Boosting Static Analysis Accuracy with Instrumented Test Executions

DynaBoost FSE 2021 Artifact

Introduction

Static program analyzers have traditionally faced the challenge of emitting too many false warnings, leading to developer dissatisfaction.

In previous work, we developed Bingo (PLDI 2018), where we effectively improved the accuracy of a static analyzer with an interactive alarm prioritization system. Bingo constructs a Bayesian network from the analysis results, and ranks the alarms according to their probabilities. The programmer uses the system over a sequence of rounds: in each round, they look at the ranked list generated by Bingo, triage a set of alarms (usually just the first alarm in the list), and give feedback to the system. In response, Bingo computes conditional probabilities given this feedback, and reranks the remaining alarms. Through this process, we showed that the user is able to discover bugs much faster than they could otherwise have found them.

In this paper, we develop DynaBoost, a system which uses data from test cases to further improve the ranking emitted by Bingo. We use Sparrow (https://github.com/KihongHeo/sparrow) as the underlying static analyzer, and use it to find buffer-overflow and format string bugs in a set of Unix command line programs.

This artifact contains all data and programs required to run the experiments reported in the paper.

Summary of Experimental Results

1. We have applied DynaBoost to 13 Unix command line programs, of which we analyze 10 for buffer overflow bugs, and 3 for format string bugs. Each of these programs has a set of known bugs. See Table 1 of the paper.

2. The major experimental results are contained in Table 2, Figure 6, Table 3, and Figure 7 of the main paper, and Figures 8 and 9 of the appendix. In turn, these results come from the following experiments:

    a. Experiment 1: Applying DynaBoost and Bingo (as a baseline) to all benchmark programs, with their standard set of test cases.

    Depending on the Bayesian network we use, we run each tool in two different

settings: Bingo_zero, Bingo_all, DynaBoost_zero, and DynaBoost_all. Traditional Bingo corresponds to the columns named Bingo_zero, while full DynaBoost corresponds to the columns named DynaBoost_all.

From the results of this experiment, we construct:

i. Table 2: Main statistical summary comparing the number of iterations in 4 different configurations. The column marked Init indicates the rank of the bug in the first round, and the column marked Iter indicates the number of rounds for the bug to rise to the top.

ii. Figure 6, 8: Evolution of the bug rank over the course of interaction.

iii. Table 3: Magnitude and frequency of false generalization events, i.e. the prominent spikes that occur in Figures 6 and 8.

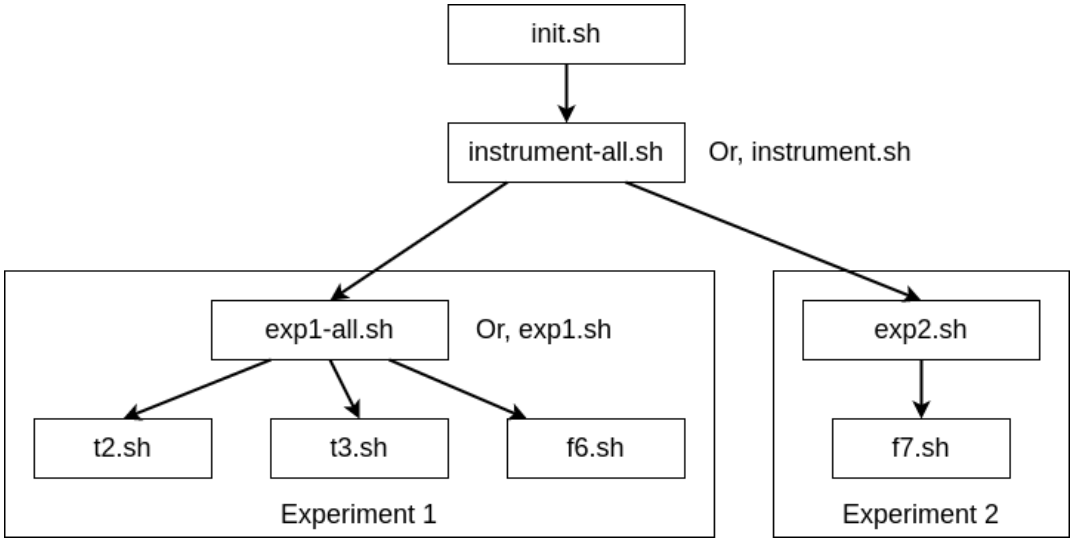
b. Experiment 2: Since the performance of DynaBoost is intimately tied to the test cases, in this experiment, we study how its performance degrades as we withhold test cases. For each fraction p in {0.1, 0.2, …, 0.9, 1}, we withhold all but p fraction of the test cases, chosen uniformly at random, and measure the number of iterations needed to find the bug. Since this number depends on which test cases were randomly chosen, we repeat this experiment 10 times for each value of p to obtain the corresponding box in the box plots of Figures 7 and 9.

Structure of the Artifact

The artifact contains 8 main scripts, organized as follows:

1. To initialize and instrument the programs: init.sh, instrument.sh
2. To run Experiment 1: exp1.sh, t2.sh, t3.sh, f6.sh
3. To run Experiment 2: exp2.sh, f7.sh

These scripts are dependent on each other as shown in the following dependency diagram.



System Requirements

To perform the experiments reported in the paper, we used a computer with an 8-core Core i9-9900 CPU and 128 GB of RAM, running a recent version of Ubuntu Linux. We recommend running the experiments on a machine with at least 32GB RAM.

Directory Structure

1. /home/ubuntu/dynaboost: This directory contains the main programs and scripts we developed as part of this project.
    a. dfsan-plugin: This directory contains the programs which perform runtime instrumentation.
    b. utils: This directory contains other scripts.

2. /home/ubuntu/bingo-ci-experiment: This directory contains preprocessed benchmark files in the benchmark subdirectory. These directories are used for the DynaBoost_all and DynaBoost_zero configurations in Table 3.

3. /home/ubuntu/vanilla-experiment: Similar to bingo-ci-experiment, but contains the data necessary for the Bingo_all and Bingo_zero configurations in Table 3.

4. /tmp: Contains target programs, and information produced during instrumentation, and test-runs.

5. /home/ubuntu/llvm, nichrome, bingo: These directories contain software dependencies.

Running the Experiments

1. Obtaining the artifact.

    a. Download the Docker image from the following URL: https://zenodo.org/record/4902828

    b. Load the Docker image using:

    $ [sudo] docker load < dynaboost-docker.tar.gz

    c. Some of our scripts require deep recursion. To allow unlimited stack sizes, execute:

    $ [sudo] ulimit -s unlimited

    d. Run the Docker container using:

    $ [sudo] docker run -it dynaboost

2. ~/dynaboost $ Initialize the environment by sourcing the init.sh script in the dynaboost directory:

    ~/dynaboost: $ source init.sh

3. **[Optionally,]** Run the underlying static analyzer, Sparrow, to obtain the list of alarms and associated provenance graph. For each program in the ~/bingo-ci-experiment/benchmark/ or ~/vanilla-experiment/benchmark directories, run:

    ~/dynaboost/bingo-ci-experiment: $ ./run.sh ./benchmark/program/program.c [interval / taint]

    ~/dynaboost/vanilla-experiment: $ ./run.sh ./benchmark/program/program.c [interval / taint]

    Choose either the interval or taint analysis as discussed in Table 1 of the paper.

    Note that this step is optional, and that we have pre-loaded results of static analysis into the Docker image. Depending on the program, this command might take up to 15 minutes to complete.

4. Instrument the program and run the test cases. Run:

~/dynaboost/eval: $ ./instrument-all.sh

On our computers, this command takes 3 hours to complete. If you wish to instrument and run the test cases for a single program, say cflow-1.5, run:

~/dynaboost/eval: $ ./instrument.sh cflow-1.5

5. To run Experiment 1:

   ~/dynaboost/eval: $ ./exp1-all.sh

   The above script generates data corresponding to the DynaBoost_all and DynaBoost_zero configurations of Table 2, in addition to Table 3 and Figure 6. These may generated using:

   ~/dynaboost/eval: $ ./t2.sh
   ~/dynaboost/eval: $ ./t3.sh
   ~/dynaboost/eval: $ ./f6.sh

   The t2.sh and t3.sh scripts emit their output to stdout, while f6.sh saves its output in the /tmp/plots directory. The exp1-all.sh script takes about 3--4 hours to finish on our machines.

   Subsequently, to generate the results in Table 2 corresponding to the Bingo_zero and Bingo_all columns, run:

   ~/dynaboost/eval: $ BINGO_CI=/home/ubuntu/vanilla-experiment ./exp1-all.sh
   ~/dynaboost/eval: $ BINGO_CI=/home/ubuntu/vanilla-experiment ./t2.sh

   Note:

   a. The above commands run Bingo and DynaBoost, for all of the benchmarks. If you wish to run the tools for a single benchmark, say cflow-1.5, run:

      ~/dynaboost/eval: $ ./exp1.sh cflow-1.5

      As before, to run Bingo, set the BINGO_CI variable before calling exp1.sh, as follows:

      ~/dynaboost/eval: $ BINGO_CI=/home/ubuntu/vanilla-experiment ./exp1.sh cflow-1.5

   b. Because of interaction between buggy programs and the dynamic instrumentation, and because of non-determinism in some parts of the ranking

algorithm, there might be minor differences between the values obtained for Table 2 and those reported in the paper.

6. To run Experiment 2, say for cflow-1.5, run:

   ~/dynaboost/eval: $ ./exp2.sh cflow-1.5

   ~/dynaboost/eval: $ ./f7.sh

   The second script places its output in the following directory: /tmp/sampleplots. Note that it is not necessary to have run exp2.sh on all benchmarks before running f7.sh, because the second script will simply skip benchmark programs where exp2.sh has not been previously run.

   This script takes considerably longer---approximately 20 hours for the largest program---and we therefore do not provide an exp2-all.sh script.