Non-Termination Proving: 100 Million LoC And Beyond CAV 2025 Artifact

Julien Vanegue[1], Jules Villard[2], Peter O'Hearn[3], Azalea Raad[4]

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

Abstract

This document contains instructions to build the pulse$^\infty$ tool and reproduce the experiments listed in submission 253 for the CAV 2025 conference. The project is indexed using a DOI url `https://zenodo.org/uploads/15220259`

Keywords: *Infer Pulse Incorrectness Non-Termination Scale*

1 Introduction

The artifact is made of the source code of our tool pulse$^\infty$ (*Pulse Infinite*) provided as an addition to the Infer framework by Meta. Experiments are shown to run on a new test suite of divergent and non-divergent examples, as well as a number of open source projects totally 50 million lines of code of C and C++. All experiments are automated and can be reproduced using the instructions below. Based on the performance of your machine, you should expect several hours for this process to complete.

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. Running the entire set of experiments as documented below will require 300 GB of free storage space to download source code and analyze the projects entirely.

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 these 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
```

Make sure the bear tool is compiled from sources and installed on your machine:

3.1 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* to perform these two steps. These steps can be used for any project to allow infer to work even with a completely bespoke build system.

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

3.2 Compiling Infer and Pulse$^\infty$

Once bear is installed, download the CAV 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-cav2025
./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
```

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

You should expect to see the following after running the test suite:

```
Found 20 issues (console output truncated to 5 ...)
```

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 a list of 50 million lines of C and C++ code made of the following projects:

- ◇ openssl: a popular cryptographic library written in C.
- ◇ libxml2: a popular XML parser written in C.

- ◇ CryptoPP: a C++ cryptographic library.
- ◇ libxpm: an XPM image parser.
- ◇ libexpat: a fast streaming XML parser.
- ◇ wireshark: a popular network analysis tool.
- ◇ The linux kernel: The most used open source operating system in the world.
- ◇ comdb2: The Bloomberg database back-end.
- ◇ bde: The Bloomberg Development Environment library.
- ◇ Blazing MQ: A low level real-time message queue library.
- ◇ bind : a DNS server.
- ◇ Bitcoin: The core of the bitcoin blockchain.
- ◇ Exim: the most used mail server in the world.
- ◇ libgit2: a source control repository library.
- ◇ FreeImage: a diverse collection of image parsers.
- ◇ libpng: a PNG format parsing library.
- ◇ mbedtls: a lightweight cryptographic library for embedded systems.
- ◇ lua: A popular scripting engine.
- ◇ pcre2: a regular expression library.
- ◇ zlib: one of the most used compression library.
- ◇ sqlite: a popular in-process database engine.
- ◇ open5gs: a powerful 5G communication library.

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

For example:

```
export INFER_HOME=$HOME
```

Then we move to the experimental folder:

```
cd $INFER_HOME/infer/tests/codetoanalyze/c/pulse/pulseinf
```

From there we launch scripts to prepare the experiments. This will download all targets in the *projects* folder and prepare them for analysis:

```
$ ./pulseinf-prepare-experiments.sh
```

Finally, we execute the analysis wrapper:

```
$ ./pulseinf-runall-experiments.sh
```

If any project fails to build, you can go to projects/*name*/ where *name* is the target name. Sometimes building one project after another one is the easier way to diagnose analysis errors. Once analysis has completed, use a third wrapper script to copy all result files in the *results* folder:

```
$ ./pulseinf-collect-results.sh
```

All results are stored in a neatly indented json file for each target. Look for INFINITE_LOOP and INFINITE_RECURSION types of errors in the json file. All targets analyzed so far constitutes about 24 million lines of code. If for whatever reason, you fail to build and analyze some targets, the set of all analysis results is stored in the following folder:

```
$ ls -l results_archive/*.json | wc -l
21
$
```

5 Analyzing the Linux Kernel

We now switch to analyzing the linux kernel, which constitutes another 26 millions lines of code in a single project. 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, first copy the scripts in the kernel source folder, and then launch the analysis script.

```
cp $INFER_HOME/infer/tests/codetoanalyze/c/pulse/pulseinf/
    kernel-scripts/* .
./termination-run-linux_kernel.sh
```

All results will appear in the file *pulseinf-results.json*. You can inspect these scripts to see 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.

6 Filtering alerts

Some projects are really large and looking at all alerts is itme consuming. To prioritize the review of the most important alerts, we provide project-specific strategies for wireshark and the linux kernel.

6.1 Linux kernel

Since the kernel is such a large target, we only look at alerts appearing in functions reachable from system calls or driver entry points as these are the likely most critical bugs. We extract these hits using the following helper scripts:

```
./dump_infinite_for_drivers.sh
./dump_infinite_for_syscalls.sh
```

Note: these scripts take a long time to execute as a post-processing phase using internal infer tool `infer-debug`. For the kernel, all found bugs were in a single device driver in the *ftdi-elan.c* file. This is easy to grep in the pulseinf-results.json file.

```
grep ftdi-elan pulseinf-results.json
```

6.2 Wireshark

Wireshark is another large codebase which contains a lot of intended non-termination due to UI widgets. In order to focus on unintended non-termination alerts, consider grepping out alerts in the UI code.

```
grep INFINITE_LOOP ./results/pulseinf-report-wireshark.json
```

References

[1] Julien Vanegue, Jules Villard, Peter O'Hearn and Azalea Raad "Non-Termination Proving: 100 Million LoC And Beyond" CAV'25 Submission