IEEE Symposium on Security and Privacy ’26 Artifact Appendix: *SeqAss*: Using Sequential Associative Caches to Mitigate Conflict-Based Cache Attacks With Reduced Cache Misses and Performance Overhead

Wei Song[1,2], Zhidong Wang[1,2], Jinchi Han[1,2], Da Xie[1,2], Hao Ma[1,2], Peng Liu[3]
1. State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China
2. School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
3. The Pennsylvania State University, University Park, USA
Email: {songwei, wangzhidong, hanjinchi, xieda, mahao}@iie.ac.cn, pxl20@psu.edu

A Artifact Appendix

A.1 Abstract

SeqAss is a randomized cache structure using sequential associativity. Without relying on techniques intrusive to the traditional cache structure, such as cache skews, over-provided metadata space, and random replacement policy, SeqAss retains the set-associative structure and supports LRU. It achieves a defense as strong as Mirage. To be specific, SeqAss achieves around the same defense strength with Mirage in thwarting both Evict+Time and Prime+Probe attacks. Instead of raising cache miss rate, SeqAss actually reduces it by 11.4%, incurring the lowest cache miss rate in all existing randomized cache structures.

A.2 Description & Requirements

The artifact contains source code of the cache model simulator, i.e. FlexiCAS, the instruction level processor simulator, i.e. Spike, the evaluation programs, and instructions on running experiments. The artifact package is organized as follows:

`bin/` Common scripts used in installation and experiments.

`infrastructure/` Tools installed locally for running experiments.

`artifact/spike-flexicas/` The modified Spike simulator.

`artifact/spike-flexicas/flexicas/` The cache model implementing the behavioral level model of all cache structures.

`artifact/spike-flexicas/flexicas/rsa` The source code of the proposed SeqAss cache.

`claims/` Scripts for running experiments.

`env.sh` Script for setting up environment variables.

`install.sh` Script for installation.

A.2.1 Security, privacy, and ethical concerns

All of the experiments described in this artifact execute on simulation models and present no risk to the host machine running the models. All experiments in this artifact runs inside the artifact directory with no read/write to the outside. Some commonly utilized packages are installed to the OS. Section A.2.4 provides a detailed list of them.

A.2.2 How to access

The artifact is uploaded to the evaluation system as a tar file, while the up-to-date version is also accessible through a public github repo: `https://github.com/comparch-security/sp2026-seqass.git`, which will be packaged along with all source code and uploaded to Zenodo for permanent accessibility in final version of the artifact.

A.2.3 Hardware dependencies

The artifact should work on any Linux running on an modern Intel machine, but we prefer Ubuntu and the artifact was prepared on Ubuntu 22.04 on a Xeon-5220 machine. We recommend running the artifact on a machine with at least 32GB memory and 8 processing cores. Most of the experiments are launched in parallel runs by default.

A.2.4 Software dependencies

Most of the source files are written in C++. Some of them requiring C++20. We recommend GNU GCC version $\geq$ 11.4.

Besides GNU GCC, the following packages are installed during the installation: autoconf, automake, autotools-dev, curl, python3, python3-pip, libmpc-dev, libmpfr-dev, libgmp-dev, gawk, build-essential, bison, flex, texinfo, gperf, libtool, patchutils, bc, zlib1g-dev, libexpat-dev, libglib2.0-dev, sshfs, device-tree-compiler, libboost-regex-dev, libboost-system-dev, libboost-program-options-dev, python3-dev, libpixman-1-dev, unzip, rsync, wget, cpio, gnuplot, xterm.

A.2.5 Benchmarks

The SPEC CPU 2017 benchmark is used in this paper for evaluating the runtime performance of different randomized cache structures. However, this benchmark is a proprietary software that cannot be publicly distributed. For the purpose of artifact evaluation, a compiled set of binaries of the SPEC CPU 2017 rate subset is provided at: https://github.com/comparch-security/sp2026-seqass/releases/download/v1.0.0/spec2017-rate.zip, which will be removed after the evaluation. Detailed script for compiling the benchmark is provided in Section A.5.

A.3 Set-up and installation

Please extract the artifact uploaded to the HotCRP system, or clone the up-to-date artifact from github:

```
  git clone \
https://github.com/comparch-security/sp2026-seqass
```

Afterwards, run the following installation commands:

```
  cd sp2026-seqass
  source install.sh
```

which will download the source code into `artifact/spike-flexicas`, compile the FlexiCAS cache model, and finally install Spike simulation.

SPEC CPU 2017 is required for running experiment **E4**. This can be down by downloading https://github.com/comparch-security/sp2026-seqass/releases/download/v1.0.0/spec2017-rate.zip and extract it to the correct directory:

```
  unzip -P sp2026 spec2017-rate.zip -d \
$RISCV/spec2017
```

At the beginning of all experiments, please ensure the environment variables are set properly. If in doubt, `echo $CM` should point to the `flexicas` directory. Source `env.sh` in the artifact directory should have all variables set properly.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SeqAss cache achieves similar defense strength against Evict+Time attacks with the Mirage cache as described in Section 5.1. This is proven by an eviction rate estimation using cache models (**E1**), whose results are depicted in Figure 10.

(C2): SeqAss cache demonstrates strong defense against both Prime+Probe and prefetch-based Prime+Probe attacks as described in Section 5.2. This is proven by evaluating the success rate of both Prime+Probe attacks (**E2.1**) and prefetch-based Prime+Probe attacks (**E2.2**) in cache models. The results are illustrated in Table 5 and 6 and depicted in Figure 11 and 12.

(C3): SeqAss cache successfully prevents attackers from collecting congruent addresses using CTPP and PPP while slowing down the speed of CT and CT-prefetch as described in Section 5.3. This is proven by searching congruent addresses using CT, CT-prefetch, CTPP and PPP (**E3**) in cache models. The result is described in Table 7.

(C4): SeqAss cache significantly reduces LLC miss rate by $\sim$11% as described in Section 5.4. This is proven by running the SPEC CPU 2017 benchmark (**E4**) on the Spike simulation model attached with different cache models. The result is described in Table 9 and Figure 14.

A.4.2 Experiments

(E1) Eviction rate estimation (2 hours): This experiment tries to evict a target address from a cache using eviction sets comprising different numbers of congruent addresses. The size of the eviction sets required to evict the target address increases significantly for the caches providing stronger defense against Evict+Time attacks. SeqAss achieves similar strength with Mirage.

Execution: Run:

```
  cd $ROOT$/claims/claim1/exp
  source run.sh
```

Results: Seven tests for all cache structures, i.e. `baseline`, `sp2021`, `ceaser-s`, `chameleon`, `mirage-50`, `mirage-75`, and `seqass`, are launched and running in parallel. Please wait until all tests are finished. The result will be recorded in log files. Expected result is provided in directory `expected`. In the result,

```
  evset size = 166: 4/5
```

denotes using an eviction set of 166 congruent addresses has successfully evicted the target address for 4 out of 5 times. 80% is the estimated eviction rate. *Shown by the result, evicting the target address on SeqAss and Mirage requires thousands of congruent addresses, while tens of addresses are sufficient for other cache structures. SeqAss is as secure as Mirage.*

Note: Due to the random nature of the experiment, results would not be exactly the same with the expected. The provided experiment is a shortened version running only 5 samples for each size of the eviction set. To reproduce the results depicted in Figure 10, the sample number should rise to 1000 and warm-up between tests are required. See the commented lines in script `run-eviction-rate`. Execution time rises accordingly.

(E2.1) Prime+Probe attack (1 hour if escaping mirage-50 and mirage-75): This experiment launches Prime+Probe attacks on different caches using eviction sets sets comprising different numbers of congruent addresses. The experiment calculates the F-scores for the success rate of attacks using different eviction set sizes. It is shown that both SeqAss and Mirage fail Prime+Probe attacks as the F-score remains at 0.

Execution: Run:

```
cd $ROOT$/claims/claim2/exp1
source run.sh
```

Results: Seven tests for all cache structures are launched and running in parallel. Please wait until all tests are finished. The result will be recorded in log files. Expected result is provided in directory `expected`. In the result,

```
evset size = 41: [6,4,10,0] F1=0.75
```

denotes using an eviction set of 41 congruent addresses in 20 Prime+Probe attacks (10 positive and negative each), 6 true-positive (TP), 4 false-negative (FN), 10 true-negative (TN), and 0 false-positive (FP) cases have been recorded. These result in an F1 score of 0.75. *Shown by the result, Prime+Probe attacks fail on both SeqAss and Mirage.*

Note: Due to the random nature of the experiment, results would not be exactly the same with the expected. The provided experiment is a shortened version running only 20 samples for each size of the eviction set. To reproduce the results depicted in Figure 11, the sample number should rise to 1000. See the commented lines in script `run-prime-probe`. Execution time rises accordingly.

(E2.2) Prefetch-based Prime+Probe attack (45 minutes if escaping mirage-50 and mirage-75): This experiment launches prefetch-based Prime+Probe attacks on different caches using eviction sets sets comprising different numbers of congruent addresses. The experiment calculates the F-scores for the success rate of attacks using different eviction set sizes. It is shown that both SeqAss and Mirage fail prefetch-based Prime+Probe attacks as the F-score remains at 0.

Execution: Run:

```
cd $ROOT$/claims/claim2/exp2
source run.sh
```

Results: Please proceed in the same way with **E2.1**. *Shown by the result, prefetch-based Prime+Probe attacks fail on both SeqAss and Mirage.*

Note: To reproduce the results depicted in Figure 12, the sample number should rise to 1000. See the commented lines in script `run-prime-probe-prefetch`. Others, proceed in the same way with **E2.1**.

(E3) Eviction set search algorithms (17 minutes): This experiment searches congruent addresses using various search algorithms (CT, prefetch-based CT, CTPP, and PPP) on different cache structures. The experiment records the success rate of finding congruent addresses and the averaged number of cache evictions incurred by each search. It is shown that SeqAss fails CTPP and PPP, but it is possible to search congruent addresses using CT and prefetch-based CT (CT-prefetch).

Execution: Run:

```
cd $ROOT$/claims/claim3/exp
source run.sh
```

Results: Please note that there are 28 combinations of search algorithms and cache structures. The `run.sh` will launch 28 separate processes running all combinations in parallel. If this is too heavy for you machine, please read the script and run manually. The result will be recorded in log files. Expected result is provided in directory `expected`. At the end of the result, the success rate and averaged number of evictions are reported, e.g.

```
success rate: 0.89 using 341885 evictions.
```

denotes that the search algorithm achieve a 89% success rate in searching congruent addresses while each search incurs around 341.9K cache evictions. *Shown by the result, CT and CT-prefetch can be used to search congruent addresses on SeqAss while CTPP and PPP fail. The number of evictions is increased from 26K to 34K.*

Note: Due to the random nature of the experiment, results would not be exactly the same with the expected. The provided experiment is a shortened version running only 20 samples for each search algorithm on a specific cache. To reproduce the results depicted in Table 7, the sample number should rise to 1000. See the commented lines in scripts `run-ct`, `run-ct-prefetch`, `run-ctpp`, and `run-ppp`. Execution time rises accordingly.

(E4) SPEC CPU 2017 benchmark (50 minutes if run in parallel): This experiment runs the SPEC CPU 2017 benchmark on the Spike simulation, which boots a Linux kernel. The experiment records runtime traces and summarizes the cache performance for running each benchmark cases. It is shown that SeqAss incurs the lowest cache misses and memory accesses in all cache structure.

Execution: Tests on different cache structures needs to execute independently and these tests MUST execute sequentially, as the Spike would using the file system to record logs in the background and parallel runs would corrupt the logs. However, it is possible to have multiple copies of the whole `sp2026-seqass` artifact and run a Spike from each copy. For each cache structure in `[baseline, sp2021, ceaser-s, chameleon, mirage-50, mirage-75, seqass]`, run the following command (using `seqass` as an example):

```
cd $ROOT/claims/claim4/exp

# re-compile Spike to update the LLC
./update-spike seqass

# launch Linux on Spike
xterm-spec &

# inside the booted Linux
# Login use id: root
# execute: run-spec --run-size 200
# CTRL-C and q after execution finished

# post result analysis
./gen_sum.py < summary.csv > seqass.csv

# clean up
rm *.log summary.csv $SPECKLE/*.default
```

Results: The `gen_sum.py` script produces the summarized result. Expected result is provided in directory `expected`. In the CSV files, the column labeled "MPKI" records the MPKI of the LLC for each benchmark cases. The final two row shows the average and geomean of the all benchmark caces. We use the geomean of MPKI for comparison between cache structures. *Shown by the result, SeqAss achieves the lowest MPKI in all cache structures. Compared with baseline, SeqAss actually reduces MPKI.*

Note: Due to the random nature of the experiment, results would not be exactly the same with the expected. The provided experiment is a shortened version running only 200M instructions for each benchmark case. To reproduce the results depicted in Table 9, run the SPEC CPU 2017 benchmark for 10G instructions per benchmark case:

```
# inside the booted Linux
# execute: run-spec --run-size 10000
```

Execution time rises accordingly.

Please note that case 525.x264_r is removed from the artifact evaluation due to its size. The input of this case takes 1.7GB storage which is too large for convenient file sharing. If you have the full SPEC CPU 2017 benchmark, Section A.5 provides script for compiling the benchmark locally.

A.5 Notes on Reusability

The cache model used in this work, i.e. FlexiCAS, can be directly used to quickly investigate different cache structures, defenses and attacks. The test programs in `artifact/spike-flexicas/flexicas/alg` provide good examples on how to drive the cache model and collect performance information.

When combined with Spike, FlexiCAS is used to replace the internal cache model of Spike and provide much more accurate cache performance. `artifact/spike-flexicas/flexicas/spike-cache.cc` is the interface file used by Spike to initiate its cache model using FlexiCAS.

Running the SPEC CPU 2017 benchmark would require booting a Linux on Spike and compile the benchmark into RISC-V executable binaries. Both of which require a RISC-V cross-compiler. In this research, we have compiled and installed a local copy of `https://github.com/riscv-collab/riscv-gnu-toolchain/releases/tag/2023.12.12`. The Linux kernel is generated from Buildroot using `https://github.com/sycuricon/riscv-spike-sdk`. An installation script is provided as `bin/install-spike-sdk`.

The SPEC CPU 2017 benchmark is compiled using `https://github.com/ccelio/Speckle`. We have made some changes in the compiler flags to make the benchmark pass the compiler. These changes and the compilation instructions are provided in `https://github.com/comparch-security/speckle-2017`.