L3 internship: Shallow description of a Virtual Machine

Mathis Bouverot-Dupuis

June & July 2021

Contents

1 Introduction

This report accounts for the work done with Alexandre Talon and Guillaume Bonfante, during June and July of 2021 in the LORIA laboratory (Nancy). Over the course of these two months, I studied the problem of recognizing and describing virtual machines (VMs) from their execution traces.

A VM is an abstract version of a computer system : it is a program that reads a list of instructions (often called bytecode) and executes them. It's design is based on that of physical machines. The execution of an instruction consists in :

1. Fetch : read the instruction from memory.
2. Decode : parse the instruction. A typical format divides an instruction into an opcode followed by zero or more arguments (similar to machine code).
3. Dispatch : based on the opcode, decide which code (or function) to run.
4. Execute : carry through the semantics of the instruction. This is where the VM's internal state is updated and input/output operations are performed.

The serial execution of several instructions forms the VM loop.

An important class of virtual machines are programming languages interpreters, such as the Python interpreter (CPython) and the Java Virtual Machine (JVM). During this internship I focussed on different Python VMs.

2 Recognizing the VM execution

We will now describe in more detail our starting point. We give ourselves a VM to analyze (for instance the CPython interpreter). We assume that we do not have access to the source code or to the machine code of the VM executable. What we do allow ourselves to use is the execution trace of single runs of the VM : the ordered list of machine instructions the VM executable uses to run a given program. Note that this does not give us access to the complete machine code of the VM executable, nor to the bytecode instructions of the program the VM runs.

The first step to analyzing the VM is thus to recognize it in the execution trace : can we recognize the fetch-decode-dispatch-execute pattern ?

2.1 Building a CFG

We start by building the control flow graph (CFG) from the VM trace. We group instructions by basic blocks : a sequence of instructions that has only one entry point and one exit point, and build a graph whose vertices are basic blocks and whose edges indicate branches taken during the execution.

More precisely : for an machine instruction I, we call $next(I)$ (resp. $prev(I)$) the set of instructions that immediately follow (resp. precede) I in the execution trace. A basic block is then an ordered list of instructions $I_1, \dots I_n$ such that $\forall k \in [1, n-1], next(I_k) = \{I_{k+1}\}$ and $prev(I_{k+1}) = \{I_k\}$. We of course require basic blocks to be non-empty, and maximal for inclusion (i.e. either $prev(I_1)$ is of size at least 2 or I_1 is the first instruction in the trace, and similarly for I_n). Note that two consecutive instructions in a basic block do not have to be consecutive in memory (example : an unconditional branch).

We then build a graph on the basic blocks. We add an edge from a block $(I_k)_{k\in[1,n]}$ to a block $(J_k)_{k\in[1,m]}$ if and only if $J_1 \in next(I_n)$, which is equivalent to $I_n \in prev(J_1)$.

As the size and complexity of CFGs built this way grows very fast, we apply some additional rules to simplify the graph. Each time we see a *call* instruction,

[->, scale=0.9, transform shape, thick,bb/.style=draw,box,thick, inner sep=0pt,minimum size=5mm]
(call) at (0,0) [bb] ...
call; (ret) at (-2,2) [bb] ...
ret; (fallthrough) at (-4,0) [bb] ...;
(call) -> (ret) (ret) -> (fallthrough)

Figure 1: Caption

2.2 Finding the opcode traces

3 Abstract VM model

3.1 Definition

3.2 Additional (basic) assumptions

3.3 Finding registers

4 Opcode semantics

5 Experimental setup & conclusion