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 or opcodes) 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 (see Figure 2a).

An important class of virtual machines are programming language interpreters, such as the Python interpreter (CPython) or the Java Virtual Machine (JVM). During this internship I focussed on 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 ? In what follows we will often write 'finding the fetch' to avoid repeating 'fetch-decode-dispatch'.

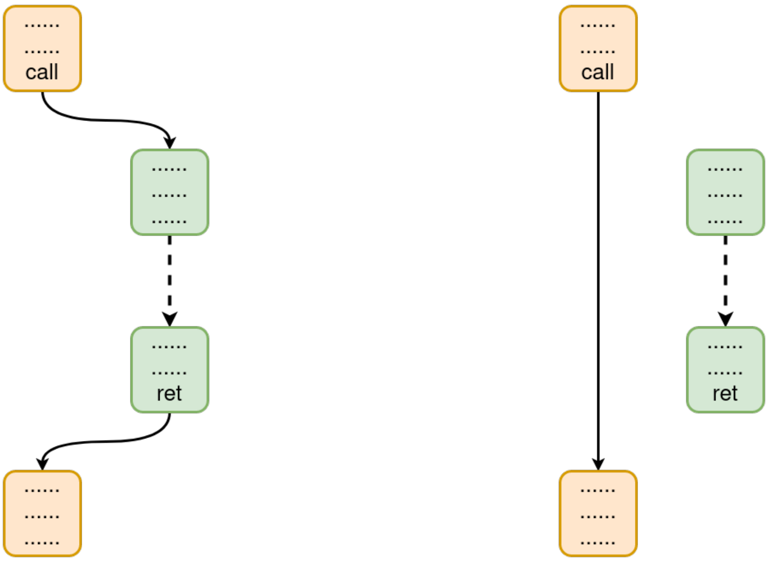
2.1 Building the CFG

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 transformations to simplify the graph. Each time we see a *call* instruction, we find the corresponding *ret* instruction, and delete and add edges as shown in Figure 1. As a consequence of this transformation, the CFG is split into several connected components, each corresponding to a single function (this is the convention adopted by the binary analysis framework IDA).



(a) Original CFG

(b) Transformed CFG

FIGURE 1 – Transformation of the CFG around call/ret

2.2 Finding the fetch

2.2.1 Basic Method

The basic idea to find the fetch is to assume that all opcodes share the same basic block for the fetch, as in Figure 2a. Note that this is a somewhat restrictive assumption, albeit necessary for this method to work. Some VM implementations use a fetch block for each different opcode (see Figure 2b) : 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.

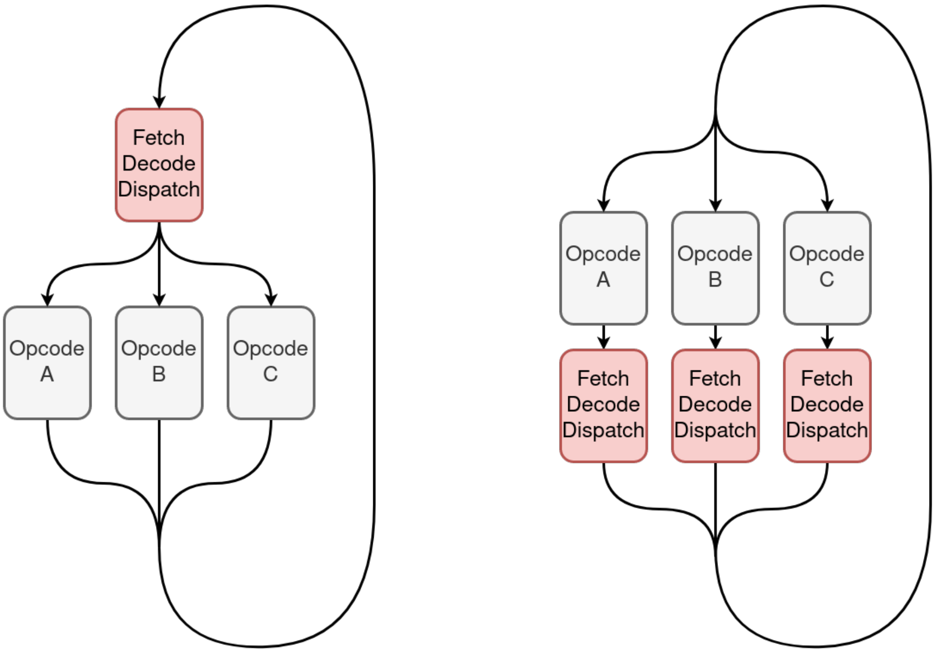
We thus build the CFG $\mathcal{C}$ and look for a basic block that :

— is executed a large number of times (because it is shared between all opcodes).

— contains at least one instruction that reads a few bytes from memory (the fetch).

— has a large number of outgoing edges (the dispatch).

Unfortunately, this method is too restrictive : the fetch, decode and dispatch aren't always in the same basic block in $\mathcal{C}$. There can be branches (e.g. to check for a termination condition, etc.) between the fetch and the dispatch, as shown in (ref to unfiltered cfg image).



(a) Centralized fetch

(b) Decentralized fetch ('computed-gotos')

FIGURE 2 – Examples of VM control flow graphs

2.2.2 Using the filtered CFG

To handle this issue, we first filter out irrelevant detail from $\mathcal{C}$, and then hope that the fetch-decode-dispatch are in the same basic block in the new CFG $\mathcal{C}_{filtered}$.

Given two parameters *exec_blocks* and *exec_edges*, we build $\mathcal{C}_{filtered}$ as follows :

- For each block $B \in \mathcal{C}$, if B is executed at least *exec_blocks* times, add B to $\mathcal{C}_{filtered}$.
- For each edges $(E : B \rightarrow B') \in \mathcal{C}$, if E is executed at least *exec_edges* times and both B and B' are executed at least *exec_blocks* times, add E to $\mathcal{C}_{filtered}$.

We then apply a simple transformation on $\mathcal{C}_{filtered}$ to merge all blocks B and B' such that $next(B) = \{B'\}$ and $prev(B') = \{B\}$. During my experiments I had to tune the values of *exec_blocks* and *exec_edges*, and found values around 1000 to give satisfactory results. This method thus only works on Python programs that contain at least a few thousand opcodes.

The CFG $\mathcal{C}_{filtered}$ only retains the main structure of the graph, abstracting away the code paths rarely taken. However we might remove some or most of the opcode execution code from the graph : see figures 4 and 5 for a comparison between $\mathcal{C}$ and $\mathcal{C}_{filtered}$. The original CFG $\mathcal{C}$ is a good place to look for the dispatch, and $\mathcal{C}_{filtered}$ is the place to look for the fetch-decode-dispatch basic block.

We thus look for a basic block in $\mathcal{C}_{filtered}$ that :

- is executed a large number of times.
- contains at least one instruction that reads a few bytes from memory (the fetch).
- contains at least one instruction that has a large number of outgoing edges in $\mathcal{C}$.

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.

```
f = open("file", "w")
for i in range(1000):
        f.write(str(i))
```

Figure 3 – An example python program, used to generate the graphs in Figures 4 and 5.

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.

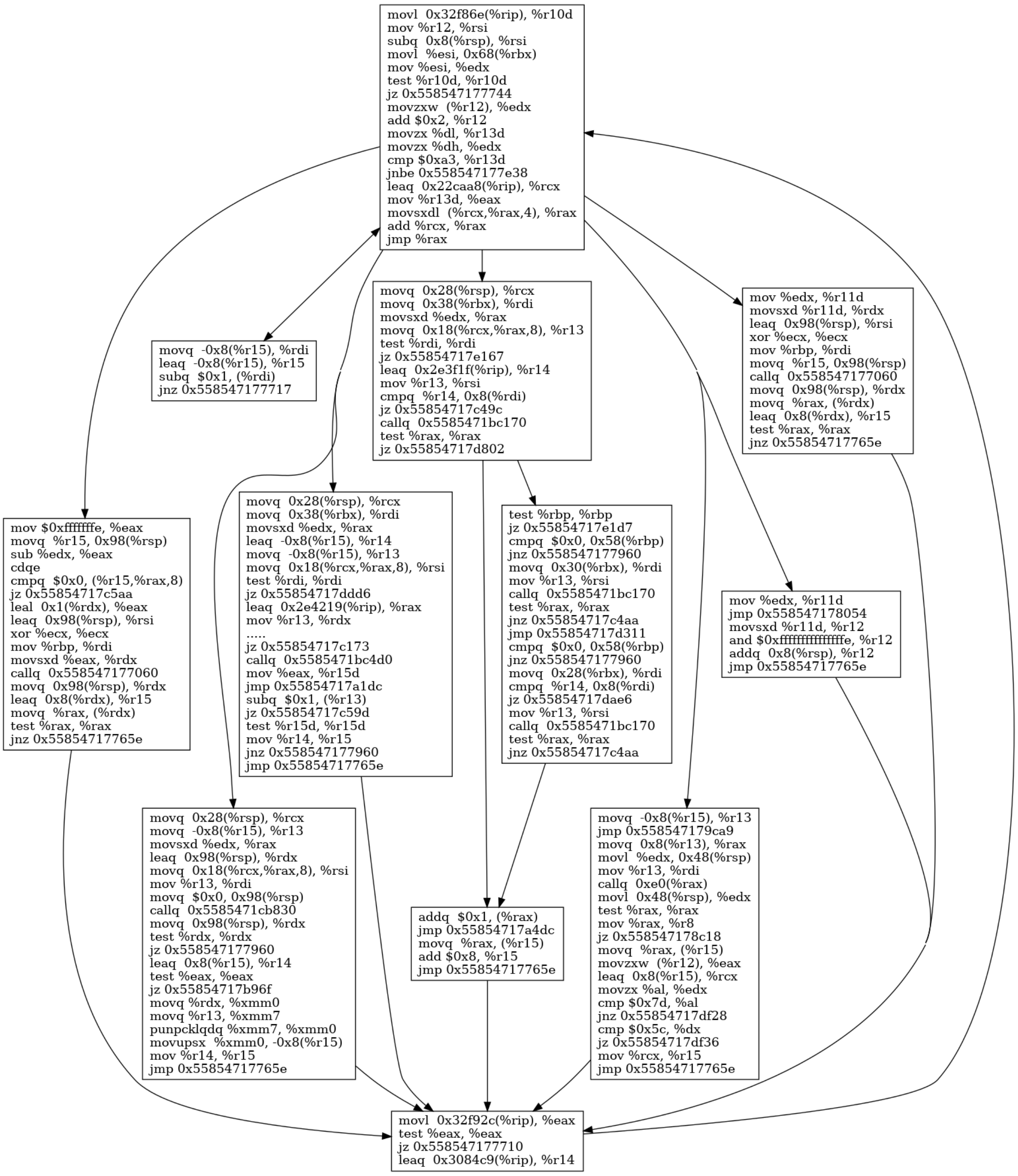
We will write $ptr \leftarrow f(ptr)$ where f is any function and ptr is ip, sp or fp to indicate that a given opcode changes the value of ptr to $f(ptr)$.

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.



```
movl  0x32f86e(%rip), %r10d
mov %r12, %rsi
subq  0x8(%rsp), %rsi
movl  %esi, 0x68(%rbx)
mov %esi, %edx
test %r10d, %r10d
jz 0x558547177744
movzxw  (%r12), %edx
add $0x2, %r12
movzx %dl, %r13d
movzx %dh, %edx
cmp $0xa3, %r13d
jnbe 0x558547177e38
leaq  0x22caa8(%rip), %rcx
mov %r13d, %eax
movsxdl  (%rcx,%rax,4), %rax
add %rcx, %rax
jmp %rax
```

```
movq  -0x8(%r15), %rdi
leaq  -0x8(%r15), %r15
subq  $0x1, (%rdi)
jnz 0x558547177717
```

```
movq  0x28(%rsp), %rcx
movq  0x38(%rbx), %rdi
movsxd %edx, %rax
movq  0x18(%rcx,%rax,8), %r13
test %rdi, %rdi
jz 0x55854717e167
leaq  0x2e3f1f(%rip), %r14
mov %r13, %rsi
cmpq  %r14, 0x8(%rdi)
jz 0x55854717c49c
callq  0x5585471bc170
test %rax, %rax
jz 0x55854717d802
```

```
mov %edx, %r11d
movsxd %r11d, %rdx
leaq  0x98(%rsp), %rsi
xor %ecx, %ecx
mov %rbp, %rdi
movq  %r15, 0x98(%rsp)
callq  0x558547177060
movq  0x98(%rsp), %rdx
movq  %rax, (%rdx)
leaq  0x8(%rdx), %r15
test %rax, %rax
jnz 0x55854717765e
```

```
mov $0xfffffffe, %eax
movq  %r15, 0x98(%rsp)
sub %edx, %eax
cdqe
cmpq  $0x0, (%r15,%rax,8)
jz 0x55854717c5aa
leal  0x1(%rdx), %eax
leaq  0x98(%rsp), %rsi
xor %ecx, %ecx
mov %rbp, %rdi
movsxd %eax, %rdx
callq  0x558547177060
movq  0x98(%rsp), %rdx
leaq  0x8(%rdx), %r15
movq  %rax, (%rdx)
test %rax, %rax
jnz 0x55854717765e
```

```
movq  0x28(%rsp), %rcx
movq  0x38(%rbx), %rdi
movsxd %edx, %rax
leaq  -0x8(%r15), %r14
movq  -0x8(%r15), %r13
movq  0x18(%rcx,%rax,8), %rsi
test %rdi, %rdi
jz 0x55854717ddd6
leaq  0x2e4219(%rip), %rax
mov %r13, %rdx
.....
jz 0x55854717c173
callq  0x5585471bc4d0
mov %eax, %r15d
jmp 0x55854717a1dc
subq  $0x1, (%r13)
jz 0x55854717c59d
test %r15d, %r15d
mov %r14, %r15
jnz 0x558547177960
jmp 0x55854717765e
```

```
test %rbp, %rbp
jz 0x55854717e1d7
cmpq  $0x0, 0x58(%rbp)
jnz 0x558547177960
movq  0x30(%rbx), %rdi
mov %r13, %rsi
callq  0x5585471bc170
test %rax, %rax
jnz 0x55854717c4aa
jmp 0x55854717d311
cmpq  $0x0, 0x58(%rbp)
jnz 0x558547177960
movq  0x28(%rbx), %rdi
cmpq  %r14, 0x8(%rdi)
jz 0x55854717dae6
mov %r13, %rsi
callq  0x5585471bc170
test %rax, %rax
jnz 0x55854717c4aa
```

```
mov %edx, %r11d
jmp 0x558547178054
movsxd %r11d, %r12
and $0xfffffffffffffffe, %r12
addq  0x8(%rsp), %r12
jmp 0x55854717765e
```

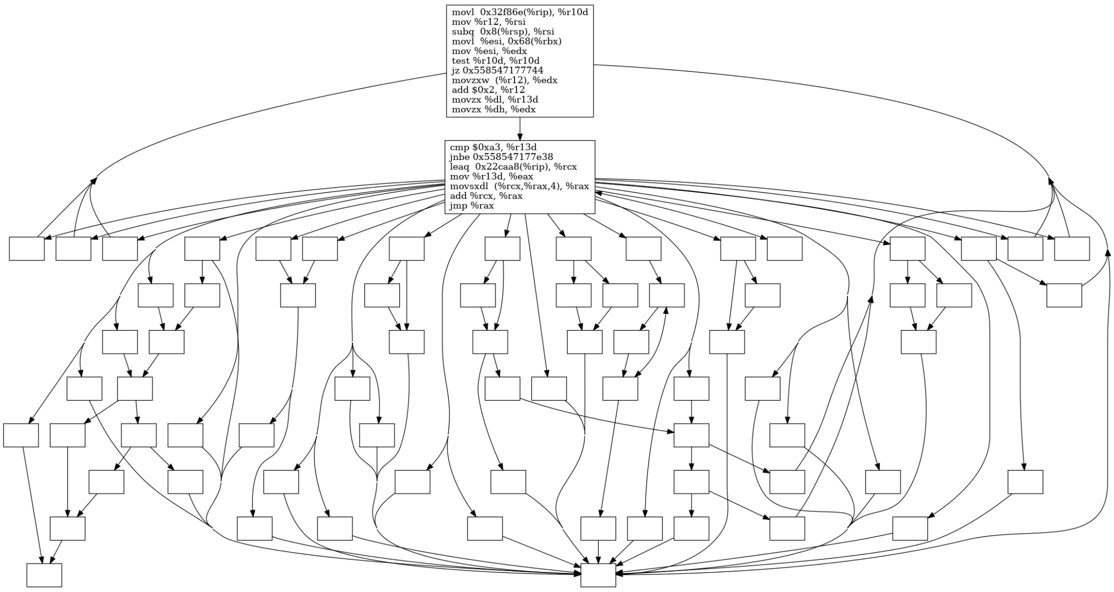
```
movq  0x28(%rsp), %rcx
movq  -0x8(%r15), %r13
movsxd %edx, %rax
leaq  0x98(%rsp), %rdx
movq  0x18(%rcx,%rax,8), %rsi
mov %r13, %rdi
movq  $0x0, 0x98(%rsp)
callq  0x5585471cb830
movq  0x98(%rsp), %rdx
test %rdx, %rdx
jz 0x558547177960
leaq  0x8(%r15), %r14
test %eax, %eax
jz 0x55854717b96f
movq %rdx, %xmm0
movq %r13, %xmm7
punpcklqdq %xmm7, %xmm0
movupsx  %xmm0, -0x8(%r15)
mov %r14, %r15
jmp 0x55854717765e
```

```
addq  $0x1, (%rax)
jmp 0x55854717a4dc
movq  %rax, (%r15)
add $0x8, %r15
jmp 0x55854717765e
```

```
movq  -0x8(%r15), %r13
jmp 0x558547179ca9
movq  0x8(%r13), %rax
movl  %edx, 0x48(%rsp)
mov %r13, %rdi
callq  0xe0(%rax)
movl  0x48(%rsp), %edx
test %rax, %rax
mov %rax, %r8
jz 0x558547178c18
movq  %rax, (%r15)
movzxw  (%r12), %eax
leaq  0x8(%r15), %rcx
movzx %al, %edx
cmp $0x7d, %al
jnz 0x55854717df28
cmp $0x5c, %dx
jz 0x55854717df36
mov %rcx, %r15
jmp 0x55854717765e
```

```
movl  0x32f92c(%rip), %eax
test %eax, %eax
jz 0x558547177710
leaq  0x3084c9(%rip), %r14
```

Figure 4 – The function implementing the VM loop in $\mathcal{C}_{filtered}$, constructed from a trace of CPython running the program in Figure 3. Notice how the fetch (movzxw (%r12), %edx)) and the dispatch (jmp %rax) are in the same block (the top-most block), and that this block has relatively few outgoing edges.



```
movl  0x32f86e(%rip), %r10d
mov %r12, %rsi
subq  0x8(%rsp), %rsi
movl  %esi, 0x68(%rbx)
mov %esi, %edx
test %r10d, %r10d
jz 0x558547177744
movzxw  (%r12), %edx
add $0x2, %r12
movzx %dl, %r13d
movzx %dh, %edx
```

```
cmp $0xa3, %r13d
jnbe 0x558547177e38
leaq  0x22caa8(%rip), %rcx
mov %r13d, %eax
movsxdl  (%rcx,%rax,4), %rax
add %rcx, %rax
jmp %rax
```

FIGURE 5 – The function implementing the VM loop in $\mathcal{C}$, constructed from the same trace as in Figure 4. Only the blocks implementing the fetch and the dispatch have their code shown. Notice how the fetch and the dispatch are in different blocks, and that the dispatch has many outgoing edges.

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 (the 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

4.1 Examples & Challenges

Once we have the values of *ip* and *sp* for each opcode, we try to attribute semantics to each opcode, i.e. characterize how they change the values of *ip* and *sp*. One limitation of our 'shallow' description is that we do not known what values are stored on the stack : typical Python interpreters use the stack to store pointers to objects which we do not know the memory layout of. We thus can't differentiate between an ADD and a MUL operation for instance. All we can detect is the number of push/pops each opcode performs and how it changes the value of *ip*.

We would like to build semantics of the following form, e.g. for the simple case of an ADD opcode :

- $ip \leftarrow ip + 2$
- $sp \leftarrow sp - 8$

Meaning that when we execute an ADD, *ip* always increases by 2 and *sp* always decreases by 8 (ADD pops its two arguments, pushes the result and advances to the next instruction). Remember that Python instructions are 2 bytes long, so *ip*+2 is the instruction directly following *ip*. However *sp* can be the address of the top of the stack (in which case $sp - 8$ is the value directly under *sp*, since stack values are 8 byte pointers) or can be something else, e.g. an index. In this case *sp* will be aligned on 1 rather than on 8, and ADD would have the following semantics :

- $ip \leftarrow ip + 2$
- $sp \leftarrow sp - 1$

Likewise, one could imagine a Python interpreter that grows the stack towards the bottom (CPython and PyPy both grow the stack towards the top), in which case it could become $sp \leftarrow sp + 1$. The precise semantics of an opcode thus depend on the particular interpreter we choose.

Another characteristic of our approach is that a single execution trace often doesn't give us enough information to find the most general semantics for an opcode. Take for example a conditional jump instruction such as POP_JUMP_IF_TRUE, which performs a jump if the top of the stack is *true* and otherwise advances to the next instruction. In a trace where each time we see this opcode the top of the stack is $false$, we can't detect that this is a conditional jump (or even a jump) and must give it the semantic $ip < -ip + 2$. We thus compute the most precise semantics for a given execution trace.

Note that for a given opcode, several semantics may be acceptable. For instance the ADD opcode always has an argument equal to 0 : thus we could characterize it by $ip \leftarrow ip + 2$ as well as by $ip \leftarrow ip + arg + 2$. In the second case we would interpret it as a relative jump, which it obviously isn't. I thus made the choice to only give the most 'general' characterization (in a sense I will define later on) for each opcode : ADD would thus be classified as $ip \leftarrow ip + 2$.

4.2 Definition

Building the *ip*-related semantics of an opcode is relatively straightforward. Most opcodes never change the current frame. For these, we five one of the following characterizations :

1. Fall-through : $ip \leftarrow ip + k$. This corresponds to opcodes that don't alter the flow of control of the program.
2. Relative jump : $ip \leftarrow ip + arg + k$.
3. Absolute jump : $ip \leftarrow$ block-start $+ arg + k$.
4. Conditional jump : either $ip \leftarrow ip + k_1$ or $ip \leftarrow$ block-start $+ arg + k_2$. Note that we cannot find what the condition is, since this would require inspecting values in the stack.

Here k, k_1 and k_2 are small constants (typically $-2 \leq k \leq 2$), arg is the value of the argument of the opcode, and 'block-start' is the address of the first instruction in the current code block. The different characterizations are listed (top to bottom) from the most 'general' to the most precise : for each opcode, we give it the most 'general' characterization that is acceptable.

We didn't manage to find much about instructions that jump between frames, for lack of proper frame identification (we only know when frames change, not what frame we are in). We thus didn't characterize CALL or RET instructions. One additional difficulty is that CALL-like Python instructions don't always correspond to a change of frame. They are sometimes handled internally by the interpreter (e.g. calls to internal libraries such as I/O functions) : the next opcode in the trace is thus the fall-through instruction, and no frame change has been made.

Building the *sp*-related semantics is similar. Again, dealing with opcodes that change the current frame is too difficult, but for those that never change it, we give one of the following characterizations (from the most general to the least) :

1. $sp \leftarrow sp + k$. This corresponds to most simple pop/push opcodes, as well as most binary operations (ADD, SUB, MUL, ...).
2. $sp \leftarrow sp \pm A * arg + k$. This corresponds to opcodes that either consume a number of values given by arg (e.g. BUILD_TUPLE) or produce a number of values given by arg (e.g. UNPACK_SEQUENCE).

Here k is a small constant (typically $-2 \leq k \leq 2$), arg is the value of the argument of the opcode, and A is the alignment of sp (typically 1 or 8).

5 Experimental setup & conclusion