A Pin Tool for Memory Tracing

This Pin tool can instrument the main image by dynamically tracing every executing instruction and the corresponding registers and memory access.

Code

The source code of this Pin plugin is all in `project1.cpp`.

How it works

1. Detect Main Image

The tool instruments all Images at first with a callback function, which checks if the current Image is the Main Executable. If it is, the tool will record the High and Low boundaries of this image. Meanwhile, also marks the flag to indicate the Main has been loaded. Otherwise, the image will be ignored.

2. Detect Instructions within Main

The tool instrument every Instruction with a callback function to check if it is Main Instruction. It verifies the Main Loaded flag from the above step at first. The Instructions executed before the Main Loaded are ignored. Secondly, it checks if the address of the the current instruction (`INS_Address`) is within the Main boundaries. It ignores the unqualified instructions.

3. Log Instructions & Registers

For each of the above qualified instructions, the tool uses Pin API `INS_Mnemonic` to get its meaningful name. This name and the address are logged at first.

Then is uses `INS_MaxNumRRegs`, `INS_MaxNumWRegs` and `INS_RegR` to separately get which Read and Write registers it will access. Then through `REG_StringShort`, it logs the registers' name respectively. Furthermore, it also conducts an additional check for the floating register and logs the this information as well.

(However, I found some instruction will be instrumented more than once, so need to avoid overwrite or duplicate log here, but the following call needs re-insertion)

4. Log Execution & Memories

For each of the logged instructions, with the help of Pin API (`INS_MemoryOperandCount`, `INS_MemoryOperandIsRead` and `INS_MemoryOperandIsWritten`), the tool checks if the instruction need to access any memories and if it does, check its operation type: read/write. Then through `INS_InsertCall` it inserts three kinds of callback for Instructions with different memory requirements, which are read memory, write memory and no memory. These callback functions records the order index of the executing instruction, and the memory address and size if any.

5. Output Logs

The tool adopts a key-value map to store all the information mentioned above. The key is the unqiue instruction address. The value is the log message which consists of the address, instruction name, register operations, and the details of the memory access of each time the instruction is executed. The details of the memory access of each execution keep being appended to the original message.

After the target program ends, the tool will sort the log by its instruction address in ascending order so that the output aligns with how the instructions are stored in memory in reality. Then it prints the messages following this order.

Results

The output logs are grouped by instructions and ordered by instruction address in ascending order, not the order of being instrumented. Reading the instruction address order grant the convenience to refer the source code. Below is a sample snippet

```
10ddc8cd0 MOV -w-> al
10ddc8cdb MOVZX -r-> dl -w-> edi
        [14]  no mem access
10ddc8cde MOV -r-> rbp edi
        [15]  -w-> 7ffee1e374f8 <4>
10ddc8ce4 MOV -r-> rbp
        [16]  -w-> 7ffee1e374f4 <4>
10ddc8cee CMP -r-> rbp -w-> rflags
        [17]  -r-> 7ffee1e374f4 <4>
        [40]  -r-> 7ffee1e374f4 <4>
```

Each instruction is in the format of

```
{address} {name} -{read}-> {register}... -{write}-> {registers}...
```

An instruction can read or write multiple registers. They are grouped by the symbol `-r->` and `-w->`. Since the logs has been grouped by instructions, each instruction only appears once. However, existing such record does not mean it must has been executed. It only means the instruction is been loaded into memory and been instrumented by Pin, such as `10ddc8cd0` in the example.

Execution logs has two formats. If the instruction need to access memory, the format is

```
        [{order index}] -{read/write}-> {memory address} <{size}>
```

If it has no memory access, the format is

```
        [{order index}] no mem access
```

These detailed execution logs are appended to the instruction logs in new line with obvious indentation. The `order index` is an increasing integer to track the order of the execution so that we can easily follow jumps and notice loops like `10ddc8cee`.

Limitations

This tool cannot trace any memory without being accessed by at least one instruction. For example, a 255-long character array has been declared in the code `target/target.c`, but the program only uses the first 13 elements. Through the log, we have no way to know there are 242 characters in the memory. While recoving the memory, we will have a large unknown space following the 13 characters.

This tool also has no access to the actual value of each read and write operation for both registers and memories.

Some buffer types can not be concretely inferred merely through the information logged by this tool, such as `long`. Furthermore, no way we can tell if the type is `unsigned`.

Recover the Buffer Layout

To recover the buffer layout, we need to know its boundaries (`rbp` & `rsp`) and content types.

rbp

A function always pushes its previous `rbp` into the top of the stack and uses the current top `rsp` as the `rbp`. So the top address written by `PUSH -r-> rbp` followed by a `MOV -r-> rsp -w-> rbp` is the new `rbp`.

```
10ddc8c90 PUSH -r-> rbp rsp -w-> rsp
        [0]   -w-> 7ffee1e37720 <8>
10ddc8c91 MOV -r-> rsp -w-> rbp
        [1]   no mem access
```

rsp

When another function is called, the current instruction address `rip` is written upon current top (`rsp`) so that the program can return to the current execution later. Therefore, add the address written by any `CALL_NEAR -r-> rip rsp -w-> rip rsp` with the pointer size to get the `rsp`.

```
10ddc8cfb CALL_NEAR -r-> rip rsp -w-> rip rsp
        [19]  -w-> 7ffee1e374b8 <8>
```

float

A buffer is float if its size is `4` and modified by any floating registers. The logs additionally marks such registers by appending `(float)`. However, even without such marks, those register names like `xmm0` and `xmm1` are also distinguishable enough.

```
10ddc8ef3 CVTSS2SD -r-> rbp -w-> xmm1 (float)
        [1008]  -r-> 7ffee1e37628 <4>
```

double

A buffer can be detected as `double` through the same registers defining `float` as above but with a size of `8`.

```
10ddc8eeb MOVSD_XMM -r-> rbp -w-> xmm0 (float)
        [1007]  -r-> 7ffee1e37678 <8>
```

int

A buffer of `int` has the size of `4` and is operated by the instructions for numbers.

```
10ddc8d3b MOV -r-> rbp -w-> eax
        [36]  -r-> 7ffee1e374f4 <4>
10ddc8d41 ADD -r-> eax -w-> eax rflags
```

long

A buffer may be `long` if it has the size of `8` (64 bit) and is operated by the instructions for numbers without being loaded into the floating registers. Or it is read from a known `int`.

```
10ddc8dc4 MOVSXD -r-> rbp -w-> rax
        [748]  -r-> 7ffee1e374f4 <4>
10ddc8dd2 MOV -r-> rbp rax
        [750]  -w-> 7ffee1e374e0 <8>
```

char

A buffer is detected as `char` is it only has 1 byte.

```
[763]  -r-> 10ddc8f90 <1>
```

pointer

A pointer's buffer should be `8` bytes long at first.

If `rbp` is pushed to or pop from the buffer, it is a stack pointer.

```
10ddc8c90 PUSH -r-> rbp rsp -w-> rsp
        [0]  -w-> 7ffee1e37720 <8>
10ddc8f28 POP -r-> rsp -w-> rbp rsp
        [1025]  -r-> 7ffee1e37720 <8>
```

If `rip` is written to or read from the buffer by `CALL` and `RET` instructions, it is a instruction pointer.

```
10ddc8d66 CALL_NEAR -r-> rip rsp -w-> rip rsp
        [387]  -w-> 7ffee1e374b8 <8>
10ddc8f29 RET_NEAR -r-> rsp -w-> rip rsp
        [1026]  -r-> 7ffee1e37728 <8>
```

Some pointer types can be inferred by combining other inferred types. For example, a `char` is gotten by combining an `int` and a pointer. The pointer might be `char*` and the `int` is the index.

```
10ddc8dee MOV -r-> rbp -w-> rax
        [761]  -r-> 7ffee1e37500 <8>
10ddc8df5 MOVSXD -r-> rbp -w-> rcx
        [762]  -r-> 7ffee1e374f4 <4>
10ddc8dfc MOV -r-> rax rcx -w-> dl
        [763]  -r-> 10ddc8f90 <1>
```

Memory Recocery Figure

Please find the sample here. This sample is generated towards the `target/target.c`. Besides the memory layout and type, the actual variables are also mapped into the sample by referring the source code and assembly dumped.