
# USENIX'23 Artifact Appendix: Controlled Data Races in Enclaves: Attacks and Detection

Sanchuan Chen
Fordham University
schen409@fordham.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Yinqian Zhang
Southern University of Science and Technology
yinqianz@acm.org

## A Artifact Appendix

### A.1 Abstract

The artifact SGXRACER is a controlled data race detection tool analyzing Intel SGX enclave binary code. It is implemented atop `angr` binary code analysis tool. SGXRACER performs static analysis, particularly data flow analysis, to detect shared variables and lock variables in binary code, and then use a lockset based algorithm to detect data races. To evaluate SGXRACER, we have tested four well-known SGX SDKs and eight widely-used SGX applications. We have open-sourced SGXRACER on GitHub.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Any discovered 0-day vulnerability should be reported to its software vendor and vulnerability databases such as NVD.

#### A.2.2 How to access

SGXRACER can be accessed via the following GitHub repository: https://github.com/OSUSecLab/SGXRacer.

#### A.2.3 Hardware dependencies

The suggested hardware configuration is an x86-64 PC with eight Intel Core i7-7700 processors and 32GB memory or better.

#### A.2.4 Software dependencies

SGXRACER was originally developed and tested on Ubuntu 20.04. SGXRACER requires Python 3 environment, including command line tool `python3` and `pip3`. SGXRACER also requires Python 3 package `angr`.

#### A.2.5 Benchmarks

**(B1:)** SDK binaries: In our repository, we have provided our pre-built binary code for four well-known SGX SDKs: Intel SGX SDK, Microsoft Open Enclave SDK, Apache Teaclave Rust-SGX SDK, and Fortanix Rust EDP SDK.

**(B2:)** Application binaries: In our repository, we have also provided our pre-built binary code for eight widely used SGX Applications: mbedtls-SGX, intel-sgx-ssl, TaLoS, LibSEAL, SGX_SQLite, stealthdb, SGXDeep, and hotcalls.

### A.3 Set-up

#### A.3.1 Installation

(1) Set up an OS environment: Ubuntu 20.04. (2) Install pip3 for Python 3:

```
sudo apt install python3-pip
```

(3) Install binary code analysis framework angr:

```
sudo pip3 install angr
```

(4) Clone `SGXRacer` GitHub repository:

```
git clone https://github.com/OSUSecLab/SGXRacer.git
```

(5) Read the README.md file for `SGXRacer` tool description and usage details.

#### A.3.2 Basic Test

Run the following command detects the controlled data races in Intel SGX SDK and will test the basic functionality of all software components:

```
python3 sgxrace.py -input \
./enclave_binaries/intel_sgx_sdk/enclave.signed.so  \
-output intel_sgx_sdk_results.txt \
-output1 intel_sgx_sdk_results1.txt \
> intel_sgx_sdk_stdout
```

This command may take minutes to execute. Please ignore warnings. The command should execute successfully without any exception. It will output three files:

```
intel_sgx_sdk_results.txt
intel_sgx_sdk_results1.txt
intel_sgx_sdk_stdout
```

The content of these files should match our corresponding same name result files in the folder `results/sdk_results`. `intel_sgx_sdk_results.txt` is detailed detection results. `intel_sgx_sdk_results1.txt` is concise version detection results. `intel_sgx_sdk_stdout` is detection statistics.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1:)** SGXRACER has been used in detecting controlled data race vulnerabilities in four well-known SGX SDKs: Intel SGX SDK, Microsoft Open Enclave SDK, Apache Teaclave Rust-SGX SDK, and Fortanix Rust EDP SDK. This is proven by the experiment (E1) described in Section 6 whose results are reported in Table 2 in our paper.

**(C2:)** SGXRACER has been used in detecting controlled data race vulnerabilities in eight widely used SGX Applications: mbedtls-SGX, intel-sgx-ssl, TaLoS, LibSEAL, SGX_SQLite, stealthdb, SGXDeep, and hot-calls. This is proven by the experiment (E2) described in Section 6 whose results are reported in Table 3 in our paper.

### A.4.2 Experiments

**(E1):** SGX SDK Data Race Detection [60 human-minutes + 20 compute-hour + 5GB disk]:
**How to:** Run the commands in README.md file evaluation part 1: *To detect data races in SGX SDKs*.
**Results:** Each command will generate three files similar to the ones in basic test:

```
xxx_sdk_results.txt
xxx_sdk_results1.txt
xxx_sdk_stdout
```

The content of these files should match our corresponding same name result files in the folder `results/sdk_results`. `xxx_sdk_results.txt` is detailed detection results. `xxx_sdk_results1.txt` is concise version detection results. `xxx_sdk_stdout` is detection statistics.

**Table 2 Variables Part:** Please refer to the detection statistics file `xxx_sdk_stdout`.
Example: In file intel_sgx_sdk_stdout:

```
sv_r_count: 317
sv_w_count: 119
sv_rw_count: 6
len(info.gv_reverse_map): 143
```

`sv_r_count` is *# Shared Var. Access (R)*. `sv_w_count` is *# Shared Var. Access (W)*. `sv_rw_count` is *# Shared Var. Access (R&W)*. `len(info.gv_reverse_map)` is *# Uniq. Shared Var.*.
Example: In file intel_sgx_sdk_stdout:

```
mutex_count: 7
spin_count: 53
once_count: 0
unique_locks: 9
```

`mutex_count` is *# Lock Var. Access (Mutex)*.
`spin_count` is *# Lock Var. Access (Spinlock)*.
`once_count` is *# Lock Var. Access (Others)*.
`unique_locks` is *# Uniq. Lock Var.*.

**Table 2 Lockset and Acquisition History Part:**
Please refer to the data race detection statistics file `xxx_sdk_stdout`.
Example: In file intel_sgx_sdk_stdout:

```
max_lockset_size: 2
min_lockset_size: 0
average_lockset_size: 0.46113479324725687
max_history_size: 8
min_history_size: 0
average_history_size: 3.3398070205136885
```

`max_lockset_size` is *Ins. Lockset Size (Max.)*.
`min_lockset_size` is *Ins. Lockset Size (Min.)*.
`average_lockset_size` is *Ins. Lockset Size (Ave.)*.
`max_history_size` is *Acquisition History Size (Max.)*.
`min_history_size` is *Acquisition History Size (Min.)*.
`average_history_size` is *Acquisition History Size (Ave.)*.

**Table 2 Var. and Func. Distribution (On table right):** The variable and function distribution are identified by manually inspecting SDK source code, which are in folder `enclave_source`. The result of variable and function distribution can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_sdk_lib_distribution`.

**Table 2 Performance Part:** Please refer to the concise detection results file `xxx_sdk_results1.txt` and statistics file `xxx_sdk_stdout`.
Example: In file intel_sgx_sdk_results1.txt:

```
ULx86_64_init_done*_ULx86_64_init*unw_init_local_common
```

Each line is a detected data race, which is separated by * into three parts: The first part is the shared variable name in the race, the second part is the function name in the first thread, and the third part is the function name in the second thread. By counting the number of lines in this file and checking the unique shared varaible names, we can get `# Shared Variables` and `# Data Races`. The false positives are identified by manually inspecting SDK source code, which is in folder `enclave_source`. The result of our false positive analysis can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_sdk_fp` (false positives highlighted).
Example: In file intel_sgx_sdk_stdout:

```
potential racing pairs: 1567
...
phase 1 time:
0.16607975959777832
phase 2 time:
19.74062967300415
```

`potential racing pairs` is *Shared Variable Access Pairs*. `phase 1 time` is *Variable Analysis Time (m)*. `phase 2 time` is *Data Race Detection Time (m)*. The

sum of `phase 1 time` and `phase 2 time` is *Total Time (m)*. Note the phase 1 and phase 2 are using cached file in our repository thus may not reflect the real analysis time. Also note that time may vary due to different software and hardware configuration and work load on the machine.

**Heap variables (reported in Section 6.1.2):** The heap variable allocations are identified by manually inspecting SDK source code and find out unique heap allocation sites, which are in folder `enclave_source`. The result of heap variable allocations can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_sdk_heap`.

**(E2):** SGX Application Data Race Detection [60 human-minutes + 10 compute-hour + 5GB disk]:

**How to:** Run the commands in README.md file evaluation part 2: *To detect data races in SGX applications*.

**Results:** Each command will generate three files similar to the ones in basic test:

```
xxx_results.txt
xxx_results1.txt
xxx_stdout
```

The content of these files should match our corresponding same name result files in the folder `results/app_results`. `xxx_results.txt` is detailed detection results. `xxx_results1.txt` is concise version detection results. `xxx_stdout` is detection statistics.

**Table 3 Detected Data Races Part:** Please refer to the concise detection results file `xxx_results1.txt` and statistics file `xxx_stdout`.

Example:

In file 001_mbedtls-SGX_results1.txt:

```
add_count*ecp_add_mixed*ecp_add_mixed
```

Each line is similarly divided by * as in SDK cases. By counting the number of lines in this file and checking the unique shared varaible names, we can get the number of shared variables in the detected data races `Var.` and the number of detected data races `Races`. The false positives are identified by manually inspecting application source code, which are in folder `enclave_source`. The result of our false positive analysis can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_fp` (false positives highlighted).

Example: In file 001_mbedtls-SGX_stdout:

```
potential racing pairs: 7817
potential racing interleavings: 15317
```

`potential racing pairs` is *Acc. Pairs*.
`potential racing interleavings` is *# Total Inter.*.

**Table 3 Variables Part:** Please refer to the detection statistics file `xxx_stdout`.

Example: In file 001_mbedtls-SGX_stdout:

```
sv_r_count: 234
sv_w_count: 138
sv_rw_count: 0
len(info.gv_reverse_map): 84
```

`sv_r_count` is count of shared variable accesses (R).
`sv_w_count` is count of shared variable accesses (W).
`sv_rw_count` is count of accesses (R&W).
The sum of these three numbers is *# Shared Var. Access*.
`len(info.gv_reverse_map)` is the number of unique shared variables, *i.e.*, *# Var.*.

Example: In file intel_sgx_sdk_stdout:

```
mutex_count: 2
spin_count: 66
once_count: 0
unique_locks: 3
```

`mutex_count` is count of lock variable accesses (mutex).
`spin_count` is count of accesses (spinlock).
`once_count` is count of lock variable accesses (others).
The sum of these three numbers is *# Lock Var. Access*.
`unique_locks` is the number of unique lock variables, *i.e.*, *# Lock Var.*.

Example: In file intel_sgx_sdk_stdout:

```
average_lockset_size: 0.1978053439017108
...
average_history_size: 9.270506565018288e-05
```

`average_lockset_size` is *Ave. Lockset*.
`average_history_size` is *Ave. Acq. History*.

**Table 3 Performance Part:** Please refer to statistics file `xxx_stdout`.

Example: In file 001_mbedtls-SGX_stdout:

```
phase 1 time:
0.27547693252563477
phase 2 time:
307.8826234340668
```

`phase 1 time` is *Variable Ana. (m)*. `phase 2 time` is *Race Det. (m)*. The sum of `phase 1 time` and `phase 2 time` is *Total Time (m)*. Note the phase 1 and phase 2 is using cached file in our repository thus may not reflect the real analysis time. Also note that time may vary due to different software and hardware configuration and work load on the machine.

**Heap variables (reported in Section 6.1.2):** The heap variable allocations are identified by manually inspecting application source code and find out unique heap allocation sites, which are in folder `enclave_source`. The result of heap variable allocations can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_heap`.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.