Non-Termination Proving At Scale (Artifact) OOPSLA 2024

Azalea Raad[1], Julien Vanegue[2], Peter O'Hearn[3]

[1]Imperial College London & Bloomberg, azalea.raad@imperial.ac.uk
[2]Bloomberg & Imperial College London, jvanegue@bloomberg.net
[3]University College London, peterohearn0@gmail.com

Abstract

This document contains instructions to reproduce the experiments listed in submission 173 for the OOPSLA 2024 conference. The project is indexed using a DOI url `https://zenodo.org/records/12637589`

Keywords: *Infer Incorrectness Non-Termination Separation Logic*

1 Introduction

The artifact, *Pulse Infinite*, is provided as an addition to the Infer framework by Meta. Experiments are shown to run on a new test suite of 100+ divergent and non-divergent examples, the software verification benchmark for non-linear arithmetic (NLA), as well as several open source projects.

2 Hardware Dependencies

All experiments were run on a Linux Ubuntu 22.04.3 LTS *jammy* on an AMD EPYC 7543P 32-Core processor. Instructions are known to build as well on Linux Ubuntu 20.04. as well as other processors of the ia32 or ia64 families.

3 Getting Started

Make sure your version of Cmake is at least 3.20. If this is not the case on your machine, please follow the official Ubuntu instructions to update cmake:

`https://askubuntu.com/questions/355565/how-do-i-install-the-latest-version-of-cmake-from-the-command-line`

Make sure all needed external packages are installed on your machine:

```
apt-get upgrade
apt-get install sqlite3 libsqlite3-dev libgmp3-dev opam
```

Then download the artifact. We use the DOI url for the purpose of this documentation:

```
wget 'https://zenodo.org/records/12637589/files/infer-oopsla24.
   zip?download=1' -O pulseinf.zip
unzip pulseinf.zip
cd infer-oopsla24
./build-infer.sh clang
```

This will compile the needed clang version automatically, and then will build the artifact as part of the infer project. Note that it can take from a few minutes to several hours to build infer with clang depending on the number of cores and memory of your machines. You may want to leave it alone for some time if you see that progress is slow.

Once your build terminates, you can run the following test script to confirm the build is operational:

```
cd infer/tests/codetoanalyze/c/pulse
./termination-run-all.sh > log
```

Make sure to edit termination-run-all.sh to set the correct infer path for your built.

This script by default will run all the non-termination software verification benchmark for non-linear arithmetic. You should see the result for each benchmark (one per file) showing either *No issue found* or a summary of the issue detected for each file of the benchmark. These results correspond to table 2 in section 7.1 of our submission [1].

Inspecting the *log* file will reveal which tests are flagged as non-terminating and which tests are not. Note that the file names includes *-t-* for terminating tests and *-nt-* for non-terminating tests, allowing to cross-reference the true positives, false negatives, false positives, etc. For each found non-termination issue, the log file contains a trace of the recurring states forming the lasso in the state space witnessing the divergence bug.

4 Step-by-step Instructions

Now that you ran pulse infinite on the benchmark, let us run the tool on several real-world projects written in C and C++. We will do this on openssl, libxml2 and cryptoPP in which we identified several non-termination issues.

First, we define our INFERHOME, which is the location where you downloaded the original artifact.

For example:

```
export INFERHOME=$HOME
```

4.1 OpenSSL

Run infer as such:

```
git clone https://github.com/openssl/openssl
cd openssl
./config
time ${INFERHOME}/infer/bin/infer run --pulse-only -- make -j
    2> infer-run-openssl.log
python3 -m json.tool infer-out/report.json > pulseinf-report.
    json
```

The detail of the analysis and results can be found in *infer-run-openssl.log*. See the number of PULSE_INFINITE alerts to focus on divergence bugs. See the full listing in *pulseinf-report.json* file. You can confirm that our finding in function *aes_ecb_cipher*

reported in our submission's listing 1 [1] is present in the list (as well as other similar bugs reported in table 3).

4.2 LibXML2

Run infer as such:

```
git clone https://github.com/GNOME/libxml2/
cd libxml2
time ${INFERHOME}/infer/bin/infer run --pulse-only -- make -j
   2> infer-run-libxml.log
python3 -m json.tool infer-out/report.json > pulseinf-report.
   json
```

The detail of the analysis and results can be found in *infer-run-libxml.log*. See the number of PULSE_INFINITE alerts to focus on divergence bugs. See the full listing in *pulseinf-report.json* file. You can verify that our reported findings in *xmlDictGrow*, *xmlHashGrow*, *xmlHashScanFull* and *xmlHashScanFull3* are in the list.

4.3 CryptoPP

Run infer as such:

```
git clone https://github.com/weidai11/cryptopp
cd cryptopp
time ${INFERHOME}/infer/bin/infer run --pulse-only -- make -j
   2> infer-run-cryptopp.log
python3 -m json.tool infer-out/report.json > pulseinf-report.
   json
```

The detail of the analysis and results can be found in *infer-run-cryptopp.log*. See the number of PULSE_INFINITE alerts to focus on divergence bugs. See the full listing in *pulseinf-report.json* file. You can verify that our finding in function *AlignedAllocate* in our submission's listing 2 is present in the list.

5 Reusability Guide

In general, we recommend using the latest version of pulse-infinite to detect divergence bugs. While these instructions are self-sufficient for *OOPSLA*, new versions of the tool are fine-tuned to have an increased signal/noise ratio and an better performance runtime.

To download the latest version and build it, execute the following commands:

```
git clone https://github.com/jvanegue/infer
cd infer
./build-infer.sh clang
```

Then one can analyze any software using the methodology shown above.

5.1 Using Bear

For some particularly large software which are built with different systems (for example: the linux kernel), it is useful to perform analysis in two steps:

1. Create the compilation database from a regular build.
2. Invoke infer with the compilation database.

We will use an open-source project *bear* for step 1:

```
git clone https://github.com/rizsotto/Bear
cd Bear
cmake -DENABLE_UNIT_TESTS=OFF -DENABLE_FUNC_TESTS=OFF
make -j
make install
```

5.2 Analyzing the Linux Kernel

For the particular case of the linux kernel, you need have the linux kernel source code and its header files installed (apt get usually includes packages with these for your version). For example, for Ubuntu 20.04 it is likely that you are using linux 5.4.0. The instructions below can work for any linux kernel version with the string modified:

```
sudo apt-get install linux-headers-5.4.0-1001-gkeop linux-
    source-5.4.0
cd /usr/src/linux-source-5.4.0/
tar xjvf linux-source-5.4.0.tar.bz2
cd linux-source-5.4.0
# you need to be root to build the linux kernel
sudo /bin/bash
make menuconfig
```

You can then save the default kernel configuration in the *.config* file and exit the menuconfig.

Now to analyze the linux kernel:

```
bear -- make -j
time ${INFERHOME}/infer/bin/infer --pulse-only --compilation-
    database compile_commands.json --keep-going 2> log
```

Note how we use *bear* to build the compilation database first in *compile_commands.json* then run infer in pulse only mode with *–pulse-only* instructing infer to keep going if any non-fatal error occurs during the analysis (*–keep-going*). As in previous experiments, you can inspect the *log* file for additional tracing information during the analysis. Once this finishes, you can witness that it takes approximately 18 minutes to analyze the linux kernel. These experiments on one of the largest open source project are proof of the reusability of the framework.

References

[1] Azalea Raad, Julien Vanegue And Peter O'Hearn "Non-Termination Proving At Scale" OOPSLA'24 Submission