Reverse Engineering

Class 7

Binary Instrumentation

Binary Instrumentation

- What's this?
  - Addition of code to the application original code that, generally, does not seek to alter its functional result (transparent)
  - Trampolines injection (callbacks)
  - Instructions modification (binary translation)
  - Source code or binary instrumentation
  - Instrumentation previous to execution or while executing

Binary Instrumentation

- Why?
  - Profiling – gather data for performance optimization
  - Code-coverage
  - Behavior analysis (understand functionality)
  - Memory analysis (leaks, dangling pointers)
  - In-memory fuzzing
  - Execution on a different architecture (binary translation)
  - Testing (trigger execution flows)

Binary Instrumentation

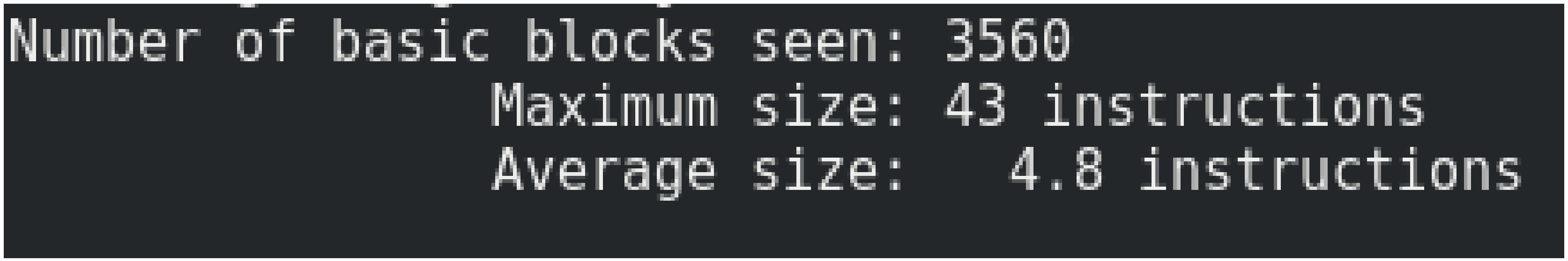
- Applicable to binaries (PE, ELF, classfiles, etc.)
- Binary instrumentation frameworks:
  - DynamoRIO (Windows, Linux, Android)
  - PIN (Windows, Linux)
  - Windows API Monitor (Windows)
  - QEMU (Linux)
  - ASM (Java)
  - Byteman (Java)

Binary Instrumentation

- DynamoRIO
  - Windows, Linux, Android
  - Open source (BSD license)
  - AArch32, AArch64, IA-32, x86_64
  - http://dynamorio.org

Binary Instrumentation

- Examples
  - `./bin64/drrun -c ./samples/bin64/libbbsize.so -- ls /`



```
Number of basic blocks seen: 3560
               Maximum size: 43 instructions
               Average size:   4.8 instructions
```

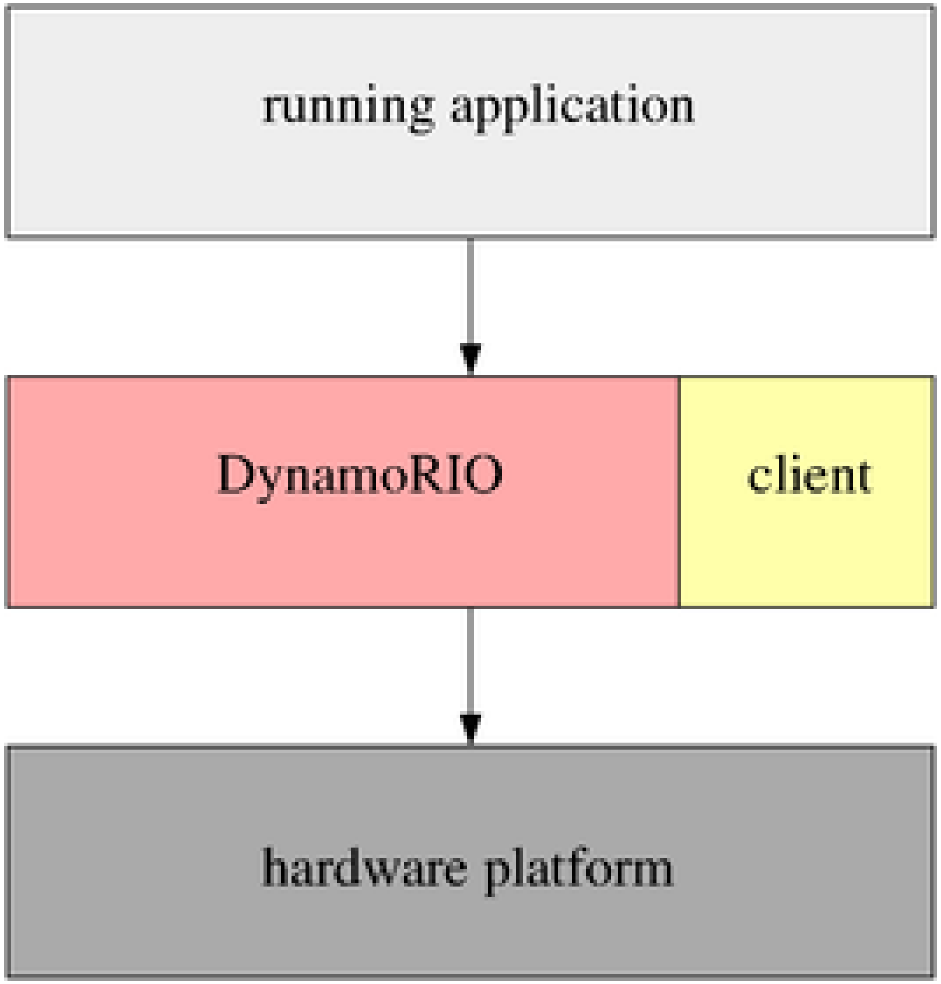
Binary Instrumentation

- Examples
  - `./bin64/drrun -c ./samples/bin64/libopcodes.so -- ls /`

```
Top 15 opcode execution counts in 64-bit AMD64 mode:
      11137 : xor
      14808 : shr
      16073 : pop
      16742 : sub
      20663 : push
      21278 : jnz
      21753 : jnz
      24025 : jz
      27753 : jz
      29073 : movzx
      29707 : and
      32082 : lea
      52862 : add
      57644 : test
      59205 : cmp
      90661 : mov
     101790 : mov
```

Binary Instrumentation

- Architecture



Binary Instrumentation

- Client library
  - Dynamic library (PIC)
  - Has instrumentation hooks implementation
  - Developed by the one that wants to instrument
  - Dynamically links DynamoRIO libraries
  - It’s loaded to the instrumented process from the beginning

Binary Instrumentation

- Client library receives events from DynamoRIO through registered callbacks
- Multiple callbacks may be registered for the same event and there can be multiple client libraries
- `dr_client_main`
  - Client library entry-point
  - Extensions initialization and callbacks registration
  - Called when process is created

Binary Instrumentation

- DynamoRIO has a general purpose API: it’s not advisable to “trust” in libraries loaded in the instrumented process
  - Open, read, write files
  - Synchronization primitives (I.e. Mutex)
  - Memory allocation
  - Threads creation
  - Etc.

Binary Instrumentation

- Examples of events to which the client library can subscribe:
  - Basic blocks or instructions creation
  - Threads initialization/finalization
  - Library loading/unloading
  - Syscalls interception
  - Signals or exceptions interception

Binary Instrumentation

- There are multiple instrumentation APIs
- Multi-Instrumentation Manager
  - Works on a 4-pass scheme over the executable code
  - 1) App2App
    - Application code transformations, previous to instrumentation
  - 2) Analysis
    - Application code analysis, once App2App transformations are applied. Code is not modified during this stage

Binary Instrumentation

- Multi-Instrumentation Manager
  - 3) Instrumentation
    - Application code transformations due to instrumentation. Can be high level transformations, that require multiple instructions. I.e. clean-calls insertions
  - 4) Instrumentation2Instrumentation
    - Pass to view and transform code generated during instrumentation. It's possible, for example, to make optimizations
  - Each stage is optional

Binary Instrumentation

- Multi-Instrumentation Manager
  - Callbacks registration for different instrumentation stages

```
if (!drmgr_register_bb_instrumentation_ex_event(app2app_cb, analysis_cb, instruction_cb, instr2instr_cb, NULL))
        DR_ASSERT(false);
```

Binary Instrumentation

- Multi-Instrumentation Manager
  - Instrumentation stage callback
    - Called once per basic block instruction

```
static dr_emit_flags_t
instruction_cb(void* drcontext, void*
tag, instrlist_t* bb, instr_t* instr, bool
for_trace, bool translating, void*
user_data);
```

Binary Instrumentation

- Basic blocks creation
  - Basic block: instructions sequence that ends in a flow control instruction
  - Instructions representation: `instr_t` and `instrlist_t` (*dr_ir_instr.h* and *dr_ir_instrlist.h*)
  - It's possible to modify, add or remove instructions

Binary Instrumentation

- Basic blocks creation
  - Previous to the execution of an application basic block, it’s copied to the “code cache” and instrumentation events are triggered
  - DynamoRIO keeps control of execution at the end of the basic block to continue instrumenting with the same strategy (as new basic blocks are executed)
    - Program is not instrumented upfront. Parts never executed are not instrumented

Binary Instrumentation

- Instructions insertion
  - Meta-instructions
    - Transparent for the application, used for monitoring purposes
    - I.e. call to a client library function
    - Not instrumented by DynamoRIO
  - Application instructions
    - Modify application state

Binary Instrumentation

- APIs to encode, decode and disassembly instructions
  - Structure: `instr_t`
- Clean Calls
  - Insert a C function call (hook) in the middle of a basic block
  - Function is invoked each time the basic block is executed
  - Application state is preserved (general purpose registers, floating point registers, stack, etc.)

Demo 7.1

Instrumentation

Binary Instrumentation

- How does instrumentation internally work?
  - *drrun* does an execve and *libdynamorio.so.6.2* starts executing
  - *_start* is the first function to execute in this library (implemented in assembly for x86)
  - *_start* relocates the library and calls *privload_early_inject*
  - This function uses a loader from DynamoRIO to load the ELF binary -to be instrumented- and initializes it (*dynamorio_app_init*)
  - Finally *dynamo_start* is called

Binary Instrumentation

- At this point, the process has mapped both the application to be instrumented (i.e. *main*) and the client library where hooks are implemented (i.e. *ins_example.so*)
- “dispatch” function is called so DynamoRIO can keep control of instrumented execution
  - This is an infinite loop that executes until process finishes
  - “dispatch” instruments basic blocks, put them to execute and recovers control (because instrumented basic blocks return to “dispatch”)

Binary Instrumentation

- *build_basic_block_fragment* function, called by “dispatch”, creates instrumented basic blocks
  - Instrumented basic blocks are called “fragments”
  - Fragments are represented by `fragment_t` structure
  - Example of a call to the 1st instrumented basic block from *main*: *start* parameter has value 0x400144

Binary Instrumentation

- Main original code:

```
(gdb) x/10i 0x400144
0x400144: push   %rbp
0x400145: mov    %rsp,%rbp
0x400148: mov    $0x0,%eax
0x40014d: callq  0x400166
0x400152: nop
0x400153: mov    $0x3c,%rax
0x40015a: mov    $0x0,%rdi
0x400161: syscall
0x400163: nop
0x400164: pop    %rbp
```

```
void _start() {
    foo();
    asm(
        "nop\n"
        "mov $60, %rax\n"
        "mov $0, %rdi\n"
        "syscall\n"
    );
}
```

Binary Instrumentation

- *build_basic_block_fragment* calls library client hooks to obtain the final list of instrumented instructions
- Once the list is obtained, *emit_fragment_common* function creates the new fragment
  - An executable segment has to be allocated in memory for the instructions (as a JIT compiler would do)

Binary Instrumentation

- Example of a `fragment_t` created out of *main's* first basic block:

```
$2 = {tag = 0x400144 "UH\211",
<incomplete sequence \345\270>, flags =
16777264, size = 435, prefix_size = 0
'\000', fcache_extra = 9 '\t',
  start_pc = 0x54691008 "eH\243",
in_xlate = {incoming_stubs = 0x0,
translation_info = 0x0}, next_vmarea =
0x0, prev_vmarea = 0x546c3090, also = {
    also_vmarea = 0x0, flushtime = 0}}
```

Binary Instrumentation

- You can see there information such as:
  - tag: original basic block virtual address
  - start_pc: instrumented basic block virtual address
- In */proc/<PID>/maps* we can verify how start_pc address (0x54691008) corresponds to an executable segment:

54691000-54692000 rwxp 00000000 00:00 0

Binary Instrumentation

- Instructions at 0x54691008 (instrumented basic block):

```
(gdb) x/50i 0x54691008
0x54691008:  movabs %rax,%gs:0x0
0x54691013:  movabs %gs:0x20,%rax
0x5469101e:  mov    %rsp,0x18(%rax)
0x54691022:  mov    0x2e8(%rax),%rsp
0x54691029:  movabs %gs:0x0,%rax
0x54691034:  lea    -0x2a8(%rsp),%rsp
0x5469103c:  callq  0x5468acc0
0x54691041:  callq  0x11087
0x54691046:  callq  0x5468ad80
```

Binary Instrumentation

- These instructions are instruction2instruction pass output, and what is finally executed
- In the previous listing, a *callq 0x11087* instruction can be spotted
  - *ins_example.so* is mapped to 0x10000
  - In 0x1087 offset *runtime_cb* function is located
  - In instruction2instruction a clean call to this function was inserted in each basic block

Binary Instrumentation

- ins_example.so

```
0000000000001087 <runtime_cb>:
1087:    push   %rbp
1088:    mov    %rsp,%rbp
108b:    lea    0x2d7(%rip),%rdi
1092:    mov    $0x0,%eax
1097:    callq  ba0 <dr_printf@plt>
109c:    nop
109d:    pop    %rbp
109e:    retq
```

```c
static void
runtime_cb(void) {
    dr_printf("runtime
call to hook
method!\n");
}
```

Binary Instrumentation

- These instructions (*callq 0x11087*) are clean calls
- Clean calls are preceded by a call to a function that saves the context (*callq 0x5468acc0*) and succeeded by one that restores the context (*callq 0x5468ad80*)

Binary Instrumentation

- Code seen in instrumentation2instrumentation pass has a call to 0x400166
  - At a C source code level (*main.c*), this call corresponds to *foo* function
- However, if instrumented block calls directly 0x400166, DynamoRIO loses control and won’t be able to continue instrumenting basic blocks
- Thus, at a fragment level, call to 0x400166 was substituted by the following code:

Binary Instrumentation

```
0x54691144:  mov    $0x0,%eax
...
0x5469118c:  movabs %rax,%gs:0x0
0x54691197:  movabs %gs:0x20,%rax
0x546911a2:  mov    0x18(%rax),%rsp
0x546911a6:  movabs %gs:0x0,%rax
0x546911b1:  pushq  $0x400152
0x546911b6:  jmpq   0x546b1030
```

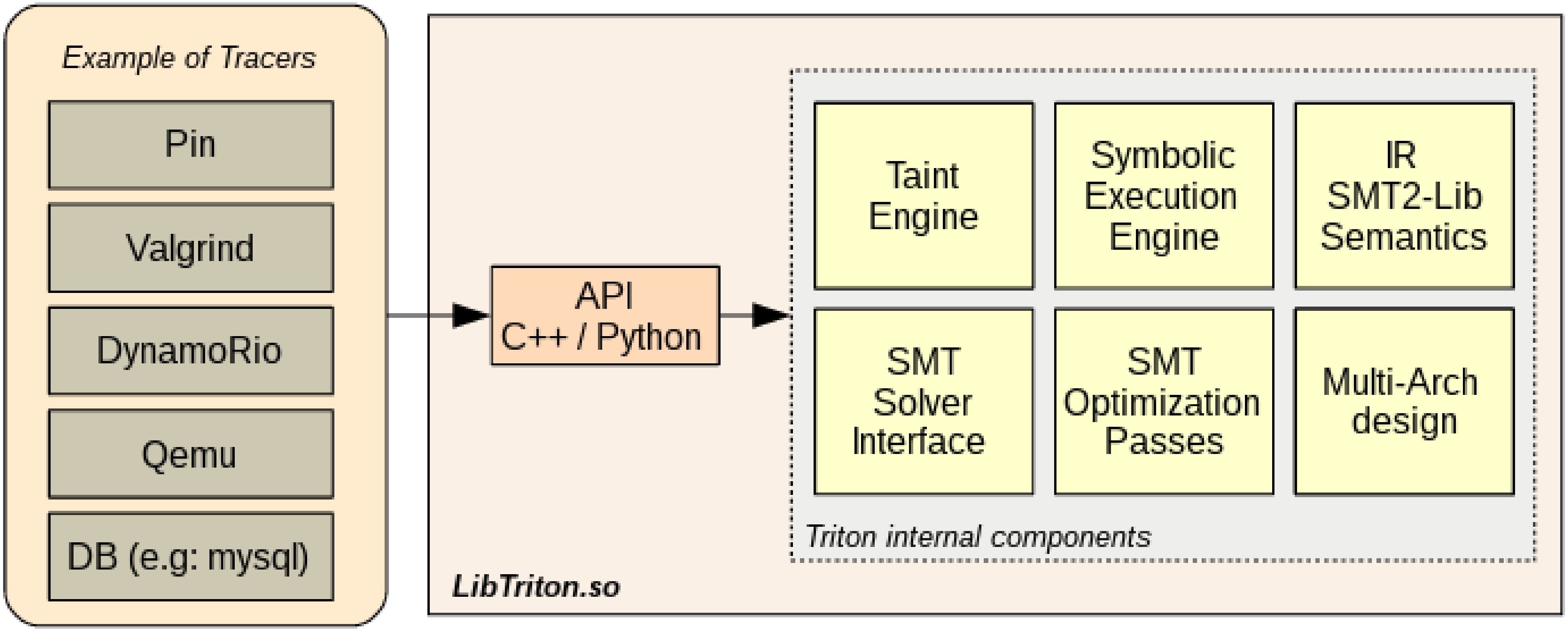
Binary Instrumentation

- Instead of calling 0x400166, a jump is made to 0x546b1030
- *foo* call return address is pushed to the stack
- What does 0x546b1030 code do?
  - Saves the context
  - Calls “dispatch”
- The cycle repeats, instrumenting *foo* basic block this time

Dynamic Binary Analysis

- Based on binary instrumentation frameworks, high level tools can be built to do run time checks on the binary
- In example, Valgrind has the capability of hooking memory allocations and freeings to detect leaks
- Triton is a DBA framework developed by Quarkslab with open license and multiplatform
  - Combines symbolic execution capabilities with SMT solvers

Dynamic Binary Analysis



SMT engine used by Triton is z3

Dynamic Binary Analysis

- Taint analysis
  - Trace memory and registers that are controlled by the user (input)
  - Inputs are considered insecure or untrusted. Every instruction that handles input is particularly interesting from the security point of view. This is “what the attacker controls”
  - A taint analysis policy has 3 components: 1) introduction rules, 2) propagation rules, and 3) check rules

Dynamic Binary Analysis

- Taint analysis
  - Introduction rules: registers, memory
  - Propagation rules:
    - Over-approximation (Triton)
      - False positives
    - Accurate approximation
    - Sub-approximation
      - False negatives
  - Propagation is a trade-off between precision and efficiency (memory + CPU)

Dynamic Binary Analysis

```
mov ax, 0x1122                  ; RAX is untainted
mov al, byte ptr [user_input] ; RAX is tainted
cmp ah, 0x99                    ; can we control this comparison?
```

In this case, over-approximation is going to assume that the comparison can be controlled by the user. That's a false positive

In these cases, symbolic execution can be used to ask the SMT solver if there is any value that satisfies the constraint

Dynamic Binary Analysis

- Symbolic execution
  - Convert values from registers and memory to symbolic
  - Make questions that can be answered by an SMT solver
  - Example:
    - convert eax register to symbolic
    - process an instruction that involves eax symbolic value
    - ask an initial value for eax such that once the instruction is executed, a specific condition is satisfied

Dynamic Binary Analysis

```
Triton = TritonContext()
Triton.setArchitecture(ARCH.X86)

# rax is now symbolic
Triton.convertRegisterToSymbolicVariable(Triton.registers.eax)

# process instruction
Triton.processing(Instruction("\x83\xc0\x07")) # add eax, 0x7

# get rax ast
eaxAst =
Triton.getAstFromId(Triton.getSymbolicRegisterId(Triton.registers.e
ax))

# constraint
c = eaxAst ^ 0x11223344 == 0xdeadbeaf

print 'Test 5:', Triton.getModel(c)[0] # Out: SymVar_0 = 0xCF8F8DE4
```

Dynamic Binary Analysis

- Code emulation
  - Process instructions located at a specific virtual address range:

```
0x40056d: "\x55",                              #   push    rbp
0x40056e: "\x48\x89\xe5",                      #   mov     rbp,rsp
0x400571: "\x48\x89\x7d\xe8",                  #   mov     QWORD PTR [rbp-0x18],rdi
0x400575: "\xc7\x45\xfc\x00\x00\x00\x00",      #   mov     DWORD PTR [rbp-0x4],0x0
0x40057c: "\xeb\x3f",                          #   jmp     4005bd <check+0x50>
0x40057e: "\x8b\x45\xfc",                      #   mov     eax,DWORD PTR [rbp-0x4]
0x400581: "\x48\x63\xd0",                      #   movsxd  rdx,eax
0x400584: "\x48\x8b\x45\xe8",                  #   mov     rax,QWORD PTR [rbp-0x18]
```

Dynamic Binary Analysis

- Code emulation
  - Create instructions (opcode + virtual address)
    - Instruction(), setOpcode, setAddress
  - Ask Triton to process instructions
    - Triton.processing(inst)
  - Obtain RIP value after executing them (in terms of virtual addressing)
    - ip = Triton.buildSymbolicRegister(Triton.registers.rip).evaluate()

Dynamic Binary Analysis

- Code emulation
  - Set concrete values to memory and registers
    - Triton.setConcreteMemoryValue(0x601040, 0x00)
    - Triton.setConcreteRegisterValue(Triton.registers.rdi, 0x1000)
  - Symbolize memory
    - Triton.convertMemoryToSymbolicVariable(MemoryAccess(address, CPUSIZE.BYTE))

Dynamic Binary Analysis

- Code emulation
  - Obtain concrete values from the memory and registers
    - Triton.getConcreteMemoryValue(MemoryAccess(write+4, CPUSIZE.DWORD))
    - Triton.getConcreteRegisterValue(Triton.registers.rax)
  - Instructions can be disassembled and operands obtained
    - inst.getDisassembly()
    - inst.getOperands()

Dynamic Binary Analysis

- Code emulation
  - It’s possible to analyze “micro-instructions” or “atomic instructions” that constitute an instruction
    - Many compilers use an intermediate representation (IR) for this type of instructions
  - I.e. movabs rax, 0x4142434445464748 involves:
    - Set rax with a specific value
    - Increase rip to point to the next instruction
  - inst.getSymbolicExpressions()

Dynamic Binary Analysis

- Code emulation
  - It’s possible to analyze which “micro-operation” modified a registry or a memory address
    - Triton.getSymbolicRegisters().items()
    - Triton.getSymbolicMemory().items()
  - When memory or registers are symbolic (Triton.buildSymbolicRegister(Triton.registers.ah)), it’s possible to get the micro-operations that modified it, or get a concrete value

Dynamic Binary Analysis

- Code emulation
  - Once performed the emulation, it’s possible to obtain all execution path constraints (result of each branch)
    - getPathConstraints → getBranchConstraints
    - I.e: 0x11223344: jne 0x55667788
    - Flag: true if branch was taken
    - Source address: 0x11223344
    - Destination address: 0x55667788 if branch is taken or next address in case not
    - pc: node that represents the branch within the Abstract Syntax Tree (AST)

Demo 7.2

Symbolic execution (Triton)

Lab

7.1

Create a client library for DynamoRIO capable of detecting function parameters that are pointers to dynamically allocated memory (x86_64, SystemV ABI)

Lab

7.2: Use symbolic execution in Triton to find an input that makes *check* function return 1:

```
int check(int i) {
  const unsigned char* c = (unsigned char*)&i;
  if (((c[0] ^ c[1]) == 0x3C) && ((c[0] * c[3]) ==
0x40) && c[1] != 0) {
    return 1;
  }
  return 0;
}
```

References

- http://dynamorio.org/docs/
- Triton - dynamic binary analysis framework
  - https://github.com/JonathanSalwan/Triton
  - https://triton.quarkslab.com