L3 internship : Shallow description of the Python Virtual Machine

Mathis Bouverot-Dupuis

June & July 2021

Table des matières

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 two Python VMs : CPython (the standard python interpreter) and PyPy.

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. We obtain this execution trace using a custom Intel PIN tracer.

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,

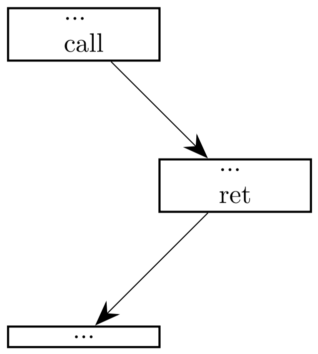

FIGURE 1 – Caption

2.2 Finding fetch and dispatch

The basic idea to find the fetch, decode and dispatch is to assume that all bytecodes share the same machine instructions for the fetch-decode-dispatch, as in (ref to cfg image, with filtered blocks). Note that this is a somewhat restrictive assumption, albeit necessary for this method to work. Some VM implementations duplicate the fetch-decode-dispatch code for each different bytecode (see other cfg image) : most notably CPython does so. This optimization ("computed-gotos") can thankfully be disabled when compiling CPython. In all my experiments I used this modified CPython version.

We look for a basic block that has the following properties :

— It is executed a large number of times (because it is shared between all bytecodes).
— It contains at least one instruction that reads a few bytes from memory (the fetch).
— It has a large number of outgoing edges (the dispatch).

Unfortunately, this method is too restrictive : the fetch-decode-dispatch aren't always in the same basic block. There can be branches (e.g. to check for a termination condition) between the fetch and the dispatch, as shown in (ref to unfiltered cfg image). To handle this issue, we assume the fetch-decode-dispatch are in the same basic block in the CFG after we have filtered out some irrelevant details.

The way we filter the CFG is by removing basic blocks and edges that aren't executed a significant number of times. This way we only retain the main structure of the graph, abstracting away the less often taken branches. However we might remove some or most of the bytecode execution code from the graph (we only retain the opcodes executed a lot of times) (maybe image comparing filtered and unfiltered ?) : the unfiltered CFG is a good place to look for the dispatch, and the filtered CFG is the place to look for the fetch-decode-dispatch block.

We thus look for a basic block in the filtered CFG that has the following properties :

— It is executed a large number of times.
— It contains at least one instruction that reads a few bytes from memory.
— The corresponding block in the unfiltered CFG has a large number of outgoing edges.

Once we have found the fetch, it is easy to find the bytecode : we simply get the value read by the fetch. We can also divide the trace into chunks corresponding to each bytecode, and remove the irrelevant parts of the VM trace (initialization and finalization code) : we obtain a list of small traces for each bytecode.

3 Abstract VM model

Before we can continue the analysis any further, we give ourselves an abstract VM model. While the previous section didn't assume much about the VM, this model follows more closely the Python VM.

3.1 Definition

We assume a stack-based VM, with no registers. It's internal state consists in :

— a set of code blocks

— a set of value stacks

— a stack of frames.

— pointers to the current frame (fp), the current bytecode (ip) and the top of the current value stack (sp)

A code block is a list of instructions. Each instruction is 2 bytes wide, the first byte being the opcode (example : ADD, POP, JMP) and the second byte being an argument. A code block contains all the code a given Python function can execute.

A value stack is a stack of Python values (objects). The contents of the stack are opaque to us. However all the values in the stack are contiguous in memory and of the same size.

Each frame corresponds to the execution of some Python function. A frame has its own code block and value stack.

The term "pointer" is intentionally vague : a pointer could be stored in a physical register or in a memory cell, it could contain the address of the object it points to or be an index into a list. During experiments I had to make additional assumptions about pointers, as will be explained in the following sections.

We will write $ptr \leftarrow f(ptr)$ where f is any function and ptr is ip, sp or fp to indicate that a given opcode changes the value of ptr to $f(ptr)$.

3.2 Finding pointers

We assume the state of the VM is valid at the fetch (and at the dispatch). At other points in the execution trace, the registers and memory cells used to store the VM pointers may be used for different purposes. Our goal is then to find the most information we can about the VM state before each bytecode.

3.2.1 Finding *ip*

Our method for finding the fetch gives us the instruction(s) that read the bytecode, and thus the address of the bytecode in memory : this gives us the value of ip before each opcode.

3.2.2 Finding *fp*

To find the frame pointers, we must make another assumption : that the C stack of the interpreter corresponds to the stack of the Python program it is running (remember we only look at the program state between bytecodes). Changes in $\%rsp$ should then give us the points at which the frame changes. Looking at registers that also change exactly and only at the same time as $\%rsp$ (and are also aligned on 8 bits) gives us additional frame pointers : pointers to some frame data (for instance, a C struct on the heap).

Knowing the values of the frame pointers, we then tried to recognize the stack structure of frames, but some opcodes seem to operate on the stack of frame in non-standard ways : i.e. other than push or pop one, a even several, frame(s).

3.2.3 Finding *sp*

A first challenge is where to look for *sp* : registers or memory ? It turns out it can be in both, depending on the interpreter (in a register for CPython, in memory for PyPy). Looking for *sp* in registers is straightforward, as there are a very limited number of them. However memory is more complex : we can't scan through each memory cell. What we can hope for is that *sp* is stored in a C struct pointed to by a frame pointer : *sp* must be used very often, so requiring more than one pointer indirection to access it seems very inefficient. We thus look for *sp* near addresses pointed to by frame pointers. More precisely, we assume *sp* will always be stored at $[fp + ofs]$ for fp a frame pointer (whose value can change during the program execution) and ofs a small constant offset (a multiple of 8).

To detect whether a location (register or memory cell) stores the value of *sp*, we take the list of values this location holds for each opcode. The basic idea is that many opcodes modify *sp* in the same way : pop or push a few values. But we must be careful : we do not know whether *sp* is stored as the address of the top stack cell, or as the index of the top stack cell, or something else. Depending on the case, a PUSH might increment *sp* by 8, 1 or something else. The solution is to compute the alignment of *sp*, i.e. the largest power of 2 that divides all the values in the list. We implement this idea as :

— Compute the maximum consecutive number of times that $sp \leftarrow sp + k * align$ where k is a small constant (e.g. $-2 \leq k \leq 2$). This number shouldn't change much with the size of the program (we check consecutive occurences) : we require it to be larger than a constant (e.g. 10).

— Compute the maximum (not necessarily consecutive) number of times that $sp \leftarrow sp - align$, i.e. the number of POPs. This number should be proportional to the size of the program : we check it is larger than a percentage of the total opcode count (e.g. 10%).

The second test used is to distinguish between *sp* and ip/fp : *ip* and fp might be stored in a similar way as *sp*, and pass the first test.

3.2.4 Code blocks

We compute the code blocks by partitioning the bytecode instructions according to (transitive closure of) the following relation. Two bytecode instructions are in the same code block if :

— They are executed in the same frame (i.e. there is no frame change between two of their executions). This is because all instructions executed inside a frame are in the same Python function.

— They are close in memory. This is because different code blocks are likely to be far apart in memory.

4 Opcode semantics

5 Experimental setup & conclusion