Gamifying Program Analysis

Daniel Fava,[1] Julien Signoles,[2] Matthieu Lemerre,[2]
Martin Schäf,[3] Ashish Tiwari[3]

[1] University of California, Santa Cruz
[2] CEA, LIST, Software Reliability and Security Laboratory,
P.C. 174, Gif-sur-Yvette, 91191, France
[3] SRI International

Abstract. Abstract interpretation is a powerful tool in program verification. Several commercial or industrial scale implementations of abstract interpretation have demonstrated that this approach can verify safety properties of real-world code. However, using abstract interpretation tools is not always simple. If no user-provided hints are available, the abstract interpretation engine may lose precision during widening and produce an overwhelming number of false alarms. However, manually providing these hints is time consuming and often frustrating when re-running the analysis takes a lot of time.
We present an algorithm for program verification that combines abstract interpretation, symbolic execution and crowdsourcing. If verification fails, our procedure suggests likely invariants, or program patches, that provide helpful information to the verification engineer and makes it easier to find the correct specification. By complementing machine learning with well-designed games, we enable program analysis to incorporate human insights that help improve their scalability and usability.

1 Introduction

Abstract interpretation [1] is a powerful technique for program verification. Tools like Astrée [2] and Frama-C [11] have successfully demonstrated not only that abstract interpretation is able to prove the absence of run-time errors in real-world C programs, but also that it is commercially viable to do so.

To verify a given program P, abstract interpretation approximates the semantics of P based on monotonic functions. The analysis symbolically executes P keeping a set of possible states at each program point. If an error is not reachable in this abstraction, we have a proof that this error is also not reachable in the original program.

Unfortunately, even if these tools are fully automated, it does not mean that using them is simple. Sometimes, in particular when analyzing looping control-flow, abstract interpretation loses precision and the set representing the possible states of the analyzed program becomes too imprecise. This can result in a large number of false alarms, up to a point where the only option is to abort the analysis. In these cases, to help the analysis regain precision, a verification engineer

has to step in and provide hints in the form of code annotations or custom parameterizations. In the case of large programs, writing these annotations can be a painful experience. The process tends to be incremental because an annotation that was used to drive the analysis forward may be insufficient a few statements later. In other words, previous annotations which were considered sufficient may have to be revised because they were either too weak or too strong to continue the analysis at a later point in the program. This leads to a labor intensive process that is also costly because, in order to provide useful annotations, the analyst not only has to understand the analyzed code, but also the details of the abstraction used by the verification engine.

In an effort to lower the cost of applying abstract interpretation, we have seen a new trend of using machine learning to identify likely invariants. The idea is to collect two sets of concrete program states that are either part of a successful execution (*good states*) or failing executions (*bad states*), and use machine learning to find a classifier that separates those sets. Approaches such as Daikon [5], ICE [8], and work by Sharma et al. [15–17], have successfully demonstrated that machine learning can be used to learn likely invariants. Unlike widening, which is commonly used in abstract interpretation to generalize program behavior, machine learning can also provide generalization guarantees.

However, there are limitations to using machine learning for finding likely invariants. First, collecting good states and bad states is expensive (if it were easy to enumerate them, we would not need abstraction) and thus the machine learner has to operate on a small data set. This increases the risk of over-fitting. Second, learners have a tendency to produce large invariants that are not fit for "human consumption." And third, machine learners operate on a hypothesis space which allows them to express certain kinds of knowledge and empowers them with the ability to generalize. However, there can be mismatches in the type of representation strength of a classifier and the domain of the program under analysis.

We present an approach that combines abstract interpretation, machine learning, and crowdsourcing to learn likely invariants. We have developed a system called Chekofv that maintains three values at each program point:

1. a set of states, which represents our current estimate of the likely invariant at that program point,
2. a set of good states, which are concrete states such that executions starting from those states do not cause any assertion violations, and
3. a set of bad states, which are concrete states such that executions starting from those states cause an assertion violation.

None of these sets is necessarily a strict over- or under-approximation of the reachable set at that program point. We use abstract interpretation to initialize the first set, but later update it with likely invariants learnt using machine learning or crowdsourcing. The set of good and bad states are collected using testing and symbolic execution, and they form the inputs for the machine learning and crowdsourced games. In particular, Chekofv complements machine learning pro-

cedures by using two games, Xylem [14] and Binary Fission[4]. These games enable the non-expert crowd to solve the problem of finding likely invariants. The first game, Xylem, resembles Daikon: for a given set of states, the player has to find a predicate that describes all states. The second game, Binary Fission, gamifies a decision tree learning procedure: the player is presented with a set of good states and bad states, and she has to generate a classifier to separate these sets.

The intuition is that crowdsourcing has three major benefits over machine learning: 1) invariants are not limited by a particular kernel function or hypothesis space; instead, we can obtain a very diverse set of solutions from different players; 2) humans tend to produce invariants that are readable (unlike the machine, which can produce illegible predicates); 3) given our natural limitations handling large amounts of data, we believe that humans are less likely to produce a solution that overfits. The crowdsourced experiment has to run long enough for a reasonable set of solutions to be available. However, compared to the several man-months of effort of verifying a real system, this may still be a cheap preprocessing step. Another potential problem is that human intuition breaks at high dimensions, and the dimensionality of the data to be classified depends on the number of variables in scope at a particular program point. This is why, when designing a verification game, the choices of visualization and data representation are important.

Our tool Chekofv shares several similarities with machine learning based approaches such as [16]. We perform an abstract interpretation of a given `C` program using the plug-in Value of Frama-C. Each time when we reach a program location where Value loses precision (e.g., due to widening or unspecified inputs), we use dynamic or symbolic execution to collect good and bad states. Unlike previous approaches, we use these sets as input to the two games described above. The games produce likely invariants which are then inserted as assertions into the program. This process is iterated until we cannot find any bad state that satisfies our current invariant and we cannot find a good state that violates this invariant. Unlike [16] where the focus is on verification, we use the approach to also generate preconditions and checks as suggestive program patches for the developer.

In the following, we discuss our infrastructure, provide a motivating example and an overview of our crowdsourcing game. Our main contribution is the program analysis and patching procedure that combines abstraction interpretation with machine learning and crowdsourcing via gamification. A secondary goal is to increase the visibility of our games and get feedback from the community. We hope to collect enough data this way to perform a statistically significant study that compares the quality of crowdsourced invariants with the quality of machine learned ones.

[4] `http://xylem.verigames.com` and `http://binaryfission.verigames.com`

2 Related Work

The idea of learning likely invariants from program states goes back to Daikon [5]. Daikon learns likely invariants from a given set of (good) program states by working with a fixed set of grammar patterns. Numerous approaches have used Daikon; for example, iDiscovery [18] uses symbolic execution to improve on Daikon's invariants. Similar to our approach, it inserts the learned invariants back in the code under analysis and then uses symbolic execution to confirm or break these candidate invariants. This process generates new states that can be fed to Daikon and can be iterated until either an inductive invariant is found, or symbolic execution fails to generate new states.

Sharma et al. [16] formulate the problem of extrapolation in static analysis as a classification problem in machine learning. They also use *good* and *bad* states and a greedy set cover algorithm to obtain loop invariants. In a follow up work, a similar algorithm to detect likely invariants using randomized search is described [15]. While our approach is similar in the sense that we learn invariants from good and bad examples, our application is different. Rather than finding accurate loop invariants, we are interested in finding human-readable annotations using crowdsourcing that prevent abstract interpretation from losing precision.

The architecture of our approach strongly resembles the decision tree learning based approach of DTInv [12]. In fact, the authors of that paper kindly provided their implementation which we use to test our approach. The key difference between the two techniques is that we use gamification instead of machine learning to find invariants.

Another popular approach for learning likely invariants is the ICE-learning framework [8]. Similar to Daikon, ICE-based algorithms search for invariants by iterating through a set of templates. Unlike Daikon, ICE does not discard likely invariants that are inductive. Instead, it checks a set of implications to decide if the counterexample is a new good or bad state.

Predicate abstraction [9] based on abstract interpretation has also been used to learn universally-quantified loop invariants [7] and was implemented in ESC/-Java [6]. This approach may require manual annotations to infer smart invariants. It is a 100% correct technique but at the price of precision. Counterexample driven refinement has been used to automatically refine predicate abstractions and reduce false errors [10]. Fixpoint-based approaches have also been studied [3]; however they do not explicitly generate bad states, unlike the work we describe here.

An approach to gamify type checking has been presented by Dietl *et al.* [4].

3 Motivating Example

We explain how Chekofv works by dissecting the famous Heartbleed bug in OpenSSL.[5] The code snippet that caused the bug is sketched in Figure 1. For space reasons, we omit a few lines from the original code which are not relevant to understanding the bug.

```
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);
```

Fig. 1. Heartbleed bug in OpenSSL. The problem in this snippet is that, when calling `memcpy` in line 21, we cannot guarantee that `pl` is actually of size `payload`. That is, by providing a wrong `payload`, an attacker is able to read a few bytes of arbitrary memory.

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.

[5] http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html

Figure 1 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 put them into `payload`. However, since 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. Indeed `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`. Since the `payload` can be longer than the actual size of `pl`, close-by data in the memory (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 enforces `bp` and `pl` to be at least of size `payload`. Since it cannot prove this property, it warns about a potential bug. However, since this is not the only warning emitted by Frama-C, chances are it will go unnoticed.

Let us now see how our approach can make it easier for a human analyst who is employing Frama-C to notice this bug. First, even if it does not appear in Figure 1, the length of `&s->s3->rrec.data[0]` 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:

$$
\begin{aligned}
\texttt{hbtype} &\in [0, 255] && \textit{(a one-byte positive integer)} \\
\texttt{payload} &\in [0, 2^{16} - 1] && \textit{(a two-byte positive integer)} \\
size_p &= \texttt{SSL3\_RT\_HEADER\_LENGTH} - 3 \\
\texttt{padding} &= 16
\end{aligned}
$$

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 to note 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` $- 3$.

Since none of the variables in the abstract state depicted above is modified by any statement until line 21, these variables will have the same intervals. Hence, the implicit assertion that `payload` $\leq size_p$ 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, etc), or we lost information due to widening (e.g., after loops).

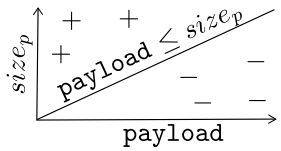

Fig. 2. Data points collected by our symbolic execution for `payload` and $size_p$. A plus indicates a good state and a minus indicates a bad state.

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 2 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 crowdsourcing 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 $\texttt{payload} \leq size_p$. However, let us assume that our symbolic execution picked extreme values and we get an over-fitted invariant $2 * \texttt{payload} \leq size_p$.

We merge the invariant $2 * \texttt{payload} \leq size_p$ 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., $2 * \texttt{payload} > size_p$ and thus the abstract value of `payload` is $[size\_p/2 + 1, 2^{16} - 1]$). This will reveal 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 crowdsourcing immediately because there might be several invariants that have this property.

Eventually, Chekofv finds the invariant $\texttt{payload} \leq size_p$ for line 10 which is sufficient to prove the assertion in line 21. Note that 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 shall help the verification engineer when verifying the program. In the remainder of this paper, we show the architecture of Chekofv and how it finds likely invariants using crowdsourcing.

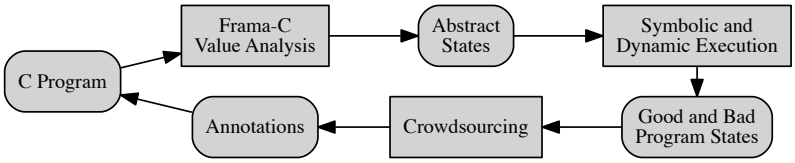

Fig. 3. Overview of our Chekofv system. Chekofv takes a `C` program as input and performs an abstract interpretation. If abstract interpretation fails to verify the program, the computed abstract states are passed to a symbolic execution engine to sample concrete good and bad states. These sets are then passed to our crowdsourcing games to compute likely invariants which are inserted back into the program. This loop terminates if either the program is verified or the invariants cannot be improved further.

4 Overview of the Chekofv System

Our approach to learn likely invariants to assist abstract interpretation is implemented as part of the Chekofv system outlined in Figure 3. The system 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.

Our procedure for program analysis is as follows:

1. *Initialize:* At every program point, initialize the likely invariant to *true*, good states to $\emptyset$ and bad states to $\emptyset$.
2. *Update1:* Update the likely invariant at each program point using abstract interpretation. Terminate with success if all assertions are verified. If the likely invariants are left unchanged, goto *Terminate*.
3. *Update2:* Find new good states that lie outside the current likely invariant, and new bad states that lie inside the current likely invariant. If such states are found, *add* them to the set of good and bad states at each program point. Otherwise, goto *Terminate*.
4. *Update3:* Use the current set of good and bad states to learn an invariant, using either machine learning or crowdsourcing, and use it to update the likely invariant at each program point. If we fail to separate good and bad states, then goto *Terminate*, else goto *Update1*.
5. *Terminate:* Terminate with the likely invariants as hints for the verification engineer.

We now describe the different pieces of the procedure above as implemented in Chekofv.

Abstract Interpreter. Chekofv uses the Frama-C plug-in Value to perform abstract interpretation, which computes, at each program point, an abstract state

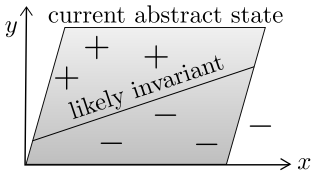

Fig. 4. Example of abstract states, concrete states, and likely invariants. Assuming a program over two variable x and y, the possible values of these variables form a two-dimensional space. The parallelogram describes a possible abstract state. Plus and minus refer to known concrete good and bad states. A likely invariant is a plane that cuts the parallelogram in two parts, one containing only good states, one containing only bad states.

that over-approximates the set of all possible states the program may be at that point. The abstract state is a mapping from every memory location to the set of possible values that this location may have at the current program point. 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 are represented using an interval per memory region where the pointer may point. Frama-C emits 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 emit any warning, we have a proof that the program is safe and our analysis terminates.

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

Good and Bad States. For a given program point in our input program, Frama-C gives us the corresponding abstract state. This abstract state, as depicted in Figure 4, 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). Our goal is to learn an invariant for this 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, we cannot compute the set of good and bad states automatically (otherwise we would not need abstract states), so we can only approximate the invariant that we are looking for. To that end, we use 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.

Sampling Bad States. To find a state which results in an assertion violation, we employ a symbolic execution tool to check if an error state is reachable from any state in the abstract domain of the current program point. That is, we turn the current abstract state into a precondition (or an assume statement) for the symbolic execution. For each variable v with an abstract domain $v \in [min, max]$, we add a conjunct $min \leq v \leq max$ to the precondition. If symbolic execution finds a reachable error state under this precondition, we add it to the set of bad states. If the program point we are analyzing is the program entry, or if we know that our precondition only describes reachable states, we have found a genuine error.

Sampling Good States. The easiest way to collect good states is to run the program and monitor its state with a debugger. If no test cases are available, finding good states is more challenging and we can instead employ symbolic execution (similar to how we have described the generation of bad state above). However, since we might have inserted a too strong invariant in a previous iteration of a loop, symbolic execution may fail because the set of possible states to start from is, for example, empty. To avoid this problem, we also check if there exists a state outside the current abstract domain from which an execution terminates normally. Here, we proceed in a similar way as for the bad states but we compute 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.

Once we have collected the sets of good and bad states, we can start looking for a likely invariant. Finding this likely invariant can be seen as a binary classification problem in machine learning. We are looking for an approximation of a function that labels all good states as good and all bad states as bad. The connection between invariant generation and classification has been explored in many recent works [8, 12, 15–17]. Instead of using machine learning, we propose a crowdsourcing solution to perform this classification.

Gamification of Machine Learning. The main contribution of this paper is the use of crowdsourcing as an alternative to machine learning. The motivation is to avoid two problems that are inevitable when using machine learning: over-fitting, and limited expressiveness of the kernel function. Over-fitting is an inherent problem to machine learning when operating on small data sets. If only a small number of points is available, the machine learner may find a formula that describes exactly this set, resulting in a large formula with no predictive power. This is in particular relevant because we cannot collect arbitrary large sets of good and bad states. Gamification reduces this risk because different players may come up with different solutions, and humans are usually good at finding the *easiest* solution.

The second issue that we are trying to tackle by gamifying machine learning is the limitation of using a fixed kernel function in machine learning. Fixing a kernel function (e.g., conjunctions of linear inequalities) is vital for a machine learner to find a good solution, but it is not clear a priori which kernel function to pick. By gamifying machine learning, we do not have to fix a kernel function and can allow players to come up with arbitrary invariants.

We have developed two games, Xylem and Binary Fission, that crowdsource the machine learning aspect of Chekofv. Xylem is a gamification of Daikon while Binary Fission is a gamification of decision tree learning. These games are discussed in detail in Section 5. Both games interface with Chekofv in the same way as a machine learner would. They receive sets of good and bad states and return likely invariants. We merge the learned (likely) invariants into the program and start over with the first step of our analysis by recomputing the abstract states with value analysis.

Termination. Chekofv terminates if either the system is verified, or one of following situation occurs:

- *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 crowdsourcing 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.

5 Crowdsourcing games

To crowdsource the problem of finding likely invariants, we have developed two games, Xylem [14] and Binary Fission. Both games are available online at verigames.com.

Xylem. The goal of Xylem is to generate new predicates that can be used for invariant construction. Players are presented with a sequence of (good or bad) program states and are asked to find a non-linear inequality that is satisfied by these

states. In that sense, Xylem can be seen as a crowdsourced version of Daikon. To cap the cognitive load on players, Xylem splits the predicate construction problem into several game levels, with each level being composed of a limited number of states and a subset of variables. To ensure that we obtain a diverse set of solutions, Xylem gives different subsets of states and variables to different players. Figure 5 shows a scene of the game. Players takes on the role of a botanist exploring new forms of plant life on a mysterious island. Program states are presented as growth phases of a plant in the top half of the screen. Each variable in the state is presented as a blossom of distinct color. The number of petals per blossom represents a variable value in the current state. At each level, players are presented between four and ten states and are asked to create a predicate that holds on all of the states. The bottom half of the screen contains a toolbox that is used to assemble the predicate; the toolbox contains variables (i.e., blossoms), numbers, operators, and helper functions such as array length. This toolbox was a challenging part of the game design and remains an open area of research. The major difficulty is in striking a balance between user interface simplicity at the same time as providing sufficiently expressive constructs with which players can generate predicates. As players assemble predicates from the elements in the toolbox, the growth phases that satisfy the current predicate turn green while others turn red. Players can submit a predicate once all growth phases are green. That is, the game guarantees that the resulting predicate is a valid invariant for the given subset of states and variables (note that the predicate does not have to be an invariant on the data as a whole).

Fig. 5. Screenshot from Xylem

Binary Fission. In Binary Fission, players construct likely invariants from a fixed set of predicates. To that end, players are presented with good and bad states represented by blue and brown dots as shown in Figure 6. The objective is to separate the two sets by building a decision tree. All states are initially mixed together in a single root node. The player can then choose a predicate from the given set and apply it to the root. Applying a predicate to a node generates two child nodes, one containing all states that satisfy the applied predicate and another containing all states that falsify it. The player grows this decision tree until each leaf becomes pure (i.e., contains only good or only bad states), or until a depth limit is reached. Loosely speaking, Binary Fission is the gamification version of DTInv, a decision tree based invariant learner [12]. Once a tree is built, we can trace down from the root to a pure good node (a node composed of good states only) taking the con-

junction of predicates along the way. A conjunction of predicates from the root of the tree to a leaf containing only good states is a likely invariant for our program. Note that there may be several leaf nodes containing only good states. We combine them into one likely invariant by forming a disjunction.

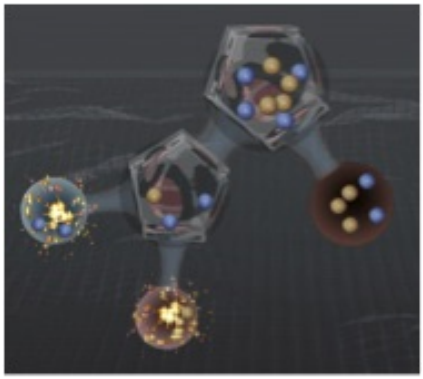

Fig. 6. Screenshot from Binary Fission

The set of predicates available to player can be obtained from different sources. Ideally, they are generated by Xylem as discussed previously. Currently we seed this set using Daikon.

If the available predicates are not sufficient to separate the good from the bad states within a given tree-depth, some leaf nodes will contain both good and bad states. These states are then taken aside and re-entered as input into Xylem and/or Daikon. Since these sets are smaller than the initial set, Xylem and Daikon are allowed to search for more complex likely invariants (which would otherwise be computationally expensive and lead to many useless candidates).

Implementation Notes. The Chekofv system provides both games with sets of states and collects the predicates and candidate invariants provided by the players. Chekofv uses Frama-C for abstract interpretation and extends it by several plugins to extract the abstract state at particular program points, and to insert likely invariants. This implements the steps *Initialize* and *Update1* from our abstract algorithm in Section 4. Note that Chekofv only samples states at entry points of procedures and before procedure calls that precede Frama-C warnings. As shown in Section 3 this is sufficient to find real bugs.

For practical reasons, we use the bounded model checker CBMC [13] instead of symbolic execution to implement step *Update2* from Section 4. We perform minor program transformations (e.g., insert assumptions and non-determinism) to make the result resemble a symbolic execution. The collected good and bad states are stored in a database and serve as input to both games.

For testing, we also seed or database the sets of good and bad states from the experiments in [12] and [18] to our database. The sets of good and bad states differ greatly between the benchmarks. For example, the TCAS benchmark in [18] comes with hundreds of states collected from dynamic execution, while other benchmarks come with less than a dozen states obtained by symbolic execution.

Step *Update3* from Section 4 is realized by the two games; each game has its own web server that pulls program states from the database and presents them to the game client. Predicates produced by the players are sent back to

the server and stored in a database. Likely invariants generated from game play are post-processed and then used by Frama-C, thus closing the loop in Figure 3.

6 Conclusion

We have presented an approach that uses crowdsourcing to learn likely invariants that assist abstract interpretation. Our approach extends previous machine learning based techniques by reformulating the machine learning problem of finding a classifier that separates two sets as puzzle games.

Crowdsourcing the invariant learning has several potential benefits over machine learning: finding good and bad states is expensive and often only possible to a limited extend. Hence, these sets are small which often causes machine learning to over-fit and find correlations that may be true on the observed data, but irrelevant (or even wrong) in the program. Crowdsourcing can avoid this problem. First, given our natural limitations handling large amounts of data, we believe that humans prefer shorter solutions that are less likely to overfit. Second, crowdsourcing returns a diverse set of different likely invariants from which we can choose. Beyond that, crowdsourcing is not limited by a set of templates or kernel functions when constructing likely invariants. That is, unlike machine learning, we do not have to limit the search space for likely invariants a priori. This may lead to a more diverse set of invariants and allow us to discover invariants and requires less interaction through the verification engineer, like trying different kernel functions.

The games are now open to the public. We hope that the interested reader will enjoy playing them and help us to collect valuable data along the way.

7 Acknowledgement

This work was supported in part by the National Science Foundation under grant contracts CCF 1423296 and CNS 1423298, and DARPA under agreement number FA8750-12-C-0225.

We gratefully acknowledge the contributions of our collaborators at UCSC especially Kate Compton, Heather Logas, Joseph Osborn, Zhongpeng Lin, Dylan Lederle-Ensign, Joe Mazeika, Afshin Mobrabraein, Chandranil Chakrabortii, Johnathan Pagnutti, Kelsey Coffman, Richard Vallejos, Lauren Scott, John Thomas Murray, Orlando Salvatore, Huascar Sanchez, Michael Shavlovsky, Daniel Cetina, Shayne Clementi, Chris Lewis, Dan Shapiro, Michael Mateas, E. James Whitehead Jr., at SRI John Murray, Min Yin, Natarajan Shankar, Sam Owre, and at CEA Florent Kirchner, Boris Yakobowski.

References

1. 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.

2. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *PLS*. 2005.
3. P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*. 2007.
4. 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*, pages 42–49. ACM, 2012.
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. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.
7. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.
8. P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *CAV*, 2014.
9. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, jun 1997.
10. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. In *TACAS*, 2008.
11. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, pages 1–37, Jan. 2015.
12. S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *CoRR*, 2015.
13. D. Kroening and M. Tautschnig. CBMC - C bounded model checker. In *TACAS*, 2014.
14. 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. 2014.
15. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, 2014.
16. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*. 2013.
17. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*.
18. L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *ISSTA*, 2014.