# Information Security Lab
# Module 4: Trusted Execution Environments

Teaching Assistants: Benedict Schlüter and Clément Thorens

v2025.0
November 20, 2025

Responsible: Prof. Dr. Shweta Shinde (SECTRS)

# Contents

# 1 Organisation

Welcome to Module 4 of the Information Security Lab! This module is organized by the Secure & Trustworthy Systems Group.

## 1.1 Exercise Sessions

We use the exercise sessions only to prepare you for your lab assignments. We will not introduce additional tasks or exercises in these sessions. However, the material is complementary to the lecture.

**25./26.11.25:** Introduction to the grading environment and required background knowledge for the Page Fault exercises.

**02./03.12.25:** Exercise is **omitted**.

**09./10.12.25:** Exercise is **omitted**.

## 1.2 Lab Sessions

The lab sessions are intended to discuss any problems that you might have with the assignment. You are also welcome to post your questions and join ongoing discussions on the Moodle forum. Please make sure that you do not share solutions when asking or answering questions. Lab session attendance is **not** mandatory.

# 2 Introduction

The main goal of this lab is to understand Trusted Execution Environments, in particular, how they secure data in use and how one could attack them.

## 2.1 Grading

Note that all our instructions are tailored towards Linux. Therefore:

**Important**

**Please use a Linux system** to run the tools (objdump, SSH, etc.) we are about to introduce. Any distribution should do, and using a VM is perfectly fine. If you decide to develop your exploit on another OS, be aware that it may be troublesome, and that debugging potential issues is your responsibility.

### Grader

The grader allows you to submit your solutions and get immediate feedback. Since the grader has to evaluate the solutions of all students, they are executed in order of submission. Although you may submit as many attempts as you want, we encourage you to use the grader with caution, since delays induced by a high workload affect all students. Develop your exploits on your own machine and submit them to the grader once you are fairly confident that they work. The grader considers your last working submission for your final mark, meaning you will not risk losing your points with additional submissions.

**Important**

Be aware that we will perform plagiarism checks on **all** your submissions. Apart from that, passing an exercise on the grader means that you get the corresponding point for your final grade. We reserve the right to grade your submission after the deadline manually. Thus the result of the grader does not entitle you to the score displayed.

### ISL Tool

The ISL tool is a npm package that you can run on the command line of your host system to perform the following tasks:

1. Authenticate with the ISL lab environment using your ETH credentials.
2. Submit solutions to the grader.
3. Show your current score.

# 3 ISL-Tool Installation and Workflow

First, install Node.js (`https://nodejs.org/en/download/package-manager/`) on your Linux system. Make sure you have at least version 16.15.0 installed.

**Important**

The version provided by packet managers (like apt/snap) is usually too old, so we recommend installing Node.js from an alternative repository as shown here `https://github.com/nodesource/distributions#debian-and-ubuntu-based-distributions`. Using version "`NODE_MAJOR=21`" has proven to work fine.

If the installation results in an error, check this post `https://unix.stackexchange.com/questions/627635/upgrading-nodejs-on-ubuntu-how-to-fix-broken-pipe-error`.

Now you can run the `isl-tool`. By using `@latest` you always get the latest version of the tool:

```
> npx isl-tool@latest help
Usage: isl-tool [options] [command]
Information Security Lab Tool (ETH Zurich)

Options:
  -h, --help                display help for command

Commands:
  login                     authenticate for access to student environment
  submit <exercise> <file>  submit solution for grading
  results                   list the current grading results for your submissions
  submission <exercise>     show the submission file considered for the grading
                            result of an exercise
  help [command]            display help for command
```

**How to use the tool:**

Submit your exploit:

```
npx isl-tool@latest submit ex1a exploit1a.py
```

Show the solution currently considered for your grade:

```
npx isl-tool@latest submission ex1a
```

Show your score:

```
npx isl-tool@latest results
```

# 4 Exercises

This module comprises a total of 7 exercises grouped in two parts.

## 4.1 Part 1 - Page Fault Side-Channel

**Important**

- **DO NOT** recompile the code as this might change the function alignment. You are not required to run any of the provided binaries to solve the exercise successfully

### Exercise 1.1 - Double and Add - 2 points

In the first part of this module, you will steal a Diffie-Hellman key from an SGX enclave. Intel SGX is known for containing many side channels. This task focuses on page faults in particular. SGX's memory is protected against direct access from applications outside the enclave, but the OS is still responsible for paging the enclave's memory. Concretely, this implies that the OS can change certain bits in the page table. The modification can induce page faults when the enclave tries to write/read/execute the page. The page fault is delivered to the OS, which can infer information about the enclave's execution state. Due to the hardware requirements for SGX and the complex setup for emulating enclaves, we provide traces.

*Goal:* Your task is to write a **generic** python script capable of extracting a secret key from **one** execution trace (series of page faults) of the binary. For simplicity, we removed all external library calls from the trace. Thus the trace only contains page faults in the binary's executable `PT_LOAD` segment itself. This also includes the `.plt` section.

- The traces were generated with ASLR **enabled**. This implies that the function addresses change across different executions. However, the **relative** position within the segment remains the same.
- The filename of the trace is equal to the secret key generating it.
- The page size is 4 KiB.
- $k \in [1, |\texttt{G}| - 1]$. Where $k$ is the private key and `|G|` the order of the Elliptic Curve.

```
npx isl-tool@latest submit m4_page_fault_1 <your_solution_filename>.py
```

### Exercise 1.2 - Non-Adjacent Form - 2 points

The implementation in the previous exercise was too slow. Thus we decided to implement a faster algorithm for the group operations on an elliptic curve. The algorithm makes use of the **non-adjacent form**.

*Goal:* Your task is to write a **generic** python script capable of extracting a secret key from **one** execution trace (series of page faults) of the binary. For simplicity, we removed all external library calls from the trace. Thus the trace only contains page faults in the binary's executable `PT_LOAD` segment itself. This also includes the `.plt` section.

- The traces were generated with ASLR **enabled**. This implies that the function addresses change across different executions. However, the **relative** position within the segment remains the same.
- The filename of the trace is equal to the secret key generating it.
- The page size is 4 KiB.
- $k \in [1, |\texttt{G}| - 1]$. Where $k$ is the private key and `|G|` the order of the Elliptic Curve.

```
npx isl-tool@latest submit m4_page_fault_2 <your_solution_filename>.py
```

## Exercise 1.3 - K-Ary Multiplication - 2 points

After implementing key generation on elliptic curves we were asked to implement a fast modular multiplication algorithms that can be used for RSA decryption. However, when executed in an enclave context, the implementation is subject to a page fault side channel.

*Goal:* Your task is to write a **generic** python script capable of extracting a secret key from **one** execution trace (series of page faults) and **one** memory read access trace (series of page faults) of the binary. For simplicity, we removed all external library calls from the trace. Thus the execution trace only contains page faults in the binary's executable `PT_LOAD` segment itself. This also includes the `.plt` section. However, the memory reads, originate also from shared library accesses.

- The traces were generated with ASLR **enabled**. This implies that the `SYMBOL` addresses change across different executions. However, the **relative** position within the segment remains the same.
- The filename of the trace is equal to the secret key generating it.
- The page size is 4 KiB.

```
npx isl-tool@latest submit m4_page_fault_3 <your_solution_filename>.py
```

## 4.2 Part 2 - GPU TEEs

In this part, you will have to leak and modify the computation performed on a mock GPU implementing the basic functionalities of a TEE. The system is composed of a Guest (`guest.py`), a Hypervisor (`hypervisor.py`), a GPU (`gpu.py`), and a DMA controller (`dma_engine.py`).

**Guest-GPU Communication** The Guest communicates with the GPU by sending pre-defined commands. In `donotmodify.py`, you can find two examples of sequences, `sequence_1` and `sequence_2`. These two sequences will be used to grade your attacks. The GPU replies to the Guest by sending an acknowledgment. Each command has an associated acknowledgment; for example, the Guest will expect to receive `CONFIDENTIAL-SET` in response to the `CONFIDENTIAL-ON` command. In `donotmodify.py`, you can find the list of acknowledgments the Guest is expecting from the GPU, `expected_reply_1` and `expected_reply_2`, in response to `sequence_1` and `sequence_2`, respectively. All sequences are Python lists saved as pickle files in the `keys/` directory. All commands and acknowledgments are signed using the Edwards-curve Digital Signature Algorithm (EdDSA).

**DMA Requests** DMA messages are specific commands for the DMA controller. They are not signed and the controller sends an acknowledgment to the Guest in response. When the DMA controller receives a DMA request, it directly reads/writes from/to the GPU memory, represented by the file `dram.pkl`.

**Data Encryption** If the command `CONFIDENTIAL-SET` has been issued by the Guest, the GPU will expect values encrypted with AES-GCM as input to the computation. The inputs are encrypted by the Guest before being sent to the DMA controller and placed in the GPU memory. In turn, the GPU will encrypt the result of the computation before writing it back to the memory, where it can be accessed by the Guest through a DMA request.

**(No) Key Exchange** For simplicity, no key exchange is implemented between the Guest and the GPU. Instead, the AES key and EdDSA key pairs are generated by `helper.py` and stored in a memory region accessible by both the GPU and the Guest.

**Hypervisor** The untrusted Hypervisor relays all messages between all components, Guest, GPU, and DMA controller. For your attacks, you are only allowed to modify the Hypervisor (`hypervisor.py`). You have no limitation in what you can do with the messages, however the Guest or the GPU might detect your attack and terminate the execution (or crash).

Figure 4.1 shows a visual representation of the system.

**Important**

- **Do not implement brute force attacks.** A correct solution should run in, at most, few seconds on a modern machine.
- **Do not attack the PyCryptodome implementation.**
- **Do not read or write the `aes_key.bin`, `*.pkl` or the `*.pem` files to implement your attacks.** You must not read or write files located in the `keys/` directory in your solution, otherwise your solution will fail in the grading environment.
- **Do not install external Python modules** The only external Python module required is PyCryptodome. Do not install any other external module. You can import modules from the Python Standard Library.

### Setup

You have to use Python >3.10 and PyCryptodome. To run the full system, you can either use the script `run.py` we are providing or run all component scripts in the correct order: `dma_engine.py` → `hypervisor.py` → `gpu.py` → `guest.py`.

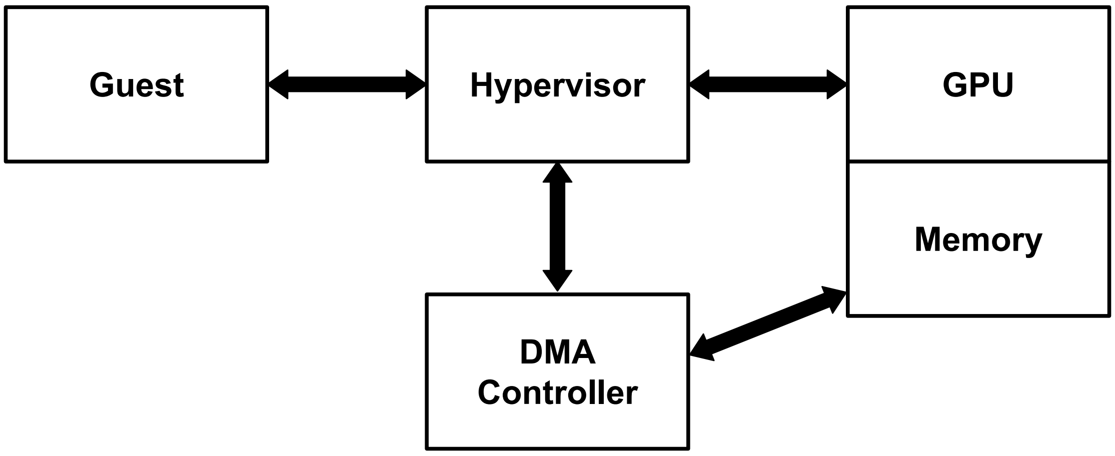

**Figure 4.1:** Mock GPU system

## Exercise 2.1 - Inputs and result leak - 2 points

**For this exercise, use** `sequence_1` **and** `expected_reply_1` **by passing** 1 **as the first parameter to** `guest.py` **or to** `run.py`

The inputs of the first sequence of commands are -14578, -7105, -2, 2 and the result is 21685. These numbers are sent encrypted by the Guest to the GPU (inputs) and by the GPU to the Guest (result). Your goal is to print these numbers from the Hypervisor (`hypervisor.py`). During the evaluation, the commands executed will be the same, but the values will change (same computation different inputs and result).

The grader will expect the solution to be printed in this format:

$$[-14578.0, -7105.0, -2.0, 2.0, 21685.0]$$

where `input1` $= -14578$, `input2` $= -7105$, `input 3` $= -2$, `input 4` $= 2$ and `result` $= 21685$.

```
npx isl-tool@latest submit m4_tee_ex2.1 hypervisor.py
```

## Exercise 2.2 - Modify the computation - 6 points

**For this exercise, use** `sequence_2` **and** `expected_reply_2` **by passing** 2 **as the first parameter to** `guest.py` **or to** `run.py`

This sequence starts with the same computation as in the first exercise, but then another set of computations is run. When running without interference, the results of these two computations are 21685 and 8. Your goal is to modify the computation results. The Guest or the GPU must not detect that an attack has happened. During the evaluation, we will be using the **exact same** sequence of commands as the one you have been given (`sequence_2`), including the input values. You have to implement three attacks:

- **2 points - Attack 1:** Results must be 0 and 15
- **2 points - Attack 2:** Results must be 14576 and 17
- **2 points - Attack 3:** Results must be 21685 and 14588

The Guest prints the computation result and the grader reads it from there. You do not have to do anything for the grader to evaluate your solution. Only submit `hypervisor.py` to the grader.

```
npx isl-tool@latest submit m4_tee_ex2.2.1 hypervisor.py
npx isl-tool@latest submit m4_tee_ex2.2.2 hypervisor.py
npx isl-tool@latest submit m4_tee_ex2.2.3 hypervisor.py
```

### Exercise 2.3 - Fix the vulnerabilities - 1 point

For the last part, you have to fix the vulnerabilities that are allowing previous attacks. For that, you can change `guest.py`, `gpu.py`, and `helper.py`. Do **not** modify `hypervisor.py`, `donotmodify.py` and `dma_engine.py`. To evaluate your changes, we will execute different sequences than the ones you have been working with in exercises 2.1 and 2.2.

To get the points, your solution should print the correct results on the Guest when the Hypervisor is not tampering with the computation and should terminate the computation when the Hypervisor implements the attacks of exercises 2.1 and 2.2. The results must be printed in the format given in the original `guest.py`, for example:

```
Result 1 is 21685.0;Result 2 is 8.0;
```

The grader expects the output of a normal run to contain only the results in this specific format, do not print anything else or change the format while running an unmodified hypervisor.

Submit your modified files to the grader as `.tar` archive.

```
tar -cvf solution.tar guest.py gpu.py helper.py
npx isl-tool@latest submit m4_tee_ex2.3 solution.tar
```

# 5 Appendix

## 5.1 Additional Material

Part of the linked material will be discussed in the exercise session. It covers basics some of you might have already learned during your bachelor's.

**GOT and PLT** https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html
**ELF Internals** https://github.com/PacktPublishing/Learning-Linux-Binary-Analysis
**ELF Introduction** https://lwn.net/Articles/631631/
**Square and Multiply** https://en.wikipedia.org/wiki/Exponentiation_by_squaring
**Non-Adjacent Form** https://en.wikipedia.org/wiki/Non-adjacent_form
**K-ary Multiplication** Encyclopedia of Cryptography andSecurity

## 5.2 Non-x86 / Non-Linux Operating System

For the 1st part, you might need `objdump` to calculate offsets. For non-x86/non-Linux operating systems, you can either install the necessary tools on your own, use the computers in the lab, or rely on https://www.isg.inf.ethz.ch/Main/HelpRemoteAccessSSH.