USENIX Security '25 Artifact Appendix: Systematic Evaluation of Randomized Cache Designs against Cache Occupancy

Anirban Chakraborty*† Nimish Mishra*‡ Sayandeep Saha§
Sarani Bhattacharya‡ Debdeep Mukhopadhyay‡

†Max Planck Institute for Security and Privacy, Germany
‡Indian Institute of Technology Kharagpur, India
§Indian Institute of Technology Bombay, India

anirban.chakraborty@mpi-sp.org nimish.mishra@kgpian.iitkgp.ac.in {sarani, debdeep}@cse.iitkgp.ac.in sayandeepsaha@cse.iitb.ac.in

A Artifact Appendix

A.1 Abstract

This work performs a systematic evaluation of 5 randomized cache designs- CEASER, CEASER-S, MIRAGE, ScatterCache, and SassCache against cache occupancy wrt. *both* performance as well as security. This work fills in a crucial gap in current literature on randomized caches: currently most randomized cache designs defend only contention-based attacks, and leave out considerations of cache occupancy. With respect to performance, this work proposes a new and uniform benchmarking strategy, which allows us to perform a fair and comparative analysis across all designs under various replacement policies. Likewise, with respect to security against cache occupancy attacks, this work evaluate the cache designs against various threat assumptions: ① covert channels, ② process fingerprinting, and ③ AES key recovery (to the best of our knowledge, this work is the first to demonstrate full AES key recovery on a randomized cache design using cache occupancy attack). The main takeaway of our work is to establish the need to *also* consider cache occupancy side-channel in randomized cache design considerations.

In this artifact appendix, we detail steps to reproduce our results from the paper. We first detail how to build and run the different randomized cache designs in gem5. MIRAGE and ScatterCache are already open-sourced, while CEASER and CEASER-S have been implemented in-house by us and open-sourced. The implementation of SassCache has been generously contributed by SassCache's authors and adapted to our setup. Thereafter, we detail steps to reproduce the various experiments related to performance and security analysis detailed in the paper.

*Equal Contribution.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact does not have any destructive tendencies, and is thus safe to use. Moreover, we do not attack any commercial library/implementation in this work, and thereby do not come under the purview of disclosure and release of security advisories.

A.2.2 How to access

The permanent link for the source codes used in this paper can be found at: `https://github.com/SEAL-IIT-KGP/randomized_caches/tree/main` (tag: v3.0). The artifact is also hosted at Zenodo; url: `https://doi.org/10.5281/zenodo.15198048`.

A.2.3 Hardware dependencies

We do not have any specific requirements for hardware. However, it is recommended to have at least 16 MB of RAM for smooth gem5 simulations. For all our experiments reported in this appendix, we used Intel(R) Xeon(R) Gold 6226*R* CPU as the evaluation platform (running Ubuntu 20.04.6 LTS, 32 GB RAM).

A.2.4 Software dependencies

We have the following major requirements: `gcc-9`, `g++-9`, `python2.7`, and `virtualenv`. ScatterCache specifically requires `gcc-7` and `g++-7`. Other dependencies include: `git`, `python2.7-dev`, `wget`, `curl`, `pip`, `zlib`, `HDF5`, `M4 macroprocessor`, `matplotlib`, and `python-dev`. Rest dependencies are installed automatically by the setup scripts (detailed below).

A.2.5 Benchmarks

Experiments related to performance benchmarking and fingerprinting attack require SPEC2017 should our results be reproduced from scratch. Alternatively, the results can also be derived from the in-house datasets related to these experiments that we open-source. We do not provide SPEC2017 sources as part of the artifact due to license constraints.

A.3 Set-up

A.3.1 Installation

Navigate to the subdirectory `randomized_cache_hello_world/` and execute `setup.sh`. This script sets up a virtual environment in Python 2.7, installs the needed dependencies (like `scons` and `six`), and builds the gem5 binaries of all cache designs.

The bottleneck step in this installation is the number of threads used to build gem5. The parameter `THREADS` in `setup.sh` controls the amount of parallelization: more threads imply shorter build time, but also uses more RAM. We have set `THREADS` to a conservative 1, but recommend tweaking it based on the system configuration.

A.3.2 Basic Test

The subdirectory `randomized_cache_hello_world/` contains scripts: `run_ceaser.sh`, `run_ceaser_s.sh`, `run_mirage.sh`, `run_scatter.sh`, and `run_sass.sh`. Each of these scripts build a sample binary (i.e. `spurious_occupancy.c`) and executes the same within the gem5 simulation of its respective cache architecture. A successful simulation is implied by an exit message of form:

***Spurious Occupancy step finished**. Exiting @ tick 67659457149 because exiting with last active thread context*

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Performance Evaluation: One of the main claims of the paper is that prior benchmarking techniques are unsuitable for a fair evaluation of modern randomized cache designs. Section 3.3 introduces the idea of benchmarking under spurious occupancy. We claim this benchmarking strategy provides a fair evaluation for all cache designs by (1) removing any implementation specific assumption, and (2) harmonizing the parameters considered for benchmarking different designs.

(C2): Performance Evaluation under different Replacement Policies: This work also shows the effect of replacement policies on the performance of different cache designs, allowing for a better understanding and takeaways for the performance different cache designs. This claim pertains to Section 3.4 in the main paper.

(C3): Covert Channel: An important claim of the paper is that certain kinds of randomized cache designs alleviate creation of covert channels. We claim that design rationale based on pseudo-fully associative design are more amenable to creation of covert channels than set-associative designs. This claim pertains to Section 5 in the main paper.

(C4): Process Fingerprinting: We claim that pseudo-fully associative design are more amenable to process fingerprinting side-channel attacks than set-associative designs. This claim pertains to Section 6 in the main paper.

(C5): AES key recovery: We claim that pseudo-fully associative design are more amenable to AES key recovery side-channel attacks than set-associative designs. This claim pertains to Section 7 in the main paper.

A.4.2 Experiments

(E1): *[Performance Evaluation] [10 human-minutes + 24 compute-hours + 3GB disk]:* To reproduce results related to claim **C1**.

How to: Navigate to subdirectory `perf_runs`. It has the scripts *run_baseline_benchmark.sh*, *run_ceaser_benchmark.sh*, *run_ceaser_s_benchmark.sh*, *run_mirage_benchmark.sh*, *run_sass_benchmark.sh*, and *run_scatter_benchmark.sh* that help run SPEC2017 benchmarks on the associated cache designs.

Preparation: The environment variable `BASE_DIR` must point to the top-level directory of the artifact, while the environment variable `SPEC_PATH` points to the SPEC2017 installation.

Execution: Run one of the scripts, providing a benchmark name as input. The list of supported benchmarks is captured in the README file of the `perf_runs` subdirectory. For example, *bash run_mirage_benchmark.sh blender* runs blender on MIRAGE.

Results: The collected measurements must be put into `perf_runs/perf_data/randomized_caches.xlsx` to reproduce the results in Section 3.3.

Data: Alternatively, we also provide our dataset in `perf_runs/perf_data/`. The sub-directory names are a combination of the cache design and the replacement policies. To reproduce results related to claim **C1**, consider only the subdirectories with suffix `RandomRP`.

Each subdirectory contains further directories for each benchmark. For instance, `ceaser_perf_runs_RandomRP/gcc` contains the `stats.txt` for running the SPEC2017 benchmark gcc on CEASER under the default replacement pol-

icy. A helper script `compile_stats.sh` will aid in constructing the requisite data from these sub-directories

We have already abstracted out results of `compile_stats.sh` for each subdirectory of `perf_runs/perf_data/` into `perf_runs/perf_data/randomized_caches.xlsx`. This spreadsheet contains formulae for automatically computing the LLC miss ratio, normalize it against the baseline, and perform performance evaluation. It reproduces Table 2 of the paper, as well as computes data used by the scripts in `$BASE_DIR/scripts` which plot Figure 5 in the main paper..

(E2): *[Performance Evaluation under Replacement Policies] [10 human-minutes + 24 compute-hour + 3GB disk]:* To reproduce results related to claim **C2**.

How to: Same as **E1**.

Preparation: Same as **E1**.

Execution: Same as **E1**, except that the replacement policy can also be provided as input. Refer to the README in the artifact for more details.

Results: Same as **E1**

Data: Same as **E1**. This experiment reproduces Table 2 of the paper.

(E3): *[Covert Channel] [10 human-minutes + 8 compute-hours]:* To reproduce results related to claim **C3**.

How to: Navigate to sub-directory `llc_simulator`.

Preparation: None. This experiment does not require any additional setup. The source codes are mostly written in `python3` and uses basic packages such as `numpy` and `matplotlib`.

Execution: Execute `bash run_simulation.sh`. This script executes occupancy based covert channel setup for MIRAGE, CEASER, ScatterCache, and Baseline. The scripts collect covert channel data and write to `outfile_*.txt` for both bits 0 and 1. Finally, `covert_channel_plot.py` plots the collected data.

Results: This experiment validates the claim **C3**. Overall, this experiment establishes that randomized caches that rely on pseudo-fully associative design rationale are susceptible to covert channel setup (with high accuracy) even with 10% accuracy. On the other hand, comparatively, it is difficult to setup a covert channel of acceptable accuracy with 10% accuracy.

(E4): *[Process Fingerprinting] [10 human-minutes + 48 compute-hours + 8GB disk]:* To reproduce results related to claim **C4**.

How to: Navigate to sub-directory `fingerprinting`.

Preparation: Same as **E1** and **E2** since this experiment requires setting up SPEC2017.

Execution: Follow the steps README illustrates in the subdirectory. Essentially, this experiment is similar to **E1**, except that the benchmarks are randomly selected.

Results: Use the scripts `ceaser.py`, `ceaser_s.py`, `mirage.py`, `sass.py`, and `scatter.py` to construct accuracy metrics for the fingerprinting experiment.

Data: We have open-sourced the data collected internally in `fingerprinting/data/` as `*.log` files. Running the scripts `ceaser.py`, `ceaser_s.py`, `mirage.py`, `sass.py`, and `scatter.py` on these log files creates the data used to construct Figure 8 in the main paper (and the inferences drawn in Section 6).

(E5): *[AES] [10 human-minutes + 1 compute-hour + 0.5 GB disk]:* To reproduce results related to claim **C5**.

How to: Navigate to the directory `aes`. It has two subdirectories: `aes_profiled_key` and `aes_victim_key`. The C implementations of the AES T-Box within these subdirectories contain the attack and victim code. `aes_profiled_key` contains the key being used for profiling: **0xffeeddccbbaa99887766554433221100**. On the other hand, `aes_victim_key` contains the key being recovered from the constructed profile: **0x7766554433221100ffeeddccbbaa9988**. All data is collected at 50% occupancy (Section 7.3 and 7.4) and suffices to establish the reproducibility of our attack.

Preparation: None

Execution: The `analysis` subdirectory contains the dataset for each randomized cache design. For each cache design, we provide a script `analysis.py` that constructs the profile and mounts the attack (as per Section 7.1 of the paper), as well as a script `guessing_entropy.py` that computes the guessing entropy.

Results: The computed guessing entropy for the cache design comes out as: MIRAGE : 33.0, CEASER-S : 92.0, SassCache : 104.0, and ScatterCache : 100.0. Note that the results in Section 7.3 have been compiled over runs over multiple different keys while the artifact contains representative runs of a single key. This implies the exact value of the guessing entropy in the paper is different that what these scripts output. However, the *general trend* of strength/weakness of each design against AES key recovery attacks still stands. More precisely:

1. GE for MIRAGE: Paper [30.05], Representative data in artifact [33]
2. GE for SassCache: Paper [109.31], Representative data in artifact [104]
3. GE for ScatterCache: Paper [100.97], Representative data in artifact [100]

4. GE for CEASER-S: Paper [90.56], Representative data in artifact [92.0]

As evident, MIRAGE is the weakest with GE . Then comes CEASER-S with a GE of ; key recovery is not possible with this GE, but CEASER-S does show leakage. Then we have ScatterCache with a GE of 3, demonstrating *some* leakage, but not exploitable. Finally, SassCache is the most resilient against our attacks.

AES data collection from scratch: We encourage the user to try out data collection with different keys to get a trend (and GE) closer to what is reported in the paper. However, as also noted in the paper (footnote 21), the rate of data collection is at best 500 observations per hour. We were able to thereby deploy about 350 cores per Intel Xeon server, across three such servers. The overall data collection for *all* designs and multiple keys took over 2 weeks of compute hours. We thus do not include any data collection experiment for AES in artifact evaluation because of the abnormally high runtime of completing the experiment.

Note that such an inhibitory rate is *not a problem of attack design*, but rather is the consequence of the gem5 simulations (which is the go-to simulation strategy for state-of-the-art randomized cache literature). In a realistic setting (when these randomized caches are deployed on real hardware), our attack will be much faster.

A.4.3 Plot Scripts

We also provide the scripts in `scripts/` that we used to generate the plots in the paper.

A.5 Notes on Reusability

The findings of this thereby establish an interesting open problem: Design of a randomized cache of comparable efficiency with modern set-associative LLCs, while still resisting *both* contention-based and occupancy-based attacks. We hypothesize that randomized caches designed around dynamically changing partitions would be capable of restricting LLC occupancy for critical workloads (thereby providing security), while also generously allowing LLC occupancy for non-critical workloads (thereby providing better performance in general). More research however is required to concretely establish how such a cache design shall function, how process requests for dynamic changes in partitioning will be handled, and how to implement this design in an uncomplicated manner. For such future research, our artifacts will enable designers to test upcoming designs against occupancy attacks as well.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

A.7 Acknowledgements

We would like to thank the authors of SassCache who very generously provided us with their custom simulator for SassCache upon request, which we have adapted as part of this artifact.