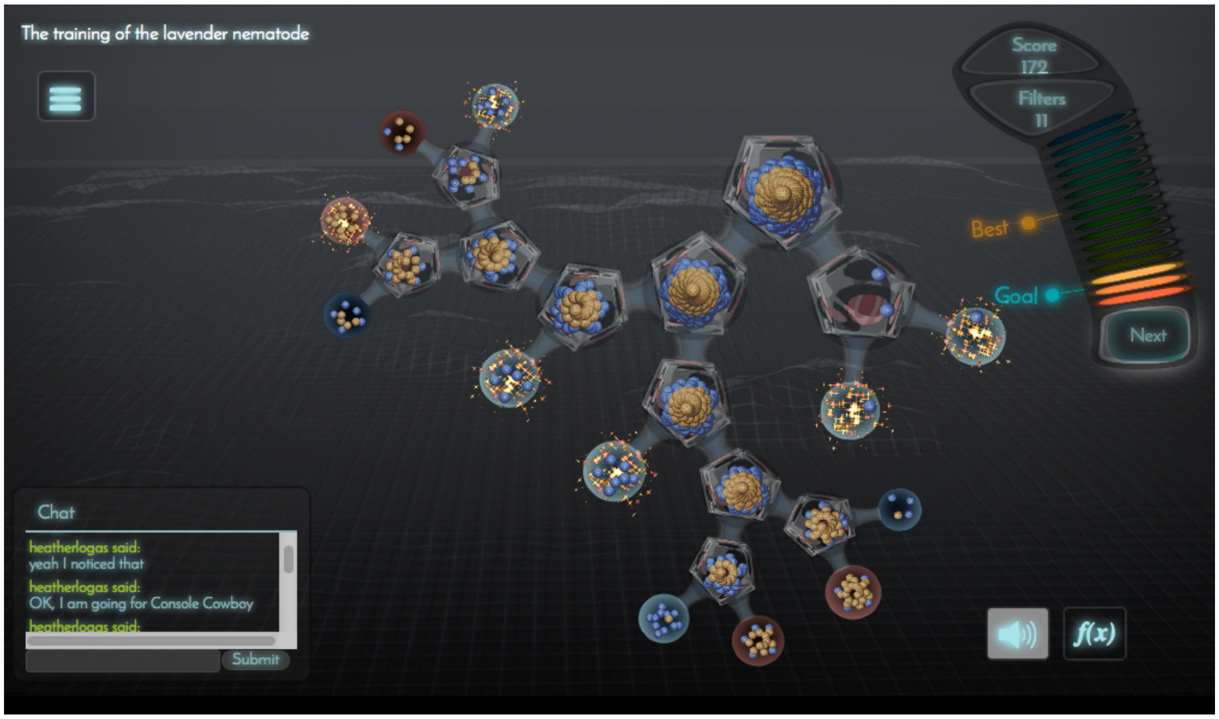
New tool unlocked!
Random Solver
Just put some filters on, ya know?

LightheartedUnicorn45
Level 2
647 points, 2
puzzles completed

Figure 27. Motivating players by acknowledging their achievements

3.5.4 *Binary Fission.* *Binary Fission* employs a classification metaphor for finding invariants. Although it applies to several forms of invariant finding tasks, our deployed version specifically addresses precondition discovery. At the technical level, the game inputs a program annotated with postconditions, a set of predicates relating program variables, and two sets of initial program states (each state is a vector of variable values), where ``good" states satisfy the assertions, and ``bad" states violate those assertions on program execution. Each *Binary Fission* player employs the available predicates to find a classification tree that separates good states from bad. This tree defines a logical formula representing a likely invariant.

At the game level, *Binary Fission* hides the nature of the program, states, and predicates from the player. Instead, it presents players with a set of gold and blue nodes (representing good and bad states, internally), mixed together inside a container. The player's goal is to separate the gold from the blue using a set of filters (corresponding internally to predicates), which are capable of partitioning the states. Different filters create different splits, and the player's job is to decide which filters to apply, and in what order. The recursive application of filters leads to the creation of a binary tree, as shown in Figure 28. Here, the conjunction of filters leading to a leaf node characterize the states in that leaf, and the disjunct of expressions describing each leaf constitutes the classification function defined by the entire tree.



heatherlogas said:
yeah I noticed that
heatherlogas said:
OK, I am going for Console Cowboy
heatherlogas said:

Figure 28. The *Binary Fission* player interface

Playing Binary Fission produces classification trees. Players act by selecting a node from the tree, and clicking on a filter, which bifurcates the selected node into two child nodes containing the states that pass, or fail to pass the filter. The game offers the player several hundred filters to choose from. The are presented in a rollover format, with visual feedback illustrating each filter's ability to separate good from bad states, together with its impact on game score. This format allows players to rapidly search the space of possible filters, and generate many classification trees. This design is intended to provide citizen scientists with an undisguised science task, and an easy to use interface that facilitates rapid progress and many incremental achievements.

Each classification tree produced through *Binary Fission* is typically partial: some leaf nodes only contain good states, some only contain bad states, while others contain a mixture. In addition, the solutions are idiosyncratic, as the players generally employ different subsets of filters during game play. As a result, the game software combines descriptions of pure good nodes and pure bad nodes across solutions to obtain a consensus view of the likely invariant.

3.6 Abstract interpretation, invariant learning, and crowd-sourcing

Binary Fission and *Xylem* form part of the suite of tools and techniques for invariant learning in our overall system. As shown in Figure 29, they join *Daikon* and decision-tree learning as invariant learning resources for CHEKOFV; other capabilities supported by the system are the Artisan&Crafters interface for paid workers, and the published CHEKOFV Robot API, for use by external automated-tool builders.

Figure 29 illustrates the iterative flow in CHEKOFV that supports the learning of likely invariants to assist abstract interpretation, which is based on *Frama-C Fusy*, *Value*, and other related plug-ins. The process takes a given terminating C program as input and returns either a proof of correctness or a copy of the input program annotated with the learned invariants and a set of assertions that could not be verified.

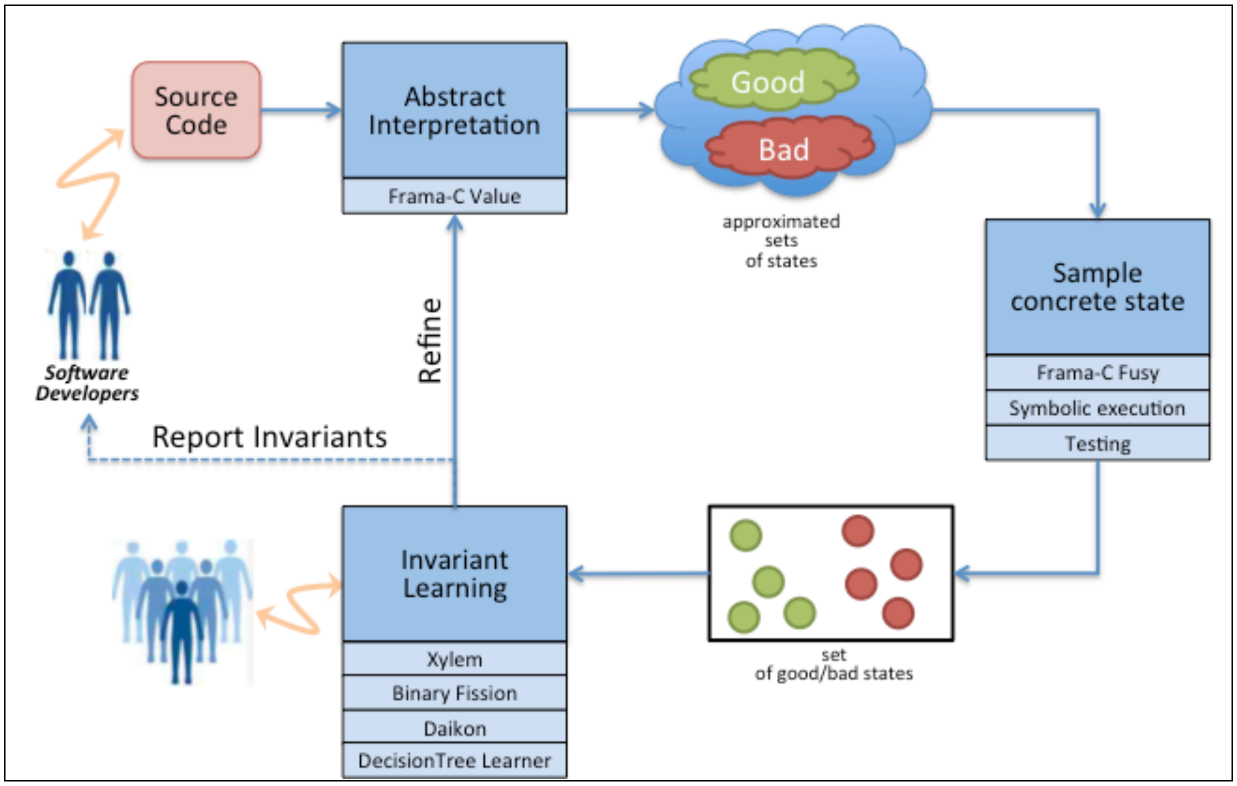


Figure 29. Abstract interpretation and invariant learning in CHEKOFV

The procedure for program analysis consists of the following steps:

1. *Initialize:* At every program point, initialize the likely invariant to true, the sets of good states and bad states to the empty set, and go to step 2.
2. *Update1:* Update the likely invariant at each program point using the abstract interpretation provided by *Frama-C*. Terminate with success if all assertions are verified. If the likely invariants are left unchanged, go to step 5.
3. *Update2:* Find new good states that lie outside the current likely invariant, and new bad states that lie inside the current likely invariant. When found, add them to sets of good and bad states at each program point. Else, go to step 5.
4. *Update3:* Use the current set of good and bad states to learn an invariant, using machine learning and crowd sourcing. Update the likely invariant at each program point using the newly learned likely invariants. If we fail to separate good and bad states go to 5, otherwise go to set 2.
5. *Terminate:* End, with likely invariants as hints for the verification engineer.

Here are the different pieces of the above procedure, as implemented in CHEKOFV.

3.7 *Frama-C* Value plug-in

CHEKOFV abstract interpretation is undertaken using the *Frama-C Value* component. This plug-in computes, at each program point, an abstract state that over-approximates the set of all possible states the program may be in at that point. The abstract state is a mapping from every memory location to the set of possible values that this location may currently have.

If the value is an integer, possible values are represented using an interval and a modulo as soon as the number of such values becomes too large (small sets are represented in an exact way). If the value is a floating point, only an interval is used.

Pointers in *Value* are represented using an interval per memory region where the pointer may point. *Frama-C* generates a warning if it cannot prove that the execution of an (implicit) assertion always succeeds from the current abstract state. If *Frama-C* does not generate any warnings, then the program is provably safe and our analysis terminates.

If *Frama-C* fails to prove that the given program is safe, the program either has a genuine error, or some of the abstract states were too imprecise to prove the program's safety. To refine this result, CHEKOFV try to learn likely invariants for each program point.

3.8 Sample concrete states

For a given program point in the input program, *Frama-C* computes a corresponding abstract state. This abstract state, as depicted in Figure 30, contains a subset of good states and bad states. Good states are program states from which the program terminates normally. Bad states are (possibly unreachable) program states, which lead to an assertion violation. Further, the abstract state may contain states that are not reachable but also do not violate any assertion and states that are reachable but lead to non-termination (we do not handle non-termination). CHEKOFV now tries to learn an invariant for this (abstract) program point that excludes all bad states and preserves all good states.

Note that, if the program is actually unsafe, such an invariant cannot be established because there exists a reachable bad state starting from this program point. That is, these invariants (when violated) can help the verification engineer to trace a safety property violation back to its origin.

Unfortunately, the set of good and bad states cannot be computed automatically (otherwise we would not need abstract states), so CHEKOFV can only approximate the invariant that it is looking for. To that end, CHEKOFV uses dynamic execution where available (i.e., test cases) or symbolic execution to sample good and bad states. As sampling the good and bad states is only an under-approximation, the likely invariants that we learn may be too strong or too weak. Hence, we may need several passes through the program until we find a suitable likely invariant.

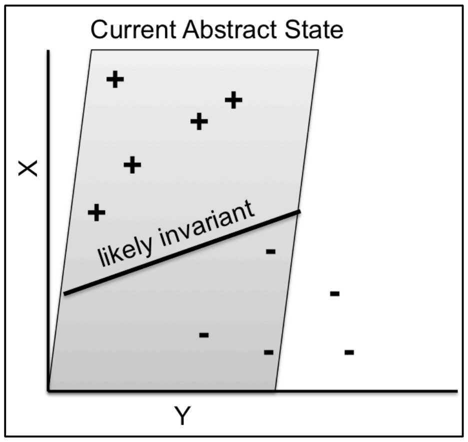


Figure 30. Example of an abstract state

3.8.1 Collecting states with unit testing. The most pragmatic way to collect program states is the run the program analysis on concrete inputs and monitor the programs state during the execution. This can be done either if the application under analysis comes with a set of test cases, or by using randomized test input generation on isolated units. To that end, we implemented an application similar to Randoop [26] that automatically generates test cases for each method in a file. For each method, we generated several sets of test inputs, which are assignments to all parameters that the procedure expects and to the global variables that may be used by this procedure. As test oracle (i.e., to decide whether the test succeeds or fails), we used a simple crash oracle; if the execution of the procedure on the generated input raises a segmentation fault or violates a (implicit) run-time assertion, we marked the test as failed.

For each test (either provided or generated), we ran the application under analysis with a debugger (gdb) attached. At each procedure entry, procedure return, and loop entry, we paused the programs execution using the debugger and record the current value of all program variables and stored them in a file. We automated this process using the Debuggers Python interface. If the test case failed (either due to exceptional termination, or because the test oracle marked it as failed), we added all collected states to the set of bad states. If the test succeeds, we add all states to the set of good states.

As file format for the recorded states, we used the dtrace-format that was introduced by the *Daikon* tool. Using this file format immediately allowed us to generate likely invariants using *Daikon* and also made it easy to interface to other machine-learning-based invariant-discovery tools like *DTInv* [14] or MCMC [27].

3.8.2 Collecting states with symbolic execution. For smaller the examples in our experiments, such as TCAS (See Appendix 5) or certain modules of BIND, this dynamic approach for collecting states worked well and collected large quantities of good and bad states in a very reasonable time. For other applications, however, this approach did not work well or could not be applied at all. For example, for parts of BIND, the available regression tests only executed small parts of the system, and randomly generating inputs for an application that modifies the file system and accesses the network is not feasible. For the autopilot software in the *Paparazzi* benchmark, we could not apply this approach at all, because the software only runs on ARM architectures and thus, it is not possible to attach a debugger in the same way as for other applications. For TCAS and other smaller benchmarks, however, testing turned out to be an efficient way to sample good and bad states as we will discuss later.

3.9 *Fusy* plug-in for *Frama-C*

For those benchmarks where sampling states using testing was not feasible, we developed a symbolic execution. The symbolic execution was developed as a plugin for *Frama-C* called *Fusy*. To sample bad states, *Fusy* checks if an error state is reachable from any state in the abstract domain of the current program point. That is, it turns the current abstract state, computed by the abstract interpretation, into a precondition (or an assume statement) for the symbolic execution. For each variable v with an abstract domain $v \in [min, max]$, *Fusy* then adds a conjunct $min \leq v \leq max$ to the precondition. If *Fusy* finds a reachable error state under this precondition, this state is added to the set of bad states. If the program point being analyzed is the program entry, or if we know that our precondition only describes reachable states, we have found a genuine error.

However, because CHEKOFV may insert a too strong invariant as intermediate result, the symbolic execution of *Fusy* may fail because the set of possible states to start from is, for example, empty. To avoid this problem, *Fusy* also checks if there exists a state outside the current abstract domain from which an execution terminates normally. Here, *Fusy* proceeds in a similar way as for the bad states but it computes a precondition for the complement of the current abstract state. This step is important to prevent the machine learning from producing overly strong likely invariants.

Fusy can also identify good states by symbolically executing the program from a given location until the end. Because it only has to find one symbolic execution that terminates normally, this is usually not prohibitively expensive, even for large programs. The bottleneck for this step was the handling of loops. If symbolic execution had to unroll a loop that cannot be exited after a few iterations, *Fusy* often got stuck if no invariant for the loop was present. To avoid this problem, we implemented several optimizations in *Fusy* and in the overall CHEKOFV loop to handle loops with priority.

3.9.1 Extensions to symbolic execution. *Fusy* first searched the different loops of the program or of a particular function. It attributed a unique number to each of them. This number had then been used by all the components of the CHEKOFV architecture to refer to identify each loop.

```
int fact(int x) {
 if (x < 0) return -1;
 if (x <= 1) return 1;
 return x * fact(x - 1);
}
int main(void) {
 int x = 5; int y = 10;
 int z = fact(100);
// removed statement
 while (y > 0) {
  x++;  y--;
 } return 0;
}
```

Figure 31. Example program to illustrate *Fusy*'s slicing functionality

For each loop, *Fusy* first sliced the program to preserve the loop's behavior by removing all statements that were not required to preserve the behavior. Consider for instance the C program in Figure 31. The slicing step generates a new program that does not include the marked statement because the variable *z* is not necessary to preserve the loop's behavior. That is, the slicing generates smaller programs in terms of number of statements and number of variables, which makes it more admissible to the symbolic execution in *Fusy*.

To ensure the correctness of the slicer (that is, not to remove useful statement), *Fusy* must resolve pointers: for instance, if 2 pointers *p* and *q* are aliased and hence point to the same memory location, and if **p* is used in the loop under analysis, statements modifying **q* must not be removed.

For such pointer resolutions, the *Frama-C* slicer used the abstract states computed earlier by *Frama-C*'s value analysis. Value analysis computed an over-approximation of all the possible values of each variable (including pointers) of the program at each program point. All these approximations are stored in an internal table and any *Frama-C* plug-in (e.g., the slicer) can then ask for an approximation of the possible values of a particular variable at a particular program point.

To generate sequences of values for the variables of each loop, *Fusy* performed symbolic execution on each previously generated program by running Value Analysis. However, it might be required to set some Value Analysis' specific options to be precise enough.

During this execution, *Fusy* modified the standard behavior of Value Analysis to track and register each value of the variables used in a loop at the point of interest (before entering the loop and after each loop iteration), whenever these values was precise enough.

When *Fusy* got the expected length of the sequence of values, it stopped the symbolic execution to run faster. That is, the states computed by the symbolic execution only represent a fixed number of executions of the loop body. While this may not be sufficient to find a proper loop invariant that holds in a larger context, this step helps in practice to facilitate the generation of good and bad states needed for the crowd sourcing.

3.10 Invariant learning

Once CHEKOFV has collected sets of good and bad states via testing or symbolic execution, it starts looking for likely invariants. Finding such likely invariant can be seen as a binary-classification problem in machine learning. CHEKOFV is looking for an approximation of a function that labels all good states as good and all bad states as bad at a given program point. The connection between invariant generation and classification has been explored in many recent works. While one of the key assumptions of CHEKOFV is that crowd sourcing can outperform machine learning in this domain, we still applied machine learning to get some initial candidate invariants and iterate the loop from Section 0 to collect additional good and bad states. To that end, we used two machine-learning tools: *Daikon* [5] and *DTInv* [14].

Daikon takes a set of program states as input, and infers a logic formula that holds true for each state in the given set. We applied *Daikon* separately to the set of good states and to the set of bad states, for each program point for which we collected such states. As a result, we obtained an initial set of likely invariants.

We also used *DTInv*, which is a decision-tree learner that takes a set of good states and a set of bad states as input and generates an invariant that includes all good states and excludes all bad states. Because *DTInv* performs a classification task, the invariants generated turned out to be logically stronger than the invariants generated by *Daikon*, but they often over-fitted the desired solution.

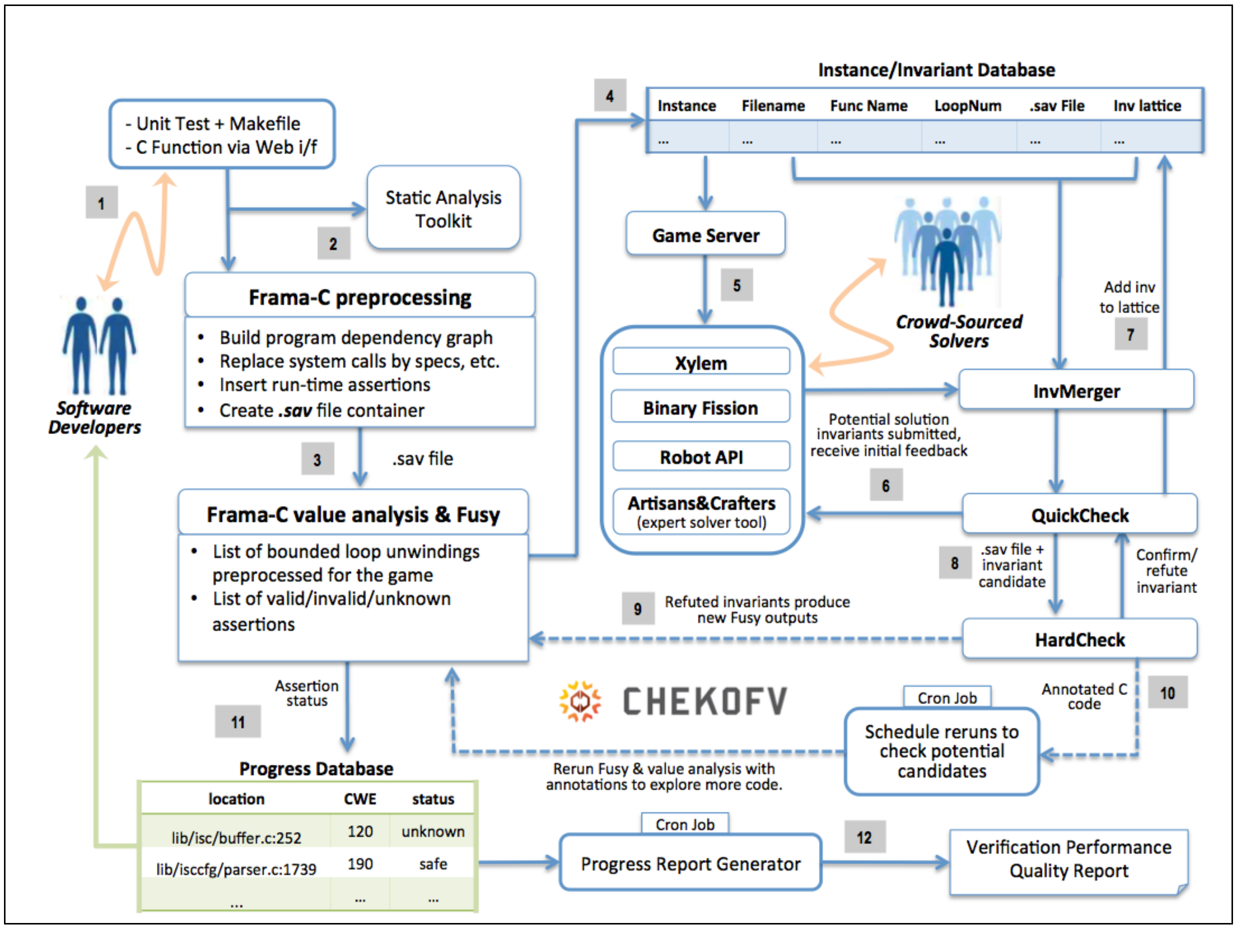
Later we discuss how we can crowd source this step using games. However, we emphasize that the CHEKOFV algorithm theoretically can be applied purely using machine learning without any crowd-sourced inputs. In this case, CHEKOFV will produce results that are very similar to the ones of *DTInv* and MCMC. Being able to run the system start to end without using the crowd sourcing was important to us because it allows us to efficiently evaluate and benchmark the benefit of using crowd sourcing.

3.11 Closing the loop

After performing the initial abstract interpretation, collecting good and bad states, and learning likely invariants using either machine learning or crowd sourcing, CHEKOFV has completed one iteration of its process. The task now is to feed the newly learned likely invariants back into the program under analysis.

To that end, we developed several plug-ins for *Frama-C* and other components. The key new elements in this part of CHEKOFV are Invemerger, Quickcheck, and Hardcheck. The role of these tools in the broader context of CHEKOFV is shown in Figure 32.

In Step 5 of this figure, the Game Server issues instance sets to *Xylem* or *Binary Fission* players. Solvers explore the patterns presented, and provide potential invariants. Artisans & Crafters (the expert solver web interface) and the Robot API may also be used to receive instance groups. The solutions generated are passed to Invmerger and Quickcheck (step 6), which adds them to the Hasse invariant lattice (step 7) to provide an initial ranking evaluation. Quickcheck returns this result to the solver, and also draws upon the instance database source code information to generate a .sav file for further invariant validation by Hardcheck (step 8). In step 9, Hardcheck confirms or refutes the invariant, and updates Quickcheck with its findings. Refuted invariants are transitioned back to *Frama-C* to generate more *Fusy* instances.



Instance/Invariant Database
Instance
Filename
Func Name
LoopNum
.sav File
Inv lattice
- Unit Test + Makefile
- C Function via Web i/f
1
2
Static Analysis Toolkit
Frama-C preprocessing
Build program dependency graph
Replace system calls by specs, etc.
Insert run-time assertions
Create .sav file container
Software Developers
3
.sav file
Frama-C value analysis & Fusy
List of bounded loop unwindings preprocessed for the game
List of valid/invalid/unknown assertions
4
Game Server
5
Xylem
Binary Fission
Robot API
Artisans&Crafters (expert solver tool)
Crowd-Sourced Solvers
Add inv to lattice
7
InvMerger
Potential solution invariants submitted, receive initial feedback
6
QuickCheck
8
.sav file + invariant candidate
Confirm/ refute invariant
9
Refuted invariants produce new Fusy outputs
HardCheck
CHEKOFV
Cron Job
Annotated C code
10
Schedule reruns to check potential candidates
Assertion status
11
Rerun Fusy & value analysis with annotations to explore more code.
Progress Database
location
CWE
status
lib/isc/buffer.c:252
120
unknown
lib/isccfg/parser.c:1739
190
safe
...
Cron Job
Progress Report Generator
12
Verification Performance Quality Report

Figure 32. Verification flow supporting multiple crowd-sourced tools

We now discuss each of these new components in more detail.

3.11.1 Invmerger. Invmerger is a *Frama-C* plug-in that inserts invariants learned by CHEKOFV in the C program at their right places. Invmerger takes a list of pairs of invariants and labels as input. The labels refer to *Frama-C*s internal representation of C programs to ensure that the location in the code is not affected when inserting or removing invariants from the code.

Further, Invmerger supports a special keyword *slideNum* which referred to the *Fusy* variable corresponding to the number of loop iterations. This keyword can be seen as a ghost variable that can be used by the games to allow players to build more expressive invariants. Invmerger can be easily extended to use other types of ghost variables.

3.11.2 Hardcheck. The goal of Hardcheck was to verify each candidate invariant that has been inserted in the C program by Invmerger. Hardcheck is primarily based on the *Frama-C* WP (Weakest Precondition) plug-in to verify the candidate invariants. If Hardcheck succeeded in proving that a candidate invariant is in fact an invariant, the invariant is left in the code. If Hardcheck failed to prove that a candidate invariant is an actual invariant, it tried to generate a counter example that witnesses why that the candidate invariant can be violated. Such a counter-example is a sequence of program states along a control-flow path from an entry point of the analyzed program to the location where the candidate invariant has been inserted, such that the candidate invariant is violated when reaching that location. That is, we can add this counter-example immediately to the sets of learned program states and remove the candidate invariant from the program.

The *Frama-C* plug-in WP is based on Dijskra's Weakest Precondition Calculus. For each program property to be verified, this analysis technique generates one or several so-called proof obligations (PO). If all POs are verified, then the corresponding program property is verified. These POs used to be verified by other means (usually either automatic theorem provers or proof assistants). In CHEKOFV, we relied on the automatic theorem prover *Alt-Ergo*.

Weakest precondition calculus is a modular-analysis technique. Thus WP analyzed a single C function *f* without the implementation of the others. However, if *f* calls another function *g*, WP requires a contract (i.e., specification) for *g* to reason about *f*. If the function *g* has no specification, it is not possible to prove anything interesting about the function *f*. Because writing such specifications by hand would contradict with the goals of CHEKOFV, we chose to inline every function call that WP needs to examine to generate the POs of all the loop invariants. This inlining step is done prior running the above-mentioned algorithm.

3.11.3 Generation of Candidate Counter-examples. Some automatic theorem provers, in particular Alt-Ergo, are able to generate a 'counter-model' when they conclude that a given proof obligation is not valid. This counter-model explains why the property is not satisfiable using uses the prover's internal logic constraints.

From this prover's counter-model, the plug-in 'Counter-example' generates a new function 'main' for the C program in order to express the logic constraints according to the input variables of the C program.

Running this C program from the new function 'main' should violate the unproved property. However, there is no guarantee that the prover's counter-model does violate the property and thus that the generated counter example does violate it. Indeed, it used to not be the case if the prover reaches its timeout before either validating or invalidating the property: here its returned counter-model is just its current state and there is little chance that it is a true counter-model. Thus Hardcheck tried to check that the generated counter example is a real one.

Hardcheck used *Frama-C Value* plug-in to validate candidate counter-examples. It ran the new program from the generated function 'main' and checked the validity status of the relevant candidate invariant. Because the input was very precise thanks to the plug-in Counter-example (the function 'main' initializes all required inputs), it should be possible to get precise analysis results as soon as *Value* is configured enough.

3.11.4 Quickcheck. Checking invariants with Hardcheck was expensive. The step required several calls to a theorem prover, which can take several seconds or even minutes. In particular due to the inlining of procedure calls, Hardcheck often took very long to provide an answer. Hence, checking each candidate invariant immediately with Hardcheck would have been prohibitively expensive. In particular, because it is to be expected that many players provide similar or even equivalent solutions to the same problem.

We also developed a tool called Quickcheck to perform a cheap analysis of the candidate invariants returned by players of the games to decide if Hardcheck should be employed to check the invariant, or if we already have a better solution. Quickcheck uses a partially ordered directed graph known as Hasse diagram to maintain a set of candidate invariants for each program point. See Figure 33.

Initially, the set contains only the trivial candidate invariants True and False. Because all invariants refer to the same program point, they are over the same set of program variables and thus they have an implicit partial ordering over logical implication. That is, given two candidate invariants A and B, they are equal in our partial ordering if A=>B and B=>A, if only one direction holds we know that one dominates the other, and if neither holds, they are unrelated.

In this figure, we can see that false is the source because false implies anything, and true is the sink because anything implies true. In this example, the candidate invariant C3 implies C1 but not vice versa, C5 implies C2, and neither of the others imply each other.

The Hasse diagram is used to prevent CHEKOFV from performing redundant invariant checks. If a new invariant is received, we first check its position in the Hasse diagram. This check is cheap because it only requires checking logical implications of candidate invariants.

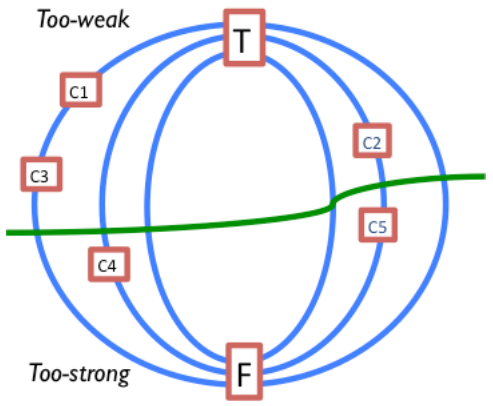


Figure 33: Example of a Hasse diagram for categorizing candidate invariants

Once the position of a new candidate invariant has been found, Quickcheck checks if this invariant is stronger (i.e., implied by) than known candidate invariants for which WP already has concluded that they are too strong to be program invariants, or if this invariant is weaker (i.e., implies) than candidate invariants that are known to be too weak. In both cases, Quickcheck concludes that checking this candidate invariant is not necessary. If Quickcheck establishes that the new candidate invariant is logically equivalent to a known candidate invariant, it also discards it. In any other case, Quickcheck adds the candidate invariant to the Hasse diagram and calls Hardcheck to establish if it is a true invariant, too strong, or too weak. Figure 34 depicts part of a Hasse diagram, with more detail about the relationships among individual candidate invariants.

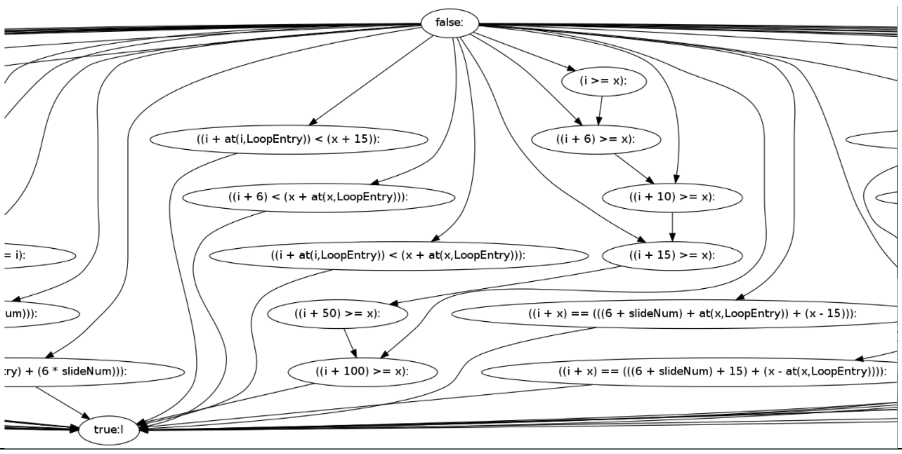

false:
(i >= x):
((i + at(i,LoopEntry)) < (x + 15)):
((i + 6) >= x):
((i + 6) < (x + at(x,LoopEntry))):
((i + 10) >= x):
= i):
((i + at(i,LoopEntry)) < (x + at(x,LoopEntry))):
((i + 15) >= x):
um))):
((i + 50) >= x):
((i + x) == (((6 + slideNum) + at(x,LoopEntry)) + (x - 15))):
:ry) + (6 * slideNum))):
((i + 100) >= x):
((i + x) == (((6 + slideNum) + 15) + (x - at(x,LoopEntry)))):
true:l

Figure 34. Hasse diagram of relationships among sample candidate invariants

Quickcheck was able to prevent a significant amount of redundant computation and also provided interesting insight on the type of candidate invariants provided by players. For debugging purposes, we also added a time lapse to replay how new candidate invariants are being analyzed by Quickcheck and how the Hasse diagram evolves over time.

3.12 Analysis Termination

The analysis terminates either upon system verification or when one of following failures occur:

– Failure to find new good and bad states: Symbolic execution can fail to find new states. This may happen because the problem of finding good and bad states is undecidable in general and very expensive in practice. In this case, we terminate with the last learned invariants as a hint for the verification engineer.

– Failure to classify good and bad states: For crowd sourcing this may happen because the games do not have enough players, or the needed invariant is not expressible with the tools offered by the game. The latter case is equivalent to the case where a machine learner fails due to the choice of the kernel functions. Assuming that the language of the game or the kernel function of the machine learner are strictly more expressive than the abstract domain of *Frama-C*, we can terminate reporting the last learned invariants.

– Failure to improve abstract domain with the learned invariants: This may happen because the language of the likely invariants is more expressive than what can be expressed in the abstract domain. In this case, we know that there are bad states that cannot be excluded in the current abstract domain and we can report a warning that the current abstract domain is not sufficient to verify the program.

3.13 *Sample* Plug-in

To collect data about the candidate invariants provided by players, the progress of the verification, and the performance of our tools, we developed several plug-ins and tools, the most important of which was *Sample*.

Sample is a *Frama-C* plug-in developed in the context of this project. Its function is to extract a concrete state from the results of a Value analysis. This concrete state can then be tested to determine if it is actually reachable or if it has been produced by an over-approximation. In the latter case, it may be possible to improve the analysis by adding specification statements to the code, which exclude this unreachable state from the analysis. That is, the plug-in does not output a concrete state but a *sample*.

For a given tuple of C expressions, a sample is a corresponding valuation of these expressions in a chosen control point of the program. These valuations can either be reachable values from the real program or unreachable, i.e. the result of some over-approximation during the analysis.

The plug-in is invoked through the command line. The user must supply a statement identifier where the sampling has to be done and the variables that have to be sampled. For reproducibility reasons, the seed used by the random number generator can also be provided. When several samples are needed, the user can supply a file in comma-separated-values (CSV) format of previous samples. These sample will be excluded from the possible outputs, and the plug-in will produce, when possible a new sample.

The CSV file must start with a line containing the variables sampled, separated with commas. In case this line is not exact, Sample will give the expected line. Each following line must have the same number of values (one for each variable) separated by commas. The plug-in will check that all these numbers are valid integers or floating-point numbers, based on the variable type.

We use the result of the *Value* plug-in execution. This result describes, for each variable of the program, a set of possible value. For integers, if this set is small enough, it is given explicitly, otherwise, it is given as an interval with modulo information. For floating points, it is always given as an interval. The plug-in can also compute such sets for expressions: it combines the information it has about each variable of the expression and deduces a set of values for the expression. We then sample a value in these sets in a way described in the next paragraph.

The plug-in integrates custom pseudo-random number generators to generate integers and floats. In general, the sampling of values doesn't follow a uniform distribution of number among the reachable values. There is only one exception. When the set of possible values for an expression is small enough, say less than ten, the value is selected uniformly between those values.

Otherwise, the value is randomly chosen in the following way. If the value can be either positive or negative, its sign is chosen with equal probability. Then, a positive or negative interval of possible value is built from this choice and a number is generated in this interval, non-uniformly once again. For instance, for integers, the integral base two logarithm (the number of significant bits) is chosen first. Then a number with this logarithm is uniformly picked.

3.14 Plug-ins for CWE progress metrics

We developed two additional *Frama-C* plug-ins to evaluate the progress of the verification effort in terms of the CWEs that were to be checked. *Frama-C* automatically inserts run-time assertion into the analyzed code for the CWEs 120, 134, 190, and 476. Other CWEs such as 250, 306, 434, 672, 732, 807, and 863 that depend on domain knowledge can be added by hand. The verification progress can now be measured by counting how many of these assertions can be verified and how many assertions are rendered unreachable by the generated invariants.

The first plug-in that was developed collects all warnings emitted by the *Value* plug-in of *Frama-C* and stores them in a database in a way that these warnings can be queried by category or by location in the source code. Before adding invariants to the code, most of these warnings will be false alarms that are generated because the value analysis lost precision while analyzing the program. As we add invariants to the program, the number of warnings is expected to decrease because the value analysis becomes more precise. The main goal of this plugin was to identify hotspots where the CHEKOFV failed to improve the precision of value analysis and take appropriate measures.

The second plug-in that we developed collects all statements in an analyzed program that are considered unreachable by value analysis. Similar to the previous plug-in, all unreachable statements are stored in a database. This plug-in was necessary to identify the impact of overly strong invariants. For example, an invariant that strongly constrains input variables may eliminate many value-analysis warnings by rendering parts of a program unreachable. To get an accurate view of the progress of the verification in CHEKOFV, we used this plug-in to weigh the number of verified assertions against the number unreachable statements.

The resulting metrics from an exploratory case study using the BIND source code are reported in Section 4.5. While these metrics features are not part of the final integrated system, they provides valuable insights for the development of CHEKOFV, and for spotting bugs during integration.

4. RESULTS AND DISCUSSION

To showcase the capabilities of the CHEKOFV system, we carried out two case studies to detect two known security vulnerabilities – the Heartbleed bug and a recent critical bug discovered in BIND. We detected these bugs under lab conditions. We already knew where the bug was located, and isolated the relevant program parts accordingly. After the case studies, we discuss the progress and problems we observed when applying CHEKOFV to the applications BIND and *Paparazzi*. Both are large-scale real world applications that are challenging to any form of program analysis.

This is followed by a summary of our work on state-space metrics, and a feasibility study of *Xylem* using *SV-COMP* verification benchmarks. Next, we report on insights gathered on the benefits of using crowd sourcing compared to existing machine-learning techniques, which is based on data collected from applying CHEKOFV tools to the standard verification benchmarks.

At the end of the section, we provide some player productivity data for *Binary Fission*.

4.1 Case Study 1: *OpenSSL* – Heartbleed Bug

We explain how CHEKOFV works by dissecting the well-known Heartbleed bug in *OpenSSL*. The code snippet that caused the bug is sketched in Figure 35. For space reasons, we omit a few lines that are not relevant to understanding the bug.

The bug is a missing bounds check in the heartbeat extension inside the transport layer security-protocol implementation. A heartbeat essentially establishes whether another machine is still alive by sending a message containing a string (called payload) and expecting to receive that exact same message in response. The bug is that, although the message also contains the size of this payload, the receiver does not check if this size is correct. Therefore, an attacker can read arbitrary memory by sending a message that declares a payload size that is greater than the actual message.

Figure 35 shows the part of the code that processes a heartbeat message. On line 3, the pointer *p* is set to point to the beginning of the message. Then, on line 8, the message type is read, and on line 9, the size of the payload is read through the macro *n2s* which reads two bytes from *p* and puts them into payload. However, because the whole incoming message might be controlled by an attacker, there is no guarantee that this payload really correspond to its actual length and there is no check in the code. Payload might be as much as $2^{16} - 1 = 65535$. Line 10 then puts the heartbeat data into *pl*.

In line 15, a buffer is allocated and its size is actually as much as $1 + 2 + 65535 + 16 = 65554$. Then lines 18 and 19 fill the first bytes of the buffer with the type and the size of the response message. Finally, line 21 attempts to copy the heartbeat data from the incoming message to the response through a call to *memcpy*. Because the payload can be longer than the actual size of *pl*, nearby memory data (included potential confidential user data) may be inadvertently copied.

Frama-C can detect this bug. It adds an implicit assertion just before the *memcpy* that *bp* and *pl* must be at least of size payload. Because it cannot prove this property, it warns about a potential bug. However, because this is not the only warning emitted by *Frama-C*, chances are it will go unnoticed.

```
1  int dtls1_process_heartbeat(SSL *s)
2  {
3    unsigned char *p = &s->s3->rrec.data[0], *pl;
4    unsigned short hbtype;
5    unsigned int payload; // message size
6    unsigned int padding = 16;
7
8    hbtype = *p++;
9    n2s(p, payload); // read message size from input
10   pl = p;
11
12   if (hbtype == TLS1_HB_REQUEST)
13   {
14     unsigned char *buffer, *bp;
15     buffer = OPENSSL_malloc(1 + 2 + payload +
          padding);
16     bp = buffer;
17
18     *bp++ = TLS1_HB_RESPONSE;
19     s2n(payload, bp);
20
21     memcpy(bp, pl, payload);
```

Figure 35. Faulty code that processes a heartbeat message in *OpenSSL*.

Let us now see how our approach can make it easier for a human analyst who is using *Frama-C* to notice this bug. First, even if it does not appear in Figure 35, *p* is fixed and equal to *SSL3 RT HEADER LENGTH*. Starting with the abstract state computed by value analysis at line 10, the abstract state looks roughly as follows:

hbtype $\in$ [0, 255] (a one-byte positive integer)

payload $\in$ [0, $2^{16} - 1$] (a two-byte positive integer)

size_p = SSL3 RT HEADER LENGTH − 3

padding = 16

For readability, we use this abbreviated version of the abstract state computed by *Frama-C*. The actual abstract state would contain a lot more information about the input parameter *s*, about the value of *p, pl*, and about other global variables. The important thing in this abstract state is that payload can be an arbitrary two-byte unsigned integer, while the size of the allocated memory for pointer *p* is fixed and equal to *SSL3 RT HEADER LENGTH − 3*.

Because none of the variables in the abstract state depicted above are modified by any statement until line 21, these variables will have the same intervals. Hence, the implicit assertion that payload $\leq \text{size}_p$ which is required by *memcpy* does not hold.

Now, we use symbolic execution to refine our abstract state just before line 10. We pick this program point because it assigns a value from an unknown source to a variable. CHEKOFV refines all states where we receive unknown inputs (user input, files, network, and so forth), or we lost information due to widening (e.g., after loops).

First, we collect bad states that lead to assertion violations. To that end, we construct a precondition that ensures that the symbolic execution may only pick initial values that are in our current abstract state. The symbolic execution will then search for concrete states from which the assertion can be violated. Next, we need to collect good states from which the assertion is not violated. We can either use the same symbolic execution approach that we used to collect bad states or fall back on data from previously recorded test cases, if available.

Figure 36 shows the distribution of the collected data points for payload and $size_p$. As discussed above, all good states (depicted by a plus sign) are states where $size_p$ is greater or equal to payload. All bad states (shown as a minus sign) are states where payload is greater than $size_p$. Using these data points, we can now employ our crowd-sourcing games (or a machine learner) to find a classifier (that is a likely invariant) that separates the good states from the bad states. The ideal classifier would be payload $\leq size_p$. However, let us assume that our symbolic execution picked extreme values and we get an over-fitted invariant $2 * \text{payload} \leq size_p$.

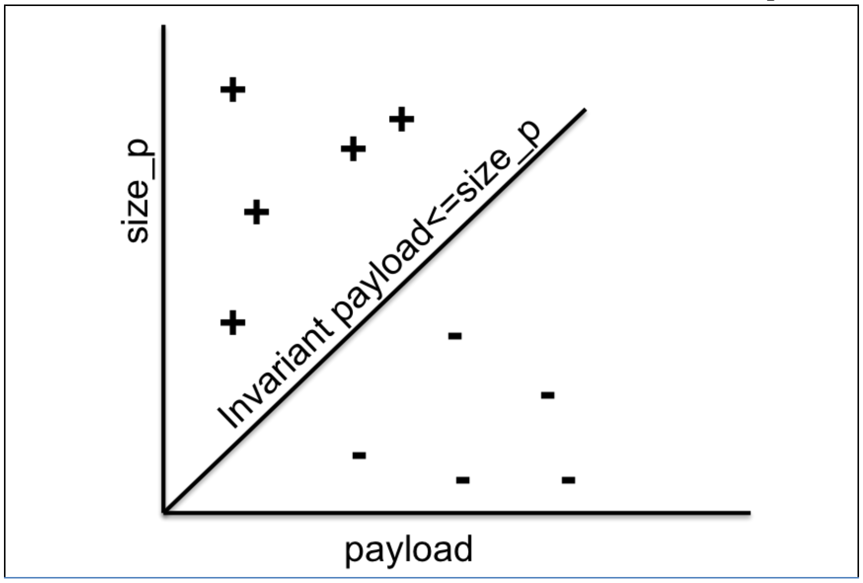


Figure 36: Distribution of data points for payload and $size_p$.

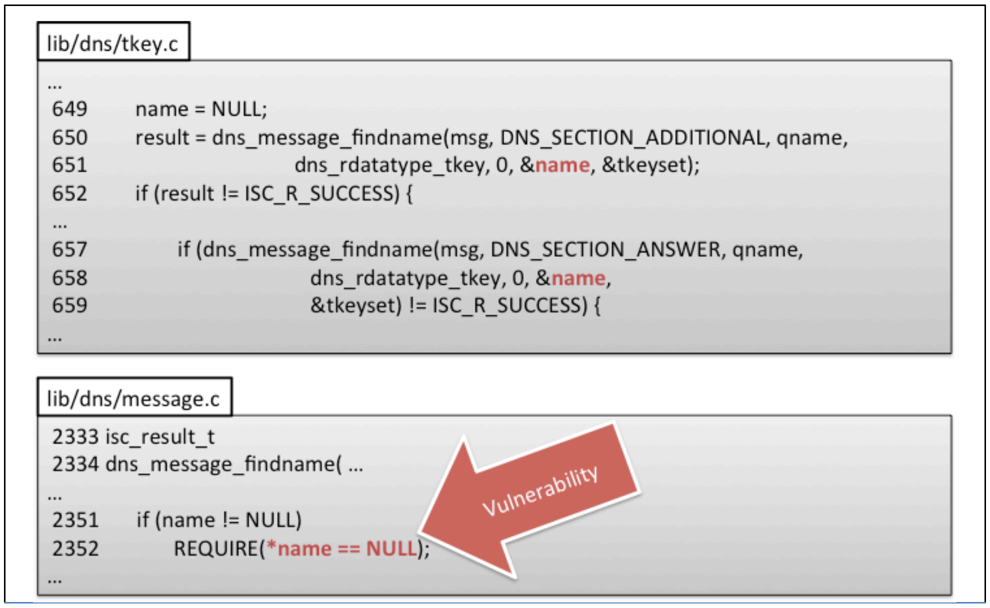
We merge the invariant $2 * \text{payload} \leq size_p$ into the program at line 10 and re-run our value analysis. The invariant refines the abstract state at line 10 such that payload is in the interval [0, $size_p$/2]. Hence, the assertion violation in line 21 is now gone and we know that we cannot find new bad states that violate this assertion. However, we still have to ensure that the inserted invariant did not throw away too many good states. Thus, we start our symbolic execution again, this time with the precondition that the invariant does not hold (i.e., $2 * \text{payload} > size_p$ and thus the abstract value of payload is [size p/2 + 1, $2^{16} - 1$]). This reveals new good states that ensure that we cannot find the same invariant again. This loop is repeated until we cannot find new good or bad states. We mark likely invariants where this is the case as potential solutions. However, we do not stop the crowd sourcing immediately because there might be several invariants that have this property.

Eventually, CHEKOFV finds the invariant payload ≤ size_p for line 10 which is sufficient to prove the assertion in line 21. We cannot actually prove that this is an invariant (in fact it is not an invariant because there is a bug). It is a likely invariant that helps the verification engineer when verifying the program.

4.2 Case Study 2: BIND – CVE-2015-5477

Our second case study tried to use CHEKOFV to find a recent critical bug discovered in BIND, which was reported July 28, 2015. Similar to the Heartbleed bug in the previous case study, this bug can be exploited by an attacker by sending manipulated packages to BIND. Even though the bug does not immediately leak confidential information as does Heartbleed, it can still be exploited to shut down BIND remotely, acting as a denial-of-service attack.

Figure 37 shows the code snippet from BIND that is relevant to the vulnerability. In the method *dns_tkey_processquery* in *tkey.c*, BIND handles a message received from the network. To that end, it uses the method *dns_message_findname* to extract information from the received message. One of the arguments to *dns_message_findname* is a piece of memory called *name* where the method can write its response.



lib/dns/tkey.c

```
...
649     name = NULL;
650     result = dns_message_findname(msg, DNS_SECTION_ADDITIONAL, qname,
651                        dns_rdatatype_tkey, 0, &name, &tkeyset);
652     if (result != ISC_R_SUCCESS) {
...
657         if (dns_message_findname(msg, DNS_SECTION_ANSWER, qname,
658                         dns_rdatatype_tkey, 0, &name,
659                         &tkeyset) != ISC_R_SUCCESS) {
...
```

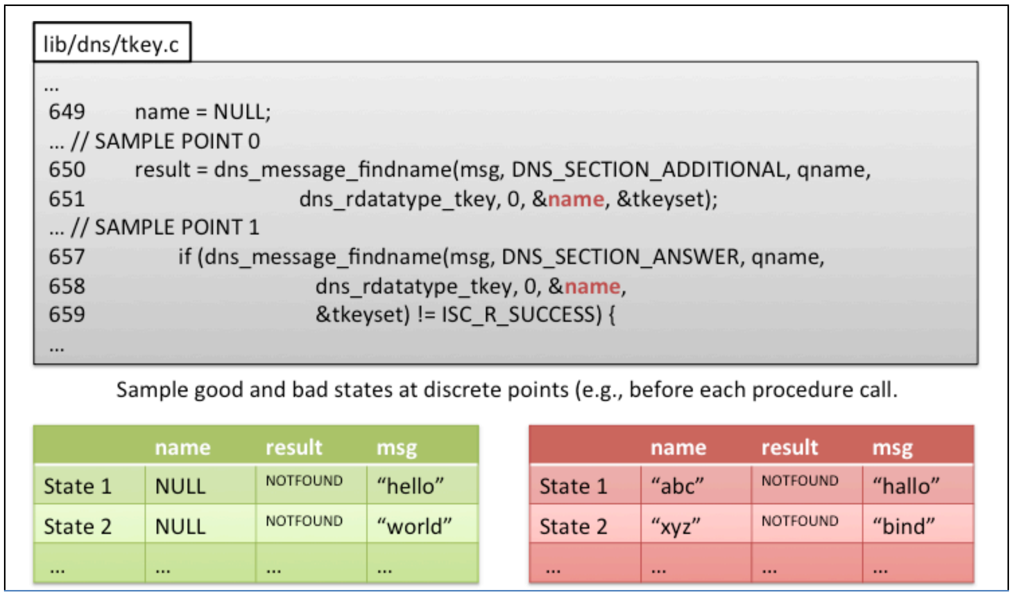
lib/dns/message.c

```
2333 isc_result_t
2334 dns_message_findname( ...
...
2351     if (name != NULL)
2352         REQUIRE(*name == NULL);
...
```

Figure 37. Code snippet illustrating the vulnerability in BIND

To ensure that *dns_message_findname* does not overwrite received data, it asserts that this variable *name* points to unused memory (see line 2352 in Figure 37). The problem is that *dns_message_findname* might be called twice in *tkey.c*: first in line 650 and then again in line 657, depending on the content of the received message. However, the variable *name* is not being reset after the first call and thus, the assertion may be violated by appropriate input.

Figure 38 shows how CHEKOFV sampled data in our case study. CHEKOFV treats the method *dns_tkey_processquery* as an entry point (because it is reachable from network input), and starts sampling good and bad states using symbolic execution (because we assume that there is no test case witnessing this bug, otherwise it would have been fixed earlier).


lib/dns/tkey.c

```
…
649       name = NULL;
… // SAMPLE POINT 0
650       result = dns_message_findname(msg, DNS_SECTION_ADDITIONAL, qname,
651                           dns_rdatatype_tkey, 0, &name, &tkeyset);
… // SAMPLE POINT 1
657            if (dns_message_findname(msg, DNS_SECTION_ANSWER, qname,
658                           dns_rdatatype_tkey, 0, &name,
659                           &tkeyset) != ISC_R_SUCCESS) {
…
```

Sample good and bad states at discrete points (e.g., before each procedure call.

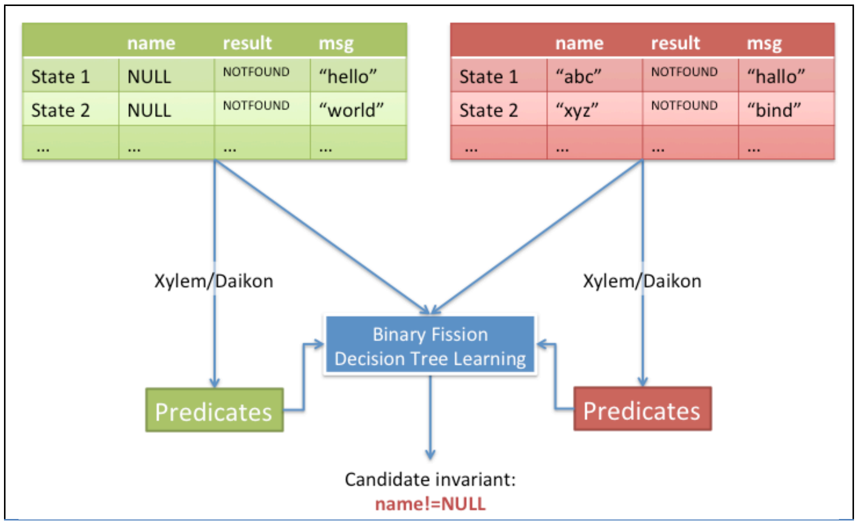
	name	result	msg
State 1	NULL	NOTFOUND	“hello”
State 2	NULL	NOTFOUND	“world”
…	…	…	…

	name	result	msg
State 1	“abc”	NOTFOUND	“hallo”
State 2	“xyz”	NOTFOUND	“bind”
…	…	…	…

Figure 38. Example of how CHEKOFV samples data for invariant learning.

As sample points, CHEKOFV uses the entry and exit point of *dns_tkey_processquery* as well as one point before and after each method call inside *dns_tkey_processquery* (including the points before and after the calls to *dns_message_findname*). Figure 38 shows a simplified version of what the collected data may look like for the sample point before line 657. The value result must be *NOTFOUND*, otherwise, the line would not be reachable. The value of *msg* can be arbitrary (for both good and bad states), and the value of *name* must be null for all good states, and can be anything other than null for all bad states.

Figure 39 shows how CHEKOFV learned an invariant from the sampled data. The good and the bad states separately are processed by *Daikon* and passed to *Xylem* to find predicates summarizing these sets. Then, the good and bad states, together with the set of learned predicates, are passed into a decision-tree learner and *Binary Fission* to find the likely invariants.



	name	result	msg
State 1	NULL	NOTFOUND	“hello”
State 2	NULL	NOTFOUND	“world”
…	…	…	…

	name	result	msg
State 1	“abc”	NOTFOUND	“hallo”
State 2	“xyz”	NOTFOUND	“bind”
…	…	…	…

Figure 39. Learning invariants from sampled data.

For our example, CHEKOFV easily identified the invariant that name must be different from null before line 657. Because this invariant does not hold, symbolic execution can find a counter example that exposes the vulnerability.

4.3 BIND Analysis

During the project, it became clear that analysis of the entire BIND source code base was essentially out of reach for the current generation of *Frama-C* analyzers. This challenge code base also posed significant problems for the game-instance generation and results-integration parts of CHEKOFV. However, there were several valuable insights in this work, outlined in the next section, which were drawn from our work with certain subsets of the code. Subsequent sections discuss a number of the core challenges encountered with the BIND code base.

4.3.1 Research Contributions from BIND Analysis. Key findings that emerged from our work with certain subsets of the BIND codebase include:

- **Unit Tests**: These are a simplified setting, as they are supposed to have all values deterministic. This is not the case in practice, e.g., because of imprecision of our modeling of functions in the C library or of the testing framework, or because some of the loops could not be completely unrolled without a prohibitive cost. Despite these problems with the simplified settings of unit tests, we determined the correct process for replacing the custom allocator by standard *malloc/free* calls in this setting. We were able to improve the completeness of our C library stub, as well as its accuracy.
- **Small Binaries**: The main binary of the BIND code base is the *named* binary that implements the name server. However, BIND also includes a number of other "small binaries", for example a series of command-line utilities. Using the work done on the unit tests, we were able to successfully analyze many of them (*genrandom*, *arpaname*, *nsec3hash*,...). Unfortunately, these binaries did not contain an adequate selection of comprehensive loops that could be used for game-level instance generation.
- **Modular Analysis of BIND Code**: Instead of starting from *main*, we ran *Value* starting from all the functions of BIND that contained loops, in a restricted setting where arguments to functions could not alias each other. We stopped the execution after a two-minute timeout. About half of the functions were successfully analyzed using this setting.

Additional new tools and techniques developed in this work are described in Sections 5 and 6.

4.3.2 Large code base. The size of the source code for the main BIND binary (*named*) is very large, which implies that a whole-program analysis takes a long time. This is especially the case for value analysis, where function calls are handled using in-lining. The result was a significant difficulty in debugging the analysis; as the analysis did not terminate, it was difficult to understand what went wrong. In particular, the loss of precision when analyzing *named* appeared after several minutes, which made the cycle modification/test lengthy.

4.3.3 Precision loss. As a complex project, BIND defines a lot of data structures. Some are implemented using tagged or discriminating union, i.e. a given region of memory may have different, unaligned contents, depending on the value of one field, the tag. It is important to keep the different cases separated according to the value of the tag; else the possible values of the fields in overlapping memory locations are misinterpreted, and the precision loss becomes large (e.g. when mixing pointers and integers).

If the path-sensitivity of *Value* allows it to separate these cases inside of a function, the fact that all states are merged on each function return means that we could not avoid merging unrelated values, and thus, this precision loss.

A solution would have been to add relational invariants describing that the value of some fields depend on the value of the tag. But these are difficult to add to the current memory model of *Value*, for which the fact that abstract values are non-relational is a fundamental assumption (the memory model associates to each memory location a set of value, which prevents storing relations between memory locations).

However, trying to combine this modeling (memory location -> set of values) with relational information between the values led to interesting results in the design of the Codex analyzer.

4.3.4 Custom memory allocator. Several of our value-analysis problems were exacerbated due to the use of a custom memory allocator in BIND. By nature, a custom memory allocator uses the same memory region to store multiple pieces of information, and uses discriminating unions everywhere; this means that a slight imprecision in the allocation leads to a disorganization of all the heap-allocated memory content. For example, in a conditional, if one branch does an allocation while the other does not, there is an imprecision in the index and the contents of the heap quickly becomes completely imprecise.

The most practical way we found to handle this problem was to replace this custom allocator with q standard call to *malloc* and *free*, which are handled directly by *Frama-C* internal primitives. However, the interface of this allocator is large and correct handling of it required a large effort. Even with this effort, the memory model used by *Value* assumes a finite number of memory allocations, which still leads to dynamic allocations done in loop and dynamic allocation of an unknown size being handled imprecisely.

4.3.5 Recursion. *Frama-C* uses a finite memory model, so that essentially local variables can be allocated at most once, like global variables. It also uses dynamically in-lined procedure calls to every function. The result is that recursion cannot be directly supported by the analyzer. Approaches to handling recursive cases are either to rewrite the affected functions by hand, or introduce an ANSI/ISO C Specification Language (ACSL) function contract to be used by the recursive calls. Both of these solutions require a deep understanding of the code under analysis.

Undertaking this manual work for BIND was unrealistic within the scope of the project. On this project, we developed a syntactic recursion-detection tool that detects connected components in the syntactic call graphs (i.e., that ignore calls occurring through function pointers). This tool found 28 distinct connected components, often with 8 functions inside the connected components.

However, building an ACSL specification for large connected components is difficult, because it must summarize accurately the action of a large chunk of code, and is therefore a difficult undertaking. Furthermore, such contracts often need to address recursive data structures such as red-black trees. While these can be expressed in ACSL through the use of logic functions, value analysis is not equipped to use them.

Finally, it is likely that syntactic recursive components are just a small part of the BIND challenge. Because BIND uses function pointers extensively, it is possible that our tool overlooked many potential recursive components. For example, a significant recursion error that we encountered was a recursive call to *isc_assertion_failed,* where the assertion check routine called an error routine through the use of a function pointer, that itself contained assertions.

4.3.6 Precision loss due to ACSL. We were unable to use a function definition for various reasons, including the recursion challenges and the excessive time needed for analysis. We also had to deal with external libraries whose code was not available or was not relevant for the analysis of the core BIND source code.

The result was to introduce ACSL specifications as a way to avoid expansion of parts of the program. However, writing ACSL stubs requires an understanding what the code does, to define an abstraction of its behavior. Unfortunately, sometimes ACSL is too imprecise for that purpose. In particular, ACSL does not distinguish between "possible" and "certain" stores, and thus performs imprecise weak updates in many cases.

In general, when evaluating an ACSL specification, one has to assume the worst, which leads to imprecise results. For instance, ACSL allows stating that a memory location contains a value computed from two pointers. Unfortunately, in the worst case it is possible to "mix up" (e.g., using a xor) the values of the pointers, which leads to an imprecise value that *Value* cannot eliminate. These problems occur frequently, particularly given BIND's use of complex data structures (and thus, many pointer manipulation).

4.3.7 Summary of BIND analysis. Our extended analysis of BIND indicated that whole-program analysis, like Value, works poorly with such a large code base. Further observations may be found in Section 5 – Conclusions. It also became clear that, to address elaborate systems like BIND, we need to eliminate some of the fundamental limitations of our tools, in particular the finite memory model and the intra-procedural trace partitioning. Some future directions for this tool development are outlined in Section 6 – Recommendations.

4.4 *Paparazzi*

4.4.1 Overview. *Paparazzi* is a complete system of open source hardware and software for Unmanned Aircraft Systems (UAS), which is composed of both airborne autopilot and ground control components. The ground station component includes mission planning and monitoring software, and utilizes a bi-directional data link for telemetry and control. The autopilot is written in C; part of the code is generic and compiled in libraries; other parts are compiled after generation from a GUI (written in OCaml). Beyond the obvious safety and security critical issues, *Paparazzi* was selected for CHEKOFV analysis for the following reasons:

- It has a large code base (200K lines of code), with many interesting patterns to analyze.
- The code is embedded, which is generally simpler to analyze. For example, embedded code usually does not feature recursion or dynamic allocation, which are not handled by *Value*. *Value* has a large record of successful analyses in embedded C code.
- The C code in *Paparazzi* consists in several different configurations that are assembled from different components, each of which are typically much smaller than 200kloc.

We successfully completed the analysis of all configurations of *Paparazzi* that were selected. In all cases, the analysis time was relatively short, at worst several minutes.

The main issue when analyzing *Paparazzi* was parsing. The compilation of a complete *Paparazzi* autopilot happens in two steps, where one part of the code is written "by hand", but uses preprocessing directives to handle different configuration. The other part of the code is generated and compiled by the GUI after selection of the configuration options.

4.4.2 Results. Our difficulty was to get the source code with the correct configuration option. A generated file was identified that contained most of the configuration necessary, from which we could semi-automatically write a Makefile from it.

Our analysis addressed the following *Paparazzi* source code configurations (fixed wing or helicopter, different system boards...):

- Microjet :: A fixed wing aircraft using the LPC21 board.
- Bixler :: A fixed wing aircraft using the STM32F1 board.
- Booz2 :: A quad-rotor using the STM32F1 board.
- Quadlisam2 :: Another quad-rotor using the STM32F1 board

The results are summarized in Table 7. The analysis was relatively fast in all cases. The number of alarms remained important and could have been reduced by unrolling more loops in the analysis. However, this would also have meant less interesting levels for the game players, so we set the trace partitioning and loop unrolling parameter to a small value ("*-slevel 10*").

Table 7. Analytical results for four *Paparazzi* configurations

Configuration	Time	Number of alarms	Number of analyzed statements
Microjet	16.3s	121	5029
Bixler	18.3s	427	5660
Booz2	217.3s	590	5720
Quadlisam2	34.6s	900	6085

4.4.3 *Paparazzi* Problem Identified. In the "Booz2" configuration, in src/sw/airborne/led.h, there is a led_init() function, which calls a lot of LED_INIT macros, one of set translating to:

*((gpioRegs_t *)0xE0028000)->dir1 |= (unsigned long)(1 << 31);*

This code is incorrect, as 1 is signed, 1 << 31 results in a signed overflow which is an undefined behavior. The correct replacement is ((unsigned long) 1 << 31), or (1UL << 31).

As this is an undefined behavior, the compiler is free to compile code that uses this expression arbitrarily. However, this bug is quite common, and in practice the code is compiled as expected; however a compiler that would be "too smart" (for instance using the LLVM pass that uses value information from the *Value* plugin) could do something wrong.

This problem was reported to the *Paparazzi* coordination group.

4.5 Cardinal of the state space metrics

As discussed in Section 3.14, several *Frama-C* plug-ins were developed to help determine the progress of verification efforts. However, measuring such progress was not always producing the desired results. Often, the progress appeared to remain constant even after several iterations of the verification loop and after adding multiple invariants. This was in part due to that fact that invariants provided by the game were limited to expressing invariants about numerical data types, or because Weakest Precondition (WP) failed to prove the necessary invariants due to missing environment assumptions that have to be provided by a human expert.

For these cases, we still wanted to measure if the invariants generated by CHEKOFV are beneficial to the value analysis by *Frama-C*. The metrics we chose is the cardinal of abstract states computed by value analysis; the smaller the cardinal, the more precise is the abstract interpretation. We illustrate this approach using the following program:

```
void main(void){
 int n, i;
 n = 17;
 for(i = 0; i < n; i++);
}
```

If we run *Value* on this procedure, it estimates that the procedure has approximately 111 reachable states and the intervals n ∈ {17} : i ∈ [17..127] for the local variables n and i. The number 111 corresponds to the cardinal of the state space at the end of function main. Because the analysis it non-relational, the concretization of this abstract value is:

$$\{n = 17 \wedge i = 17, n = 17 \wedge i = 18, \ldots n = 17 \wedge i = 127 \backslash\}$$

That is, the concretization consists in the Cartesian product of all possible values for every memory location.

Now, if we add an invariant to this program:

```
void main(void){
   int n, i;
   n = 17;
   /* loop invariant 0 <= i <= 17; */ :
   for(i = 0; i < n; i++);
 }
```

Value uses of the provided invariant to refine its results and estimates that there is now only 1 possible state at the end of this function (which is entirely deterministic) with the intervals n ∈ {17} : i ∈ {17}.

At the end of the increment statement i++, there are 17 possible values for i (between 1 and 18), which is also the best possible abstraction for this example.

The size of this set can quickly become huge. For instance, in a trivial program that takes an integer *i* as input and returns the same integer, we do not know the value of *i*, which can take 2^32 possible values (32 being the size of int). If there are *m* such unknown memory locations, the cardinal is 2^(32)^m, which is huge for a program with as many global variables as BIND. Using "big integers" to compute this cardinal exactly would have taken a prohibitive amount of CPU time and memory.

For this reason, we decided to use a logarithmic scale to compute the size of the cardinal when it can become big, i.e., when computing it for several memory locations. When there is a single memory location, 2^32 is a number with a reasonable size, and we used the normal scale, to increase precision.

This "logarithmic cardinal" is stored in a floating point value for two reasons. The first is that the result of the logarithm is generally not integral, and using floating point is thus appropriate. The second is that we do want a precision loss that is proportional to the cardinal, which is exactly what floating point provides.

In an exploratory case study, we ran the analysis on each function with loops in BIND and computed the size of the state for each statement of the main function. We compared the results with and without invariants. The results we have using the found invariants show that adding invariants reduces the state space in 285 functions (1534 statements). The average state space reduction is of 10^13,145 (to compare with the average state space of the program, in the range of 10^218,916,217,7821,975).

While this metric is not part of the final CHEKOFV system, it provides valuable insights for debugging the overall system, and spotting bugs during integration.

4.6 *Xylem* case study on SV-COMP Benchmarks

In a first feasibility study, we have generated puzzles for a set of programs from the sv-comp loop benchmarks [28]. So far, players of *Xylem* have solved 9589 puzzles. Out of these solutions, 5395 were duplicated answers (either exact duplicates or logically equivalent). For 1488 solutions, *Frama-C* could verify that they are valid loop invariants, for 6590 *Frama-C* could show that they are not invariants, and for 1511 invariants *Frama-C* failed to produce a result because they contained non-linear expressions that could not be handled by the employed theorem prover. This gives hope that even the relatively simple predicates that can be generated by *Xylem* are suitable to assist formal verification.

We carried out several player interviews to assess the usability of the game. One of the main complaints was that players wanted to be able to express transition predicates rather than invariant properties. Players complained that they want to state properties such as “x always increases by one”, or “y is equal to x from the previous state”. This may be a weakness of using the concept of plant growth in our narrative and we are currently exploring ways to improve this.

Probably the biggest challenge that we are facing for our future work is the scoring system. For reasonably large programs, it is not feasible to check player solutions on the server within a reasonable time. However, because there are many formulas that hold for a bounded sequence of states (including all tautologies), it is vital for the long-term motivation of the game to provide immediate feedback about the quality of a solution to the player.

4.7 Democratizing Verification

As discussed in Section 3.X, *Binary Fission* uses a classification model that provides several advantages in crowd-sourced verification. In particular, it provides a natural method of aggregating results across the experience of the crowd. While there are millions of programs that could benefit from formal verification, there are only a few thousand skilled logicians to undertake that activity. However, classification eliminates all symbolic reasoning, and *Binary Fission* only requires players to visually distinguish two classes of object patterns. Gamification also expands the user base. The net result is that using classification democratizes the verification task.

Several features of *Binary Fission* that make the game easy to play also impact the quantity and type of solutions found. First, the game suppresses the identity of program states and the mathematical content of the predicates used as filters. While this design limits the intuition that people can bring to bear on the verification task, it frees players to think less, and explore more options while generating classification trees. In conjunction with the roll-over mechanic for selecting and applying filters, *Binary Fission* encourages each player to work quite rapidly.

This approach has two, possibly hidden, benefits. Relative to the use of an automated classification tool (that selects each filter based on some greedy metric), players are free to explore locally non-optimal choices. By extension, the crowd as a whole will conduct a very broad search over the space of possible classification trees. The net effect is that the crowd will look under unexpected rocks, or if you prefer, for treasures in unanticipated locations. This is a good use of the crowd for verification tasks.

Overall, it is unclear whether a game design that facilitates rapid, broad search is better than a design that exposes details of the verification task and asks players to apply more problem-specific intuitions. We developed the relatively abstract classification model in *Binary Fission* because we saw that it would supply the kind of rapid feedback that would make the game appeal to citizen scientists. However, a different design might supply the same benefits while exposing more of the verification task's structure. That would be a subject for a future CHEKOFV game, after *Xylem* and *Binary Fission*.

4.8 *Binary Fission* Evaluation

Once players produce a classification tree, it is easy to read out logical expressions that characterize good states and bad states, and those expressions constitute likely invariants. *Binary Fission* classification trees are typically partial, such that leaf nodes can contain either good or bad states, or a mixture of both. The conjunction of predicates that links the root to a pure good node describes a set of states that satisfy program assertions, and expresses a likely invariant. As shown in Figure 40, tracing from the root node to the two pure positive nodes produces P ^ Q and P ^ R which form the candidate invariant (P ^ Q) (P ^R). A single player solution can contain several such paths. By extension, the disjunction of paths to pure good nodes across all player solutions forms the consensus, likely invariant. This results in an expression of the form:

$$PureGoodConjunct_1 \ldots PureGoodConjunct_n$$

Because these expressions are induced from data, they are only *likely*, or candidate invariants. Determining whether an expression is an actual program invariant requires logical proof. In the case of precondition finding tasks, and test the consensus likely invariants produced by *Binary Fission* by passing them through the CBMC model checker; an automated tool that calculates the logical effect of each program statement on the candidate invariant, and determines if the end result implies the desired postconditions. In general, only a small number of the clauses in the consensus likely invariant correspond to valid program preconditions. However, we have shown that the crowd collectively succeeds at this task, and for non-trivial programs (See Appendix 5).

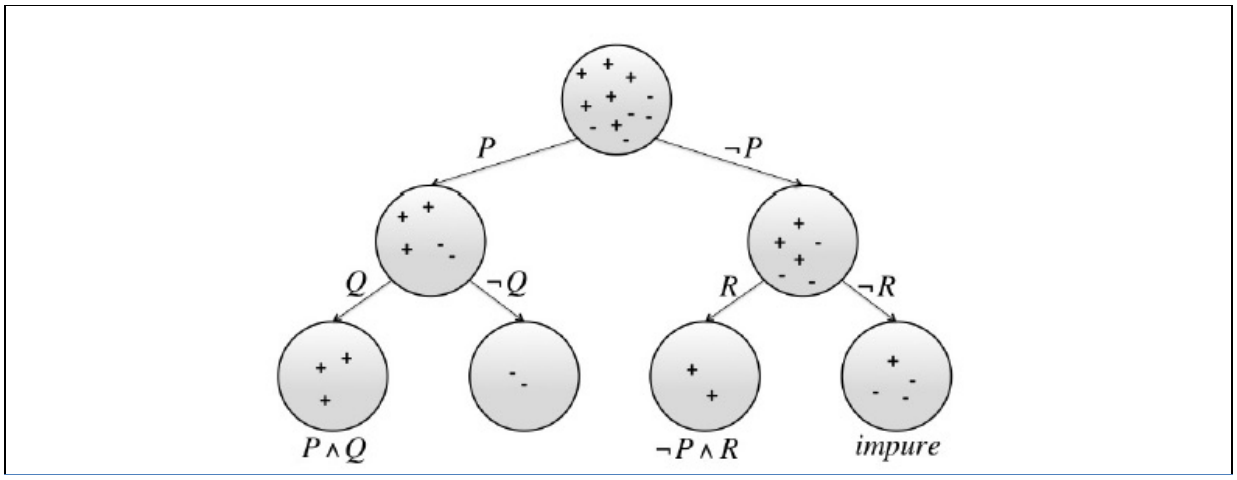


Figure 40. Example of a decision tree produced by *Binary Fission*.

To give a sense of the scale, the Traffic Collision Avoidance System (TCAS) is a ~200-line aircraft collision-avoidance program, and *Binary Fission* players found preconditions for 6 of its 7 functions (see Appendix 5). The likely invariants for those problems contained between 260 and 700 disjunctive clauses, and of those clauses, 6 to 103 of them proved to be program preconditions. For several problems, the top three preconditions explained between ~25% and ~50% of the data. In other words, the crowd employed *Binary Fission* to find valid, and general program preconditions. Said differently, the crowd (expectedly) finds quite a bit of junk, but *Binary Fission* successfully coordinates a large number of people to uncover valuable verification gems.

4.8.1 Agnosticism of the Classification Model. Our use of classification in *Binary Fission* has an additional benefit for the game's usefulness as a scientific tool; because the classification model is agnostic as to the source of the data, and the source of the primitive predicates used to separate that data, we can mate *Binary Fission* to a wide variety of automated tools. This is important because it is generally difficult to produce predicates relevant to an invariant finding problem, and to identify good, and especially bad program states. Automated tools for those tasks typically involve restrictions on the underlying program. For example, *Binary Fission* employed the *Daikon* system to suggest primitive predicates for use in precondition discovery, but this restricted our domain of application to algebraic programs with no arrays and no pointer variables. Given that the verification landscape is populated with many specialized tools that generate predicates or data for particular classes of programs, the agnosticism of the classification model opens the door to the maximum number of application paths.

We became aware of this benefit as the development of *Binary Fission* progressed. In the early stages of the design, we knew that we wanted a sorting mechanic. However, we only realized that agnosticism to data and predicate sources was an option when we began to connect the nascent game to application programs. Our interest in crowd sourced science also helped to bring this benefit into focus. Other classification oriented approaches exist for invariant discovery [14] but their emphasis is typically on optimizing execution speed vs opening paths to application.

4.8.2 Game Features that Impact Solution Quality. Two features of *Binary Fission* acutely affect the quality of the solutions that players find: the game limits the depth of the classification tree, and it motivates players via a scoring function that shapes the classification tree.

Binary Fission only allows trees of depth < = 5. We originally imposed this limit as a means of managing screen real estate, as trees of larger depth contained too many nodes to display at once, without reducing them to an impractically small size. (Building game mechanics to scan a larger tree would needlessly complicate the player interface.) However, the depth bound also had an unintended but fortuitous consequent for the verification task; it forced players to create small trees that might partially segment the data, but that could only express logical functions of limited complexity (one predicate per tree level). This complexity bound guards against overfitting (the tendency to limit the generality of learned expressions by describing every nuance in the data) which is a common failure mode of classification systems.

The scoring function for *Binary Fission* has a similar history. We introduced it as a somewhat arbitrary metric to reward incremental achievements, based on the intuition that it was useful to partially separate good states from bad, and especially useful to isolate pure nodes. We also wanted to encode several degrees of achievement into the reward function, and these factors led us to the following function:

$$N \times \sum_{i \in \text{leaf nodes}} \left(purity_i^A \times size_i^B \right)$$

Here, *purity* is the maximum over the percentage of good states and the percentage of bad states in the node, and *size* is a count of the states in the node. *A* and *B* are arbitrary constants. *N* is a constant that increases with the count of pure nodes, and decreases with maximum depth of the classification tree. This scoring function assigns moderate rewards to partial success (creating impure nodes), and significant reward to the creation of pure nodes. Moreover, it influences players to produce as many pure nodes as possible, as early in the classification process as possible.

This scoring function had its own unanticipated benefit for the verification task. By rewarding the creation of many pure nodes at small depth that describe large amounts of data, it selects for short logical expressions that have the potential to be useful, and general invariants. These are exactly the type of statements that verification engineers seek when solving invariant discovery tasks by hand.

4.8.3 Crowd-sourced Solution Progress. Figure 41 illustrates the crowd's progress towards finding a consensus likely invariant in the TCAS problem set. It plots cumulative data explained by the crowd-sourced solution, as accumulated in decreasing order of predicate quality (i.e., the number of good program states recognized by the conjunctive predicate associated with each Pure Good node). This figure supports several interesting observations. First, the top 20% of the solutions explain 80% of the data, and this pattern repeats across all problems. This suggests a statistical regularity in crowd performance, and an uneven distribution of expertise across players. Second, the consensus solution is partial, meaning it fails to explain all the data even after incorporating every player's contribution. This is an expected result, as *Binary Fission* limits the depth of player classification trees -- some truths are simply hard to express in bounded space.

To investigate this point further, we employed a greedy search algorithm to construct a classifier for the same problem, over the same primitive predicates. The method used average impurity for scoring splits. When invoked with a depth limit of 5, the resulting partial classifier explained 21 good program states. This splitting metric clearly provided insufficient motivation to distinguish Pure Good nodes early in the classification process that have utility for invariant generation. In contrast, the reward metric employed by *Binary Fission* clearly influenced players to isolate Pure Good nodes at shallower depths, with the associated benefit for explaining good program states. This pattern repeated across TCAS problems.

We also tested the expressive power of the primitive *Binary Fission* predicates by invoking the greedy classification algorithm without a depth limit. The result here, and in all 7 TCAS problems, was that the predicates had the power to correctly separate all good program and bad program states. As a result, our statistics on *Binary Fission* solutions concern the performance of the crowd, not the expressivity of the predicates at their disposal.

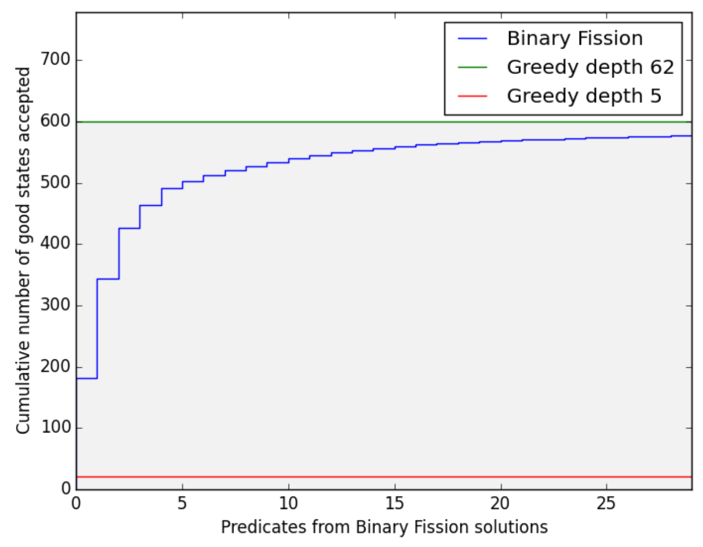


Figure 41. Progress of crowd towards consensus on invariant

4.8.4 Evaluation Summary. In this evaluation, we addressed the problem of crowd sourcing program preconditions, under the model that crowd sourcing offers an alternate, and viable method for addressing a difficult task. We have provided an existence proof in the form of the *Binary Fission* game, and we have shown that crowd sourcing is effective by employing the game to discover program preconditions for six particular problems. The preconditions are non-trivial, reasonably general (as measured by data coverage on a test set), and human readable. They are also novel, at least with respect to the output of *DTInv*, which finds likely invariants that do not qualify as program preconditions.

4.9 Recent *Binary Fission* Productivity

Figure 42 illustrates recent player participation in *Binary Fission*. Spikes may be observed at several points as a result of upturns in interest in the project. We believe that the large spike in August 2015 is associated with the publicity surrounding the 2015 Usenix Security Symposium, where various elements of our work were presented, as well as additional media outreach during that period.

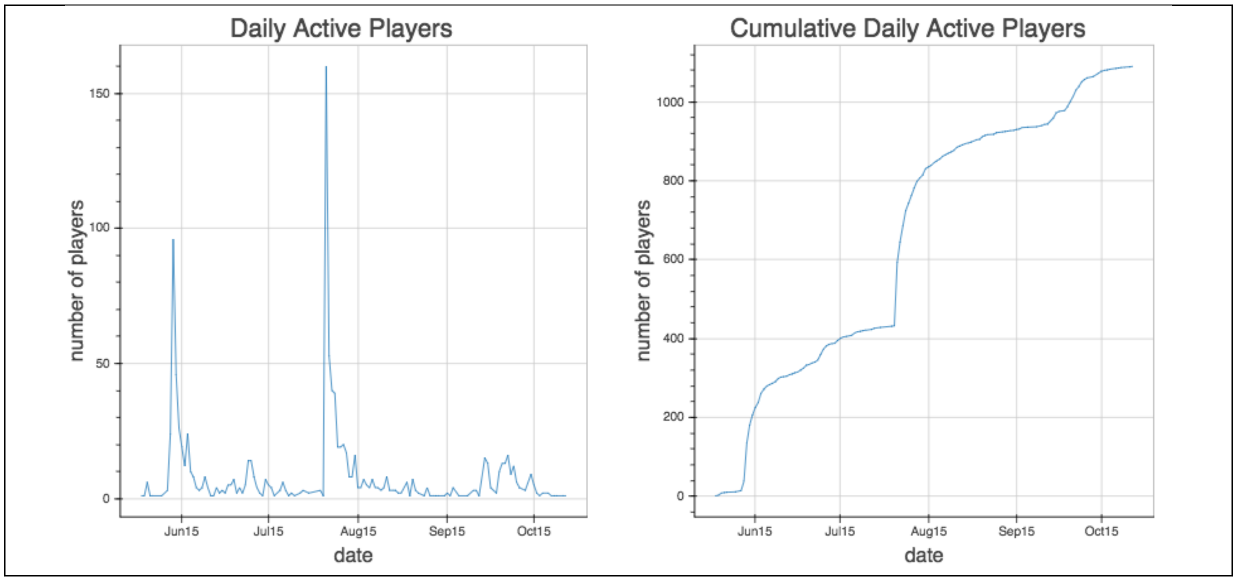


Figure 42. Player participation in *Binary Fission* – mid-May to mid-October 2015

The number of solutions submitted also tends to spike when new players are recruited. However, as demonstrated in Figure 43, we also see upticks in solution submissions that result from mailshots to current players, such as the one towards the end of September 2015.

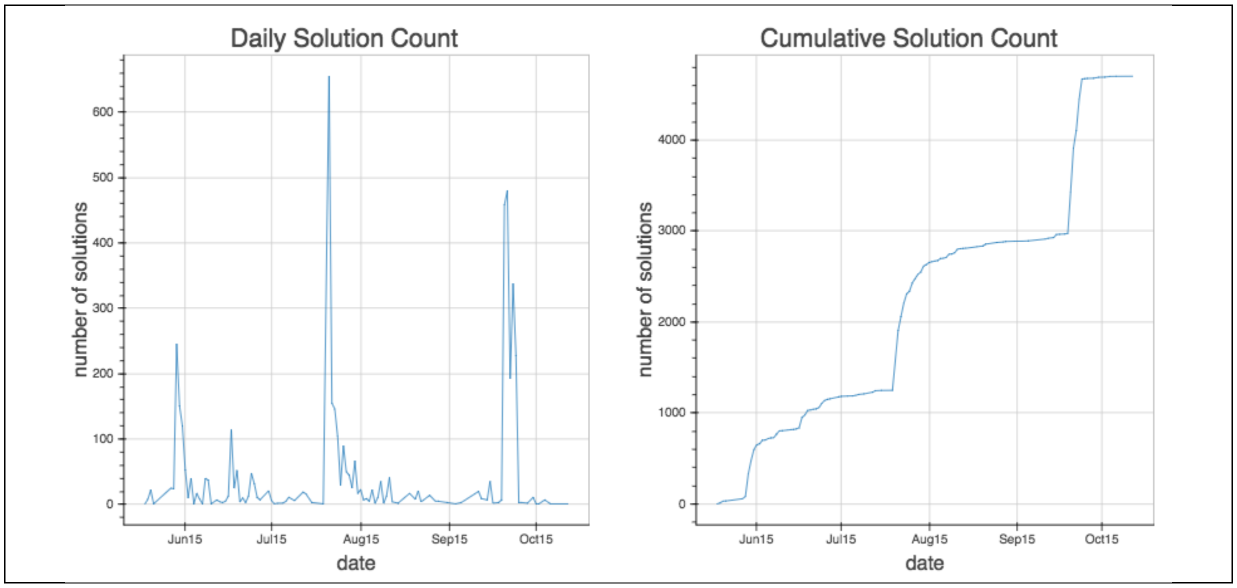


Figure 43. *Binary Fission* solutions submitted – mid-May to mid-October 2015

5. CONCLUSIONS

5.1 Whole-program analysis in context

As discussed in Section 3, our extended exploration of BIND indicated that whole-program analysis, like Value, works poorly with such a large code base. Unfortunately, modular analysis also works badly because of the heavy use of function pointers in this system. As an example, Calcagno, Distefano, O'Hearn and Yang [29] used a modular analysis on big C programs and reports the results shown in Table 8.

Table 8. Results of modular analysis on large C programs [29]

Program	Version	KLOC	Number of procs	Proven Procs	Proven	Time
Linux kernel	2.6.30	3032	143768	86268	60	9617.44
Gimp	2.4.6	705	16087	8624	53.6	8422.03
Gtk	2.18.9	511	18084	9657	53.4	5242.23
Emacs	23.2	252	3800	1630	42.9	1802.24
Glib	2.24.0	236	6293	3020	48	3240.81
Cyrus imapd	2.3.13	225	1654	1150	68.2	1131.72
OpenSSL	0.9.8g	224	4982	3353	67.3	1449.61
Bind	9.5.0	167	4384	1740	39.7	1196.47
Sendmail	8.14.3	108	820	430	52.4	405.39
Apache	2.2.8	102	2032	1066	52.5	557.48
Mailutils	1.2	94	2273	1533	67.4	753.91
OpenSSH	5	73	1329	594	44.7	217.81
Squid	3.1.4	26	419	281	67.1	107.85

As can be seen, despite the fact that programs known to be difficult were present (such as Linux), BIND was the one with the worse results.

As noted in Section 6 – Recommendations, the CHEKOFV analysis of BIND drove a series of insights for enhancing our verification tool set.

5.2 Scoring scheme in *Xylem*

Because a valid *Xylem* solution with certain numbers of variables or bonus tiles might not even be possible in a given problem, we couldn't simply provide scores based on the number of variables or bonus tiles used. Instead we used a model inspired by the science-based game *SpaceChem* (www.zachtronics.com/spacechem/). At the completion of a level, *SpaceChem* shows the player a visualization of how they have performed compared to other players of the same level. We adapted this to *Xylem* by plotting out on a chart the number of variables and bonus tiles used by a player as compared to other players of the same puzzle. As discussed earlier, if a player used the same or greater number of variables (and, separately, bonus tiles) than the highest number used so far by any player, then they receive a star. In the same way, players were scored for bonus tiles, and producing an equality in a solution gave the player another star. In this way, a player could receive a score of up to three stars per puzzle. Additionally, a special stamp is granted to the player if they have created a “novel solution”. That is, a solution that hasn’t yet been recorded for this particular puzzle.

This scoring system was included in the first release of the game. Unfortunately it had some notable problems, not the least of which was that it was very easy to “game” the system, by taking advantage of the design in order to always get a high score. This frustrated those players who did not want to cheat, but at the same time were faced with an obvious way to collect three stars every single time without putting much effort in. It also made players’ contributions feel somewhat meaningless, knowing that other players could achieve three stars without carefully considering the puzzles. Furthermore, because it was so easy to “game” the system, the process risked transitioning from “not overly contributing" to *Xylem*’s science goals to being pretty much useless for this endeavor. The original design was intended to encourage the contribution of useful invariants, reward players for their efforts while playing the game, and encourage them to continue playing. However, it did none of these things, and so revisions to the scoring system became the main impetus for releasing a major *Xylem* update post-launch.

Another *Xylem* scoring idea considered, but subsequently not implemented, was to build a restricted-inference capability into the game client that could check the relative strength of a player-submitted invariant against invariants established by the backend. This would provide immediate feedback of two types: that the player’s solution is subsumed by known invariants, or potentially unique (pending evaluation by the backend). Unique candidates would receive a high score. Defining and using such a structure is an example of a problem at the boundary between game development and verification research, which we hope to explore further in future projects.

5.3 Peer review in *Xylem*

In the process of upgrading *Xylem* during Phase One of the project, we introduced a peer review scheme, where players had the opportunity to rate the solutions previously provided by others. Figure 44 shows a sample screen from this part of the game. This technique provided us with a mechanism to crowd-source the relative quality of candidate invariants, as well as offer additional opportunities for players to earn more rewards.

5.4 Insights on *Binary Fission* evaluation

Earlier in this report, we addressed the problem of crowd-sourcing program preconditions, under the model that crowd sourcing offers an alternate, and viable method for addressing a difficult task. An existence proof in the form of the *Binary Fission* game has been provided, and we showed that crowd sourcing is effective by employing the game to discover program preconditions for 6 TCAS problems. The preconditions are non-trivial, reasonably general (as measured by data coverage on a test set), and human readable. They are also novel, at least with respect to the output of *DTInv*, which finds likely invariants that do not qualify as program preconditions.

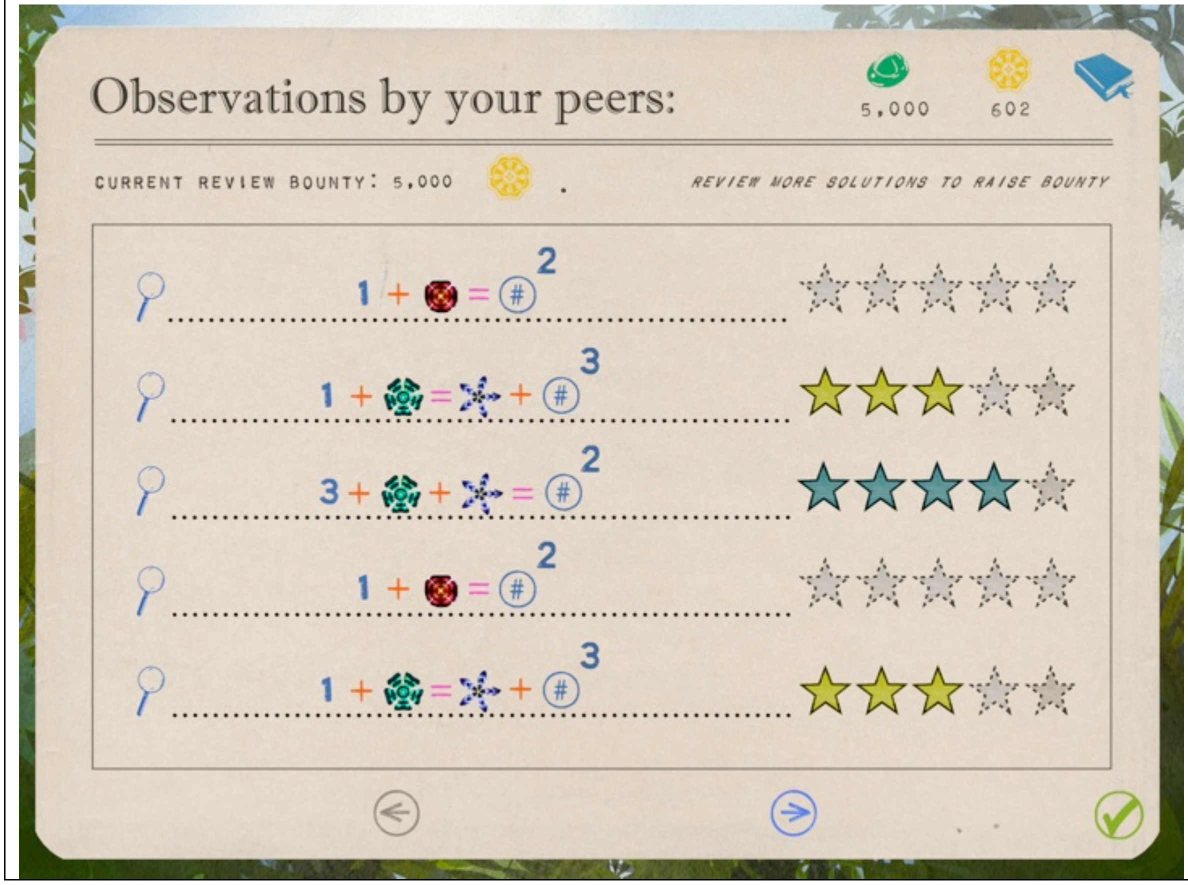

Observations by your peers:
5,000
602
CURRENT REVIEW BOUNTY: 5,000
REVIEW MORE SOLUTIONS TO RAISE BOUNTY

Figure 44. Peer review screen from *Xylem*

There are three sources of power behind *Binary Fission*: it employs an expressive representation, it relies on the crowd to conduct a thorough search, and the game imposes restrictions on that search that select for general solutions. In more detail, the representational power comes from *Daikon*, as *Binary Fission* inputs the highly structured predicates it produces. The game exploits crowd search by collecting and testing the large number of piecewise solutions that players contribute. The game influences the shape of the solution by limiting classifier depth, and by rewarding discovery of partial classifiers that isolate positive data, which has special utility for invariant construction.

While *Binary Fission* employs a classification model, improving classification technology is not our goal. Our main point is to introduce crowd sourcing as a promising approach to invariant discovery. From this perspective, the key conjecture behind crowd sourcing is that many non-expert individuals have the desire and ability to provide insight into highly technical problems when they are presented in a suitable form. This conjecture holds for *Binary Fission*. If it generalizes, related games will provide leverage on additional verification tasks, and crowd sourcing will offer an avenue for expanding the reach of verification technology.

At this stage, we can only report the first results from a crowd-sourced approach to precondition discovery. As mentioned above, the key points are that crowd sourcing is feasible, effective, and promising as a practical avenue for expanding the reach of verification methods. That said, there are several threats to the validity of these claims, as well as our more detailed results.

First, while crowd sourcing finds preconditions on TCAS, the approach may not generalize to more complex programs. In particular, TCAS is a short, straight line, arithmetic program that lacks pointers, loops, complex data structures, and a range of other language features that complicate the verification task. The counterpoint is that *Binary Fission* is agnostic to the structure of the underlying program, because it formulates precondition discovery as classification. The limits on its use come from the need for inputs common to classifiers; a base of relevant primitive predicates, and labeled data distinguishing bad program states from good. It is true that these inputs are hard to provide for more complex programs (especially the predicate base and assertion violating program states) as they are the product of deep analyses of program structure. However, *Binary Fission* is also agnostic as to the source of these data, which greatly increases its avenues for application.

Second, our results on the novelty of the *Binary Fission* solution could be the product of our choice of *DTInv* as the comparator. This is quite plausible; the likely invariants produced by other machine learning methods might qualify as preconditions. However, our experience with *Binary Fission* has illuminated constraints that should be applied to the use of classifiers for this task; they should penalize solution size (which is common wisdom), employ a powerful predicate base to support human legibility of the end result, and reward identification of pure good nodes rather than focus on an entropic measure as the splitting criterion.

A third, and broader concern, is that classification is viable but our use of crowd sourcing is superfluous, meaning that *Binary Fission* can be replaced by a suitable automated method. This argument is relevant at this stage in the development of *Binary Fission*, but it devolves to the underlying question, “What does the crowd bring to classification that is difficult to automate?”. In the case of *FoldIt,* players brought spatial intuition to the task of folding complex proteins, and obtained results never achieved through search over molecular conformations in combination with energy minimization methods.

Classification tasks also have a natural framing as search, and by analogy, the crowd may intuit which predicates to employ en route to a more general solution. *Binary Fission* currently hides a bit too much information to support this type of intuition (in service of broadening the game's appeal), but advanced versions will provide more context about the underlying task. We currently rely on the crowd to explore unexpected places relative to the greedy search conducted by automated methods, and this approach has successfully produced program preconditions.

Another consideration is that *Binary Fission* may be addressing a less salient crowd-sourcing problem. Rather than ask the crowd to combine primitive predicates, perhaps their skills would be better employed on the task of inventing the predicates themselves. As was discussed earlier, this was the intent of *Xylem* and the other Phase One CSFV games [30]. Predicate invention (including predicate abstraction from data) is a critical, but elusive, process currently performed by people, as is the process of finding the ideal crowd-sourcing techniques themselves.

In summary, *Binary Fission* was employed to analyze the implementation of an on-board aircraft collision detection and avoidance system. We found that the crowd can employ *Binary Fission* to prove program properties. They found function preconditions (statements about program variables associated with function inputs) that guarantee important safety properties hold on program exit, where those properties are encoded as postconditions. *Binary Fission* players also discovered concise, general, and human readable preconditions, which are also novel relative to the complicated logical expressions often produced by other classifications systems. The players have no special expertise in formal methods or programming, and are not specifically aware they are solving verification tasks.

Binary Fission demonstrates the feasibility of crowd-sourced invariant discover, and it illustrates the promise of crowd sourcing for other verification tasks. This suggests a pathway for expanding the reach and practical application of verification technology.

5.5 Game Features for Science Tasks

Binary Fission is both a game and a mechanism for program verification. As a result, the design has two roles that are sometimes in conflict. In particular, the game must be enjoyable to have an audience at all, which exerts a force towards simplifying or abstracting the science task in service of playability. However, as a verification technique, *Binary Fission* must also maintain fidelity to the science task to be useful at a practical level. This tension implies a small sweet spot in the game design.

Previously, we have discussed the development of *Binary Fission* as a playable game. We now examine the evolution of features that impact its usability for crowd-sourced verification, and that shape the quality of the solutions they find as perceived by verification engineers. Usability features include the use of classification trees for crowd sourcing, and *Binary Fission*'s agnostic stance towards the source of predicates and good/bad program states. The key solution-shaping features are the reward function, and depth limit on the size of the solution tree.

The classification model employed by *Binary Fission* provides several advantages for facilitating crowd-sourced verification; it captures several types of verification problems, it expands the set of people who can perform the task, and it provides a natural method of aggregating results across the experience of the crowd.

In more detail, several forms of the invariant discovery problem easily map onto the task of distinguishing good states from bad. As mentioned earlier, if the target is loop invariants, any state produced by the loop at a given iteration is good, and any state that could never be produced by the loop at that iteration is bad. If the task is finding program preconditions, good states are program inputs that satisfy postconditions on execution, while bad states violate those postconditions. In both cases, the logical function expressed by the classification tree provides encoded candidate invariants.

Once players produce a classification tree, it is easy to read out logical expressions that characterize good states and bad states, and those expressions constitute likely invariants. In *Binary Fission*, the classification trees are typically partial; some leaf nodes only contain good states, some only contain bad states, while others contain a mixture. The conjunction of predicates that links the root to a pure good node describes a set of states that satisfy program assertions, and expresses a likely invariant. A single player solution can contain several such paths.

It is worth noting that we only understood how to form a consensus invariant after *Binary Fission* had been developed and deployed. Our original thought was that we could use the pure bad nodes and the impure nodes in the classification tree. It turns out that the impure nodes are of no value for finding program invariants except as input to further search/classification. However, the pure bad nodes can offer value; the negation of a path to any pure bad node can be AND-ed into the description of any pure good node, and the result tested via the model checker. The additional restriction may turn a failed candidate into a valid precondition (or make a successful candidate less general).

Finally, as noted earlier, the classification model in *Binary Fission* is agnostic as to the source of the data, as well as the source of the primitive predicates used to separate that data. This introduces a broader benefit for the game's usefulness as a scientific tool, because *Binary Fission* can be deployed as a human-driven aggregator to a wide variety of automated tools.

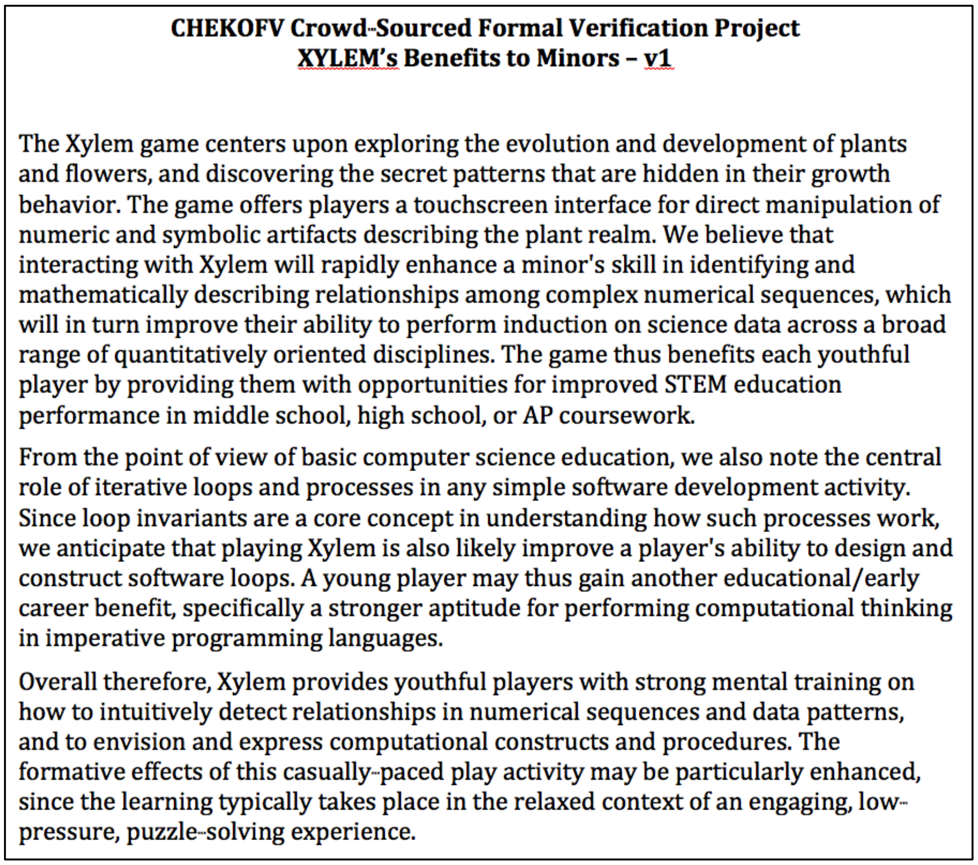
5.6 Challenges Involving Research Ethics Oversight

The fundamental public relations and marketing strategy for the overall CSFV program was to unify multiple verification games into a single Verigames identity. This provided valuable promotional leverage for all the game-development teams, and it centralized the player recruitment and signup process. The core Verigames coordination was handled by a separately-contracted team led by TopCoder (now part of Aperio Inc). This simplified things from an individual player's point of view, because they only needed to enroll once to participate in the complete program. However, a key impact of this approach was that the process for obtaining and maintaining Institutional Review Board (IRB) approvals differed substantially from the typical standard procedures used for many other behavioral programs involving human participants.

The top level IRB application strategy for CSFV was to assemble a single package covering all teams and institutions, and obtain central approval for the overall program. However, this resulted in the need for specific local IRB approvals at any institution that was involved with collecting and analyzing software verification data. For CHEKOFV, this strategy posed problems at both SRI and UCSC, because our team members were not involved with player recruitment and were not collecting or accessing any personally-identifiable information (PII). Thus, representatives of both the local IRBs considered the project to be exempt from their purview. However, these initial decisions were not considered as providing sufficient ethical oversight to justify obtaining central approval for the overall CSFV program.

In the course of managing this approval process, it was noted that players under 18 years old might be recruited as participants. The result was that additional consent forms and other document revisions, including a specific Benefits to Minors statement (see Figure 45), were needed for inclusion in an updated central application package. Because there was now a possibility that minor participants might be involved in our work, the local IRBs at both of our institutions we able to confirm that the research project no longer qualified as exempt from their oversight, and they each generated an expedited approval for CHEKOFV instead. These decisions in turn provided adequate justification for central approval for the overall program.

As it subsequently transpired, there were other administrative and logistical challenges involved with recruiting minors as players, so that part of the general CSFV program activity was later dropped in any case. Further information on undertaking online-centric behavioral research in the future may be found in Section 6: Recommendations of this report.



CHEKOFV Crowd-Sourced Formal Verification Project
XYLEM's Benefits to Minors – v1

The Xylem game centers upon exploring the evolution and development of plants and flowers, and discovering the secret patterns that are hidden in their growth behavior. The game offers players a touchscreen interface for direct manipulation of numeric and symbolic artifacts describing the plant realm. We believe that interacting with Xylem will rapidly enhance a minor's skill in identifying and mathematically describing relationships among complex numerical sequences, which will in turn improve their ability to perform induction on science data across a broad range of quantitatively oriented disciplines. The game thus benefits each youthful player by providing them with opportunities for improved STEM education performance in middle school, high school, or AP coursework.

From the point of view of basic computer science education, we also note the central role of iterative loops and processes in any simple software development activity. Since loop invariants are a core concept in understanding how such processes work, we anticipate that playing Xylem is also likely improve a player's ability to design and construct software loops. A young player may thus gain another educational/early career benefit, specifically a stronger aptitude for performing computational thinking in imperative programming languages.

Overall therefore, Xylem provides youthful players with strong mental training on how to intuitively detect relationships in numerical sequences and data patterns, and to envision and express computational constructs and procedures. The formative effects of this casually-paced play activity may be particularly enhanced, since the learning typically takes place in the relaxed context of an engaging, low-pressure, puzzle-solving experience.

Figure 45. Xylem statement on Benefits to Minors

6. RECOMMENDATIONS

6.1 Citizen Science and *Binary Fission*

The main goal of our citizen science work has been to introduce crowd sourcing as a promising approach to invariant discovery. From this perspective, the entire CHEKOFV game infrastructure, in particular the underpinnings of *Binary Fission*, is an exploration of a simple conjecture, i.e., that many non-expert individuals have the desire and ability to provide insight into verification tasks when they are presented in a suitable form. From our initial experience, this conjecture appears to hold, especially because *Binary Fission* was designed as an aggregator/consolidator for candidate invariants for other games, robot solvers, or automated verification tools. If it generalizes, the related games will provide leverage on additional verification tasks, and crowd sourcing will offer an avenue for expanding the reach of verification technology.

As a result, the work of CHEKOFV can be expanded in multiple directions. One obvious direction is to extend *Binary Fission* by addressing known flaws in existing source code. For example, we could make use of the pure bad nodes in the classification tree to improve the formal invariant test. We could introduce additional social collaboration features. More broadly, we could address the criticism that *BF* does not engage the crowd's intuitions about classification tasks by exposing more insights on the program state, and by developing game mechanics for visualizing and parameterizing the application of predicates to the data space. The goal here would be to let players more usefully employ human spatial intuition.

A second path forward would be to build (literally) an industrial strength version of *Binary Fission*. This successor game would input client code, and interface with a variety of automated techniques that provide predicates, supply good and bad program states, and test crowd sourced candidate invariants for their status as actual invariants against the client's program. This game would require surmounting or finessing significant technical challenges, as the current generation of automated techniques carry idiosyncratic restrictions on the size and content of the program under analysis, on the types of conclusions they can draw, and on the computational efficiency of deriving those results. As a consequence, *Binary Fission II* would probably accept a class of programs far larger than the algebraic functions analyzed by the current game, but far less general than the arbitrary programs that industrial clients might hope to analyze.

A third avenue to explore would be to produce a suite of games that collectively span a verification task. For example, we could let players define primitive predicates in a *Xylem*-like environment, and then compose them via *Binary Fission* to form more complex invariants.

A logical extension of this idea is to build a suite of crowd-sourced games that address each hard part of data driven invariant discovery; predicate generation (a la *Xylem*), the creation of good and (especially) bad program states, composition of likely invariants (as in *Binary Fission*), evaluating the strength of likely invariants (with feedback into the games for predicate generation), and testing likely invariants as program invariants, to name a few. Our intuition on this last task is that data driven predicate generation games could be employed to augment an automated model checker by proposing predicates that facilitate its derivation when it encounters difficulties.

As a whole, a robust development of crowd sourced verification games has the potential to greatly expand the application and utility of verification technology.

6.2 Future Directions for Verification using Value Analysis

The comprehensive investigation of value analysis within the CHEKOFV project, particularly in the context of *Frama-C*, resulted in the development of several new software tools that helped to address several of the shortcomings described previously in Section 4.3. We believe that the related lessons learned offer a number of valuable directional pointers for future research in the formal verification field.

6.2.1 Non-termination analysis. To better understand the situation where value analysis could not terminate, we developed several new tools during the course of the project.

Recursive component analysis

The first tool to be developed was used to finds the recursive functions in a program by analyzing the strongly connected components in the syntactic call graph, together with the entry points of this connected component. This tool cannot prove the absence of recursion in a program, due to function pointers (as happens in BIND); however it finds all the syntactically recursive calls (that do not go through function pointers), which is the most common case. Moreover, as it gives the entry point to these recursive components, it tells which function to stub to replace the set of function.

The development of this tool raised the more general question of how to partition large code bases automatically to make them amenable for automatic analysis. Finding suitable a partitioning of BIND for the analysis with *Frama-C* turned out to be a labor-intensive step because the verification engineer had to develop an in-depth understanding of the application logic and data structures used in BIND . E.g., BIND reads configuration data from a file into a red-black tree which needs to be stubbed manually because neither configuration data nor the data structure can be handled efficiently by *Frama-C*.

Extending the recursive component analysis to identify and stub more parts of the input program that are known to cause problems with *Frama-C* would significantly reduce the manual effort required during the verification.

Profiling

We have also instrumented the analyzer so that it reports where it spends time analyzing C functions. For every function, it reports the number of times that it was called, and the total time analyzing the code of that function. It also reports the time spent for each function in each level in each call stack, as well as the current call stack.

Using this report, one can replace the functions that takes too long to be analyzed, and are called too often, by stubs that takes a shorter time to analyze. This tool is now integrated in *Frama-C*, and has proven invaluable in many situations to improve the analysis time. We believe that other analysis tools could also benefit from this form of internal performance monitoring.

De-recursifier

We have developed a new way to handle recursive calls in *Value*, that requires no change to the memory model. The idea is that when a function is called, which was already in the call stack, the code of the function is dynamically duplicated. This handles the fact that local variables are considered as if they were global by value; by the code duplication, we have two instance of the variable. Of course this technique only works in the case where the recursion is of finite height, and *Value* is precise enough to find out that this is the case; but in some cases this spares the need to stub a recursive function.

Program transformations such as this increase the degree of automation, however, they alter the structure of the program and thus make it harder to track findings back to the original program. In the future, we plan to implement this step and similar steps as refactoring to improve traceability. This is an interesting direction because, unlike normal refactoring, the refactoring in the context of verification does not necessarily have to preserve the semantics of the original program (e.g., it might over-approximate).

Memexec cache

Because *Value* analyzes function calls by (semantically) in-lining them, analyzing a function call can take a lot of time. But many times, different function calls take the same arguments in memory; for instance function calls in a loop vary only slightly in their execution, and they may call sub-functions with the exact same arguments.

For this reason, *Value* has a "cache" which associates to each function, to each set of values of the part of memory read by the function, the part of memory that is assigned by the function. When the function is called in the same conditions, the output can be reused. This cache is crucial for execution time: for instance, its use in *Paparazzi* allowed some analyses to be executed for several seconds, instead of several minutes.

However, this cache was incompatible with the use of dynamic allocation: when the cache was hit, it behaved as if the memory allocations returned were re-using an old memory region, instead of a fresh one. During the course of the project, this memexec cache was enhanced to properly (soundly) handle dynamic allocations.

In the future, we plan to combine value analysis with shape analysis summaries like the ones computed by Facebook's Infer tool to improve the effectiveness of memexec cache.

6.2.2 Other enhancements. We developed modifications to our custom build tool that dynamically record the set of arguments used when compiling BIND, and enable it to be used by the preprocessing and parsing phase of *Frama-C*. We also implemented a preprocessing plugin that manages calls to variadic functions (i.e. those which accept a variable number of arguments) by simplify them into regular function calls, and generates ACSL contracts along the way. Others miscellaneous verification features developed for the project included the introduction of additional warnings that allows to better track when loss of precision occurs, as well as many extensions to our stub of the GNU C library.

6.2.3 New analyzer. The above extensions allows better understanding of what does not work, and provide workarounds in some cases, but the main causes we have identified (finite memory model, absence of relation and intra-procedural trace partitioning) remain. Using our experience with *Value* and the lessons learned from analysis of BIND, we have begun developing an analysis framework that could handle these issues. In particular, the memory model is parametric, so that it can be changed to handle infinite memory, or to do a pessimistic modular intra-procedural analysis (which allows analyzing recursive calls).

6.3 New Framework for Research Ethics

As noted in Section 5, the process for obtaining and maintaining Institutional Review Board (IRB) approvals for the overall CSFV program differed substantially from the typical procedures used in many other behavioral programs involving human participants. In particular, the IRB application strategy for CSFV was to assemble a single package covering all teams and institutions, and obtain central approval for the overall program.

Under the procedures that were prevailing at the time, this necessitated first obtaining local approvals at each participating institution, which can rapidly become a coordination headache. IRB guidelines and regulations are primarily designed for classical human research laboratory work in fields like medicine and psychology; however, they have limited practical relevance to modern human behavior studies, such as CSFV, that involve highly-networked information and communications technology systems.

Nowadays, numerous distributed research consortiums are working in these contexts, not just with crowd-sourcing activities, social networks and popular gaming worlds, but also with online educational environments, cybersecurity applications, and surveillance systems. The current impracticalities with research ethics oversight are exacerbated by the pervasive need to undertake comprehensive, transnational experimental projects, where much of the human data collection and analysis is undertaken remotely across varied, and often incompatible, legal regimes and social norms. In such conditions, it is inevitable that individual local IRBs will maintain differing opinions about their purview for these forms of, typically low-risk, studies.

Newly proposed regulatory changes may assist with addressing some of these practicality concerns. Revised policies and regulations have been put forward for comment with the publication of a Notice of Proposed Rulemaking (NPRM) in the U.S. Federal Register [31].

Although these updates may contribute somewhat to improving the situation, further coordinating guidelines and support is required for overseeing human participant research using online systems and other cyber-environments across multiple jurisdictions. In essence, an international ethics observance organization is needed for this purpose. This could for example be a consortium of non-profit organizations operating in several domains, which would ensure smooth transnational processing of approvals. It seems appropriate that such a consortium would need to have the backing of a recognized international entity such as UNESCO.

6.4 Educational Games: Project Fibonacci

6.4.1 Background. *Project Fibonacci* is a proposed initiative for adapting *Xylem: The Code of Plants* to a game for math anxiety reduction and algebra learning among middle school students. Math anxiety has been the subject of considerable scholarship for several decades [32][33]. For many reasons, middle school becomes a critical moment for a student's relationship with math: because there is a jump in math skills required from elementary to middle school, because math in middle school becomes very abstract, or because some middle schoolers are not developmentally ready for the abstraction.

Students at this juncture who disconnect from math and decide that they are just not good at it, that its not for them, can develop life-long math anxiety which not only affects their future schooling but can cause them problems during their adult years as well. It can have profound effects on their self-esteem and self-image and cause them to avoid activities that they may be otherwise interested in (running their own business, programming, and science). Even paying a check at a restaurant or looking over bills can cause embarrassment and anxiety symptoms.

6.4.2 Approach. *Project Fibonacci* aims to create an intervention in the mathematical lives of middle school students, shoring up their skill and confidence in math so that they have an opportunity to avoid the pain of math anxiety and build a positive relationship with it. This intervention takes the form of a game that teaches and supports math skills while being intrinsically motivating.

Fibonacci can be adapted from the existing math-puzzle game *Xylem: The Code of Plants,* if several changes are used to create an age-appropriate game that meets the science objectives:

- Replace procedurally generated puzzles with hand-crafted puzzles, with attention given to creating an appropriate difficulty curve.
- Use a reward structure to reinforce players' successes on a personal one-on-one level.
- Add new theme and narrative framing that will appeal to the target audience, while making the math more concrete by tying it to real-world phenomena.

At the same time, using the *Xylem* engine gives us several advantages over creating a game from scratch: *Xylem* includes a sophisticated equation builder, which allows players build equations from component parts.

The core gameplay of *Xylem* involves inductive reasoning, and is one of very few games to do so. This encourages players to look at mathematics from a different angle than they are accustomed to. The core gameplay of *Xylem* involves cognitive reasoning, pattern recognition and mathematical thinking. It is not simply math drills but rather puzzles to figure out, thereby supporting the philosophy of the new Common Core Standards in mathematics.

Because we are no longer tasked with deriving puzzles from actual code in a piece of existing software, we can hand-craft the puzzles presented to players and in so doing handcraft the difficulty curve of the game. We are able to teach concepts in a logical order and to allow players to gradually expand their skill set at a comfortable pace. The game will be designed such that players are building on their own improving skills and experiences as they progress.

6.4.3 Related Work. While the space of math games is well populated, most attention to gameplay and aesthetics of the game experience is spent on games for elementary school students. Games exist for middle school to adult populations, but these games are often little more than math drills. Arguably the best math game for any age group, *DragonBox* teaches algebra skills. While *Project Fibonacci* will teach math skills to middle-school aged students, its central objective is to foster greater comfort in working with numbers and the ability to think mathematically. We also wish to help students make a smoother transition from concrete to symbolic thinking in mathematics.

Wigfield and Meece [32] identified affective components in math anxiety among 6th to 12th-graders, such as nervousness, fear and discomfort, as well as cognitive components expressed primarily as worry about performance. They recommend that both components should be addressed by efforts to reduce math anxiety: the cognitive through confidence-building and the affective through "training to reduce fear and dread" of mathematics. The design of *Project Fibonacci* addresses both concerns by using inductive reasoning, a carefully controlled difficulty curve, hand-crafted puzzles, and the ability for players to work together on puzzles. By representing core math concepts as puzzles, the game offers youngsters a different way of looking at math.

7. REFERENCES

[1] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “*Frama-C*: A Software Analysis Perspective,” *Form. Asp. Comput.*, pp. 1–37, Jan. 2015.

[2] P. Cousot, P. Ganty, and J.-F. Raskin, “Fixpoint-Guided Abstraction Refinements,” in *SAS*, 2007.

[3] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *POPL*, 1977.

[4] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “The ASTRÉE analyzer,” in *PLS*, 2005.

[5] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The *Daikon* System for Dynamic Detection of Likely Invariants,” *Sci Comput Program*.

[6] P. Garg, C. Löding, P. Madhusudan, and D. Neider, “Ice: A robust framework for learning invariants,” in *CAV*, 2014.

[7] R. Sharma and A. Aiken, “From Invariant Checking to Invariant Inference Using Randomized Search,” in *CAV*, 2014.

[8] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, “Verification as learning geometric concepts,” in *SAS*, 2013.

[9] R. Sharma, A. V. Nori, and A. Aiken, “Interpolants as classifiers,” in *CAV*, 2012.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided Abstraction Refinement for Symbolic Model Checking,” *J ACM*, vol. 50, no. 5, Sep. 2003.

[11] K. L. McMillan, “Lazy Abstraction with Interpolants,” in *CAV*, 2006.

[12] I. Dillig, T. Dillig, B. Li, and K. McMillan, “Inductive Invariant Generation via Abductive Inference,” in *OOPSLA*, 2013.

[13] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, “Feedback-driven Dynamic Invariant Discovery,” in *ISSTA*, 2014.

[14] S. Krishna, C. Puhrsch, and T. Wies, “Learning Invariants using Decision Trees,” *CoRR*, 2015.

[15] S. Graf and H. Saidi, “Construction of abstract state graphs with PVS,” in *CAV*, 1997.

[16] C. Flanagan and S. Qadeer, “Predicate Abstraction for Software Verification,” in *POPL*, 2002.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended Static Checking for Java,” in *PLDI*, 2002.

[18] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, in *TACAS*, 2008.

[19] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović, “Verification games: Making verification fun,” in *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, 2012, pp. 42–49.

[20] P. C. Wason, “On the failure to eliminate hypotheses in a conceptual task,” *Q. J. Exp. Psychol.*, vol. 12, no. 3, pp. 129–140, Jul. 1960.

[21] J. M. Kleinberg, “Authoritative sources in a hyperlinked environment,” *J. ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999.

[22] A. Cooper, *The Inmates are Running the Asylum*. Indianapolis, IN, USA: Macmillan Publishing Co., Inc, 1999.

[23] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović, and F. players, "Predicting protein structures with a multiplayer online game," *Nature*, vol. 466, no. 7307, pp. 756–760, Aug. 2010.

[24] A. Kawrykow, G. Roumanis, A. Kam, D. Kwak, C. Leung, C. Wu, E. Zarour, Phylo players, L. Sarmenta, M. Blanchette, and J. Waldispühl, "Phylo: A Citizen Science Approach for Improving Multiple Sequence Alignment," *PLoS ONE*, vol. 7, no. 3, p. e31362, Mar. 2012.

[25] K. Tuite, N. Snavely, D. Hsiao, N. Tabing, and Z. Popovic, "PhotoCity: training experts at large-scale image acquisition through a competitive game," 2011, p. 1383.

[26] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA Companion*, 2007, pp. 815–816.

[27] A. Biere and R. Bloem, Eds., *Computer Aided Verification*, vol. 8559. Cham: Springer International Publishing, 2014.

[28] D. Beyer, "Status Report on Software Verification - (Competition Summary SV-COMP 2014)," in *TACAS*, 2014, pp. 373–388.

[29] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," 2008, p. 289.

[30] H. Logas, J. Whitehead, M. Mateas, R. Vallejos, L. Scott, D. Shapiro, J. Murray, K. Compton, J. Osborn, O. Salvatore, and others, "Software Verification Games: Designing Xylem, The Code of Plants," *FDG*, 2014.

[31] U. S. F. Register, "Protection of Human Subjects," *Not. Propos. Rulemaking NPRM*, vol. www.federalregister.gov/articles/2015/09/08/2015–21756/federal-policy-for-the-protection-of-human-subjects, 2015.

[32] A. Wigfield and J. L. Meece, "Math anxiety in elementary and secondary school students.," *J. Educ. Psychol.*, vol. 80, no. 2, pp. 210–216, 1988.

[33] F. Pajares and L. Graham, "Self-Efficacy, Motivation Constructs, and Mathematics Performance of Entering Middle School Students," *Contemp. Educ. Psychol.*, vol. 24, no. 2, pp. 124–139, Apr. 1999.

8. APPENDICES

Appendix 1: Jason Rohrer: Code Breaker Design Concept.

Appendix 2: Huascar Sanchez: Loop Analysis in BIND/lib.

Appendix 3: Maria Daltayanni: CHEKOFV Ranking System (CRS).

Appendix 4: Lauren Scott: CyphrSeedr Tutorial Design Document.

Appendix 5: Fava et al: Crowdsourcing Program Preconditions via a Classification Game

CHEKOFV Final Report

Appendix 1

Code Breaker Design Concept

Jason Rohrer

Design Concept: *Code Breaker*

Jason Rohrer

January 16, 2012

1 Introduction

We've already narrowed our focus down to crowd-sourced discovery of loop invariants. Even within this relatively limited realm, it seems that reasoning about pointers and arbitrary data structures, without representing those structures explicitly, is beyond the reach of a casual-friendly, sufficiently-abstract game design. During our meeting, I watched marble-and-pipe machines sprout dizzying complexity as they tried to capture the behavior of even a simple linked list.

In a last-ditch effort to produce a design that doesn't explicitly represent program structure, I'm further narrowing the focus: loops for which the full, relevant state space—for a given execution instance of the loop—can be represented by a finite set of numerical values.

2 Thematic Overview

A newly constructed radio telescope has been receiving perplexing data sequences from various points in deep space. At first, these sequences were dismissed as the output of quasars, but over time, that explanation has become less and less satisfying. First of all, the locations of these sources do not match the positions of any previously known quasars—not surprising in and of itself, given that this scope is more sensitive than previous devices. More shocking is the data itself: when interpreted numerically, it seems that clear, logical patterns emerge. We are hesitant to use the "I" word here, but we almost cannot help seeing some kind of intelligence in these patterns. Not language, as we normally think of it, but perhaps a language of numerical relationships? We don't know for sure, and that's why we need your help.

We've got thousands of sources to analyze. Furthermore, from each source, we have a virtually unlimited supply of sample messages of varying length. Each message is an example of the pattern being output by a given source. Your job is to detect and describe a pattern in each source's messages. If your pattern misses something, our database will automatically provide you with messages that break your proposed pattern—more information with which you can revise your pattern.

But just because you devise a pattern that covers all messages from a given source doesn't mean that you've nailed it. Maybe your proposed pattern isn't as specific as it could be. You'll be collaborating with others from around the world who are working on the same task. Together, you'll hone the pattern for each source down into its tightest, most well-tuned form.

3 Design Description

3.1 Related Design

The enormously popular casual puzzle game *Square Logic* gives players a grid of logical and mathematical constraints (for example, "these cells must all be odd," "these cells must sum to 6," or "the product of these cells must be 8") and then asks the player to fill the grid with numbers, Sudoku-style, so that no number occurs more than once in each row or column:

http://www.squarelogicgame.com/

Code Breaker is an inversion of *Square Logic* in that the player is given a grid of numbers and asked to find the constraints.

3.2 Extracting Messages from Running Programs

Completely ignoring program structure, we can look at the relevant program state space (data that is actually touched by the loop) as a set of anonymous numerical values. At the end of a given loop iteration, that numerical state can be represented as a *line* of numbers, like this:

0	10	10	5	12	3

Lines from multiple iterations can be stacked, in order, to form a complete, multi-line *message* like this:

0	10	10	5	12	3
1	10	10	5	12	3
2	12	10	5	12	3

Finally, columns in a given message can be assigned anonymous labels like this:

A	**B**	**C**	**D**	**E**	**F**
0	10	10	5	12	3
1	10	10	5	12	3
2	12	10	5	12	3

3.3 Player Goal

Given a multi-line messages like the one shown above, the player attempts to define constraints that describe each line in the message. Looking at the given message, the player might propose the following constraint:

$$B \leq E$$

3.4 Goal Iteration

A player-proposed constraint might be satisfied by the current messages or even by all sample messages that the player has seen so far. However, there might be other messages from the same source that break the proposed constraint. The player is presented with such a counter-example:

A	**B**	**C**	**D**	**E**	**F**
0	10	10	5	2	3
1	10	10	5	2	3
2	10	10	5	2	3

Clearly, $B \leq E$ doesn't hold for this message. The player might think that $B \leq C$ could work, but that would be violated by the previously-seen message. Stumped, the player requests another example message:

A	**B**	**C**	**D**	**E**	**F**
0	10	10	15	2	3
1	15	10	15	2	3
2	15	10	15	2	3

Now it becomes clear that B takes on the maximum value of C, D, and E, so the player might propose:

$$B \leq \max(C, D, E)$$

And this, it turns out, is satisfied by all messages with that line length. A longer message from the same source is presented to the player as a new counter-example:

A	**B**	**C**	**D**	**E**	**F**	**G**
0	10	10	15	2	16	4
1	15	10	15	2	16	4
2	15	10	15	2	16	4
3	16	10	15	2	16	4

While the same $\max$ relationship is present here, it's clear that a variable number of columns must be accounted for. The player tries:

$$B \leq \max(\text{colspan}(C, A))$$

Where the *colspan* operator extracts the set of columns starting at C and moving A columns to the right.

And, it turns out that all messages from this source satisfy that constraint. The player submits her solution and moves on to tackle message from a different source.

3.5 Meager Solutions

Of course, the rather tight constraint discovered by the previous player required quite a bit of insight and effort. A more loose constraint might be proposed by a less industrious player:

$$A \leq \text{lastColumn}$$

Where the *lastColumn* operator picks the value of the last column in a given line. Yes, all messages satisfy the constraint, and in fact, this particular constraint was missed by the previous player. These two constraints could be combined into the following set, which is stronger than either in isolation:

$$B \leq \max(\text{colspan}(C, A))$$
$$A \leq \text{lastColumn}$$

Thus, players can build on each other's constraints to find even better constraints for the messages from a given source.

3.6 Underlying Code

The above "messages" were actually the state space extracted from the end of each loop iteration in the following function:

```
arrayMax( a, n )
    m = INT_MIN

    for( i=0; i<n; i++ )
        if( a[i] > m )
            m = a[i]

    return m
```

with the following mapping:

Code Variable:	Message Column:
i	A
m	B
a	C $\cdots$ prevCol(lastCol)
n	lastCol

The key insight in this kind of mapping is that though `arrayMax` can handle input of arbitrary length, a given instance of its invocation always involves a finite state space. Furthermore, useful pattern information can be gleaned, as demonstrated above, from extremely small invocation examples. Yes, though the example function might process thousand-variable state spaces in practice, such examples don't provide more information about constraint patterns than much smaller examples. Loops behave inductively, after all, so we don't need to worry about how our system scales to huge state spaces.

4 Presentation

The enormous popularity of Sudoku, Drop7, and Square Logic suggests that casual players don't have trouble with logical or mathematical reasoning. However, there's no sense in overloading the presentation of *Code Breaker* with unfamiliar symbols (even the difference between $<$ and $\leq$ might not be clear to non-programmers).

Instead of asking the player to type in constraint formulas, we can ask them to construct a *pattern machine* that matches lines from messages. The anonymous column names (A, B, C, etc.) can be disks for dragging and dropping. Operators ($+$, $-$, $\times$, $\div$) can be blocks that connect disks together. Relationship operators, such as the aforementioned $<$ and $\leq$, can become blocks with explanatory icons (sloping right triangles).

Aggregation operations, like min and max, can become containers where other blocks can be dropped inside. The colspan operator can be a block with two slots in it (one slot for the first column name, and the second slot for an expression describing the column extent).

Players can then run their pattern machine on a message, line by line, to see where the message breaks their machine.

CHEKOFV Final Report

Appendix 2

Loop Analysis in BIND/lib

Huascar Sanchez

This talk in one slide

- How many loops there are in BIND/lib?
- What is the range of loops in BIND/lib?
- What is the range of data types and data structures used in these loops?
- Summary of results

How many loops there are in BIND/lib?

Directory	While loops	For loops	Total loops
lib/dns	1659	651	2310
lib/bind9	3	43	46
lib/export	18	32	50
lib/irs	30	22	52
lib/isc	304	158	462
lib/isccc	8	14	22
lib/isccfg	185	31	216
lib/lwres	83	60	143
lib/tests	13	0	13

Totals	2303	1011	3314

The range of loops in BIND/lib

- Loops are not created equal.
  - Some have very complex conditionals, others are simple.
  - Some have no function/method calls inside them, others have lots.
  - Some may have many internal statements, others very few---or none.
- Consequently, we *hypothesize* that by looking at a large enough sample from BIND, we'll discover logical clusterings containing loops with similar properties; i.e., **markers**.

The range of loops in BIND/lib

- This sample must be small enough to be conveniently handled by us, yet large enough to help us detect meaningful patterns
  - e.g., control breaks, early exit, continuation with next iteration
- We *randomly* selected *100* loops.
- This sample contained a mix of ***while***, ***for***, and ***do-while*** loops.

The range of loops in BIND/lib

- To help us examine loops in BIND, we've used a set of ***markers***.
- These ***markers*** were handy for reasoning about the loops' internals:
  - Update (U). e.g., result = ISC_SUCCESS
  - Control Break (CB). e.g., if(condition){ UPDATE }
  - Early Exit (EE). e.g., break, return DATA, goto LABEL.
  - Continue with next Iteration (CN). e.g., continue
  - Inner Loops (IN).
  - ANY (A).

What is the range of loops in BIND/lib (**WHILE**)?

By using these markers, we've identified 7 classes of *While* loops in Bind.

Class	Pattern	Count
W1	(U, CB)	9
W2	(U, CB, U)	4
W3	(CB, U)	4
W4	(U)	9
W5	(U, IL, U)	5
W6	(U, CB, EE, IL, U | EE)	5
W7	(U | A , EE | CN, CB | U | A)	4

The range of loops in BIND (W1)

General Case

```
while (condition) {
    UPDATE
    CONTROL_BREAK
}
```

The range of loops in BIND (W1)

Concrete Case

file: `lib/bind9/check.c`
function: `check_viewacls`

```
while (acls[i] != NULL) {
    tresult = checkacl(acls[i++], actx, NULL,
                       voptions, config,
                       logctx, mctx);
    if (tresult != ISC_R_SUCCESS)
        result = tresult;
}
```

The range of loops in BIND (W2)

General Case

```
while (condition) {
    UPDATE
    CONTROL_BREAK
    UPDATE
}
```

The range of loops in BIND (W2)

Concrete Case

file: `lib/irs/getaddrinfo.c`
function: `freeaddrinfo`

```
while (ai != NULL) {
    ai_next = ai->ai_next;
    if (ai->ai_addr != NULL)free(ai->ai_addr);
    if (ai->ai_canonname)
        free(ai->ai_canonname);
    free(ai);
    ai = ai_next;
}
```

The range of loops in BIND (W3)

General Case

```
while (condition) {
    CONTROL_BREAK
    UPDATE
}
```

The range of loops in BIND (W3)

Concrete Case

file: `lib/dns/adb.c`
function: `dump_adb`

```
while (entry != NULL) {
    if (entry->refcnt == 0)
        dump_entry(f, entry, debug, now);
    entry = ISC_LIST_NEXT(entry, plink);
}
```

The range of loops in BIND (W4)

General Case

```
while (condition) {
    UPDATE
}
```

The range of loops in BIND (W4)

Concrete Case

file: lib/export/samples/sample-update.c
function: main

```
while ((buf = ISC_LIST_HEAD(usedbuffers)) != NULL) {
    ISC_LIST_UNLINK(usedbuffers, buf, link);
    isc_buffer_free(&buf);
}
```

The range of loops in BIND (W5)

General Case

```
while (condition) {
    UPDATE
    INNER_LOOP
    UPDATE
}
```

The range of loops in BIND (W5)

Concrete Case

file: lib/export/samples/nsprobe.c
function: reset_probe

```
while ((pns = ISC_LIST_HEAD(trans->nslist)) != NULL) {
    ISC_LIST_UNLINK(trans->nslist, pns, link);
    while ((server = ISC_LIST_HEAD(pns->servers)) != NULL) {
        ISC_LIST_UNLINK(pns->servers, server, link);
        isc_mem_put(mctx, server, sizeof(*server));
    }
    isc_mem_put(mctx, pns, sizeof(*pns));
}
```

The range of loops in BIND (W6)

General Case

```
while (condition) {
    UPDATE
    CONTROL_BREAK
    EARLY_EXIT
    INNER_LOOP
}
```

The range of loops in BIND (W6)

Concrete Case

file: lib/isc/base32.c
function: base32_tobuffer

```
while (!ctx.seen_end && (ctx.length != 0)) {
    unsigned int i;
    if (length > 0) { eol = ISC_FALSE; }  else { eol = ISC_TRUE; }
    RETERR(isc_lex_getmastertoken(lexer, &token,
                                  isc_tokentype_string, eol));
    if (token.type != isc_tokentype_string) { break; }
    tr = &token.value.as_textregion;
    for (i = 0; i < tr->length; i++)
        RETERR(base32_decode_char(&ctx, tr->base[i]));
}
```

The range of loops in BIND (W7)

General Case

```
while (condition) {
    EARLY_EXIT
}
```

The range of loops in BIND (W7)

Concrete Case

file: lib/isc/include/isc/radix.h
function: has_whitespace

```
while ((c = *str++) != '\0') {
    if (c == ' ' || c == '\t' || c == '\n')
        return (ISC_TRUE);
}
```

What is the range of data types and structs used in these loops?

Structs	W1	W2	W3	W4	W5	W6	W7
cfg_obj	x						
cfg_aclconfctx	x						
isc_token	x					x	
isc_mem	x						
gai_statehead	x						
gai_resstate	x	x					
addrinfo	x	x					
gai_restrans	x	x					
isc_textregion	x					x	
base32_decode_ctx	x						
isc_region	x	x					x
isc_buffer	x					x	
isc_entropy	x					x	
isc_cbsource	x						
isc_mem_t		x		x	x		
isccc_sexpr		x				x	
isc_entropysource_t			x				
isc_buffer_t				x			
base32_decode_ctx_t				x			
__sFILE				x		x	
irs_dnsconf_dnskey_t				x			
dns_name_t				x	x		
dns_message_t					x		
dns_section_t					x		
dns_rdataset_t							
probe_ns					x		
server					x		
base64_decode_ctx						x	
isc_lex						x	
isccc_region						x	x
lwres_conf						x	

What is the range of data types and structs used in these loops?

Types	W1	W2	W3	W4	W5	W6	W7
int	x	x	x	x	x	x	x
isc_boolean_t	x		x			x	x
char *buffer	x	x	x	x		x	x
unsigned int	x	x	x	x	x	x	x
char	x		x				x
NULL	x	x	x	x	x	x	x
isc_bitstring_t			x				
isc_bufferlist_t				x			
unsigned char				x			x
unsigned char *buffer				x		x	x
query_result_t					x		
isc_tokentype_string						x	
isc_sha1_t						x	
isc_uint8_t						x	
isc_uint32_t						x	

The range of loops in BIND (FOR)

We've identified 6 classes of *For* loops in Bind.

Class	Pattern	Count
F1	(U, CB)	18
F2	(U, EE)	7
F3	(U)	10
F4	(A , INNER_LOOP, A)	7
F5	(U, CN, CB, U, CB)	4
F6	(U, EE, CN, U)	3

The range of loops in BIND (F1)

General Case

```
for (condition) {
    UPDATE
    CONTROL_BREAK
}
```

The range of loops in BIND (F1)

Concrete Case

file: `lib/bind9/check.c`
function: `check_order`

```
for (element = cfg_list_first(obj);
     element != NULL;
     element = cfg_list_next(element))
{
    tresult = check_orderent( cfg_listelt_value(element), logctx);
    if (tresult != ISC_R_SUCCESS)
        result = tresult;
}
```

The range of loops in BIND (F2)

General Case

```
for (condition) {
    UPDATE
    EARLY_EXIT
}
```

The range of loops in BIND (F2)

Concrete Case

file: `lib/bind9/check.c`
function: `bind9_check_key`

```
for (i = 0; algorithms[i].name != NULL; i++) {
    len = strlen(algorithms[i] .name);
    if (strncasecmp(algorithms[i].name, algorithm, len) == 0 &&
        (algorithm[len] == '\0' ||
         (algorithms[i] .size != 0 && algorithm[len] == '-')))
        break;
}
```

The range of loops in BIND (F3)

General Case

```
for (condition) {
    UPDATE
}
```

The range of loops in BIND (F3)

Concrete Case

file: `lib/dns/journal.c`
function: `journal_open`

```
for (i = 0; i < j->header.index_size; i++) {
    j->index[i].serial = decode_uint32(p);
    p += 4;
    j->index[i].offset = decode_uint32(p);
    p += 4;
}
```

The range of loops in BIND (F4)

General Case

```
for (condition) {
    ....
    INNER_LOOP
    ....
}
```

Approved for Public Release; Distribution Unlimited

The range of loops in BIND (F4)

Concrete Case

file: `lib/lwres/getaddrinfo.c`
function: `lwres_strsept`

```
for (s = string; *s != '\0'; s++) {
    sc = *s;
    for (d = delim; (dc = *d) != '\0'; d++)
        if (sc == dc) {
            *s++ = '\0';
            *stringp = s;
            return (string);
    }
}
```

The range of loops in BIND (F5)

General Case

```
for (condition) {
    UPDATE
    CONTINUE
    CONTROL_BREAK
    UPDATE
    CONTROL_BREAK
}
```

The range of loops in BIND (F5)

Concrete Case

file: `lib/bind9/check.c`
function: `check_dual_stack`

```
for (element = cfg_list_first(obj);
     element != NULL;
     element = cfg_list_next(element)) {
    value = cfg_listelt_value(element);
    if (cfg_obj_issockaddr(value))
        continue;
    obj = cfg_tuple_get(value, "name");
    str = cfg_obj_asstring(obj);
    isc_buffer_init(&buffer, str, strlen(str));
    isc_buffer_add(&buffer, strlen(str));
    dns_fixedname_init(&fixed);
    name = dns_fixedname_name(&fixed);
    tresult = dns_name_fromtext(name,
                    &buffer, dns_rootname,
                        0, NULL);
```

```
if (tresult != ISC_R_SUCCESS) {
        cfg_obj_log(obj, logctx, ISC_LOG_ERROR,
                "bad name '%s'', str);
        result = ISC_R_FAILURE;
    }
    obj = cfg_tuple_get(value, "port");
    if (cfg_obj_isuint32(obj)) {
        isc_uint32_t val = cfg_obj_asuint32(obj);
        if (val > ISC_UINT16_MAX) {
            cfg_obj_log(obj, logctx, ISC_LOG_ERROR,
                    "port '%u' out of range', val);
            result = ISC_R_FAILURE;
        }
    }
}
```

The range of loops in BIND (F6)

General Case

```
for (condition) {
    UPDATE
    EARLY_EXIT
    CONTINUE
    UPDATE
}
```

The range of loops in BIND (F6)

Concrete Case

file: `lib/isc/base32.c`
function: `base32_decodestring`

```
for (;;) {
    int c = *cstr++;
    if (c == '\0')
        break;
    if (c == ' ' || c == '\t' || c == '\n' || c== '\r')
        continue;
    RETERR(base32_decode_char(&ctx, c));
}
```

What is the range of data types and structs used in these loops?

Structs	F1	F2	F3	F4	F5	F6
isc_log_t	x			x	x	x
cfg_listelt_t	x				x	
cfg_obj_t	x	x		x	x	x
dns_name_t	x		x		x	
dns_fixedname_t	x		x		x	
isc_buffer_t	x	x				
isc_textregion_t	x					
dns_secalg_t	x					
dns_acacheentry_t	x			x		
dns_acache_t	x	x	x	x		
dns_rdatatype_t	x					
dns_rdataset_t	x		x			
dns_rdataclass_t	x					
dns_rdatasetiter_t	x					
addrinfo	x	x			x	
isc_logchannel	x					
isccc_sexpr_t	x					
__sFILE	x					
in_addr	x					
isc_sockaddr	x					
intervaltable	x				x	
dns_acl_t	x			x	x	
cfg_aclconfctx_t	x			x	x	
keyalgorithms		x				
dns_db_t		x		x		
dbentry		x		x		
dns_journal_t		x				
isc_mem_t		x		x		
hostent			x			
isc_entropypool_t			x			
dns_clientrestrans_t			x			
query_trans			x			
dns_message_t			x			
dns_dbnode_t				x		
isc_netaddr_t				x		
hex_decode_ctx_t						x
optionstable						x

What is the range of data types and structs used in these loops?

Types	F1	F2	F3	F4	F5	F6
unsigned int	x	x	x	x	x	x
NULL	x	x	x	x	x	x
char *buffer	x	x	x	x	x	x
int	x	x	x	x	x	x
isc_uint8_t	x			x		
unsigned char *buffer	x		x	x		
size_t	x	x				
isc_heap_t	x					
char	x			x		
isc_mutex_t	x					
irs_resconf_t	x	x				
isc_buffer_t	x				x	
isc_uint32_t	x		x	x	x	
isc_log_t	x				x	
isc_mem_t	x				x	
isc_boolean_t	x		x		x	
isc_uint16_t		x				
dns_namelist_t			x			
ISC_LIST				x		
unsigned char				x		
unsigned long				x		

A few struct definitions

cfg_obj

file: `lib/isccfg/include/isccfg/grammar.h`

```
struct cfg_obj { const cfg_type_t *type;
        union { isc_uint32_t          uint32;
                isc_uint64_t          uint64;
                isc_textregion_t string; /*%< null terminated, too */
                isc_boolean_t         boolean;
                cfg_map_t             map;
                cfg_list_t            list;
                cfg_obj_t **          tuple;
                isc_sockaddr_t        sockaddr;
                cfg_netprefix_t netprefix;
        }               value;
        isc_refcount_t  references;     /*%< reference counter */
        const char *    file;
        unsigned int    line;
};
```

A few struct definitions

isc_textregion

file: `lib/isc/include/isc/region.h`

```
struct isc_textregion {
        char *              base;
        unsigned int        length;
};
```

A few struct definitions

isc_mem

file: `lib/isc/include/isc/mem.h`

```
struct isc_mem {
        unsigned int              impmagic;
        unsigned int              magic;
        isc_memmethods_t          *methods;
};
```

A few struct definitions

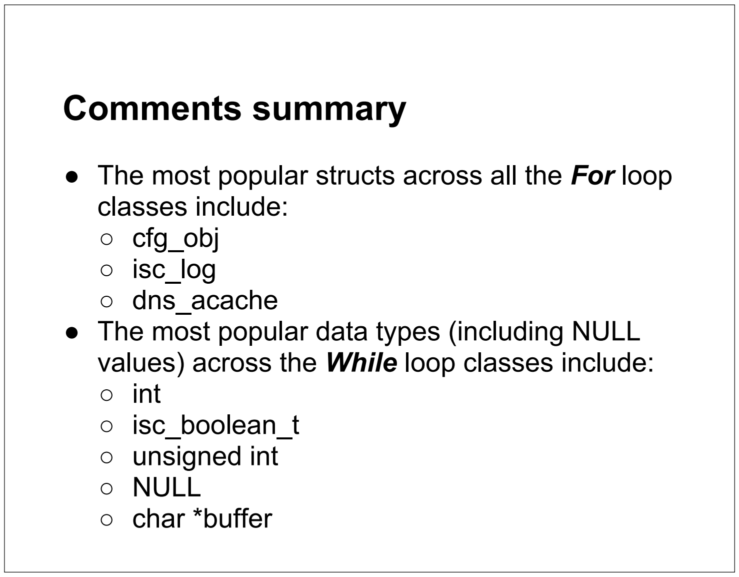
isc_buffer

file: `lib/isc/include/isc/buffer.h`

```
struct isc_buffer {
        unsigned int                   magic;
        void                          *base;
        unsigned int                   length;
        unsigned int                   used;
        unsigned int                   current;
        unsigned int                   active;
        ISC_LINK(isc_buffer_t)         link;
        isc_mem_t                      *mctx;
};
```

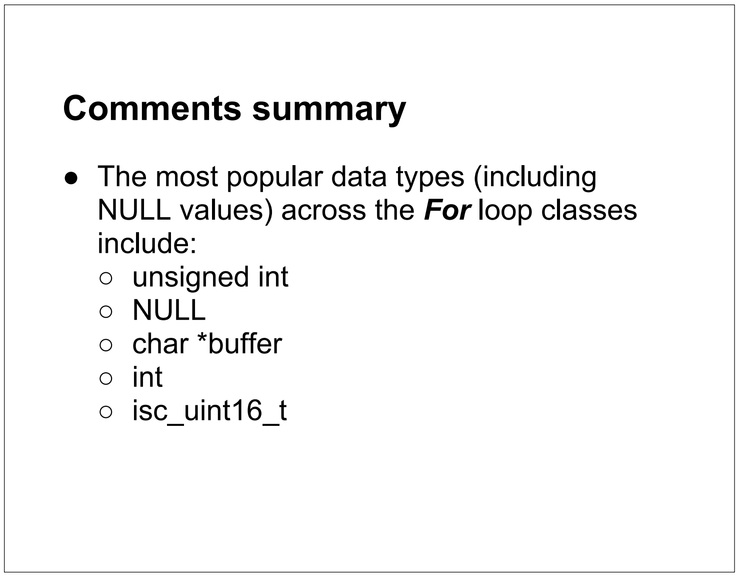
Comments summary

- The total number of loops in BIND/lib is 3314.
  - ***While*** loops are 2303
  - ***For*** loops are 1011
- We've identified 7 classes of ***While*** loops and 6 classes of ***For*** loops based on the use of a set of **markers**.
- More clusters can be found by using shared data types.
- The most popular structs across all the ***While*** loop classes include:
  - isc_region
  - isc_mem



Comments summary

- The most popular structs across all the ***For*** loop classes include:
  - cfg_obj
  - isc_log
  - dns_acache
- The most popular data types (including NULL values) across the ***While*** loop classes include:
  - int
  - isc_boolean_t
  - unsigned int
  - NULL
  - char *buffer



Comments summary

- The most popular data types (including NULL values) across the ***For*** loop classes include:
  - unsigned int
  - NULL
  - char *buffer
  - int
  - isc_uint16_t

CHEKOFV Final Report

Appendix 3

CHEKOFV Ranking System (CRS)

Maria Daltayanni

GameRank API Methods

def add_outcome(realm_id, user_id, problem_id, invariant_id, d)
This adds the result of a user solution on a problem. Check that they have unique user_id and problem_id that we can use. The score is a non-negative (?) **(? they do not know, probably non negative)** floating point number, reflecting the correctness of the solution proposed by the user. No special assumption is made on these solution scores.
Problem: Do the user_id and problem_id need to exist already on our side or not? Yes
Or do we have to implement create_user() and create_problem methods? No
For us, the main use of a create_user() … method is that it ensures that we have the same basic information for all users, since we cannot guarantee that update_user_info below is going to be called.
d is a dictionary containing:
This data is available:

- score of solution
- duration of solution attempt
- gave_up: boolean flag indicating whether a solution was entered
- List of (start_time, end_time) of all time spans when the game was played (note: do you prefer to update this information as time goes by, via a method called add_outcome_info? **No** - Or give it to us only once the play ends? **Yes**).
- Platform on which game was played (phone, tablet, PC, etc)
- Location / interactivity level info (is a player playing in a stationary room, or just toying with it in a bus?). **? not surely available**
- **played in a tournament or in collaboration: not sure they have it for everyone**

Questions: do they give us an invariant_id or do they call add_outcome and we return an invariant_id? **They give it to us**
Do we have to give methods also to delete an outcome? Or delete all outcomes for a problem? **Yes**
def delete_outcome(realm_id, user_id, problem_id, invariant_id)
def delete_all_outcomes_for_problem (realm_id, problem_id)

Or for a player? **No**

Question: should we have a single reputation system, or should we have "realms", and prefix every call with a realm_id? We need a testing realm and a production realm, at least. **Ok with these 2 realms for the beginning. If we need more reputation systems, e.g. for different skills, we can have more realms.**

def update_user_info(realm_id, user_id, d)

This is used to let our ranking system know about the existing level of a user, as chosen by the game system. d is a dictionary of things that can be changed (only the changed items are defined) :

- Player level in the game
- Player ability in a scale, as defined by developers
- How long the user exists

def update_problem_info(realm_id, problem_id, d)
d is a dictionary containing:

- Problem level as defined by developers

def get_latest_solution_date(realm_id)
Returns a datetime in UTC.
To know how old the data is - did the server get stuck, or is it fresh?

def get_user_info(realm_id, user_id)
Returns:

- Datetime in UTC of when the solution was computed.
- User rank in the system, as percentile, with an error estimate. Example: 10-15% rank.
- Floating-point rank, with no special semantics attached, except that it gives the same ordering as the percentiles above.
- N. of problems the user played.

Not necessary to be provided from our side
(
def get_play_info(realm_id, user_id, problem_id)
Returns:

- **The score of the play**
- **The time it took for the play**
- **The time when the play took place**
- **Play level: how far the user has gone**

def get_user_problems(realm_id, user_id)
Returns the list of problems the user played.

def get_problem_users(realm_id, problem_id)
Returns the list of players that played a given problem.
)

def get_problem_info(realm_id, problem_id)
Returns (in general, return a dictionary):

- Datetime in UTC of solution
- Rank and error in rank of problem among problems

- N. of users who played the problem
- Average rank in game of players who played the problem...
- How many players gave up
- Average rank of players who gave up...

def add_outcome(realm_id, user_id, problem_id, invariant_id, d)
d:

- score of solution
- duration of solution attempt
- gave_up: boolean flag indicating whether a solution was entered
- List of (start_time, end_time) of all time spans when the game was played
- Platform on which game was played (phone, tablet, PC, etc)

def delete_outcome(realm_id, user_id, problem_id, invariant_id)

def delete_all_outcomes_for_problem (realm_id, problem_id)

def update_user_info(realm_id, user_id, d)
d:

- Player level in the game
- Player ability in a scale, as defined by developers
- How long the user exists

def update_problem_info(realm_id, problem_id, d)
d:

- Problem level as defined by developers

def get_latest_solution_date(realm_id)

def get_user_info(realm_id, user_id)
Returns:

- Datetime in UTC of when the solution was computed.
- User rank in the system, as percentile, with an error estimate. Example: 10-15% rank.
- Floating-point rank, with no special semantics attached, except that it gives the same ordering as the percentiles above.
- N. of problems the user played.

def get_problem_info(realm_id, problem_id)
Returns (in general, return a dictionary):

- Datetime in UTC of solution
- Rank and error in rank of problem among problems
- N. of users who played the problem
- Average rank in game of players who played the problem...
- How many players gave up
- Average rank of players who gave up...

CHEKOFV Final Report

Appendix 4

CyphrSeedr Tutorial Design Document

Lauren Scott

1

CyphrSeedr Tutorial Design Document

Table of Contents

Overview

The aim of the CyphrSeedr tutorial is to introduce the basics of the game in a logical, coherent progression. The most important aspects of the game to be introduced here are the most basic invariants, which will reflect the more complex invariants that could be introduced in the game via "live" code, as well as the primary and secondary UI elements of the game.

Concepts to Teach Within the Tutorial

1. Preliminary User Interface
    a. How to use the slider & its representation
    b. What each symbol (i.e., Height, Leaves) means & represents
    c. How to use the toolkit/workspace
        - The less-than tool <
        - The greater-than tool >
        - The equals tool =
        - The plus tool +
        - The minus tool -
        - The constant tools 1, 100, 0, etc.
        - The variable tools (symbols)
        - The structures for the tools (Scratch-like templates)
        - The recycling bin
        - The trash (red?) button
    d. The star scoring system

2. Secondary User Interface
    a. The data overlay tools

3. Pattern finding for loop invariants
    a. One variable changing in a constant way
    b. One variable changing in a not-constant way
    c. Two variables changing in a not-constant way (no relationship)
    d. Two variables changing together in a constant way
    e. Linear relationships between two variables

4. Different Data Structures
    a. Simple integer relationships
    b. One-dimensional arrays
    c. Two-dimensional arrays
    d. Linked lists with integer data
    e. Linked lists with other data
    f. Stacks
    g. Queues
    h. Trees

5

Sample Levels

Sample Level 1

Code Sample:

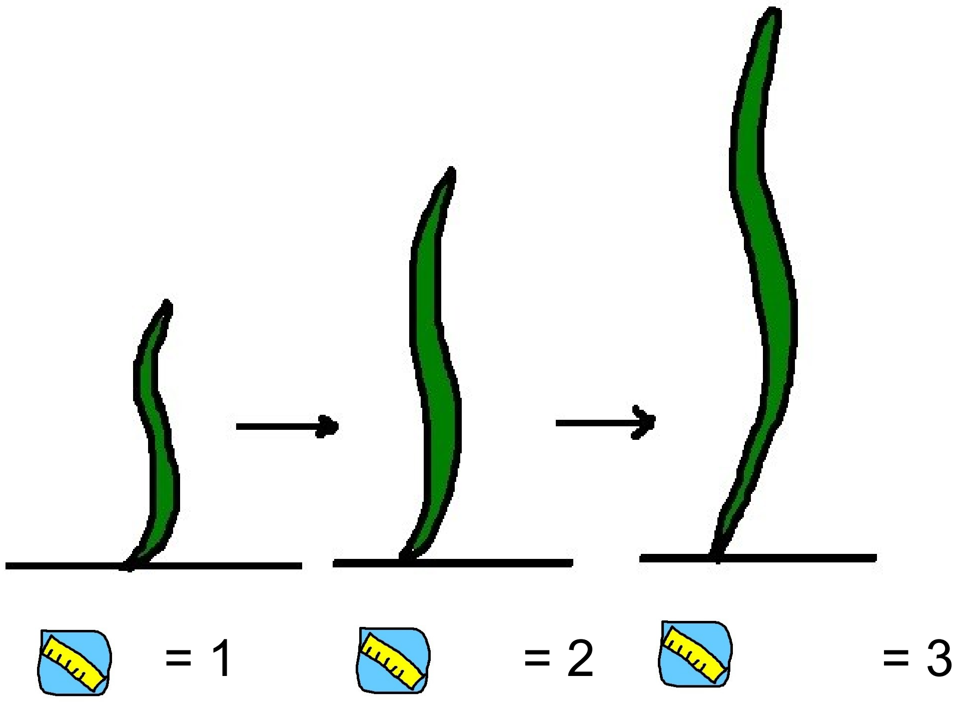
```
i = 1;
while (i < 5 && i > 0) {
        i++;
}
```

Invariant(s):

1. i >= 1
2. i <= 5

Representation:

i = Height,



Concepts to be Conveyed: Because this is the first level, it is necessary to introduce more than one basic concept; this will not be the case for the rest.

1. The slider & its representation
2. One variable changing in a constant way
3. The < and/or > and/or = tools
4. The variable tools

Sample Level 2

Code Sample:

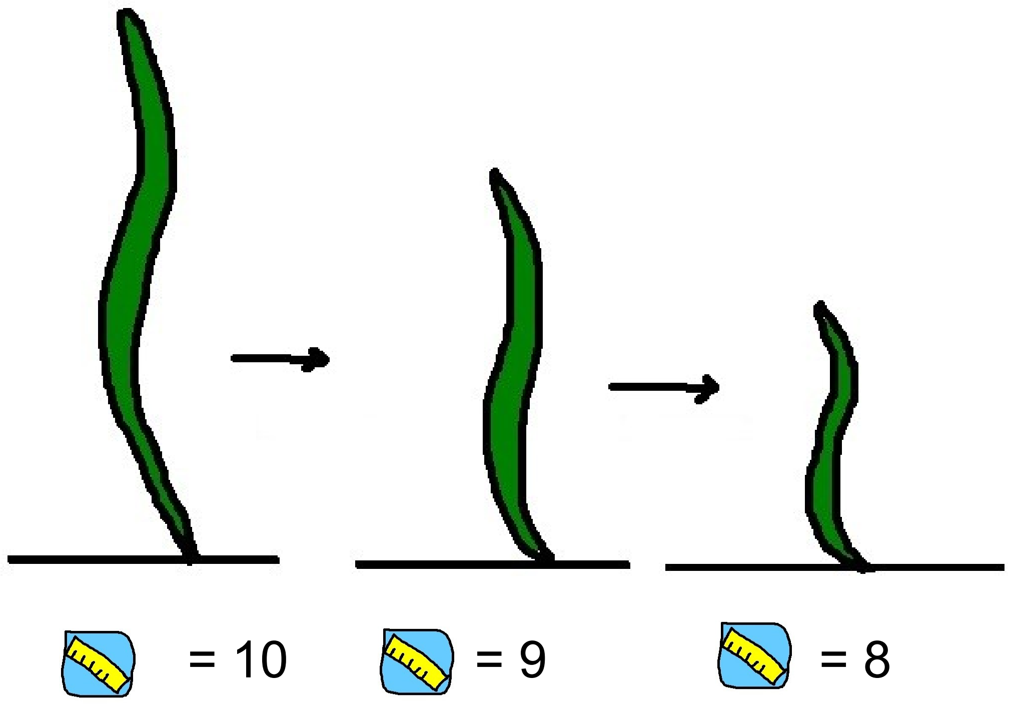
```
i = 10;
while (i > 0 && i < 15) {
        i--;
}
```

Invariant(s):

1. i >= 0
2. i <= 15

Representation:

i = Height, 📏



Concepts to be Conveyed:

1. One variable changing in a constant (but different) way.

Sample Level 3

Code Sample:

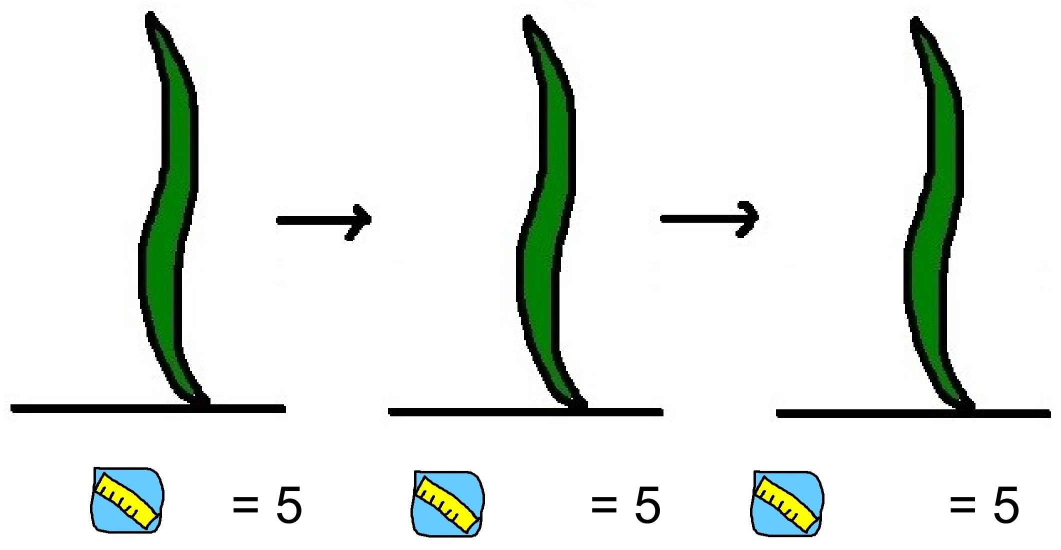
```
i = 5;
while (i > 0 && i < 5) {
        i = 5;
}
```

Invariant(s):

1. i = 5

Representation:

i = Height,



Concepts to be Conveyed:

1. The notion of a constant variable.

Sample Level 4

Code Sample:

```
i = 1;
y = 1;
while (i > 0 && y > 0) {
        i++;
        y++;
}
```

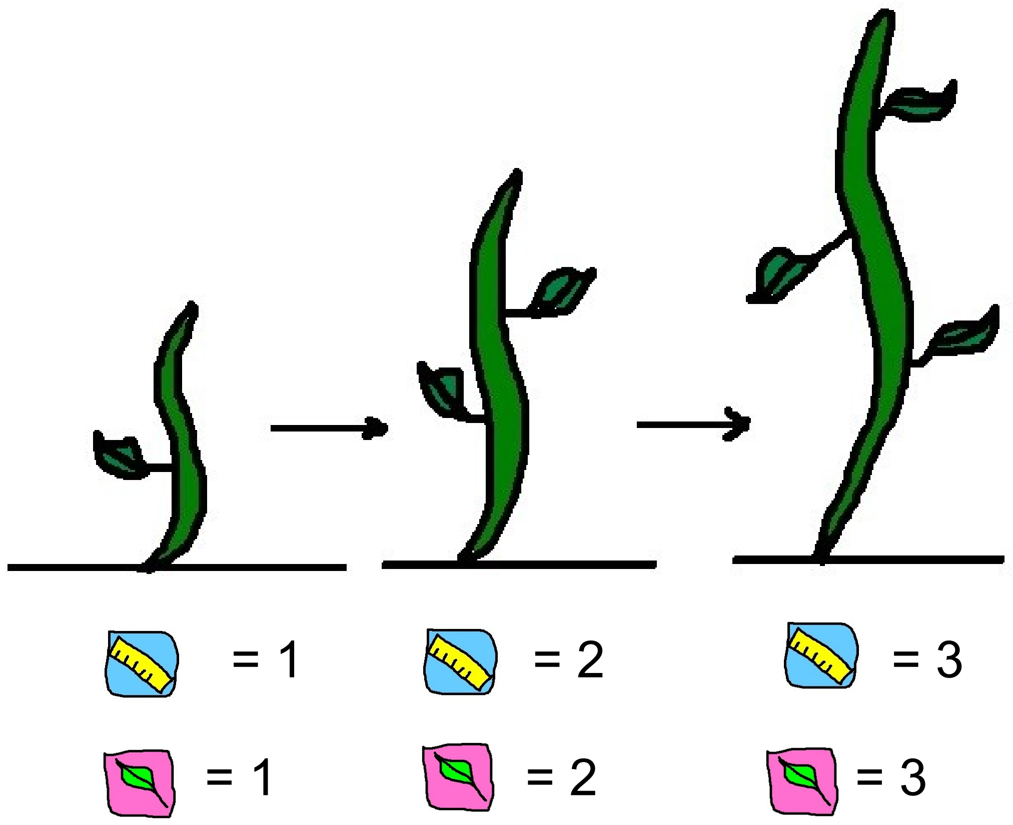
Invariant(s):

1. i = y

Representation:

i = Height,

y = Leaves



Concepts to be Conveyed:

1. Two variables.

Sample Level 5

Code Sample:

```
i = 1;
y = 0;
while (i < 5 && i > 0) {
        i++;
        y = 2i;
}
```

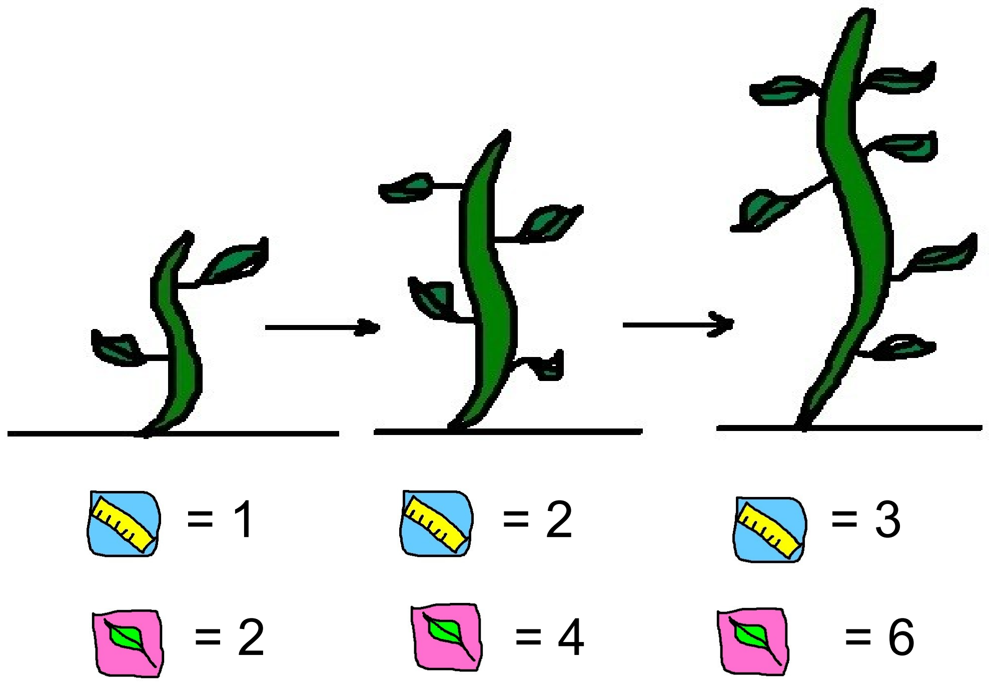
Invariant(s):

1. $y = 2i$
2. $i < 5$
3. $i > 0$
4. $y > 0$

Representation:

i = Height,

y = Leaves



Concepts to be Conveyed:

1. Two variables changing in a different constant way.

Sample Level 6

Code Sample:

```
i = 5;
y = 0;
while (i < 6 && i > 0) {
        i--;
        y ++;
}
```

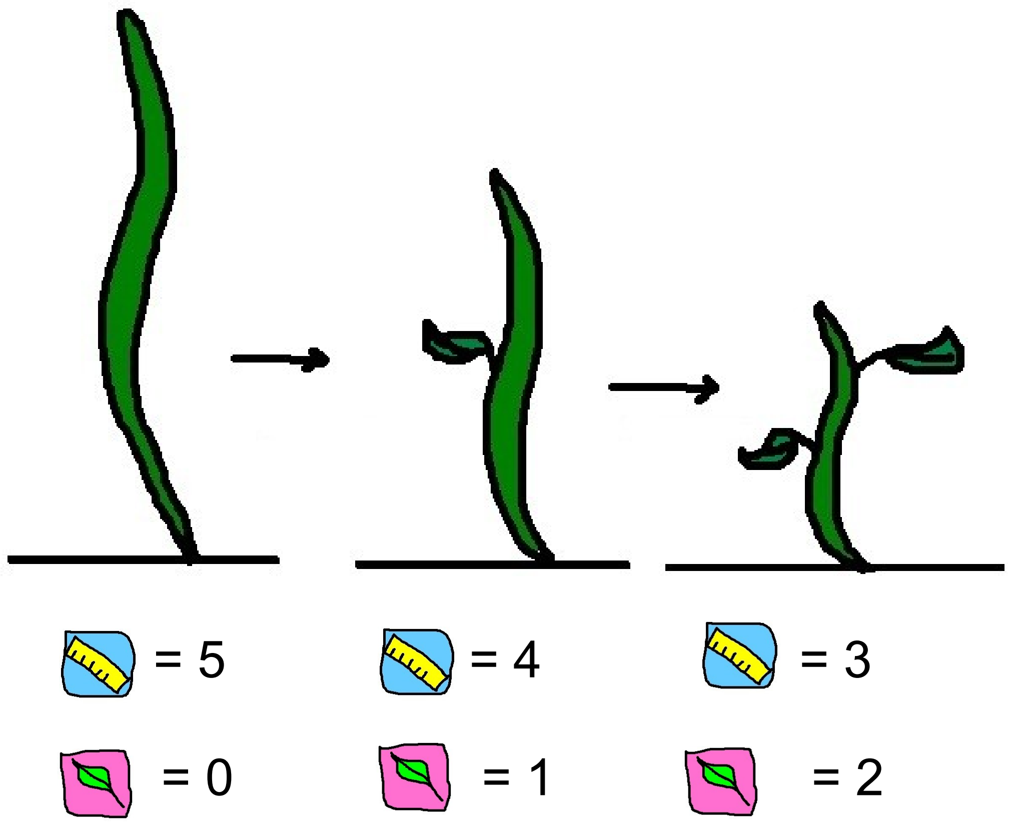
Invariant(s):

1. i > 0

Representation:

i = Height, 📏

y = Leaves 🍃



Concepts to be Conveyed:

1. Two variables changing in a different constant way.

Sample Level 7

Code Sample:

```
i = 0;
y = 0;
while (i < 5 && i > 0) {
        i++;
        y = i²;
}
```

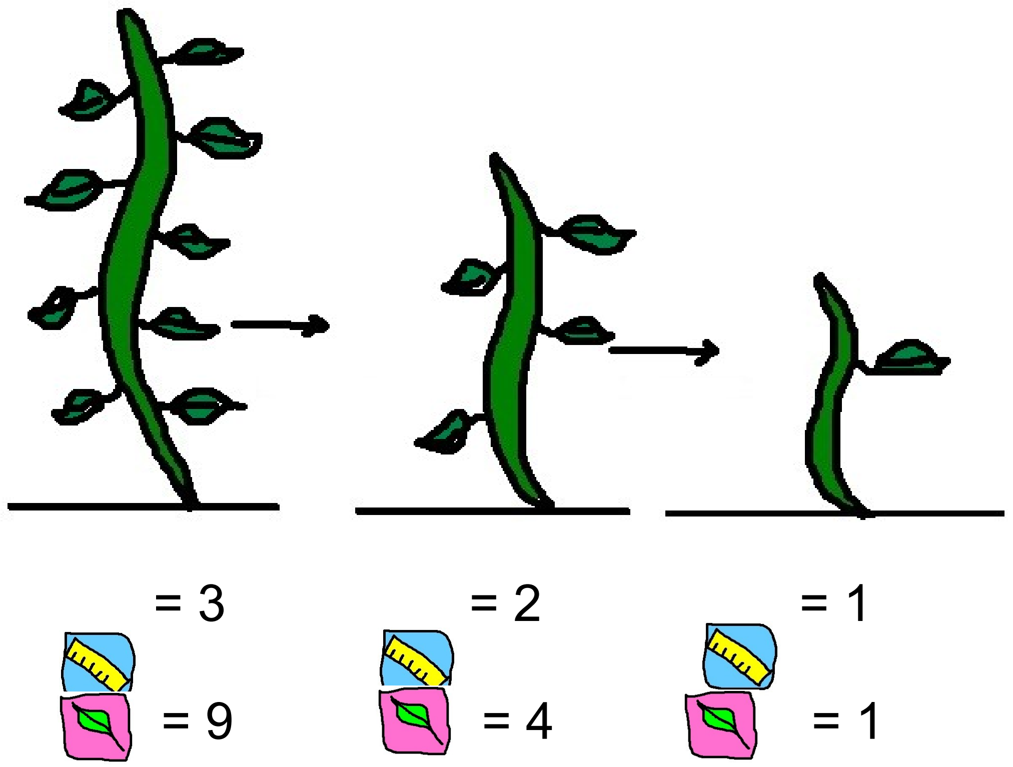
Invariant(s):

1. $y = i^2$
2. $i < 5$
3. $i > 0$

Representation:

i = Height,

y = Leaves



Concepts to be Conveyed:

1. Negative numbers.

Sample Level 8

Code Sample:

```
i = 0;
y = 0;
while (i < 5 && i > 0) {
        y = 2i + 5;
        i++;
}
```

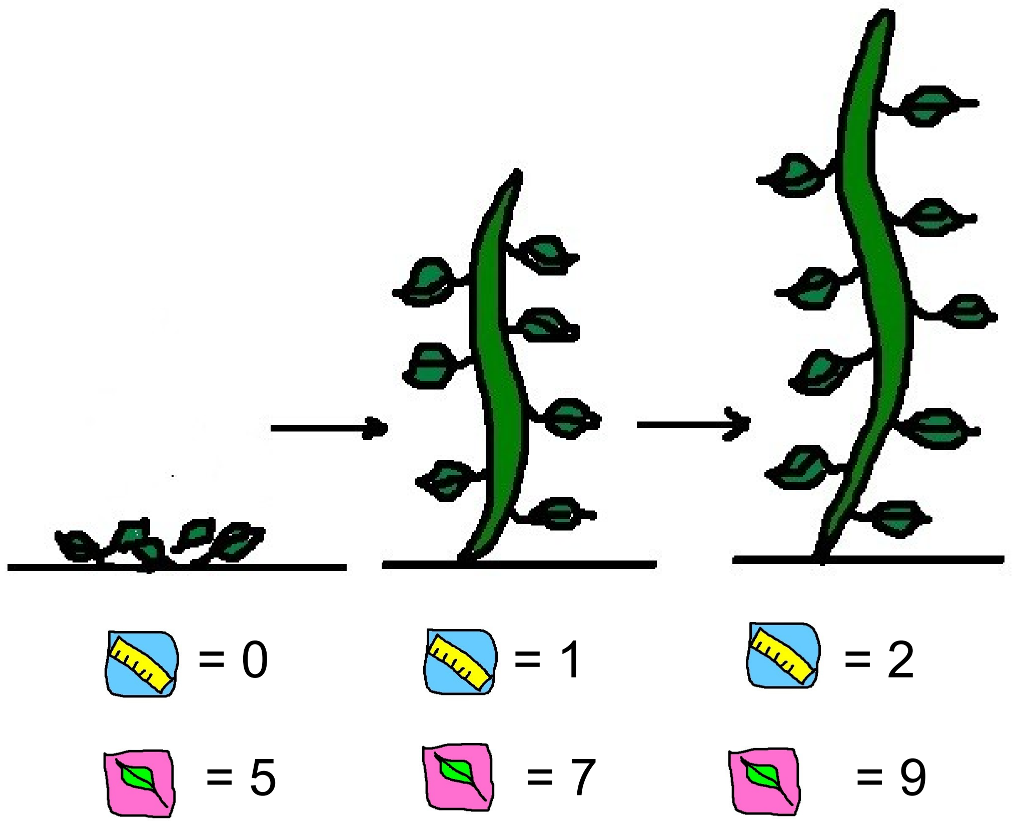
Invariant(s):

1. y = 2i + 5

Representation:

i = Height,

y = Leaves



Concepts to be Conveyed:

1. Polynomial relationships.

Sample Level 9

Code Sample:

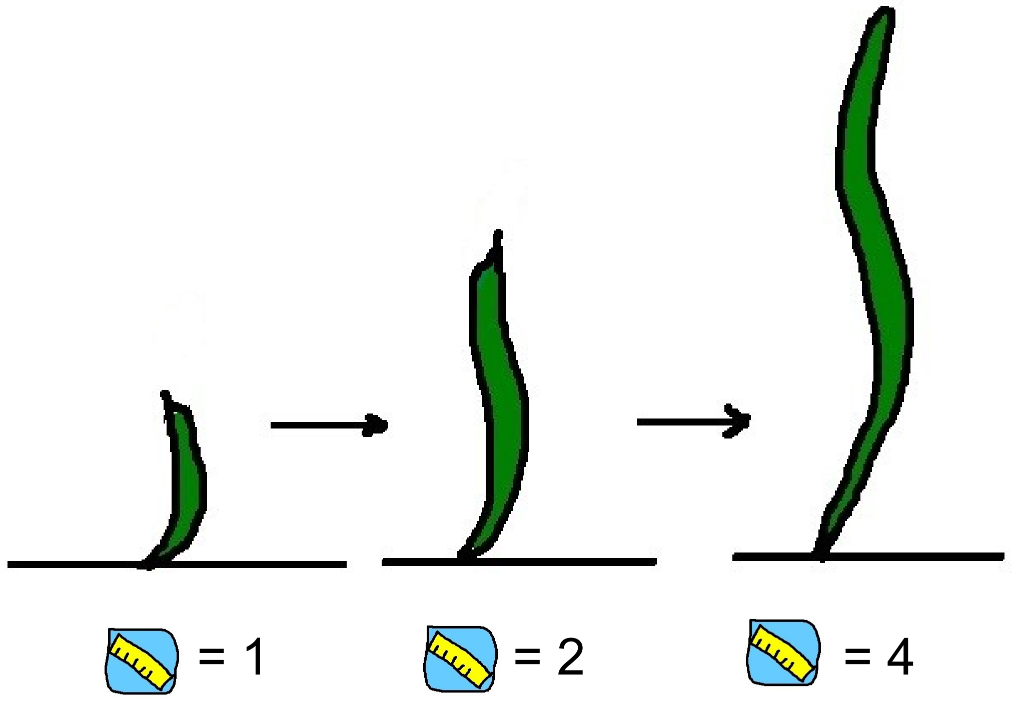
```
i = 1;
while (i < 5 && i > 0) {
        i = 2i;
}
```

Invariant(s):

1. i = 2i

Representation:

i = Height,



Concepts to be Conveyed:

1. One variable changing in a different constant way.

Sample Level 10

Code Sample:

```
i = 1;
while (i < 5 && i > 0) {
        i = 2i;
}
```

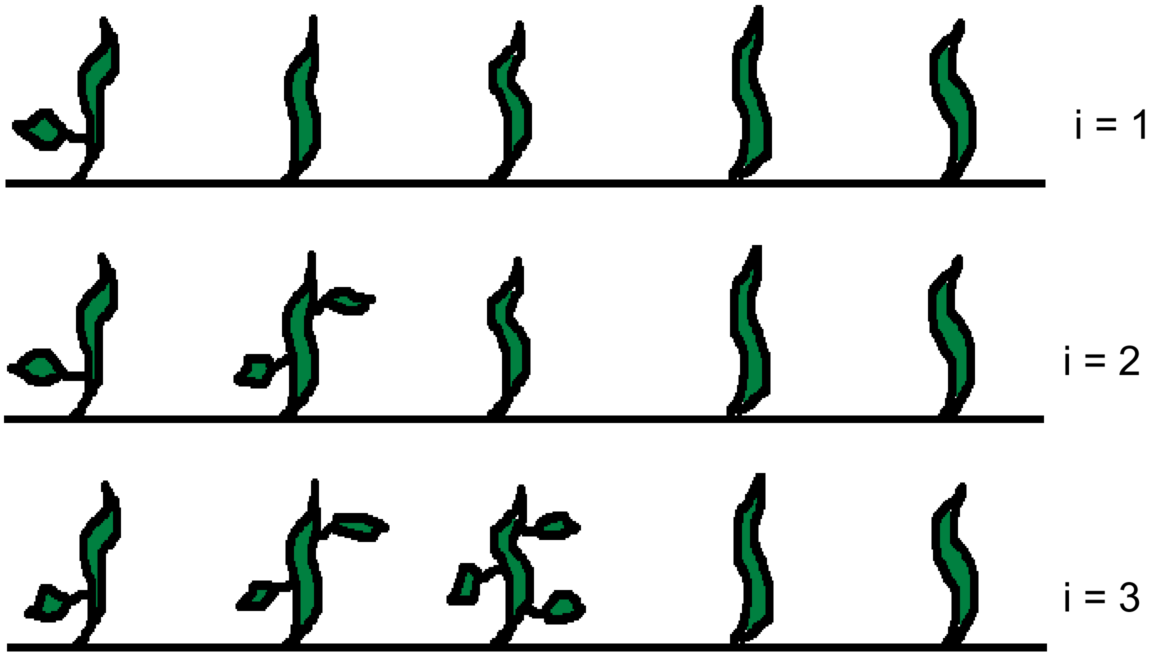
Invariant(s):

1. a[i] = i;

Representation:

Array is a rhizomatous plant (a plant with multiple identical sprouts, or nodes, whose characteristics may vary).

a[i] = ith sprout's number of leaves,



Concepts to be Conveyed:

1. Arrays
    - New tool(s)
    - Visual representation

CHEKOFV Final Report

Appendix 5

Crowdsourcing Program Preconditions via a Classification Game

Fava D, Shapiro D, Osborn J, Schäf M, & Whitehead EJ.

Crowdsourcing Program Preconditions via a Classification Game

Daniel Fava
University of California
Santa Cruz
dfava@soe.ucsc.edu

Dan Shapiro
University of California
Santa Cruz
dgs@ucsc.edu

Joseph Osborn
University of California
Santa Cruz
jcosborn@soe.ucsc.edu

Martin Schäef
SRI International
martin.schaef@sri.com

E. James Whitehead Jr.
University of California
Santa Cruz
ejw@soe.ucsc.edu

ABSTRACT

Invariant discovery is one of the central problems in software verification. This paper reports on an approach that addresses this problem in a novel way; it crowdsources logical expressions for likely invariants by turning invariant discovery into a computer game. The game, called Binary Fission, employs a classification model. In it, players compose preconditions by separating program states that preserve or violate program assertions. The players have no special expertise in formal methods or programming, and are not specifically aware they are solving verification tasks. We show that Binary Fission players discover concise, general, novel, and human readable program preconditions. This suggests that crowdsourcing offers a feasible and promising path towards the practical application of verification technology.

1. INTRODUCTION

A key problem in software verification is to find abstractions that are sufficiently precise to enable the proof a desired program property, but sufficiently general to allow an automated tool to reason about the program. Various techniques, such as predicate abstraction [2], interpolation [17], logical abduction [7], and lately machine learning (e.g., [20, 25, 11]) have been proposed to automatically find such abstractions by identifying suitable program invariants. Each of these techniques provides its own approach for *inventing* suitable predicates, but unfortunately, the space of possibilities is essentially infinite and it is not currently possible to find such predicates via automated methods.

The human process for finding invariants relies on highly skilled people, schooled in formal methods, to reason from the purpose of programs towards possible predicates. However, this approach has an issue of scale: millions of programs could benefit from formal verification, while there are only a few thousand such experts world-wide. Automated methods rely on search, and expectations to constrain the predicate invention process. White box techniques leverage knowledge about program content to propose candidate invariants, while black box methods search a space of templates (often boolean functions of linear inequalities) using comparatively little knowledge of program structure.

Recent work on classification techniques employ data to constrain predicate invention. Here, the objective is to induce a boolean expression over a base set of predicates that admits "good" program states (inputs that satisfy desired properties encoded as assertions) while excluding all "bad" states (input that violates such assertions on execution). Machine learning methods are well-suited to this task [11, 13, 21, 20]. These techniques output likely invariants that can be tested by static or dynamic analysis methods to determine if they are invariant conditions of the underlying program. The key issue in this approach is generalization; useful invariants are broad statements while classification methods tend to overfit the data. Moreover, the data on good and bad program states necessary to achieve robust generalization is in short supply, as program sampling is itself a hard task.

This paper reports on a classification based system that addresses predicate invention in a novel way; it crowdsources logical expressions for likely invariants by turning invariant generation into a computer game. This approach has several potential benefits:

- It can take advantage of the human ability to extract general predicates from small amounts of data,
- It makes predicate invention accessible to a much larger pool of individuals,
- It allows the crowd to compose unexpected, likely invariants that fully automated methods might miss.

In more detail, the game, called Binary Fission, addresses the subtask of precondition mining; it assumes a set of annotations that encode desired properties, and seeks predicates that imply the annotations hold under program execution. Players function as classification engines, by collectively composing likely invariants without any awareness that they are performing program verification.

Binary Fission is an instance of a growing number of games with a purpose [4, 12, 23], which share the premise that many difficult and important tasks can be advanced by crowdsourcing [19]. As such, Binary Fission is an existence proof for crowdsourcing precondition mining. This paper also demonstrate that it is effective. We show that:

- The crowd employs Binary Fission to compose likely invariants for non-trivial programs.
- A subset of those preconditions are program invariants.
- The program invariants are non-trivial, reasonably general, and human readable.

In addition, we show that the results of Binary Fission are novel relative to the output of DTinv [13] (a related, fully automated classification system).

The following sections describe our approach and results. We begin by framing this effort against related work, and introducing Binary Fission. Section 4 introduces the domain program we examine for preconditions, and then discusses our methodology for assembling crowdsourced likely invariants from player contributions, extracting program invariants from that set, assessing the quality of crowdsourced results. Section 5 presents results obtained with Binary Fission. Section 6 discusses the source of power behind these results, while Section 7 examines threats to validity. We end with concluding remarks.

2. RELATED WORK

The problem of finding suitable program invariants is a central part of formal verification research. Striking the balance between an abstraction that is sufficiently precise to prove a property and sufficiently abstract to reason about is what makes program analysis scalable. In static analysis, a variety of techniques exist to infer program invariants, such as CEGAR [2], Craig interpolation [17], or logical abduction [7]. However, these approaches have the inherent limitation that they rely on information generated from the source code of the analyzed program. If the needed invariant is a relation between variables that cannot be inferred from the source code, these techniques must fall back on heuristics or fail to compute an invariant.

As an alternative to static invariant discovery, we have seen an increasing activity in research on data driven approaches. A pioneer in this field is Daikon [9, 8, 10] which takes a set of good program states as input and applies machine learning to find an invariant that describes all states in this set. More recently, several approaches [11, 13, 21, 20] have extended this idea by learning invariants applying different machine learning algorithms and by also considering sets of bad states that should be excluded by a likely invariant. The benefit of machine learning or data driven approaches over static invariant discovery is that these approaches can search for invariants in a larger space and discover invariants even if they are based on relations that are not easily inferred from the program text. This paper explicitly compares results obtained by Binary Fission with results obtained through DTinv [13], which provides a classification model that is very close in spirit to our work.

Since Binary Fission is a crowdsourcing game, it can viewed as a game with a purpose (GWAP) [24]. Since Binary Fission involves people performing work that computers cannot, it can also be viewed as a form of human computation (see [1] for design issues concerning motivation and evaluation in this context, and [16] for a survey of crowdsourcing in software engineering). Since Binary Fission uses a game reward system to motivate players, it is a form of gamification [6]. We view Binary Fission as a deeper application of game design principles than typical in gamification efforts, as it simultaneously makes a hard science problem playable, and disguises the core activity more than typical human computation tasks.

Overall, the idea of building crowdsourced games for hard scientific tasks has shown enough promise to motivate a large investment in this area. Binary Fission was developed as part of the Crowd Sourced Formal Verification (CSFV) program, funded by DARPA in the United States. This program has resulted in the creation of ten games focused on the intersection with formal software verification; a summary of the games developed in this program can be found in [5], and many of the games can be played at `verigames.com`.

3. BINARY FISSION

Binary Fission is a game for crowdsourcing program invariants. It is one of several recent efforts designed to exploit the "wisdom of the crowd" by transforming hard scientific problems into games [4, 12, 23]. Binary Fission is intended for players with no expertise in formal verification methods, and the players are at most peripherally aware that they are solving verification problems through game play.

The game employs a classification metaphor for finding invariants. At the technical level, it inputs a program annotated with postconditions, a set of predicates relating program variables, and two sets of initial program states (each state is a vector of variable values), where "good" states satisfy the assertions, and "bad" states violate those assertions on program execution. Each Binary Fission player employs the available predicates to find a classification tree that separates good data from bad. This tree defines a logical formula representing a likely invariant.

At the game level, Binary Fission hides the nature of the program, data, and predicates from the player. Instead, it presents players with a set of gold and blue "quarks" (representing good and bad data, internally), mixed together inside the nucleus of an "atom." The player's goal is to separate the gold from the blue quarks using a set of filters (corresponding internally to predicates), which are capable of splitting the atom's nucleus. Different filters create different splits, and the player's job is to decide which filters to apply, and in what order. The recursive application of filters leads to the creation of a binary tree, as shown in Figure 1.

Binary Fission imposes a 5 level depth limit on player generated classification trees, which bounds the complexity of the resulting classifiers. The game also provides a scoring function (shown in Equation 1) that influences players to create leaf nodes composed purely of good, or bad program states (where the pure good nodes have special utility for defining likely invariants).

$$N \times \sum_{i \in \text{leaf nodes}} \left(purity_i^A \times size_i^B \right) \qquad (1)$$

Here, *purity* is the maximum over the percentage of good states and the percentage of bad states in the node, and *size* is a count of the quarks (states) in the node. *A* and *B* are arbitrary constants. *N* is constant that increases with the count of pure nodes, and decreases with maximum depth of

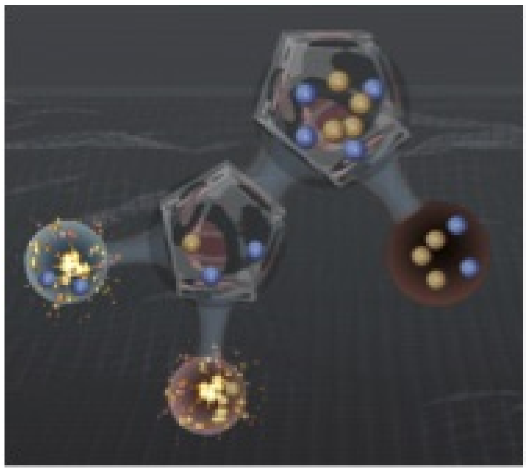


Figure 1: Binary Fission Player Interface

the classification tree. It influences players to produce as many pure nodes as possible, as early as possible, which is a force towards producing useful, and general descriptors.

Each classification tree produced through Binary Fission is typically partial: some leaf nodes only contain good states, some only contain bad states, while others contain a mixture. In addition, the solutions are idiosyncratic, as the players generally employ different subsets of filters during game play. As a result, the game software combines descriptions of pure good nodes and pure bad nodes across solutions to obtain a consensus view of the likely invariant. We discuss this process below.

4. METHODOLOGY

Our methodology for crowdsourcing precondition discovery repeats the following steps:

1. Express an invariant generation task as a data classification problem.
2. Present the problem to Binary Fission players.
3. Assemble a likely invariant across player solutions.
4. Extract clauses from the likely invariant that satisfy program assertions.
5. Assess utility of the program preconditions found.

Following these steps, we assess the value added by crowdsourcing invariants by comparing the results with the solutions produced via an automated classification technique, called DTInv [13]. The following sections clarify these tasks, after introducing the domain problem we employ as a source of invariant generation tasks.

4.1 The TCAS Program

TCAS is an aircraft collision avoidance software originally created at Siemens Corporate Research in 1993. It has become a common subject to verification methods and test case generation systems since being incorporated into the Software-artifact Infrastructure Repository [18]. The code performs algebraic manipulations of 12 integer variables and a constant four element array. It contains nested conditionals and logical operators; there are no loops, dynamic memory allocations or pointer manipulation.

TCAS is written in 173 lines of C code split into nine functions. As shown by the call graph in Figure 2, the main function calls an initialization routine before transferring control to `alt_sep_test`, which tests the altitude separation between an aircraft and intruder that has entered its protected zone. TCAS then generates warnings, called "Traffic Advisories' (TAs), and recommendations, called "Resolution Advisories" (RAs), to the pilot. The TAs alert the pilot of potential threats, while the RAs are proposed a maneuver meant to safely increase the separation between planes.

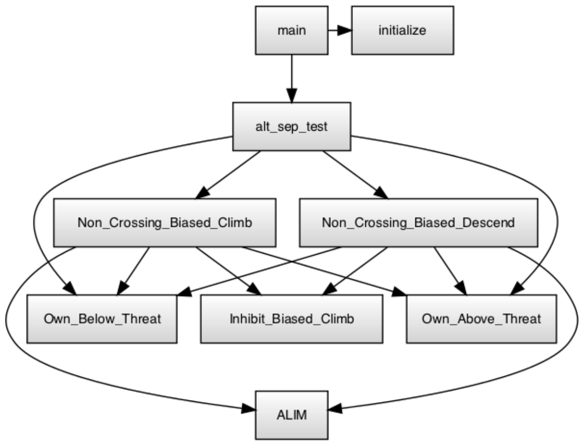


Figure 2: TCAS call graph.

A theory for avoiding aircraft collisions determines when certain maneuvers are safe; these conditions identify safety properties that the TCAS implementation should ideally guarantee. Table 1 illustrates some of these safety properties (reproduced from [3]). For example, the last two entries specify that a maneuver that reduces the separation between two planes must never be issued when the planes have intruded into each others' protected space. These safety properties can be encoded as postconditions of the TCAS program, via assertion statements at its end. The problem of proving the TCAS program safe translates into the task of verifying that the implementation cannot violate these assertions.

We tackle a subtask of the verification process, which is, to find suitable preconditions for TCAS functions. Function preconditions are conditional statements about program variables; if they hold on input to the function, program execution is guaranteed to produce the postconditions that encode desired properties.

4.2 Framing Binary Fission Problems

We define seven precondition finding tasks from the TCAS code. They are to discover preconditions for each of the functions *alt sep test*, *Non Crossing Biased Climb*, *Non Crossing Biased Descend*, *Own Below Threat*, *Inhibit Biased Climb* and *Own Above Threat* as shown in Figure 2, where those preconditions ensure the conjunction of program postconditions illustrated in Table 1.

We express these problems as Binary Fission classification

	Postcondition	Explanation

If
 	
Assert
	`Up_Separation` $\geq$ `Positive_RA_Alt_Thresh[2]` $\wedge$
	`Down_Separation` $<$ `Positive_RA_Alt_Thresh[2]`
	`result` $\neq$ `need_Downward_RA`
		A downward RA is never issued if a downward maneuver does not produce adequate separation

If
 	
Assert
	`Up_Separation` $<$ `Positive_RA_Alt_Tresh[2]` $\wedge$
	`Down_Separation` $\geq$ `Positive_RA_Alt_Tresh[2]`
	`result` $\neq$ `need_Upward_RA`
		An upward RA is never issued if an upward maneuver does not produce adequate separation

If
Assert
	`Own_Tracked_Alt` $>$ `Other_Tracked_Alt`
	`result` $\neq$ `need_Downward_RA`
		A crossing RA is never issued

If
Assert
	`Own_Tracked_Alt` $<$ `Other_Tracked_Alt`
	`result` $\neq$ `need_Upward_RA`
		A crossing RA is never issued

If
Assert
	`Down_Separation` $<$ `Up_Separation`
	`result` $\neq$ `need_Downward_RA`
		The RA that produces less separation is never issued

If
Assert
	`Down_Separation` $>$ `Up_Separation`
	`result` $\neq$ `need_Upward_RA`
		The RA that produces less separation is never issued

Table 1: TCAS postconditions.

tasks by specifying {*good states*, *bad states*, *predicates*} tuples. We obtain the state data by running a large set of test cases on the underlying program and monitoring its execution with a debugger. The TCAS repository supplies test cases with the code. We collect the program state at the entry point of each function, and monitor the program's exit status. If the input state satisfies end assertions we add that vector of program variables to the *good* states. If it violates assertions or causes the program to crash, we add it to the set of *bad* states. We augment these states by randomly sampling the variable ranges observed in the program test cases, after validating with gcov [22] that the new values exercise the same code paths. We retain these states in a hold-out set for testing the generality of any preconditions found, and do not present them to players.

Binary Fission can utilize logical predicates of any kind, obtained from any source, with the caveat that they need to be relevant to the classification task at hand in order to be useful. Because TCAS performs algebraic manipulations, we generate a base set of predicates by employing the Daikon system [10], which is able to explain regularities in program states by searching a library of structural forms. In particular, we supply Daikon with a small subset of good TCAS program states (and separately, a small set of bad states), and collect the candidate invariants it produces. For TCAS, this set consists of several hundred boolean combinations of equalities and inequalities among linear functions of 1-4 variables, including max and min operators, numeric thresholds, and explicit set membership tests.

We present each of the {*good states*, *bad states*, *predicates*} tuples generated in this way to multiple Binary Fission players who generate preconditions as a byproduct of game play. Binary Fission is available on-line at `http://binaryfission.verigames.com`, and we invite readers to try it. To date, close to one thousand players have generated about three thousand solutions for TCAS problems.

4.3 Assembling a Likely Invariant

Each classification tree generated by a Binary Fission player separates program states into a collection of Pure Good, Pure Bad, and Impure nodes (where a Pure node only contains program states of one kind). As shown in Figure 3, a conjunction of predicates that links the root to a Pure Good node describes a set of states that satisfy program assertions, and expresses a likely invariant. A single player solution can contain several such paths. By extension, we define the disjunction of paths to Pure Good nodes across all player solutions as the consensus, likely invariant. This results in an expression in Disjunctive Normal Form:

$$PureGoodConjunct_1 \vee ... \vee PureGoodConjunct_n$$

Note that the individual conjuncts might be drawn from the same or different classification trees. As a result, the conjuncts might not employ the same variables, or be mutually exclusive either as logical statements or in terms of the data they explain.

It is tempting to employ the negation of predicates describing Pure Bad nodes across players instead, since an invariant that excludes Pure Bad states is potentially weaker, and more desirable than an invariant that explicitly admits only good states. However, given a partial classifier, the logical expression $\neg(PureBadConj_1 \vee ... \vee PureBadConj_m)$ includes Impure nodes, and accepts bad states that cannot be admitted by any invariant.

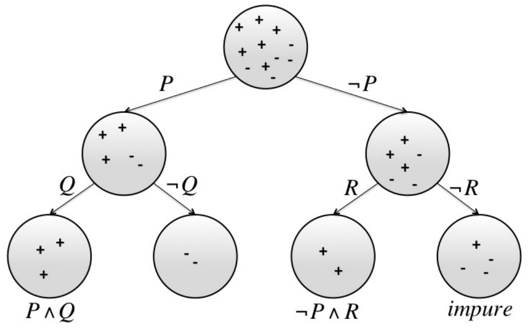


Figure 3: Example of a decision tree produced by Binary Fission. Tracing from the root node to the two pure positive nodes we have $P \wedge Q$ and $\neg P \wedge R$ which form the candidate invariant $(P \wedge Q) \vee (\neg P \wedge R)$.

4.4 Extracting Program Invariants

Given a likely invariant expressed in DNF, we use the CBMC bounded model checker [14] to identify any component conjuncts that qualify as program preconditions. That is, if $c_1 \vee c_2 \vee ... \vee c_n$ is a predicate derived from data points from function `my_func`, we consider each clause c_i for $i \in \{1, 2, ..., n\}$ in turn. We place a check of its negation at the entry of the function as shown on line 2 of Figure 4. We then run CBMC on this modified program. When CBMC

```
1 my_func(args) {
2   if !(c_i) { exit(0) }
3   // Remainder of the function...
4 }
5
6 my_func(args)
7 assert(postcondition)
```

Figure 4: Pseudocode showing program transformation for discovering function preconditions.

encounters the if-statement, it splits the analysis between the two paths. The path in which c_i is falsified *dies* when it encounters `exit(0)`. On the other hand, when c_i is satisfied, the analysis continues and the model checker attempts to find function arguments `args` that will later cause postcondition violations (line 7 of Figure 4). If CBMC cannot find inputs that satisfy c_i and violate the postconditions, then c_i is a precondition of the function. The full Binary Fission invariant is the disjunct of all clauses that satisfy this test.

4.5 Assessing Invariant Utility

Assuming Binary Fission players discover likely invariants and program preconditions, the next key concern is to evaluate the usefulness of those expressions. We would like to show that crowdsourced results enable further derivation of program properties, or facilitate practical application of the code. We do not yet have that result for Binary Fission. Instead, we assess the generality of the crowdsourced expressions by measuring their coverage against data. The more data explained, the weaker the likely invariant or precondition, and the more utility it offers for further analyses.

Binary Fission relies on a classification technique to separate good states from bad. However, classification methods are prone to overfitting; they must guard against the tendency to explain exactly and only the training data, without providing insight into the general case represented by the data *not* seen. Common defenses include penalizing overly complex expressions considered during classification, and testing against held back data to ensure the generality of the induced function. We utilize both techniques here. In particular, we rely on the Binary Fission scoring function and depth limit to prevent overfitting, and we distinguish training data from test sets.

In more detail, we measure expression generality against a set composed of Good program states. To increase the amount of data available, we interpolate between good states supplied with the TCAS code, and ensure that new states exercise the same code paths as the original states. We measure coverage of likely invariants against the training set, and coverage of preconditions against this new data, which comprises the test set.

4.6 Assessing Invariant Novelty

In addition to assessing the utility of any invariants found, we examine the conjecture that crowdsourced invariants are novel relative to the results obtained through other methods. If they are novel, it is an indication that crowdsourcing brings some special leverage to the task, and we can analyze the source of that power.

We assess novelty by comparing Binary Fission results against the output of the DTinv system [13], which is a fully automated classifer. Many invariant learners now exist, but DTinv is possibly the closest in spirit to our work. Like Binary Fission, DTinv builds a decision tree from good and bad program states (that preserve or violate end assertions), plus a set of primitive predicates that relate program variables. The key differences are that DTinv builds its own predicates from a basis set of planar cuts using the octagon abstract domain (vs importing an arbitrary predicate set), and it constructs decision trees of arbitrary depth that perfectly classify the data into Pure Good and Pure Bad sets (vs the partial classifiers of bounded depth produced by Binary Fission).

We apply DTinv to each of the TCAS problems given to Binary Fission players, and we compare the resulting likely invariants for legibility, generality in terms of data coverage, and veracity as program preconditions. To make the comparisons fair, we pre-process the TCAS code to represent arrays (which DTinv cannot currently consume) as separate variables. In addition, rather than test the DTinv solution as a whole for its status as a program precondition, we transform it into Disjunctive Normal Form and test individual disjuncts as candidate preconditions via the CBMC model checker. This approach is symmetric with our examination of disjuncts describing Pure Good nodes in the partial classifiers output by Binary Fission.

We compare the generality of the likely invariants and preconditions found by measuring their coverage of program states, as before.

5. BINARY FISSION RESULTS

Following the methodology described in the previous section, we collected crowdsourced solutions for the seven TCAS problems identified in Section 4.2. For purposes of illustration, we discuss the solution for the TCAS function *Non Crossing Biased Descend* in detail, and then summarize across the remaining six examples. We discuss the structure and coverage of the likely invariants found, we identify the valid program preconditions, and we evaluate the generality and data covered by these results. We assess novelty through comparison of the Binary Fission and DTinv solutions for the same problem.

5.1 Likely Invariants for TCAS Problems

The consensus solution for *Non Crossing Biased Descend* has 398 disjunctive clauses that represent the Pure Good nodes found across Binary Fission players. Each clause is a likely crowdsourced invariant. Figure 5 illustrates the top three, measured by their coverage over program states. Their content is syntactically similar; each clause is a conjunct of 2-3 primitive predicates (shown as top-level ANDs), where the primitives express numeric equalities and inequalities over multiple TCAS variables. These are non-trivial statements about domain variables, and they appear reasonably general; they clearly do not pick out specific data values. Following the methodology described in Section 4.5, we measure the generality of these expressions by their coverage of the training data; they each explain circa 30% of the good program states. The three likely invariants also appear

```
(not(Other_Capability > Two_of_Three_Reports_Valid))
    and (not(Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value]))

(not(Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value]))
    and ((Alt_Layer_Value <= size(Positive_RA_Alt_Thresh)-1))

(not(Alt_Layer_Value >= Up_Separation))
    and (not(Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value]))
    and ((Cur_Vertical_Sep != Positive_RA_Alt_Thresh[Alt_Layer_Value]))
```

Figure 5: The best three likely invariants measured by Good state coverage.

to be describing a similar truth, as they utilize many of the same variables and terms. As a result, they can describe many of the same states.

The solutions for all seven TCAS problems have a similar structure. Table 2 shows that they contain between 262 and 704 clauses. These solutions are simple collections, and have not been simplified; they can overlap both logically and in terms of the data covered, and their number strictly grows with the quantity of game play.

5.2 Crowdsourced Solution Progress

Figure 6 illustrates the crowd's progress towards finding a consensus likely invariant. It plots cumulative data explained by the crowdsourced solution, as accumulated in decreasing order of predicate quality (i.e., the number of good program states recognized by the conjunctive predicate associated with each Pure Good node). This figure supports several interesting observations. First, the top 20% of the solutions explain 80% of the data, and this pattern repeats across all TCAS problems. This suggests a statistical regularity in crowd performance, and an uneven distribution of expertise across players. Second, the consensus solution is partial, meaning it fails to explain all the data even after incorporating every player's contribution. This is an expected result, as Binary Fission limits the depth of player classification trees – some truths are simply hard to express in bounded space.

In order to investigate this point further, we employed a greedy search algorithm to construct a classifier for the same problem, over the same primitive predicates. The method used average impurity for scoring splits. When invoked with a depth limit of 5, the resulting partial classifier explained 21 good program states. This splitting metric clearly provided insufficient motivation to distinguish Pure Good nodes early in the classification process that have utility for invariant generation. In contrast, the reward metric employed by Binary Fission clearly influenced players to isolate Pure Good nodes at shallower depths, with the associated benefit for explaining good program states. This pattern repeated across TCAS problems.

We also tested the expressive power of the primitive Binary Fission predicates by invoking the greedy classification algorithm without a depth limit. The result here, and in all 7 TCAS problems, was that the predicates had the power to correctly separate all good program and bad program states. As a result, our statistics on Binary Fission solutions concern the performance of the crowd, not the expressivity of the predicates at their disposal.

5.3 Program Preconditions Found

We tested the likely invariants generated for *Non Crossing Biased Descend* using the CBMC model checker as dis-

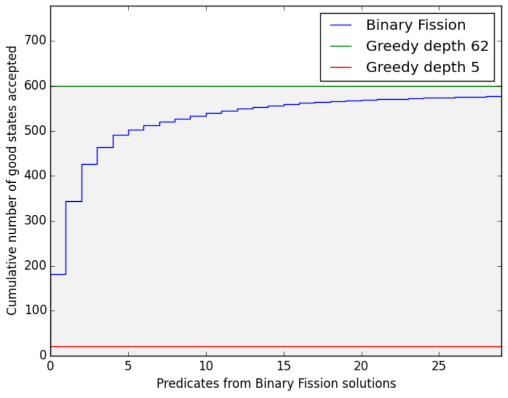


Figure 6: Crowd progress in classifying data points from `Non_Crossing_Biased_Descend`

cussed in Section 4.4. Of the 398 clauses supplied by players, 16 qualified as program preconditions. That is, if any of these preconditions hold on function entry, the postconditions described in Table 1 hold at program exit. Figure 7 lists the three most general preconditions found, ordered by their coverage over the test set of good program states. These are the first instances of program invariants found by crowdsourced methods. As with the likely invariants, these preconditions are non-trivial statements about domain variables, here relating the positions and capabilities of aircraft in the sky. For example, the first/best precondition in Figure 7 states that advising a pilot to descend (the function of *Non Crossing Biased Descend*) will satisfy safety assertions when (a) the other plane's altitude is higher, but (b) advising the pilot to climb will result in a vertical separation (*up separation*) that is less than the required tolerance.

Binary Fission players collectively found program preconditions for 6 of the 7 TCAS tasks. None were trivial. Table 2 identifies the quantity of preconditions found for each task, and the numbers are substantial.

5.4 Invariant Generality

Following the methodology described in Section 4.5, we assess the generality of the crowdsourced preconditions found by measuring their coverage over good program states in the test set. Table 3 counts the number of program states explained by for the seven TCAS problems. The best-case scenario is for the precondition to accept all good states. In the case of *Non Crossing Biased Descend*, the aggregate precondition (composed of the 16 clauses reported in Table 2)

```
(not(Other_Tracked_Alt > Own_Tracked_Alt))
    and (Up_Separation < Positive_RA_Alt_Thresh[Alt_Layer_Value])

(Other_Tracked_Alt > Positive_RA_Alt_Thresh[Other_Capability])
    and (Down_Separation >= Up_Separation)
    and (not(Up_Separation <= Positive_RA_Alt_Thresh[Alt_Layer_Value]))
    and (Other_Tracked_Alt > Own_Tracked_Alt)

(not(Other_Capability == 2))
    and (not((Down_Separation == 800) or (Down_Separation == 600)
                                      or (Down_Separation == 500)))
    and (Down_Separation != Positive_RA_Alt_Thresh[Alt_Layer_Value])
    and (not(Other_Tracked_Alt > Own_Tracked_Alt))
    and (Up_Separation < Positive_RA_Alt_Thresh[Alt_Layer_Value])
```

Figure 7: The three best crowdsourced preconditions found.

Function	Precon-ditions	Clauses from BF
ALIM	45	422
alt_sep_test	103	462
Inhibit_Biased_Climb	7	262
Non_Crossing_Biased_Climb	14	360
Non_Crossing_Biased_Descend	16	398
Own_Above_Threat	0	500
Own_Below_Threat	6	704

Table 2: Quantity of Crowdsourced Preconditions and Likely Invariants: A fraction of the likely invariants qualify as program preconditions.

Function	Good states	Total states	%
ALIM	51	95	53.7%
alt_sep_test	424	2000	21.2%
Inhibit_Biased_Climb	59	295	20.0%
Non_Crossing_Biased_Climb	60	295	20.3%
Non_Crossing_Biased_Descend	108	295	36.6%
Own_Above_Threat	0	161	0%
Own_Below_Threat	0	185	0%

Table 3: Testing preconditions' generality by comparing the number of good states accepted versus the total number of good states in the held-out test set.

explains 36.6% of the good program states withheld during the classification task. This corresponds to 2.3% of the good states per precondition clause on average, although the distribution was uneven. Figure 7 shows the best three preconditions for this problem. The first explained 53% of the data, while the second and third best preconditions captured 37% and 26% of the program states in the test set respectively. The net result is that the crowd discovers multiple program preconditions with noteworthy coverage/generality.

5.5 Novelty Relative to the DTinv Solution

As discussed in Section 4.6, we compare the Binary Fission and DTinv solutions for each TCAS problem in order to examine the conjecture that the crowd provides novel insight in the search for program invariants. We compare the legibility and coverage of the likely invariants they produce, as well as their ability to discover program preconditions.

In its raw form, the DTinv solution for *Non Crossing Biased Descend* is a depth 15 decision tree containing 65 primitive predicates that completely segments the good and bad program states. The corresponding logical expression is not human readable (nor was it intended to be). We converted this form to DNF to extract less monolithic likely invariants, and show the top three clauses (as measured by the number of Good states covered) in Figure 8.

It is immediately obvious that these expressions rely heavily on numeric thresholds. As mentioned earlier, this is by design, as DTinv's primitive predicates represent planar cuts in the octagon domain. Although it is an aesthetic judgment, this design appears to make the DTinv statements harder to interpret than the Binary Fission output in Figure 5.

Of the three DTinv expressions in Figure 8, the second overlaps the first, and the third is a specialization of the second. They cover 29%, 16%, and 11% of the Good program states, respectively. It is worth noticing that the single best likely invariant found by crowdsourcing (Figure 5) and the DTinv classifier have essentially identical capture, and that the top three employ the same variable set, though in notably different formulas. This is an indication that both systems are after similar insights.

We tested the DTinv solution for *Non Crossing Biased Descend* using the CBMC model checker to determine if it contained valid program preconditions. The surprising conclusion is that it did not, either as a whole, or when we tested individual DNF clauses. This pattern repeated across all seven TCAS problems; none of the DTinv solutions contained valid preconditions. In contrast, the crowd, acting through Binary Fission, produced preconditions for 6 of the 7 TCAS problems. As a result, the crowdsourced solutions are clearly novel relative to the DTinv output.

The cause for the lack of DTinv-based preconditions appears to be overfitting; numeric thresholds induced from data are highly likely to break in the presence of a holdback set, and the lengthy expressions DTinv discovers to explain all the training data have limited opportunity to generalize. In contrast, the more abstract predicate base and 5 conjunct limit imposed by Binary Fission essentially forces players to paint with a broader brush. Players can only produce shorter, more powerful statements, some of which generalize, as shown above.

6. DISCUSSION

This paper has addressed the problem of crowdsourcing program preconditions, under the model that crowdsourc-

```
(not(2*Positive_RA_Alt_Thresh[0] + 2*Down_Separation <= 1472))
and (2*Up_Separation -2*Alt_Layer_Value <= 801)
and (2*Alt_Layer_Value -2*Down_Separation <= -799)
and (2*Alt_Layer_Value + 2*Two_of_Three_Reports_Valid <= 9)

(not(2*Positive_RA_Alt_Thresh[0] + 2*Down_Separation <= 1472))
and (not(2*Up_Separation -2*Alt_Layer_Value <= 801))
and (not(2*Up_Separation -2*Down_Separation <= -1))
and (not(2*Own_Tracked_Alt -2*Other_Tracked_Alt <= -1203))
and (not(2*Own_Tracked_Alt_Rate + 2*Up_Separation <= 1594))
and (2*Alt_Layer_Value + 2*Other_Capability <= 5)

(not(2*Positive_RA_Alt_Thresh[0] + 2*Down_Separation <= 1472))
and (not(2*Up_Separation -2*Alt_Layer_Value <= 801))
and (not(2*Up_Separation -2*Down_Separation <= -1))
and (not(2*Own_Tracked_Alt -2*Other_Tracked_Alt <= -1203))
and (not(2*Own_Tracked_Alt_Rate + 2*Up_Separation <= 1594))
and (not(2*Alt_Layer_Value + 2*Other_Capability <= 5))
and (2*Other_Tracked_Alt -2*Down_Separation <= 95)
and (not(2*Two_of_Three_Reports_Valid + -2*Positive_RA_Alt_Thresh[3] <= -1481))
and (not(2*Cur_Vertical_Sep + 2*Other_Tracked_Alt <= 1906))
```

Figure 8: The top three DTinv likely invariants.

ing offers an alternate, and viable method for addressing a difficult task. We have provided an existence proof in the form of the Binary Fission game, and we have shown that crowdsourcing is effective by employing the game to discover program preconditions for 6 TCAS problems. The preconditions are non-trivial, reasonably general (as measured by data coverage on a test set), and human readable. They are also novel, at least with respect to the output of DTinv, which finds likely invariants that do not qualify as program preconditions.

There are three sources of power behind Binary Fission: it employs an expressive representation, it relies on the crowd to conduct a thorough search, and the game imposes restrictions on that search that select for general solutions. In more detail, the representational power comes from Daikon, as Binary Fission inputs the highly structured predicates it produces. The game exploits crowd search by collecting and testing the large number of piecewise solutions that players contribute. The game influences the shape of the solution by limiting classifier depth, and by rewarding discovery of partial classifiers that isolate positive data, which has special utility for invariant construction.

While Binary Fission employs a classification model, improving classification technology is not our goal. Our main point is to introduce crowdsourcing as a promising approach to invariant discovery. From this perspective, the key conjecture behind crowdsourcing is that many non-expert individuals have the desire and ability to provide insight into highly technical problems when they are presented in a suitable form. This conjecture holds for Binary Fission. If it generalizes, related games will provide leverage on additional verification tasks, and crowdsourcing will offer an avenue for expanding the reach of verification technology.

7. THREATS TO VALIDITY

This paper reports first results from a crowdsourced approach to precondition discovery. As mentioned above, the key points are that crowdsourcing is feasible, effective, and promising as a practical avenue for expanding the reach of verification methods. That said, there are several threats to the validity of these claims, as well as our more detailed results.

First, while crowdsourcing finds preconditions on TCAS, the approach may not generalize to more complex programs. In particular, TCAS is a short, straight line, arithmetic program that lacks pointers, loops, complex data structures, and a range of other language features that complicate the verification task. The counterpoint is that Binary Fission is agnostic to the structure of the underlying program, because it formulates precondition discovery as classification. The limits on its use come from the need for inputs common to classifiers; a base of relevant primitive predicates, and labeled data distinguishing bad program states from good. It is true that these inputs are hard to provide for more complex programs (especially the predicate base and assertion violating program states) as they are the product of deep analyses of program structure. However, Binary Fission is also agnostic as to the source of these data, which greatly increases its avenues for application.

Second, our results on the novelty of the Binary Fission solution could be the product of our choice of DTinv as the comparator. This is quite plausible; the likely invariants produced by other machine learning methods might qualify as preconditions. However, our experience with Binary Fission has illuminated constraints that should be applied to the use of classifiers for this task; they should penalize solution size (which is common wisdom), employ a powerful predicate base to support human legibility of the end result, and reward identification of pure good nodes rather than focus on an entropic measure as the splitting criterion.

A third, and broader concern, is that classification is viable but our use of crowdsourcing is superfluous, meaning that Binary Fission can be replaced by a suitable automated method. This argument is relevant at this stage in the development of Binary Fission, but it devolves to the underlying question, "What does the crowd bring to classification that is difficult to automate?". In the case of FoldIt [4] players brought spatial intuition to the task of folding complex proteins, and obtained results never achieved through search over molecular conformations in combination with energy minimization methods. Classification tasks also have a natural framing as search, and by analogy, the crowd may intuit which predicates to employ en route to a more general solution. Binary Fission currently hides a bit too much information to support this type of intuition (in service of broadening the game's appeal), but advanced versions will provide more context about the underlying task. We currently rely on the crowd to explore unexpected places relative to the greedy search conducted by automated methods, and this approach has successfully produced program preconditions.

A final, and related argument is that Binary Fission addresses the wrong crowdsourcing problem. Rather than ask the crowd to combine primitive predicates, we should unleash them on the task of inventing the predicates themselves. This step seems natural as predicate invention (including predicate abstraction from data) is a critical, but elusive process currently performed by people. We have, in fact, developed a game for this task, called Xylem [15], and it is available on-line at `xylem.verigames.com`.

8. CONCLUSION

We have employed Binary Fission, a crowdsourced game for invariant discovery, to analyze the implementation of an on-board aircraft collision detection and avoidance system. We have shown that the crowd can employ Binary Fission to prove program properties. They find function preconditions (statements about program variables associated with function inputs) that guarantee important safety properties hold on program exit, where those properties are encoded as postconditions. Binary Fission players discover concise, general, and human readable preconditions, which are also novel relative to the complicated logical expressions often produced by other classifications systems. The players have no special expertise in formal methods or programming, and are not specifically aware they are solving verification tasks.

Binary Fission demonstrates the feasibility of crowdsourced invariant discover, and it illustrates the promise of crowdsourcing for other verification tasks. This suggests a pathway for expanding the reach, and practical application of verification technology.

9. ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of our collaborators at UCSC, SRI and CEA, especially Kate Compton, Heather Logas, Zhongpeng Lin, Dylan Lederle-Ensign, Joe Mazeika, Afshin Mobrabraein, Chandranil Chakrabortii, Johnathan Pagnutti, Kelsey Coffman, John Murray, Min Yin, Natarajan Shankar, Ashish Tiwari, Sam Owre, Florent Kirchner, Julien Signoles, and Matthieu Lemerre.

This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0225. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

10. REFERENCES

[1] J. Chamberlain, U. Kruschwitz, M. Poesio, and P. Michelucci. Methods for engaging and evaluating users of human computation systems. In *Handbook of Human Computation*. Springer Science+Business Media, New York, 2013.

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

[3] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE 2001*, pages 142–151, 2001.

[4] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović, et al. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, 2010.

[5] D. Dean et al. Lessons learned in game development for crowdsourced software formal verification. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE'15)*.

[6] S. Deterding, M. Sicart, L. Nacke, K. O'Hara, and D. Dixon. Gamification. Using game-design elements in non-gaming contexts. In *CHI '11 Extended Abs. on Human Factors in Computing Systems (CHI EA '11).*, 2011.

[7] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. In *2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 443–456, 2013.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 27(2):99–123, 2001.

[9] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, Computer Science, Univ. of Washington, Nov. 1999.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[11] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *Computer Aided Verification*, 2014.

[12] A. Kawrykow, G. Roumanis, A. Kam, D. Kwak, C. Leung, C. Wu, E. Zarour, L. Sarmenta, M. Blanchette, J. Waldispühl, et al. Phylo: A citizen science approach for improving multiple sequence alignment. *PloS one*, 7(3):e31362, 2012.

[13] S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *arXiv preprint arXiv:1501.04725*, 2015.

[14] D. Kroening and M. Tautschnig. CBMC–C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[15] H. Logas, J. Whitehead, M. Mateas, R. Vallejos, L. Scott, D. Shapiro, J. Murray, K. Compton, J. Osborn, O. Salvatore, et al. Software verification games: Designing Xylem, The Code of Plants. In *Foundations of Digital Games (FDG 2014)*, 2014.

[16] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering.

RN, 15:01, 2015.

[17] K. L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 1–12, Berlin, Heidelberg, 2005. Springer-Verlag.

[18] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact infrastructure repository, 2006.

[19] N. Savage. Gaining wisdom from crowds. *Communications of the ACM*, 55(3):13–15, 2012.

[20] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*, pages 88–105, 2014.

[21] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification*, 2012.

[22] R. M. Stallman et al. Using the GNU compiler collection. 2003.

[23] K. Tuite, N. Snavely, D.-y. Hsiao, N. Tabing, and Z. Popovic. Photocity: training experts at large-scale image acquisition through a competitive game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1383–1392. ACM, 2011.

[24] L. vonAhn and L. Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, Aug. 2008.

[25] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 Int'l Symp. on Software Testing and Analysis*, pages 362–372, 2014.

List of Acronyms

AFRL	Air Force Research Laboratory
API	Application Program Interface
Astree	Static Program Analyzer for C Programming
BIND	Berkeley Internet Name Domain
C1	CHEKOFV Phase 1
C2	CHECOHV Phase 2
CFS	CHEKOFV Facilitate Server
CRS	CHEKOHV Ranking System
CS	Citizen Scientist
CS	Computer Science
CSFV	Crowd Sourced Formal Verification
CWE	Common Weakness Enumeration
DA	Dynamic Analysis
Daikon	Machine Learning Tool
DARPA	Defense Advanced Research Projects Office
DB	Database
DIInv	Machine Learning Tool
FB	FaceBook
FRAMA-C	Framework for Modular Analysis of C Programs
GS	Game Subsystem
HAS	Heavyweight Static Analysis
IFF	Identify Friend or Foe
LSA	Lightweight Static Analysis
MC	Software Model Checking
MCMC	Markov Chain Monte Carlo
mloc	Million Lines of Code
PA	Predicate Abstraction
SQL	Structured Querry Language
SV-COMP	A Competition for Software Verification
T C	TopCoder (Now Appirio)
TP	Theorem Prover
VF	Verification Framework