ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE | ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL | ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

# FloatZone: Accelerating Memory Error Detection using the Floating Point Unit

Floris Gorter*
f.c.gorter@vu.nl

Enrico Barberis*
e.barberis@vu.nl

Raphael Isemann
r.isemann@vu.nl

Erik van der Kouwe
vdkouwe@cs.vu.nl

Cristiano Giuffrida
giuffrida@cs.vu.nl

Herbert Bos
herbertb@cs.vu.nl

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

## A Artifact Appendix

## A.1 Abstract

In this artifact we provide the means to reproduce our main results. Specifically, we show that our memory sanitizer, FloatZone, can detect memory errors, and that FloatZone's performance is higher than traditional comparison-based solutions. We have validated the artifact using an Intel i9-13900K CPU running Ubuntu 22.04 with a stock v5.15 Linux kernel. Our source code is available at: `github.com/vusec/floatzone`.

## A.2 Description & Requirements

### A.2.1 Security, privacy, and ethical concerns

We require the evaluators to obtain the SPEC CPU benchmarking suites themselves, since we cannot distribute the licensed software. As a memory sanitizer, FloatZone poses no risks to the security of the target machine.

### A.2.2 How to access

The files for the artifact evaluation are available at: `https://github.com/vusec/floatzone/releases/tag/ae-final`.

### A.2.3 Hardware dependencies

While FloatZone has no strict hardware requirements (we assume x86-64), we highly recommend using a modern Intel CPU, since FloatZone's performance depends on the throughput of the floating point unit. We have ran benchmarking experiments on various CPUs (see Figure 6 for more information).

### A.2.4 Software dependencies

Some packages from the Ubuntu package manager are required to be installed to accomodate for the build process of FloatZone (e.g., for building LLVM). These are described in the Set-up section.

### A.2.5 Benchmarks

For this artifact we benchmark using the SPEC CPU2006 benchmarking suite.

## A.3 Set-up

We recommend using a bare-metal desktop system with 32GB of RAM, running Ubuntu 22.04, glibc 2.35, and a stock v5.15 Linux kernel.

### A.3.1 Installation

1. Obtain the artifact source:

```
git clone \
https://github.com/vusec/floatzone.git \
 --recurse-submodules
cd floatzone
```

2. Install some standard dependencies:

```
sudo apt install ninja-build cmake gcc-9 \
autoconf2.69 bison build-essential flex \
texinfo libtool zlib1g-dev
```

3. Configure the FloatZone environment by editing the `env.sh` file and modifying the `FLOATZONE_TOP` variable to reflect the working directory of the system, and then run:

```
source env.sh
```

4. Install the FloatZone infrastructure by running:

```
./install.sh
```

NOTE: installing LLVM can take up a lot of RAM when using multiple cores. If the compilation process crashes, use the `ninja -j <cores>` parameter inside `install.sh` to use less cores.

### A.3.2 Basic Test

To test the functionality of FloatZone, we provide a test case in the `example` directory. Run `make` to obtain three versions of the `buggy` binary: uninstrumented, instrumented by FloatZone, and instrumented by ASan. The program contains a buffer of size 16, and the command line argument is used as an index in this array. Confirm that executing:

```
./buggy_floatzone_run_base 16
```

results in an error report containing a faulting address, while using index 15 does not. See the README on GitHub for the exact expected output format.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *FloatZone can detect spatial and temporal memory errors bounded by its security guarantees (as described in Section 5). This is proven by experiment E1.*

**(C2):** *FloatZone provides high performance in terms of runtime and memory overhead (see Sections 7.3 and 7.4). This is proven by experiment E2.*

### A.4.2 Experiments

**(E1):** *[1 human-hour]: Confirming memory error detection.*

**How to:** *The Juliet Test Suite can be used to confirm that FloatZone detects memory errors. This suite contains test cases for spatial and temporal memory errors.*

**Preparation:** *Make sure that SEGFAULTS are reported: in the `runtime` directory, edit `wrap.c` and ensure that `CATCH_SEGFAULT` is set to 1. Run `make` inside this directory to ensure the shared object file is up-to-date. No further preparation is required if the `env.sh` and `install.sh` scripts have been used. If interested, FloatZoneExt (with partial overflow detection capabilities, see Section 5 and Figure 5) can be tested by modifying the FLOATZONE_MODE variable to also contain the term '`just_size`' in `env.sh`.*

**Execution:**
```
python3 run.py run juliet \
 floatzone_O0 --build --cwe 121 122 \
 124 126 127 415 416
```

**Results:** *FloatZone and FloatZoneExt can detect most of the spatial and temporal memory errors present in the Juliet Test Suite. The expected results are reported in Table 1 and Section 7.2.*

**(E2):** *[15 human-minutes + 5 compute-hours]: Confirming runtime and memory performance*

**How to:** Run the SPEC CPU2006 benchmarking suite instrumented by FloatZone and ASan, and observe the performance overhead.

**Preparation:** SPEC CPU2006 needs to be available on the system and the `FLOATZONE_SPEC06` variable in `env.sh` needs to point to the directory where it is installed. For the artifact evaluators, if they cannot obtain SPEC CPU2006, we can provide access to a machine ready to run SPEC. In order to run SPEC CPU and its benchmarks, we make use of a public infrastructure under the `infra` directory. The infra also makes sure the SPEC binaries run pinned to core 0. Make sure that the necessary python packages are installed:

```
pip3 install psutil terminaltables
```

Then, since some of the SPEC binaries contain false positives (see Table 3), in the `runtime` directory, edit `wrap.c` and ensure that `SURVIVE_EXCEPTIONS` is set to 1. Run `make` inside this directory to ensure the shared object file is up-to-date. As can be seen in the `wrap.c` source file, this only ensures that exceptions do not abort, and the program continues executing where it left off.

**Execution:** We make use of the `run.py` script to run SPEC CPU2006 along with the intended instrumentations. Execute the following command, which runs SPEC CPU2006 for three runs: the baseline, one with ASan, and one with FloatZone, and hence takes multiple hours:

```
python3 run.py run spec2006 default_O2 \
asan_O2 floatzone_O2 --build \
--parallel=proc --parallelmax=1
```

**Results:** To obtain the results from the SPEC CPU2006 runs, we again make use of the `run.py` script. Find the corresponding output folder in the `results` directory that matches the start timestamp (e.g.: `results/run.2023-06-19.13-56-59`). Then execute the following command, replacing the directory with the one just obtained:

```
python3 run.py report spec2006 \
results/run.2023-06-19.13-56-59 \
--aggregate geomean --field runtime:median \
maxrss:median
```

The output of this command can then be used to calculate the runtime and memory overheads for each individual binary, as well as for the geomean. As reported in Table 4: if ran on the i9-13900K machine, the expected runtime overhead for FloatZone is 36.4%, and 77.8% for ASan, while the memory overhead is expected to be 182% and 237%, for FloatZone and ASan, respectively.

## 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/.