USENIX Security ’25 Artifact Appendix: TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms

Daan Vanoverloop[1], Andrés Sánchez[*2,4], Flavio Toffalini[2,3], Frank Piessens[1], Mathias Payer[2], and Jo Van Bulck[1]

[1]*DistriNet, KU Leuven*, [2]*EPFL*, [3]*RUB*, [4]*Amazon*

A Artifact Appendix

A.1 Abstract

We provide the automated mitigation pipeline, including LLVM and BOLT instrumentation passes, a modified Intel SGX SDK that includes our custom prefetcher, as well as the benchmark applications to evaluate performance overhead. To assess leakage in practice, we furthermore provide an independent reproduction of the controlled-channel attack on libjpeg and a profiler tool to seamlessly extract page-access traces from victim enclaves.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact includes attacks on real-world Intel SGX processors, which require the installation of the privileged SGX-Step kernel module. We recommend that evaluators run all code on an isolated test machine.

The attack and mitigation code in this artifact is intended solely for reproducing our results. Any use of these results should be conducted responsibly. The mitigation is a research prototype, and we do not recommend its deployment in production environments as-is.

A.2.2 How to access

The artifact files are available on Zenodo (`https://doi.org/10.5281/zenodo.15194120`) and GitHub (`https://github.com/TLBlur-SGX/tlblur/tree/usenix-artifact`).

A.2.3 Hardware dependencies

Building and running TLBlur requires hardware with support for Intel SGX2 and AEX-Notify.

*All the contributions of this author were made prior to joining Amazon.

A.2.4 Software dependencies

- A Linux distribution supported by the Intel SGX SDK. We recommend using Ubuntu 22.04.
- C/C++ compiler (`gcc` or `clang`) and linker (`lld` strongly recommended)
- Build tools: `make`, `cmake`, `meson` and `ninja`.
- Stable Rust toolchain (1.76 or later)

A.2.5 Benchmarks

No external benchmarks were used for our evaluation.

A.3 Set-up

A.3.1 Installation

Download artifact. Clone the repository with all submodules.

```
$ git clone --recurse-submodules \
    --branch usenix-artifact \
    https://github.com/TLBlur-SGX/tlblur.git
```

Build. Run `build.sh` to build and install LLVM with TLBlur instrumentation passes, the Intel SGX SDK with TLBlur prefetcher, SGX-Step, benchmark enclaves and tools.

Load SGX-Step. Load the SGX-Step kernel module by running `make clean load` in `sgx-step/kernel`.

A.3.2 Basic Test

1. Run the test program with an enclave binary, e.g. `./install/bin/test-rsa ./install/lib/s-encl-rsa-` `↪ instrumented-relocs-bolt-opt.so`. Output should contain the following, indicating that the attack succeeded:

```
[../enclaves/rsa/main.c] secure enclave
    ↪ encrypted '1234' to '21921'; decrypted
    ↪ '1234'
[../enclaves/rsa/main.c] --> RECONSTRUCTED KEY
    ↪  '20771' (0x5123)
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): A practical profiler (§7.2) to extract enclave page-access traces (Figures 5, 9 and 10).

(C2): A reproduction of the seminal controlled-channel attack by Xu et al. on libjpeg, and demonstrate that it still leaks with AEX-Notify single-stepping defense (Figures 1 and 8).

(C3): An automated compiler pipeline (§6) that reduces information leakage observable to controlled-channel adversaries by "blurring" page-access traces (§9.1).

(C4): An average slowdown of $\times 2.24$ from instrumentation across a variety of benchmark programs (§9.2.1, §9.2.3, Figure 6 and Table 3).

(C5): An interrupt-resumption overhead that scales linearly with the size of the PWS and the size of the enclave (§9.2.2, Figure 7).

A.4.2 Experiments

(E1): *[Profiler] [10 human-minutes + 10 compute-minutes]: extract a page-access trace from libjpeg.*
Preparation: Run `cd sgx-step/app/libjpeg`.
Execution: Run the profiler as follows:

```
$ sudo ../profiler/target/release/sgx_tracer \
    --so ./profiler-libjpeg.so \
    -e ./Enclave/encl.so \
    --output trace.vcd \
    --args img/birds.jpg 10000000 10000000
```

Collecting the full trace can take up to several hours. Interrupt the program after a few minutes.
Results: The partial trace in the `trace.vcd` file can be opened in a VCD viewer like GTKWave. This demonstrates the practicality of using the profiler to extract page-access traces and, thus, validates C1.

(E2): *[Attack on libjpeg] [10 human-minutes + 3 compute-hours]: reconstruct libjpeg image with page-fault attack.*
Preparation: Run `cd sgx-step/app/libjpeg/attack`.
Execution: Run the attack as follows:

```
$ cargo run --release -- -o reconstruct.bmp \
    -i ../img/Wapiti_from_Wagon_Trails.jpg \
    --color enclave \
    -e ../Enclave/encl.so
```

The attack can take several hours to complete, but progress is indicated by the progress bar.
Results: The resulting `reconstruct.bmp` file should contain a reconstruction of the original image, as in Figure 1, validating C2.

(E3): *[Prefetching simulation] [10 human-minutes + 30 compute-minutes]: use the instrumented binary to evaluate the effectiveness of TLBlur.*
Preparation:

```
$ export TLBLUR_LIB="$PWD/install/lib"
$ cd sgx-step/app/libjpeg
```

Execution: Run the profiler as follows:

```
$ sudo \
  ../profiler/target/release/sgx_tlblur_sim \
  --so $TLBLUR_LIB/libprof-libjpeg.so \
  -e $TLBLUR_LIB/s-encl-libjpeg-instrumented-
      ↪ relocs-bolt.so \
  --output trace_30.vcd \
  --args img/birds.jpg 10000000 10000000 \
  --pws-size 30 --irq-pat page-fault \
  --hw-tlb set-associative \
  --ways 8 --sets 1024
```

This will run the profiler with the instrumented libjpeg binary and a PWS size of 30, simulating the effect of TLBlur with a set-associative hardware TLB with 8 ways and 1024 sets. Collecting the full trace can take up to several hours. Interrupt the program after around 30 minutes have passed to collect sufficiently long traces.
Results: The partial trace in the `trace_30.vcd` file can be opened in a VCD viewer like GTKWave. Compared to the trace from E1, page-access patterns are "blurred", as in Figure 10, validating C3. Note that the binary instrumentation pass moves code to higher addresses, hence the code pages start at page 5120 instead.

(E4): *[Instrumentation benchmarks] [10 human-minutes + 2 compute-hours]: run performance benchmarks to measure instrumentation overhead.*
Preparation: Ensure that all benchmark enclave binaries are installed in `install/lib`. Each benchmark program and combination of optimizations shown in Table 3 has a corresponding `.so` file in this directory. Refer to the table in the README for the naming scheme of enclave files. Change to the `evaluation` directory.
Execution: To run the benchmarks, execute the `run_benchmarks.sh` script. This may take several hours to complete.
Results: The results can be found in the `out` directory. Run `./plot.py` to reproduce Figure 6. Set `FULL_PLOT =` `↪ True` in `plot.py` to reproduce Table 3, validating C4.

(E5): *[Interrupt-resumption benchmark] [10 human-minutes + 5 compute-minutes]: benchmark the interrupt-resumption overhead.*
Preparation: Run the following to prepare the microbenchmark:

```
$ cd prefetch-benchmark
$ source /opt/intel/sgxsdk/environment
$ make clean all
```

Configure the enclave size in `Enclave/encl.config.xml`.
Execution: The `run.sh` script executes the microbenchmark for varying PWS sizes, and a fixed PAM size based

on the configured enclave size.
Results: The results can be found in the `out` directory. Repeat the experiment with different enclave sizes to obtain results for varying PAM sizes. Run `./plot.py` to reproduce Figure 7, showing linearly scaling overhead in both PWS and PAM size and, thus, validating C5.

A.5 Notes on Reusability

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/.