Jean-David Mukolonga (s170679)
Christal Mangolopa (s157349)

INFO0940-1 - Operating System

Project 1 : Tracer

March 2021
Academic Year 2020 - 2021

1 Implementation

For this part we start by creating a child via `fork()` and this one is traced by its parent thanks to `PTRACE_TRACEME` request of `ptrace`. The parent starts tracing the child.

In profiler mode The main implementation is in `start_tracer_p(pid_t child, char *programname)`.

The process stops the child process at each next instruction using the `PTRACE_SINGLESTEP` request. For each instruction, we detect if the scope has changed, i.e. if there has been a call or a return. For each function called, a new `fun_tree` structure is created. The latter contains information about the scope such as its number of instruction, its list of calls, its number of recursion and next scope. Step by step, we then grow the list of `fun_tree` accordingly. Finally, we can print it with the correct protocol.

In syscall mode The main implementation is in `start_tracer_s(pid_t child)`.

The process stops the child process at the next entry to or exit from a system call using the `PTRACE_SYSCALL` request. We wait to let the child finish the syscall execution. For new syscall, its name is printed. To find this name, we have put beforehand all the syscalls from the given *syscall.txt* in a dictionary. Then, we inspect the user area of the parent process to retrieve the value of the syscall execution with `PTRACE_PEEKUSER` request.

2 Questions

a. What are the opcodes that you have used to detect the `call` and `ret` instructions?

Here are the opcode used for the `call` instruction: 0xE8. We considered this one because we only take into account relative calls. Detecting a call is then simply a matter of comparing its instruction with this opcode.

For the `ret` instructions, we have first try to use opcode similarly as for the `call`. However, we could not handle relative returns with this method. Finally we decided to implement a stack structure. The latter enabled us to push an expected return instruction after each call. We then detected a relative return by checking an instruction with the expected return instruction at the top of the stack.

b. Why are there many functions called before the "real" program and what is/are their purpose(s)?

We suspected that this is a conscience of libraries. Indeed, the "real" program uses many libraries that provided essential methods. Those libraries are precompiled object files that start upstream the "real" program. The functions are called when we execute the final executable file since the library modules that have been copied into . Their purpose is therefore essential.

c. According to you, what is the reason for statically compiling the "tracee" program?

We executing statically, every library are always loaded. In contrast, dynamic execution depend on the state of the current memory. This implies that some libraries may not be loaded if they are already present in memory. A good reason for this tracing exercise is that we will always get the same output in our tests. Furthermore, static libraries are linked at the last step of compilation and not during run time.

3 Work

We have been working ruffly for 130 hours. Our main difficulties have been the implementation of extra structure not included in C language. Then, we took some time to handle call functions that do not have return instructions and all sort of unexpected behaviours.