

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

VeCT: Secure and Efficient Constant-Time Code Rewriting with Vector Extensions

Qisheng Jiang and Danfeng Zhang, *Duke University*

https://www.usenix.org/conference/usenixsecurity26/presentation/jiang-qisheng

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: VeCT: Secure and Efficient Constant-Time Code Rewriting with Vector Extensions

Qisheng Jiang
Duke University
qisheng.jiang@duke.edu

Danfeng Zhang
Duke University
danfeng.zhang@duke.edu

A Artifact Appendix

A.1 Abstract

The artifact provides the source code and experimental framework for VeCT, a compiler-based code rewriter that leverages vector extensions to retain constant-time guarantees while improving performance. It includes the VeCT core tool, a t-test framework for side-channel leakage detection, microbenchmarks and real-world applications for performance evaluation and security validation, and all scripts and data required to replicate the t-tests and performance measurements.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All experimental evaluations should be conducted within a controlled and isolated environment, ensuring no risk to external systems or user data. VeCT is designed as a defensive tool to assist researchers and developers in identifying and mitigating timing side-channel vulnerabilities. Additionally, we provide a Dockerfile to facilitate a containerized environment, which ensures isolation during the artifact evaluation process.

A.2.2 How to access

Our tool and all benchmarks used in our experimental evaluation are publicly available at `https://doi.org/10.5281/zenodo.17822446` and `https://github.com/qisheng-jiang/VeCT`. You may refer to `Evaluation.md` for the artifact evaluation. Please refer to the GitHub repository for the latest update.

A.2.3 Hardware dependencies

The artifact requires an Intel CPU with support for the **AVX-512** instruction set. Users can verify hardware compatibility by checking the CPU flags: `lscpu | grep avx512`. We performed experiments on a machine with an Intel Core i9-11900 and 32 GB DRAM running Ubuntu 24.04.

A.2.4 Software dependencies

We recommend using Docker to ensure a consistent environment. The provided environment is based on the `gcc:7.5.0`. Key software requirements include:

- LLVM and Clang.
- Build tools: `cmake`, `make`, and `ninja-build`.
- Python 3 with `pandas`, `matplotlib`, `numpy`, `click`, and `seaborn` libraries.

Please refer to the `Dockerfile` for more details.

A.2.5 Benchmarks

The evaluation utilizes several benchmarks:

- **T-test Analysis:** Statistical analysis for AVX-512 memory access instructions.
- **Microbenchmarks:** Targeted tests for performance and security validation.
- **Real-World Applications:** Benchmarks from open-source real-world applications.

A.3 Set-up

A.3.1 Installation

(I1): Build the Docker image:

```
docker build -t image-vect .
docker run --rm -it -v "$(pwd)":/app image-vect /bin/bash
```

(I2): Inside the container, compile and install VeCT:

```
cd /app/src
./install.sh
. ./setup.sh
./llvm_compile_dfsan_cpp.sh
apt update && apt upgrade -y
apt install -y llvm-13 clang-13
cd /app/src/passes && make install -j10
cd /app/src/lib && make install -j10
```

(I3): Compile and install baselines for comparison:

```
cd /app/src_constantine+
. ./setup.sh
cd /app/src_constantine+/passes && make install -j10
cd /app/src_constantine+/lib && make install -j10
```

A.3.2 Basic Test

To verify the installation, run the following script:

```
cd /app/src
bash run.sh microbenchmark/vector-perf.c
./microbenchmark/vector-perf.out < ./real-world-apps/binsec/random_input.txt
```

Successful execution should produce the similar output:

```
Success: microbenchmark/vector-perf.out
Elapsed time in microbenchmark/vector-perf.c: [2026]
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Constant-time guarantees for AVX-512 memory access instructions (Section 4).
(C2): VeCT maintains high performance across microbenchmarks and constant-time guarantees for the generated code (Section 6.1).
(C3): VeCT effectively repairs real-world applications without significant overhead (Section 6.2).

A.4.2 Experiments

(E1): *T-test Analysis [40 compute-minutes]:*
Execution:

```
cd /app/t-test
bash t-test.sh
```

Results: `t-test/results.md` shows the reproduced results for Figures 2-5 in Section 4.

(E2): *Microbenchmarks [4.6 compute-hours]:*

(E2.1): *Overview and Impact of Access Count [40 compute-minutes]:*
Execution:

```
# run VeCT (Vector and Single)
cd /app/src/microbenchmark
bash run_microbenchmarks.sh
# run baselines (Constantine+ and Original insecure code)
cd /app/src_constantine+/microbenchmark
bash run_microbenchmarks.sh
# generate results
cd /app/src/microbenchmark
python3 stats_pic.py
```

Results: `src/microbenchmark/results.md` shows the reproduced results for Figures 15 and 16 in Section 6.1.

(E2.2): *Security Validation [4 compute-hours]:*
Execution:

```
cd /app/src/microbenchmark
bash run_validation.sh
```

Results: `src/microbenchmark/validation.md` shows the reproduced results for security validation in Section 6.1. The output should look like the following for each setup:

```
Processing groups: [xxx] with masks: index 0, index 1
meas: [0.00] M, max t:   [+0.30], max tau: [6.71e-02], (5/tau)^2: [5.56e+03]. For the moment, maybe constant time.
```

(E3): *Real-World Applications [35.4 compute-hours]:*
Execution:

```
# run VeCT (Vector and Single)
cd /app/src/real-world-apps
bash run_app_tests.sh
# run baselines (Constantine+ and Original insecure code)
cd /app/src_constantine+/real-world-apps
bash run_app_tests.sh
# generate results
cd /app/src/real-world-apps
python3 stats_table.py
```

Results: `src/real-world-apps/results.md` shows the reproduced results for Table 1 in Section 6.2.

A.5 Notes on Reusability

VeCT is designed with modularity to facilitate adoption across different hardware architectures and software environments:

- **Hardware Portability:** While evaluated on Intel AVX-512, the t-test framework can be ported to other architectures with low effort by replacing architecture-dependent instructions for vector operations, cache flushes, and time measurements.
- **Toolchain Integration:** Our proof-of-concept is decoupled into four components. While our prototype integrates with Constantine for violation detection and control-flow linearization, these dependencies are not strictly required. Researchers can replace these modules with other tools with moderate engineering effort.

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/.