USENIX Security '26 Artifact Appendix: microSCALE: Static Analysis for Microarchitectural Side-channel Leakage Evaluation

Akshay Kumar E*
IIT Madras, India

Pranav Krishna N*
IIT Madras, India

Annapurna Valiveti
IIT Madras, India

Pallavi Borkar
IIT Madras, India

Aditi Roy
IIT Madras, India

Chester Rebeiro
IIT Madras, India

A Artifact Appendix

A.1 Abstract

The artifact for microSCALE is a static analysis framework for detecting power side-channel leakages in masked cryptographic implementations running on a target processor. The framework statically analyzes the processor's RTL design to detect and localize transitional leakages at both the hardware and software levels. The artifact is packaged as a Docker image that encapsulates all dependencies, and includes the microSCALE source code along with pre-built opcode-specific subgraphs for the RISC-V Rocket Core.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical risks to evaluators. microSCALE is a purely defensive static analysis tool. No system security mechanisms are disabled, no destructive operations are performed, no sensitive data is used, and no network access is required during execution. All analysis runs entirely within the provided Docker container.

A.2.2 How to access

The artifact is publicly available at:

https://zenodo.org/records/20409592

The repository contains the microSCALE source code, pre-built processor graphs for the Rocket Core, cipher assembly programs used in the evaluation, and a Dockerfile for reproducible execution.

A.2.3 Hardware dependencies

None. microSCALE requires only a standard x86-64 machine capable of running Docker.

A.2.4 Software dependencies

All software dependencies are encapsulated in the provided Docker image, ensuring a consistent and portable execution environment. The only requirement on the host machine is **Docker** (any recent version supporting `docker build` and `docker run`). Internally, the container runs Python 3.10. The following Python scripts implement the framework, and the Dockerfile provides easy, single-command access to the artifact.

`graph_maker.py` Constructs the Microarchitectural Dependency Graph (MDG) from the processor RTL.

`lookup_table_maker.py` Builds opcode-specific subgraphs from the MDG (pre-computed for RocketChip and included in the image).

`main.py` Main microSCALE pipeline: performs operand propagation and produces per-instruction data propagation maps.

`leakage_checker1.py` First-order leakage detection (default).

`leakage_checker2.py` Second-order leakage detection.

These scripts are not intended to be invoked directly. **All interaction with microSCALE should go through Docker** as described in Section A.3.

A.2.5 Benchmarks

The artifact includes the following assembly programs corresponding to the benchmarks in Table 4 of the paper: `gift_1round`, `giftsbox`, `AES_sbox`, `PRESENT_sbox`, `ASCON_sbox`, `isw_xor`, `dom_and`, `full_refresh`, and `locality_refresh`. tHE programs are included in the Docker image under `/app/programs/`.

A.3 Set-up

A.3.1 Installation

Download and extract the artifact from Zenodo, enter the directory, and build the Docker image:

```
docker build -t microscale .
```

This installs all dependencies inside the container. The RocketChip MDG and opcode-specific subgraphs are precomputed and bundled; no additional setup is required.

A.3.2 Basic Test

Run a small benchmark to verify the installation:

```
docker run microscale isw_xor
```

Expected output: a leakage report listing the leaking instruction pairs and responsible microarchitectural registers, consistent with Table 4 of the paper. The following predefined programs are also available:

```
docker run microscale dom_and
docker run microscale full_refresh
docker run microscale locality_refresh
docker run microscale giftsbox
docker run microscale AES_sbox
docker run microscale PRESENT_sbox
docker run microscale ASCON_sbox
docker run microscale gift_1round
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): microSCALE detects transitional leakage in masked cryptographic implementations running on the Rocket Core, producing a leakage report that identifies leaking instruction pairs and the responsible microarchitectural registers. This is demonstrated by experiment (E1) described in Section 6 of the paper, whose results are reported in Table 4.

A.4.2 Experiments

(E1): *Leakage detection on included benchmarks*: Run microSCALE on the included benchmark programs and verify that a leakage report is produced for each, consistent with Table 4 of the paper.
How to: Run each predefined benchmark using Docker and inspect the leakage report printed to stdout.
Preparation: Complete the Docker build described in Section A.3. No additional configuration is needed.
Execution: Run, for example, the following commands:
`docker run microscale isw_xor`
`docker run microscale AES_sbox`
For testing with a custom RISC-V assembly file, see the example in Section A.5.
Results: Each run produces a transitional leakage report.

A.5 Notes on Reusability

microSCALE accepts RISC-V assembly files. Custom masked implementations, written in the same RISC-V assembly style as the programs included under `/app/programs/`, can be analyzed by following the same workflow. To analyze a custom masked implementation, mount the file into the container and use the `custom` mode:

```
docker run \
  -v /path/to/your_file.s:/app/programs/your_file.s \
  microscale custom \
  --shares 2 --variables 2 --randoms 1 \
  --program programs/your_file.s \
  --graph Rocket --order 1
```

The `-shares`, `-variables`, and `-randoms` arguments specify the masking parameters of the input program.

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/.