Dr. Tim Blazytko
M. Sc. Philipp Koppe

Lecture Software Protection
Summer Term 2025
Ruhr-Universität Bochum

# Task Sheet 1

April 14, 2025
Submission due to May 4, 2025, 11:59 PM via Moodle.
For question, feel free to ask via Discord.

## Tasks

Course assessment is based on 4-5 task sheets which will be completed within the semester. The final grade consists of the weighted average grade of all task sheets. We will report every student to the examination office who handed in at least two tasks. In other words, we allow students to drop out after handing in a single task. Task sheets have to be worked on individually; group work is not possible.

Optional tasks are voluntary and independent of the course assessment; in other words, they do not influence the final grade. Instead, they are designed for students who want to dive deeper into individual topics. These submissions will be ranked with subjective score points. The student who has the highest score at the end of the semester wins a Binary Ninja student license.

The first task sheet covers the design and implementation of various obfuscation schemes (control-flow flattening and VMs). The optional task introduces compiler-based obfuscation and allows students to write their own obfuscation passes.

Please submit your solutions as either text files (or markdown), if several files have to be submitted, as zip files. Please do not hand in `.doc` or similar formats. If you have to include images or need some special document layouting, export the document as pdf file. The submitted binaries should be ELF files (Linux) and compiled with GCC for x86-64.

### Task 1: Control-flow Flattening (2 Points)

Hide the control flow in the function `fib` (in `fib.c`) via control-flow flattening. Re-write the function manually and use a state variable to encode the original control flow. Submit your commented source code as well as a compiled binary (for Linux, via GCC) of your protected function.

### Task 2: Virtualization-based Obfuscation (12 Points)

Your goal is to write an virtualization-based obfuscator for the function `fib` in `fib.c` in C. To achieve this, proceed as follows:

(a) Design a custom instruction set architecture which can be used for this task. In a text file, describe the VM architecture (register or stack-based), the individual instructions as well as the instruction encoding. Individual VM instructions shouldn't be too complex; for example, there shouldn't be a single handler which performs Fibonacci calculations. (2 Points)

(b) Manually re-write the function `fib` in your VM assembler and (manually) generate the bytecode in accordance with your VM design. (3 Points)

(c) Write a disassembler for your bytecode in Python or C. Disassemble the bytecode for `fib`. (1 Point)

(d) Implement an interpreter for your custom instruction set architecture which interprets the aforementioned bytecode in C. Comment your code. (3 Points)

(e) Add additional obfuscation (e.g. opaque predicates or MBAs) to at least two of your handlers. The obfuscation should be resilient against aggressive compiler optimizations; compiling with `-O3` should not remove the obfuscation. Comment your design choices and code. (1 Point)

(f) Add some simple form of control-flow sensitive bytecode encoding. In particular, it should not be possible to dump the bytecode from the binary and deploy it to a disassembler for your VM design. Instead, the bytecode should be stored in an encoded form and gradually decoded with several keys during VM execution. Describe how your bytecode blinding works. (2 Points)

Comment your design decisions and code. Submit your notes, VM assembler for `fib`, disassembler, source code of your (hardened) interpreter as well as the compiled binary.

## Task 3: Opaque Predicates (3 Points)

Write a program (in C or Python) which uses an SMT solver (for example Z3) to automatically find arithmetic opaque predicates. An opaque predicate can consist of

- up to three 64-bit variables $x$, $y$ and $z$
- random 64-bit constants
- operators for addition, subtraction, multiplication as well as bitwise and, or and xor
- comparison operators such as equal or unequal

Implement the program in a way such that it outputs 10 **unique** opaque predicates as follows:

```
opaque predicate 0: x ^ x != z ^ z & z
opaque predicate 1: x & y != y & y & x
opaque predicate 2: 7461933721599248351 ^ y | y - y == 9074880713923861056 + y ^ z * z
opaque predicate 3: 6558919088021691307 ^ z + z ^ z == x + z ^ x
opaque predicate 4: 12933530881036816921 | x + x ^ z == 6082372496972163630 & z + y + x
```

You can use `gen_opaque_predicates.py` as starting point. Submit your program as well as the output of an example run (in a text file).

## Optional Task: Compiler-based Obfuscation

In the previous tasks, we mostly implemented obfuscation manually. However, in most scenarios, obfuscation is added automatically, either on the source level, the compiler level or the binary level. While solutions for all levels exist, one of the most dominant forms is compiler-based obfuscation, especially via LLVM passes. In this task, you will dive a bit deeper into this and write your own obfuscation passes.

Your goal is to implement a LLVM compiler pass which automatically inserts opaque predicates. For this, feel free to implement opaque predicates of arbitrary complexity. While local opaque predicates (within the same basic block or function) are easier to implement, global (cross-function) opaque predicates are way harder to detect and analyze.

Alternatively, you can write a compiler pass which automatically hides the control flow via control-flow flattening. If you want to go a step further, play around with different combinations of opaque predicates & control-flow flattening. Keep an eye of the order you apply those transformations—how do they affect the code layout?

Some references to start with:

- `https://github.com/emc2314/YANSOllvm`
- `https://llvm.org/devmtg/2017-10/slides/Guelton-Challenges_when_building_an_LLVM_bitcode_Obfuscator.pdf`
- `https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation`
- `https://polarply.medium.com/build-your-first-llvm-obfuscator-80d16583392b`
- `https://www.praetorian.com/blog/extending-llvm-for-code-obfuscation-part-1/`