SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing

Tobias Cloosters, *University of Duisburg-Essen;* Johannes Willbold, *Ruhr-Universität Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Lucas Davi, *University of Duisburg-Essen*

https://www.usenix.org/conference/usenixsecurity22/presentation/cloosters

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

A Artifact Appendix

A.1 Abstract

SGXFuzz presents a novel approach to fuzz SGX enclaves in a user-space environment including the synthesis of ECall structures that automatically synthesizes a nested input structure as expected by the enclaves using a binary-only approach. The prototype consists of an enclave dumper that extracts enclaves memory from distribution formats, a fuzzing setup to fuzz extracted enclave, as well as a series of scripts to perform result aggregation. The fuzzing setup is the core of SGXFuzz and is built upon the kAFL fuzzer and the Nyx snapshotting engine. We extend the existing code of kAFL to accommodate our structure synthesis in Python. The Nyx fuzzing engine utilizes the Intel PT CPU extension to get code coverage information but does not contain any changes for SGXFuzz. Finally, we provide several scripts to process the crashes found during the fuzzing campaigns as well as the synthesized structure layouts.

A.2 Artifact check-list (meta-information)

- **Compilation:**
  recent cmake, gcc/g++, c++20, Ubuntu 22 recommended
- **Transformations:**
  custom binary-to-binary included
- **Binary:**
  Ubuntu 5.10.75 kernel
- **Run-time environment:**
  Linux/Ubuntu, custom kernel included, root access, bare metal/no VM
- **Hardware:**
  Intel CPU, Skylake or newer (Intel-PT-capable)
- **Metrics:**
  Structure Layouts, Crashes/Vulnerabilities/Bugs, Coverage
- **Output:**
  Terminal, Files (msgpack/structures, edges, crashing payloads)
- **Experiments:**
  - Extract enclaves
  - Run the Fuzzer (compile runner, start fuzzing)
  - Post-process/aggregate results
- **How much disk space required (approximately)?**
  3.5 GB install size + temporary 10–30 GB
- **How much time is needed to prepare workflow (approximately)?**
  3 h
- **How much time is needed to complete experiments (approximately)?**
  Full experiment:
  24 h per run, 30 main evaluation runs, 80 ablation runs
  = 110 days using a single machine (+ data aggregation)
  Minimal sample evaluation: 1 h
- **Publicly available (explicitly provide evolving version reference)?**
  https://github.com/uni-due-syssec/sgxfuzz/
- **Code licenses (if publicly available):**
  MIT, BSD, GPL, AGPL, Apache (see individual components)
- **Workflow frameworks used?**
  Bash and Python
- **Archived (explicitly provide DOI or stable reference):**
  https://github.com/uni-due-syssec/sgxfuzz/tree/usenix2022

A.3 Description

We will now describe the components of the artifact, how they are related and what each component is used for. The SGXFuzz artifact consists of the enclave dumper, enclave runner, the fuzzing setup, and the enclaves evaluated in the paper. The enclave dumper extracts the enclave memory from enclave distribution formats (cf. Section 5.1). This step has to be done only once per enclave, and we have already performed that step for all enclaves. The enclave runner uses the previously extracted enclave memory to run the enclave as a regular user-space process (cf. Section 5.2). The runner is a C++ program that loads the enclave memory, handles the emulation of the context switch that would usually be performed by the SGX instruction set and performs the structure allocation for each input. Finally, our fuzzing setup consists of a front end that generates fuzzing inputs and performs the structure synthesis, and a back end that executes the target and collects coverage. We use kAFL as a foundation for our fuzzing front end and add new code to the fuzzer to perform the structure synthesis. The back end consists of a patched version of QEMU and KVM to allow the collection of coverage data using the Intel PT CPU extension. We did not perform any modifications on the fuzzing back end.

A.3.1 How to access

All code relevant to the artifact and links to the components required for the fuzzing setup are publicly available on GitHub https://github.com/uni-due-syssec/sgxfuzz/tree/usenix2022.

A.3.2 Hardware dependencies

Our fuzzing back end consisting of a modified QEMU and KVM uses the Intel PT CPU extension to collect coverage data. Thus, an Intel PT-enabled CPU is required to use our fuzzing setup. However, Intel PT does not work in a virtualized environment and as such, cannot run in VM. Notice that the Intel SGX is not required at any point.

A.3.3 Software dependencies

Generally, any Linux distribution should be able to run our artifacts. However, we only tested it on Ubuntu 22.04 and the scripts we provided to set up the fuzzing setup were developed and tested with Ubuntu 22.04 in mind.

A.4 Installation

We include a setup script that should perform the major steps.

First, disable SGX in the BIOS if supported by the CPU.

Clone the repository.

Install required packages:

```
sudo apt install \
  python2 python3 libpixman-1-dev pax-utils bc \
  make cmake gcc g++ pkg-config unzip \
  python3-virtualenv python2-dev python3-dev \
  libglib2.0-dev
```

Then, you can use setup.sh to compile and install the components, or follow the steps manually. That is:

- Initialize the submodules:
  `git submodule update --init --recursive --depth=1`
- QEMU-Nyx:
  https://github.com/nyx-fuzz/QEMU-Nyx#build
- KVM-Nyx:
  https://github.com/nyx-fuzz/KVM-Nyx#setup-kvm-nyx-binaries
- (Virtual) environments for python2 and python3 and install
  - python2: configparser mmh3 lz4 psutil ipdb msgpack inotify
  - python3: six python-dateutil msgpack mmh3 lz4 psutil fastrand inotify pgrep tqdm hexdump
- Install zydis (`cd zydis && cmake . && make install`)

A.5 Experiment workflow

The experiment workflow includes three main parts: Enclave dumping, Fuzzing, Result aggregation.

A.5.1 Enclave Dumping

First, enclave dumping is used to extract the enclave memory. It is based on the Linux SGX SDK. By providing the enclave dumper with `enclave.signed.so`, a memory dump with the name `enclave.signed.so.mem`, a memory layout `enclave.signed.so.layout`, and the address of the enclave's entry point (specifically the offset of the TCS) `enclave.signed.so.tcs.txt`.

Compile is using:

```
make -C ./enclave-dumper/
```

Run it using:

```
./enclave-dumper/extract.sh [enclave.signed.so]
```

A.5.2 Fuzzing

To fuzz the previously extracted enclave, several steps are involved. We bundled all of them together in a script that runs a minimal fuzzing test:

```
./run-example.sh
```

The script runs the following steps automatically. First, the enclave runner is compiled using

```
make-enclave-fuzz-target.sh enclave.signed.so.mem \
  enclave.signed.so.tcs.txt
```

The result of the compilation is a `fuzz-generic` binary, which is the user-space version of the enclave, and a `liblibnyx_dummy.so`, which is required for the fuzzer.

In the next step, the fuzzing target is packed into a VM that is executed using the QEMU-KVM setup. The packer script can be called as follows

```
nyx_packer.py <enclave-runner> <fuzz-folder> m64 \
  --legacy --purge --no_pt_auto_conf_b \
  --fast_reload_mode --delayed_init
```

Finally, the fuzzing can be started using the kAFL fuzzing frontend. The exact command can be found in the `run_example.sh` script.

If desired, manually crafted seeds can be added to the `imports` folder. Each seed is a file consisting of the ECall ID, the serialized structure definition, and the contents of the buffers.

A.5.3 Result Aggregation

Display synthesized structures:

```
display-structs.py <path/to/fuzzing-workdir> \
  <ecall_index>
```

The script displays the evolvement of the synthesized structure in a tree format for each ECall index, with the ECall index being zero-based. The leaves show the final evolvement of the synthesized structures. Each leave shows the synthesized structure in a specific format.

Structures are serialized, e.g., `40 2 C8 4 0 C24 7 0`, and read left to right. This string denotes a structure of **40** Bytes, which has two (**2**) child structures (**C**). The first child is at offset **8** of the parent and is defined the same way: A size of **4** and zero (**0**) children. The second child has a size of **7** and also zero children. Further, the sizes may be annotated with their address (`40:0x7ffff7faafd8`). Additional types include buffers partially (on the edge) of the enclave's memory (P) and SizeOf (S) buffers of which the size is written to a defined offset.

This script shows how to parse and dump these strings:
`kafl/kAFL-Fuzzer/fuzzer/technique/struct_recovery.py`

Display crashes:

```
analyze_crashes.py <eval-dir> \
  -0 --np --no-ptr-0x7ff --no-large-diff
```

`analyze_crashes.py` script iterates through all crashes found by the fuzzer. This includes the crashes due to implementation artifacts mentioned in Section 5.5. The script performs the filtering according to Section 5.5 and will only display valid crashes. However, manual duplication of the crashes is required to find the real number of unique bugs. The flags supplied to the script do the filtering according to the described filtering techniques in the paper. The script displays useful information to understand the crash: the ECall ID, the signal (usually Segmentation Fault), pc (absolute/relative), the disassembled instruction, and addresses used for memory access.

Calculate Coverage:
`calculate-coverage.sh <path/to/fuzzing-workdir>`
Note that we recalculated the numbers of basic blocks using the basic block semantic of Binary Ninja to provide numbers comparable to TeeRex.

A.6 Evaluation and expected results & Experiment customization

The main goal of the fuzzing process is to find crashing inputs, i.e. vulnerabilities, and the synthesized structure layouts that the enclave calls expects for its input. To ensure that the artifact is functional, any enclave from the previously provided links with enclaves can be used for fuzzing, i.e., `synaTEEv2-20211105` which is the *Synaptics Fingerprint Driver Enclave* from the paper. Fuzz the enclave using the steps shown in `run_example.sh` for 960 core-hours. To fuzz another target than the example, change the `ENCLAVE_PATH` variable to the target path and change the enclave name `enclave.signed.so` to the target enclave's name, i.e., `synaTEE.signed.dll`.

After that, it is possible to use the previously described workflow to display synthesized structures using the `display-structs.py` script on the fuzzing workdir. To analyze the crashes, the workflow to display the crashes can be used. However, manual reverse engineering is required to deduplicate bugs.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.