SAFER: Efficient and Error-Tolerant Binary Instrumentation

Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R. Sekar
Stony Brook University, NY, USA.
{spriyadarsha, hnnguyen, rchouhan, sekar}@cs.stonybrook.edu

A Artifact Appendix

A.1 Abstract

The artifact submission is for the paper titled "SAFER: Efficient and Error-Tolerant Binary Instrumentation". Our tool SAFER is an efficient and safe binary-instrumentation suite that is capable of instrumenting complex programs. The tool is compatible with both position independent (PIE) and position dependent (Non-PIE) executables and has a modest overhead of ≈2%.

The artifact consists of software and will be submitted in the form of a VM containing a pre-installed version of the software and scripts needed to run the software. It will also contain all the instrumented programs used during evaluation. SAFER was used to instrument 15 real world programs along with their shared libraries, customized coreutils binaries with data embedded in code and SPEC 2006 and 2017 binaries. The total size of all programs was about 1.1GB.

A.2 Description & Requirements

SAFER's current prototype requires Ubuntu 20.04 operating system. It further requires additional packages (*Capstone* and *Ocaml*) for disassembly and static analysis. We are submitting our artifact as an Oracle VirtualBox VM. SAFER is already installed in the VM along with all the prerequisite packages. The VM also contains SAFER's source code.

Requirements to run artifact: A x86-64 system with Oracle VirtualBox is required to use our artifact. Importing the virtual box image through Oracle Virtualbox running on a x86-64 system will recreate the testing environment. The virtual machine image is configured with 8GB of RAM and 100GB of secondary storage. So we recommend to run it on a system with at least 16GB of memory and 256GB of storage space.

Benchmarks like SPEC CPU 2006 and 2017 are already installed and set up with instrumented binaries. Other datasets like instrumented real-world applications and data-in-code coreutils are also present in the virtual machine.

A.2.1 Security, privacy, and ethical concerns

Since we are providing our system in a virtual machine, there is no risk for the host machine of the evaluator.

A.2.2 How to access

SAFER's artifact http://seclab.cs.sunysb.edu/seclab/safer

A.2.3 Hardware dependencies

An x86-64 machine preferably with 16GB of RAM and 256GB of storage space.

A.2.4 Software dependencies

Oracle VirtualBox 6 is required to run the virtual machine. The system setup in the virtual machine is complete and requires no additional software installation.

A.2.5 Benchmarks

For testing the performance overhead we use SPEC CPU 2006 and SPEC CPU 2017 benchmark suites. For functionality evaluation we use 15 pre-built real-world programs and custom-compiled coreutils with data-in-code.

A.3 Set-up

A.3.1 Installation

1. Download the virtual machine image file.
2. Install and open VirtualBox.
3. Select File, Import Appliance.
4. Select the virtual machine image.
5. Click import.

A.3.2 Basic Test

Instrumenting a program **ls** with its shared libraries.

1. Start the virtual machine.
2. Login with the password 'safer'.
3. Copy the original ls binary to the test folder.

```
> cp /usr/bin/ls ~/SBI/test
```

4. Find the dependencies of ls.

```
> cd ~/SBI/testsuite
> ./find_libs.sh /home/safer/SBI/test/ls
> truncate -s 0 randomized.dat
```

5. Run instrimentation on ls.

```
> cd ${HOME}
> ./instrument_prog.sh ls
```

6. Wait for the instrumentation to finish. Printed date means the process is completed.
7. Go to the output directory.

```
> cd ${HOME}/instrumented_libs/
```

8. Run the instrumented ls. Contents of the directory should be printed.

```
> ./ls
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SAFER can handle disassembly errors in the presence of complexities such as data embedded in code.
(C2): SAFER's pointer encoding scheme can detect instrumentation errors at runtime and deterministically abort (FAIL-CRASH).
(C3): SAFERis able to instrument position dependent (Non-PIE) code.
(C4): SAFER's pointer encoding and safe jump table transformation helps in achieving safe instrumentation with a fail-crash while having a modest overhead of ≈2%.
(C5): SAFERis able to instrument a wide variety of real world programs.

A.4.2 Experiments

(E1): [**Data-in-code test**]: We use coreutils and its built-in tests to ensure that SAFER is successfully able to instrument binaries where data is present in the code section.
Generating the dataset: We make use of a linker-script to compile a version of coreutils where read-only data is embedded in the code. The binaries and pre-built and available in the VM: `/home/safer/coreutils/coreutils-data/bin`.
Test preparation: Pre-instrumented binaries with SAFER's different modes (Table 3 in the paper) are available in `/home/safer/coreutils/coreutils-data` directory as below:

- *bin_FN_PRLG:* Function prologue based pointer classification.
- *bin_FULL_AT:* run time address translation.
- *bin_valid_ins:* valid instruction boundary based pointer classification.
- *bin_ABI:* ABI specification based pointer classification.

To reuse the above pre-instrumented binaries, copy all the binaries from one of the above mention directories to `/home/safer/coreutils/coreutils-8.30/src/`. Alternatively, binaries can be re-instrumented as follows:

```
> cd /home/safer
> ./instrument-coreutils.sh \
    config=<FULL_AT/FN_PRLG/valid_ins/ABI>
```

Testing: Coreutils in-built testsuite is used to test correctness of instrumented binaries:

```
> cd /home/safer/coreutils/coreutils-8.30
> make check
```

Results: FN_PRLG and FULL_AT configurations result in 100% correct instrumented binaries. There is 1 failure due to one of the test making use of LD_PRELOAD to load a dynamically built library. The library is built by the test case at the runtime and used. SAFER requires all program modules to be instrumented for correct execution. Since, this particular library is not available to us during instrumentation time, it is left unchanged and resulted in a crash.
Other approaches (*valid_ins* and *ABI*) specifications result in failures that are detected by SAFER's fail-crash design. The test will not proceed unless the faulty binaries are replaced. The list of failed binaries is in `coreutils-data/ABI_fail.sh` and `coreutils-data/valid_ins_fail.sh`. Running these scripts will replace the faulty binaries with correctly instrumented (with FULL_AT) ones and the test can progress.

(E2): [**Non-PIE binaries**]: SPEC 2006 binaries are compiled as non-PIE and then instrumented. Successful completion of SPEC tests ensure correct transformation. Other non-PIE programs such as gcc and Python have also been tested.
How to: We are providing the instrumented non-PIE SPEC binaries to reduce the testing time. However, if you want to instrument then again follow the steps in the preparation section otherwise skip to the execution section.
Preparation: Clean up and start the instrumentation.

```
> ./instrument-suite.sh \
  /home/safer/spec-06/nopie
```

Execution: • Change the directory to the spec-06 and run the command.

```
> cd /home/safer/spec-06/
> source shrc
> runspec --config=nopie.inst.cfg \
          --noreportable \
          --iterations=1 all
```

Wait for the SPEC run to complete. This may take several hours to complete (≈ 4 hours).

- For real world programs (gcc and python), the testing process is described in the subsequent section.

Results: After the SPEC run is completed check the log file mentioned. There should be results for all the binaries except wrf and gamess whose uninstrumented versions fail on our setup.

(E3): [Runtime overhead with SPEC 2006]

How to: SPEC 2006 CPU benchmark suite is used to test the runtime overhead caused by SAFER's instrumentation. Note that the below steps are time consuming. Hence, they should be run in background (e.g., using nohup). Furthermore, we are providing pre-instrumented binaries. Hence, the preparation step can be skipped.

Preparation: Instrument the SPEC binaries.

```
> truncate -s 0
/home/safer/SBI/testsuite/randomized.dat
> cd /home/safer/
> ./instrument-suite.sh \
    /home/safer/spec-06/pie/
```

Wait for the instrumentation to complete.

Execution: Run the testsuite with uninstrumented binaries (base result).

```
> cd /home/safer/spec-06/
> source .shrc
> runspec --config=default.cfg \
    --noreportable all
```

Wait till completion and save the result directory as base. Then, run the instrumented binaries:

```
> cd /home/safer/spec-06
> runspec --config=inst.cfg \
        --noreportable all
```

Save the instrumented version results.

Results: We computed the overhead by importing the corresponding csv files onto a spreadsheet and comparing the timings of instrumented run over base run.

We are also providing the results of our experiments that have been published in the paper (`/spec-06/our_results`).

(E4): [Performance vs optimization levels] We compiled SPEC CPU 2017 benchmarks at 6 optimization levels (O0, O1, O2, O3, Ofast, and Os) and measured the runtime overhead for each of them.

Preparation: Running the uninstrumented binaries:

```
> cd /home/safer/spec-17/
> source shrc
> runcpu --config=default00.cfg \
        --noreportable intspeed
> runcpu --config=default00.cfg \
        --noreportable fpspeed
```

Execution: Running the instrumented binaries:

```
> cd /home/safer/spec-17
> runcpu --config=default.inst.00.cfg \
      --noreportable intspeed
> runcpu --config=default.inst.00.cfg \
      --noreportable fpspeed
```

Results: Other optimizations could be tested by repeating the steps above with O0 replace by either O1, O2, O3, Ofast, or Os. The results can be obtained in the same manner as SPEC 2006.

(E5): [Safe jump table transformation]: SAFER's safe jump table analysis (Section 5 in paper), helps in improving performance while maintaining correctness of instrumentation by avoiding instrumentation of indirect jumps related to jump tables.

Preparation: Safe jump table analysis code is present in `/home/safer/SBI/safe_jtable`. Steps to build this code are present in a README file in the same directory.

Execution: Safe jump table results (Figure 5 in paper) were produced on SPEC 2006 binaries. Please follow the steps in README to reproduce the results.

Results: In default mode (enable_ftype=0), no function signature matching is done and it marks 55%jump tables as safe. Enabling the function signature matching (enable_ftype=1) results in 85% safe jump tables.

(E6): [Real-world programs]: We used 15 real-world programs (`/home/safer/real-world-instrumented`) to test SAFER's applicability on real-world programs.

Preparation: Programs can be instrumented as mentioned in Section A.3.2. There is no need to generate dependency list again (step 4). Some of the programs are fairly large and cannot be instrumented with the limited amount of RAM that the virtual machine is configured to run with. Hence, we are providing instrumented programs to skip the preparation step.

Execution: Steps to execute test cases are present in `/home/safer/real-world-instrumented/README`.

Results: The instrumented program produces desired output. (e.g., gedit is able to open and edit files).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.