

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

PROBE+DETECT+MITIGATE (PDM): Enabling Cloud Tenants to Self-Defend against Microarchitectural Attacks

Arash Daneshmand, *University of British Columbia Okanagan and Concordia University;* Hugo Kermabon-Bobinnec, *Concordia University;* Lingyu Wang, *University of British Columbia Okanagan and Concordia University;* Makan Pourzandi, *Ericsson Security Research, Ericsson Canada;* Suryadipta Majumdar, *Concordia University;* Yosr Jarraya, *Ericsson Security Research, Ericsson Canada*

https://www.usenix.org/conference/usenixsecurity26/presentation/daneshmand

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology





USENIX Security '26 Artifact Appendix: PROBE+DETECT+MITIGATE (PDM): Enabling Cloud Tenants to Self-Defend against Microarchitectural Attacks

Arash Daneshmand[2,1✉], Hugo Kermabon-Bobinnec[1],
Lingyu Wang[2,1✉], Makan Pourzandi[3], Suryadipta Majumdar[1], and Yosr Jarraya[3]

[1]CIISE, Concordia University, Montreal, QC, Canada
[2]School of Engineering, University of British Columbia, Kelowna, BC, Canada
[3]Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{hugo.kermabonbobinnec,arash.daneshmand, suryadipta.majumdar}@concordia.ca
lingyu.wang@ubc.ca
{makan.pourzandi,yosr.jarraya}@ericsson.com

A Artifact Appendix

A.1 Abstract

This artifact demonstrates the functionality of the proposed detection and mitigation mechanisms as mentioned in the Open Science Section of our paper. It confirms that the provided artifacts correspond to the system design presented in the paper. The artifact includes the source code of PDM's detection and mitigation components, including probing schemes, runtime ML-based inference using ONNX, and in-memory encryption. It also includes the trained models and datasets used in our experiments, which cover the normal activities of real-world applications, attack activities, natural noises from SPEC CPU 2017, artificial noises from memory stressors, and slow evasive attacks. The complete compilation and usage instructions are included in this appendix and also in the README files provided.

A.2 Description & Requirements

This section lists all the necessary details to recreate the same experimental setup we have used to test the artifact.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, and ethical concerns for evaluators while running our artifact on their machine. No destructive steps such as disabling security features are taken in the process. The artifact uses standard user-space techniques and evaluators should run it only on processes they own.

A.2.2 How to access

Our artifact is accessible through Zenodo at https://doi.org/10.5281/zenodo.17967755, and additionally maintained in Github at https://github.com/arashd021/pdm.

A.2.3 Hardware dependencies

We implemented and evaluated our artifact using (i) an in-house testbed with an Intel Xeon Gold 5120 (Skylake) CPU with 19.25MB L3 cache, and 1MB L2 cache per core, and (ii) a Kubernetes cluster deployed on Amazon EKS based on Amazon Fargate with one vCPU and 1GB of memory. The artifact requires an x86-64 machine with a minimum of one hardware thread (as PDM does not require any additional thread).

A.2.4 Software dependencies

We tested our artifact on a fresh Ubuntu 24.04 installation as of December 2025 but it should run on any x86-64 Linux operating system. The following software dependencies are required to run the artifact:

- Python (detection datasets and models were tested with Python 3.9+)
- Python packages (pip): pandas, numpy, torch, scikit-learn, joblib (to test the detection datasets and models)
- Jupyter (to run the .ipynb notebooks)
- System build tools: gcc / g++ / make / util-linux
- libcapstone-dev (required by mitigation)
- uthash (required by mitigation)

- libssl-dev (required by OpenSSL-based victim examples for mitigation)
- libwolfssl-dev (required by wolfSSL victim examples for mitigation)
- ONNX Runtime (to enable low-overhead runtime ML inference in C)
- The CAP_SYS_PTRACE Linux capability (for runtime thread injection).
- Docker (to run the provided container for mitigation)

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Install the required software dependencies:

```
$ sudo apt update && sudo apt install -y
build-essential libcapstone-dev libssl-dev
libwolfssl-dev uthash-dev python3-pip
```

Install ONNX Runtime (v1.22.0) and set `ORT_DIR` (expected by the detection build):

```
$ cd ~
$ wget https://github.com/microsoft/
onnxruntime/releases/download/v1.22.0/
onnxruntime-linux-x64-1.22.0.tgz
$ tar -xvf onnxruntime-linux-x64-1.22.0.tgz
$ export ORT_DIR=$HOME/onnxruntime-linux-x64-1.2
2.0
```

Download the latest release of the artifact from Zenodo and extract it:

```
$ cd ~
$ wget <Zenodo Download URL>
$ unzip <PDM Zipped File>
$ cd <PDM Directory>
```

Install Python dependencies:

```
$ cd datasets
$ pip install -r requirements.txt
```

Build the artifact:

```
$ cd ../source/detection && make
$ cd ../mitigation && make
```

PDM's probing requires three distinct thresholds for different cache levels. Since these thresholds are different across environments, we provide a calibration tool to automatically identify these thresholds. Simply run the tool:

```
$ cd source/detection
$ ./calibration
```

You should then set the calibrated thresholds by updating the threshold constants (`THR_L1`, `THR_L3`, `THR_MISS`) at the top of *PDM-detection.c*, *PDM-scheme3.c*, and *PDM-scheme3+.c*, and recompile the detection components.

Finally, install the runtime `.so` library injection tool:

```
$ git clone https://github.com/kubo/injector
$ cd injector && make
```

A.3.2 Basic Test

Detection. The `example/fr` directory contains an example victim application, a FLUSH+RELOAD attack, and integration with PDM for detection:

```
$ cd source/detection/example/fr
$ make
```

The flag `#define USE_PROC_MAPS` in PDM's probing codes (enabled by default), allows the probing range to be automatically identified. Run the victim binary with PDM integrated:

```
$ sudo LD_PRELOAD=../../PDM-detection.so taskset
-c 0 ./victim
```

You should see benign datapoints where the L1hit ratio is 100% and the model output is 0:

```
# 14:55:43.967 l1hit: 100.00 ** l3hit: 0.00 **
miss: 0.00 ** bigmiss: 0.00 ===== l1hit2: 0.00
** l3hit2: 0.00 ** miss2: 100.00 ** bigmiss2:
0.00 | Model Output: 0.000000 ** batch_offset:
2560
```

Next, in another terminal, run the attack

```
$ ./F+R
```

You should see a decrease in L1 hit ratio and an increase in model output:

```
# 14:57:37.019 l1hit: 25.00 ** l3hit: 0.00 **
miss: 62.50 ** bigmiss: 12.50 ===== l1hit2: 0.00
** l3hit2: 0.00 ** miss2: 100.00 ** bigmiss2:
0.00 | Model Output: 0.800035 ** batch_offset: 0
```

Mitigation. A ready-to-use Docker image with precompiled ELF 64-bit binaries of different cryptographic primitives with and without the in-memory encryption mitigation is provided. Run the container interactively:

```
$ docker run --rm -it docker.io/pdmanon123/
pdm-artifacts
```

You can then run each binary in two modes (with and without protection). For instance:

```
$ ./ecdh 1 #without protection
$ ./ecdh 2 #with protection
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PDM *detects cache-based side-channel attacks (e.g., FLUSH+RELOAD) using only tenant-accessible features (i.e., user-space probing and ONNX-based inference), without requiring cloud-provider support. This is validated by experiment (E1).*

(C2): PDM *detects attacks relying on speculative memory accesses (e.g., Spectre variants) using the same tenant-side workflow and features, generalizing beyond eviction-based cache attacks. This is validated by experiment (E2).*

(C3): PDM *protects secrets via the paper's in-memory protection mechanism (instruction self-declaration and on-the-fly inline binary rewriting), preserving application correctness across multiple crypto libraries and primitives. This is validated by experiment (E3).*

(C4): PDM *remains effective under realistic noise and evasion conditions, including (i) natural co-resident noise (SPEC CPU 2017) on a local testbed, (ii) artificial memory stressors on a local testbed, (iii) real-cloud noise on AWS Fargate, and (iv) slow/evasive attacks. This is validated by experiments (E4).*

A.4.2 Experiments

(E1): *[FLUSH+RELOAD detection on OpenSSL AES]* [30 human-minutes + 20 compute-minute] This experiment reproduces a FLUSH+RELOAD attack against OpenSSL AES-128 (T-table implementation) and demonstrates that PDM's detection produces clearly separable outputs between benign execution and attack execution.

Preparation: This experiment requires a self-compiled version of OpenSSL to enable its T-Table-based AES implementation. Build and install a vulnerable OpenSSL with T-table AES enabled (OpenSSL 1.1.0f) under `/usr/local/ssl_vuln`, then compile the provided attacker/victim programs (this will not replace your currently installed OpenSSL).

```
$ cd source/detection/example/fr-aes
$ wget https://www.openssl.org/source/
openssl-1.1.0f.tar.gz
$ tar -xvf openssl-1.1.0f.tar.gz
$ cd openssl-1.1.0f
$ ./config -d shared no-asm no-hw --prefix=/
usr/local/ssl_vuln
$ sudo make && sudo make install_sw
```

Ensure runtime can find the vulnerable libcrypto:

```
$ export LD_LIBRARY_PATH=/usr/local/ssl_vuln
/lib:$LD_LIBRARY_PATH
```

Compile the applications:

```
$ cd ..
$ g++ spy.cpp -o spy -I/usr/local/ssl_vuln/i
nclude -L/usr/local/ssl_vuln/lib/ -lcrypto
$ gcc aes-workload.c -o aes-workload -I/usr/
local/ssl_vuln/include -L/usr/local/ssl_vul
n/lib/ -lcrypto
```

PDM can be configured to use fixed address ranges (e.g., AES T-tables addresses) in `source/detection/PDM-detection.c`. T-table addresses can simply be found by running the binary (keep the process running so addresses do not change across runs due to ASLR) :

```
$ ./spy
Probing address (Te0): 0x7ffff6fdfc00
Probing address (Te1): 0x7ffff6fdf800
Probing address (Te2): 0x7ffff6fdf400
Probing address (Te3): 0x7ffff6fdf000
```

Verify that the following macros are set using the address of `Te3` and recompile PDM:

```
#define USE_FIXED_START_ADDR
#define START_ADDR 0x7ffff6fdf000
#define SIZE 3072
```

Execution: Run the spy process, then start the AES workload, inject `PDM-detection.so` into the spy process, and toggle the attack on/off using signals.

```
# Terminal 1: spy already running (printed
probed Te* addresses)
```

```
# Terminal 2: start continuous AES
encryptions (victim workload)
$ ./aes-workload 1
```

```
# Terminal 3: inject PDM detection into the
spy process
```

```
$ cd source/detection/injector
$ sudo ./cmd/injector -p $(pidof spy)
PDM-detection.so
# Toggle attack (in the spy process)
# Enable attack
$ kill -SIGUSR1 $(pidof spy)
# Disable attack
$ kill -SIGUSR2 $(pidof spy)
```

Results: PDM prints the multivariate time-series and the ONNX model output (in Terminal 1). When the attack is enabled (`SIGUSR1`), the model output should shift compared to the benign phase (`SIGUSR2`), matching the eviction effect reported in the paper.

(E2): *[Spectre detection]* [20 human-minutes + 10 compute-minute] This experiment shows that PDM can detect *Spectre-PHT*, as an example of attacks relying on speculative memory accesses.
Preparation: Build the modified Spectre-PHT proof-of-concept.

```
$ cd source/detection/example/spectre-pht
$ make
```

Similar to the previous experiment, PDM uses fixed-address mode for this experiment. First, run the Spectre PoC to extract the address of secret:

```
$ ./spectre.out
# The virtual address of 'secret' is:
0x555555556008
# Length of the string in secret: 40
characters
```

Next, verify that the following macros are enabled in `source/detection/PDM-detection.c` with printed address of the secret, and recompile PDM.

```
#define USE_FIXED_START_ADDR
#define START_ADDR 0x555555556008
#define SIZE 40
```

Execution: Run the PoC:

```
$ taskset -c 0 ./spectre.out
```

In a second terminal, inject PDM to the PoC process:

```
$ cd source/detection/injector
$ sudo ./cmd/injector -p $(pidof
spectre.out) PDM-detection.so
```

And toggle the attack on/off:

```
# Enable attack
$ kill -SIGUSR1 $(pidof spectre.out)
# Disable attack
$ kill -SIGUSR2 $(pidof spectre.out)
```

Results: When the attack is enabled, the PDM probing output should show a noticeable increase in the `L1hit2` ratio, and the printed ONNX inference score should shift consistently relative to the benign phase.

(E3): *[In-memory Encryption]* [30 human-minutes + 10 compute-minute]
This experiment demonstrates PDM's in-memory encryption mitigation. After guarding secret pages, accesses trigger a signal handler that performs on-the-fly inline binary rewriting so that the application encrypts/decrypts secrets transparently through injected trampolines. The protected application preserves its functionality correctness and terminates normally across different crypto primitives.
Preparation: The mitigation signal handler and example crypto primitives are already compiled as mentioned in Section A.3.1:
Execution: Run each crypto primitive binary in two modes with the handler injected using LD_PRELOAD. For instance:

```
$ cd source/mitigation/example/openssl
$ LD_PRELOAD=../../PDM-encrypt.so ./ecdh 1
$ LD_PRELOAD=../../PDM-encrypt.so ./ecdh 2
```

Results: Each mode prints the stored cryptographic key in the memory and the execution time. Mode 1 runs normally using the actual key stored in memory, without protection. Mode 2 should print that the key is being guarded, show its encrypted content in memory, and still complete successfully.

(E4): *[Datasets and Models]* [30 human-minutes + 10 compute-minute] This experiment shows PDM's robustness by running the provided Jupyter notebooks on the different datasets (testbed, AWS Fargate, and noise). The notebooks load the corresponding trained model (`.pth`/`.pkl`), run inference on the selected dataset (`.csv`), and report metrics (e.g., TPR/FPR).
Preparation: Required dependencies for this experiment are mentioned in Section A.3.1. Run Jupyter Notebook:

```
$ cd datasets
$ jupyter notebook
```

Execution: Open any of the following notebooks and run the cells. Each notebook contains a variable `dataset_path` that selects which `.csv` file to evaluate.

- `datasets/testbed/Detection Engine (In-`

ference Only).ipynb (local testbed datasets, including FLUSH+RELOAD, FLUSH+FLUSH, PRIME+PROBE and Spectre variants and co-resident natural noises from SPEC CPU 2017 benchmark).

- datasets/noise/Detection Engine 8 (Inference Only).ipynb and datasets/noise/Detection Engine 9 (Inference Only).ipynb (memory stressor noise).
- datasets/fargate/Detection Engine (Inference Only).ipynb (real-cloud AWS Fargate datasets, including FLUSH+RELOAD, FLUSH+FLUSH and Spectre variants).

Results: For each run, the notebook prints detection metrics (TPR/FPR) and saves an output file named inference_output_with_metrics.csv in the same directory showing the model's output for each datapoint.

A.5 Notes on Reusability

PDM's detection and mitigation components are both intentionally modular. They are both compiled as user-space shared libraries that can be loaded into the application via LD_PRELOAD or ptrace runtime injection. To apply PDM's detection to a new application, evaluators can (i) identify the secret addresses from the running application and update detection's probing range, (ii) run the included cache threshold calibration tool on the target machine and update the probing thresholds accordingly, and (iii) compile the probing/detection shared library and load it into the target process using the provided injector. The mitigation handler can be similarly injected to a target application. The provided examples demonstrate the expected integration pattern and can be used as templates for additional crypto libraries or primitives.

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/usenixsec2026/.