Triton: Concolic Execution Framework

Florent Saudel
Jonathan Salwan

SSTIC
Rennes – France

Slides: detailed version

June 3 2015

Keywords: program analysis, DBI, DBA, Pin, concrete execution, symbolic execution, concolic execution, DSE, taint analysis, context snapshot, Z3 theorem prover and behavior analysis.

Who are we?

- Jonathan Salwan is a student at Bordeaux University (CSI Master) and also an employee at Quarkslab
- Florent Saudel is a student at the Bordeaux University (CSI Master) and applying to an Internship at Amossys
- Both like playing with low-level computing, program analysis and software verification methods

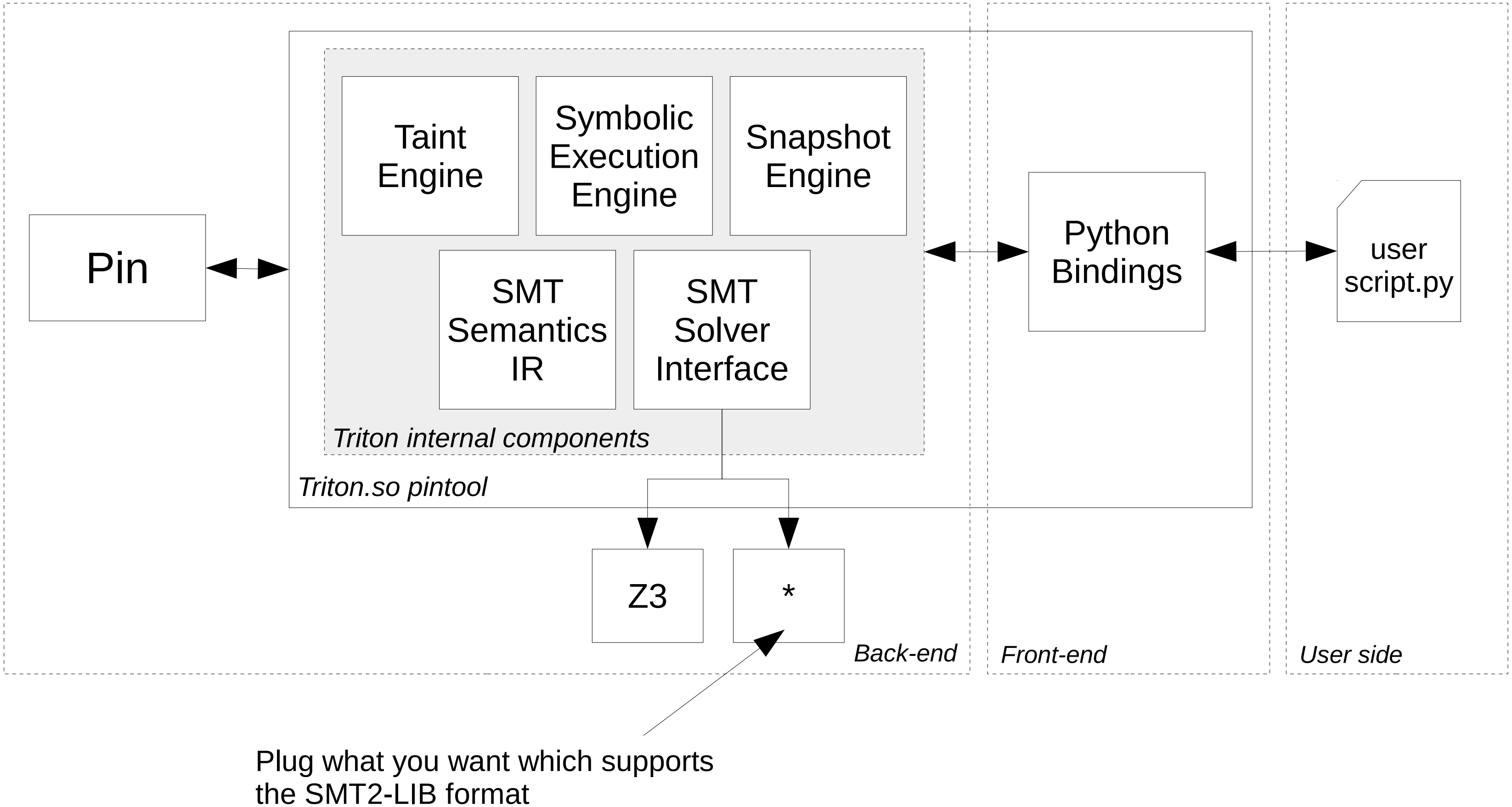
Where does Triton come from?

- Triton is a project started on January 2015 for our Master final project at Bordeaux University (CSI) supervised by Emmanuel Fleury from laBRI
- Triton is also sponsored by Quarkslab from the beginning

What is Triton?

- Triton is a concolic execution framework as Pintool
- It provides advanced classes to improve dynamic binary analysis (DBA) using Pin
  - Symbolic execution engine
  - SMT semantics representation
  - Interface with SMT Solver
  - Taint analysis engine
  - Snapshot engine
  - API and Python bindings

What is Triton?



Relative projects

- Well-known projects
  - SAGE
  - Mayhem
  - Bitblaze
  - S2E
- The difference?
  - Triton works online* through a higher level languages using the Pin engine

online*: Analysis is performed at runtime and data can be modified directly in memory to go through specific branches.

What kind of things you can build with Triton?

- You can build tools which:
  - Analyze a trace with concrete information
    - Registers and memory values at each program point
  - Perform a symbolic execution
    - To know the symbolic expression of registers and memory at each program point
  - Perform a symbolic fuzzing session
  - Generate and solve path constraints
  - Gather code coverage
  - Runtime registers and memory modification
  - Replay traces directly in memory
  - Scriptable debugging
  - Access to Pin functions through a higher level languages (Python bindings)
  - And probably lots of others things

Triton's Internal Components

Symbolic Engine

Symbolic Engine

- Symbolic execution is the execution of a program using symbolic variables instead of concrete values
- Symbolic execution translates the program's semantics into a logical formula
- Symbolic execution can build and keep a path formula
  - By solving the formula and its negation we can take all paths and “cover” a code
    - Instead of concrete execution which takes only one path
- Then a symbolic expression is given to a SMT solver to generate a concrete value

Symbolic Engine inside Triton

- A trace is a sequence of instructions

  $T = (Ins_1 \land Ins_2 \land Ins_3 \land Ins_4 \land \ldots \land Ins_i)$

- Instructions are represented with symbolic expressions
- A symbolic trace is a sequence of symbolic expressions
- Each symbolic expression is translated like this:

  REF_{out} = semantic

  - Where :
    - REF_{out} := unique ID
    - Semantic := REF_{in} | <<smt expression>>
- Each register or byte of memory points to its last reference → Single Static Assignment Form (SSA)

Register References

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

// All refs initialized to -1
Register Reference Table {
    EAX : -1,
    EBX : -1,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
}

Register references

Example:

```
► movsx eax, byte ptr [mem]    #0 = symvar_1
  add eax, 2
  mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
    EAX : #0,
    EBX : -1,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <#0, symvar_1>
}
```

Register references

Example:

```
  movsx eax, byte ptr [mem]    #0 = symvar_1
► add eax, 2                   #1 = add(#0, 2)
  mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : -1,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <#1, add(#0, 2)>,
    <#0, symvar_1>
}
```

Register references

Example:

```
  movsx eax, byte ptr [mem]   #0 = symvar_1
  add eax, 2                  #1 = add(#0, 2)
──►mov ebx, eax               #2 = #1
```

// All refs initialized to -1
Register Reference Table {
    EAX : **#1**,
    EBX : **#2**,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <**#2, #1**>,
    <**#1, add(#0, 2)**>,
    <**#0, symvar_1**>
}

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

→ What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
EAX : #1,
EBX : #2,
ECX : -1,
...
}

// Empty set
Symbolic Expression Set {
<#2, #1>,
<#1, add(#0, 2)>,
<#0, symvar_1>
}

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : **#2**,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <#2, #1>,
    <#1, add(#0, 2)>,
    <#0, symvar_1>
}

EBX holds the reference **#2**

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : **#2**,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <#2, #1>,
    <#1, add(#0, 2)>,
    <#0, symvar_1>
}

EBX holds the reference **#2**

What is **#2** ?

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

→ What is the semantic trace of EBX ?

```
// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : #2,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <#2, #1>,
    <#1, add(#0, 2)>,
    <#0, symvar_1>
}
```

EBX holds the reference **#2**

What is **#2** ?

Reconstruction: EBX = **#2**

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : **#2**,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <**#2**, **#1**>,
    <#1, add(#0, 2)>,
    <#0, symvar_1>
}

EBX holds the reference **#2**

What is **#2** ?

Reconstruction: EBX = **#1**

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
EAX : #1,
EBX : **#2**,
ECX : -1,
...
}

// Empty set
Symbolic Expression Set {
<#2, #1>,
<**#1**, **add(#0, 2)**>,
<#0, symvar_1>
}

EBX holds the reference **#2**

What is **#2** ?

Reconstruction: EBX = **add(#0, 2)**

Rebuild the trace with backward analysis

Example:

```
movsx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

→ What is the semantic trace of EBX ?

```
// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : #2,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <#2, #1>,
    <#1, add(#0, 2)>,
    <#0, symvar_1>
}
```

EBX holds the reference **#2**

What is **#2** ?

Reconstruction: EBX = **add(symvar_1, 2)**

Follow references over memory

- Assigning a reference for each register is not enough, we must also add references on memory

```
mov dword ptr [rbp-0x4], 0x0
...
mov eax, dword ptr [rbp-0x4]
```

```
push eax
...
pop ebx
```

What do we want to know?

```
Eax = 0 from somewhere
```

```
ebx = eax
```

References

```
#1 = 0x0
...
#x = #1
```

```
#2 = #1
...
#x = #2
```

References conclusion

- All registers, flags and each byte of memory are references
- A reference assignment is in SSA form during the execution
- The registers, flags and bytes of memory are assigned in the same way
- A memory reference can be assigned from a register reference (mov [mem], reg)
- A register reference can be assigned from a memory reference (mov reg, [mem])
- If a reference doesn't exist yet, we concretize the value and we affect a new reference

SMT Semantics Representation with SSA Form

SMT Semantics Representation with SSA Form

- All instructions semantics are represented via SMT2-LIB representation
- This SMT2-LIB representation is on SSA form

Assembly

```
add rax, rdx
```

SMT

```
rax = (bvadd ((_ extract 63 0) rax) ((_ extract 63 0) rdx))
... (af, cf, of, pf) ...
sf = (ite (= ((_ extract 63 63) rax) (_ bv1 1)) (_ bv1 1) (_ bv0 1))
zf = (ite (= rax (_ bv0 64)) (_ bv1 1) (_ bv0 1))
```

New rax reference

Old rax reference

Old rdx reference

SSA SMT

```
#60 = (bvadd ((_ extract 63 0) #58) ((_ extract 63 0) #54))
... (af, cf, of, pf) ...
#64 = (ite (= ((_ extract 63 63) #60) (_ bv1 1)) (_ bv1 1) (_ bv0 1))
#65 = (ite (= #60 (_ bv0 64)) (_ bv1 1) (_ bv0 1))
```

SMT Semantics Representation with SSA Form

- Why use SMT2-LIB representation?
  - SMT-LIB is an international initiative aimed at facilitating research and development in Satisfiability Modulo Theories (SMT)
  - As all Triton's expressions are in the SMT2-LIB representation, you can plug all solvers which supports this representation
    - Currently Triton has an interface with Z3 but feel free to plug what you want

Symbolic Execution Guided By The Taint Analysis

Symbolic Execution Guided By The Taint Analysis

- Taint analysis provides information about which registers and memory addresses are controllable by the user at each program point:
  - Assists the symbolic engine to setup the symbolic variables (a symbolic variable is a memory area that the user can control)
  - Limit the symbolic engine to the relevant part of the program
  - At each branch instruction, we directly know if the user can go through both branches (this is mainly used for code coverage)

Symbolic Execution Guided By The Taint Analysis

- Transform a tainted area into a symbolic variable

rax points on a tainted area

```
0x40058b: movzx eax, byte ptr [rax]
         -> #33 = ((_ zero_extend 24) (_ bv97 8))
         -> #34 = (_ bv4195726 64) ; RIP
0x40058e: movsx eax, al
         -> #35 = ((_ sign_extend 24) ((_ extract 7 0) #33))
         -> #36 = (_ bv4195729 64) ; RIP
```

Use symbolic variable instead of concrete value

```
0x40058b: movzx eax, byte ptr [rax]
         ->  #33 = SymVar_0 ; Controllable by the user
         ->  #34 = (_ bv4195726 64) ; RIP
0x40058e: movsx eax, al
         ->  #35 = ((_ sign_extend 24) ((_ extract 7 0) #33))
         ->  #36 = (_ bv4195729 64) ; RIP
```

Symbolic Execution Guided By The Taint Analysis

- Can I go through this branch?
  - Check if flags are tainted

eax is tainted

```
0x4005ae: cmp ecx, eax
          ->  #72 = (bvsub ((_ extract 31 0) #52) ((_ extract 31 0) #70))
          ...CF, OF, SF, AF, and PF skipped...
          ->  #78 = (ite (= #72 (_ bv0 32)) (_ bv1 1) (_ bv0 1)) ; ZF
          ->  #79 = (_ bv4195760 64) ; RIP

                   tainted


0x4005b0: jz 0x4005b9
          ->  #80 = (ite (= #78 (_ bv1 1)) (_ bv4195769 64) (_ bv4195762 64)) ; RIP
```

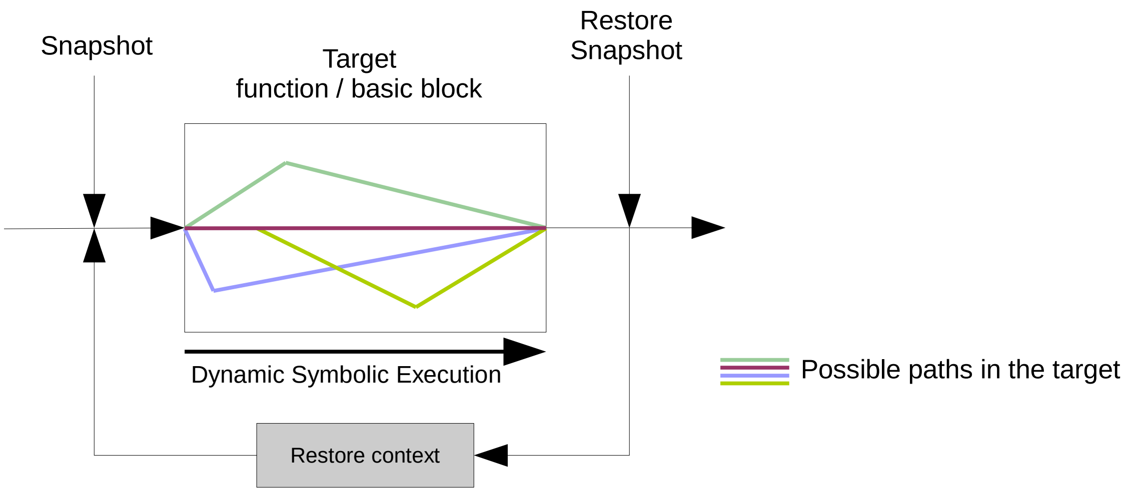
Taint Analysis guided by the Symbolic Engine and the Solver Engine

- As the symbolic execution may be guided by the taint analysis, the taint analysis may also be guided by the symbolic execution and the solver engine
- What to choose between an over-approximation and under-approximation?
  - Over-approximation: We can generate inputs for infeasible concrete paths.
  - Under-approximation: We can miss some feasible paths.
- The goal of the taint engine is to say YES or NO if a register and memory is probably tainted (byte-level over approximation)
- The goal of the symbolic engine is to build symbolic expressions based on instructions semantics
- The goal of the solver engine is to generate a model of an expression (path condition)
  - If your target is not tainted, don't ask a model → gain time
  - If the solver engine returns UNSAT → the tainted inputs can't influence the control flow to go through this path.
  - If the solver engine returns SAT → the path can be triggered with the actual tainted inputs. The model give us the set of concrete inputs for this path.

Snapshot Engine – Replay your trace

Snapshot Engine – Replay your trace

- The snapshot engine offers the possibility to take and restore snapshot
  - Mainly used to apply code coverage in memory. Useful when you fuzz the binary
  - In future versions, it will be possible to take different snapshots at several program point
- The snapshot engine only restores registers and memory states
  - If there is some disk, network,... I/O, Triton won't be able to restore the files modification



Stop talking about back-end!
Let's see how I can use Triton

How to install Triton?

- Easy is easy
- You just need:
  - Pin v2.14-71313
  - Z3 v4.3.1
  - Python v2.7

Shell 1: Installation

```
$ cd pin-2.14-71313-gcc.4.4.7-linux/source/tools/
$ git clone git@github.com:JonathanSalwan/Triton.git
$ cd Triton
$ make
$ ../../../pin -t ./triton.so -script your_script.py -- ./your_target_binary.elf64
```

Start an analysis

Code 1: Start analysis from symbols

```
import triton


if __name__ == '__main__':

    # Start the symbolic analysis from the 'check' function
    triton.startAnalysisFromSymbol('check')

    # Run the instrumentation - Never returns
    triton.runProgram()
```

Code 2: Start analysis from address

```
import triton


if __name__ == '__main__':

    # Start the symbolic analysis from address
    triton.startAnalysisFromAddr(0x40056d)
    triton.stopAnalysisFromAddr(0x4005c9)

    # Run the instrumentation - Never returns
    triton.runProgram()
```

Predicate taint and untaint

Code 3: Predicate taint and untaint at specific addresses

```
import triton


if __name__ == '__main__':

    # Start the symbolic analysis from the 'check' function
    triton.startAnalysisFromSymbol('check')

    # Taint the RAX and RBX registers when the address 0x40058e is executed
    triton.taintRegFromAddr(0x40058e, [IDREF.REG.RAX, IDREF.REG.RBX])

    # Untaint the RCX register when the address 0x40058e is executed
    triton.untaintRegFromAddr(0x40058e, [IDREF.REG.RCX])

    # Run the instrumentation - Never returns
    triton.runProgram()
```

Callbacks

- Triton supports 8 kinds of callbacks
  - **AFTER**
    - Defines a callback after the instruction processing
  - **BEFORE**
    - Defines a callback before the instruction processing
  - **BEFORE_SYMPROC**
    - Defines a callback before the symbolic processing
  - **FINI**
    - Define a callback at the end of the execution
  - **ROUTINE_ENTRY**
    - Define a callback at the entry of a specified routine.
  - **ROUTINE_EXIT**
    - Define a callback at the exit of a specified routine.
  - **SYSCALL_ENTRY**
    - Define a callback before each syscall processing
  - **SYSCALL_EXIT**
    - Define a callback after each syscall processing

Callback on SYSCALL

Code 4: Callback before and after syscalls processing

```
def my_callback_syscall_entry(threadId, std):

    print '-> Syscall Entry: %s' %(syscallToString(std, getSyscallNumber(std)))

    if getSyscallNumber(std) == IDREF.SYSCALL.LINUX_64.WRITE:
        arg0 = getSyscallArgument(std, 0)
        arg1 = getSyscallArgument(std, 1)
        arg2 = getSyscallArgument(std, 2)
        print '   sys_write(%x, %x, %x)' %(arg0, arg1, arg2)

def my_callback_syscall_exit(threadId, std):
    print '<- Syscall return %x' %(getSyscallReturn(std))

if __name__ == '__main__':
    startAnalysisFromSymbol('main')
    addCallback(my_callback_syscall_entry,  IDREF.CALLBACK.SYSCALL_ENTRY)
    addCallback(my_callback_syscall_exit,   IDREF.CALLBACK.SYSCALL_EXIT)
    runProgram()
```

Code 4 result

```
-> Syscall Entry: fstat
<- Syscall return 0
-> Syscall Entry: mmap
<- Syscall return 7fb7f06e1000
-> Syscall Entry: write
   sys_write(1, 7fb7f06e1000, 6)
```

Callback on ROUTINE

Code 5: Callback before and after routine processing

```
def mallocEntry(threadId):
    sizeAllocated = getRegValue(IDREF.REG.RDI)
    print '-> malloc(%#x)' %(sizeAllocated)


def mallocExit(threadId):
    ptrAllocated = getRegValue(IDREF.REG.RAX)
    print '<- %#x' %(ptrAllocated)

if __name__ == '__main__':
    startAnalysisFromSymbol('main')
    addCallback(mallocEntry, IDREF.CALLBACK.ROUTINE_ENTRY, "malloc")
    addCallback(mallocExit, IDREF.CALLBACK.ROUTINE_EXIT, "malloc")
    runProgram()
```

Code 5 result

```
-> malloc(0x20)
<- 0x8fc010
-> malloc(0x20)
<- 0x8fc040
-> malloc(0x20)
<- 0x8fc010
```

Callback BEFORE and AFTER instruction processing

Code 6: Callback before instruction processing

```
def my_callback_before(instruction):
    print 'TID (%d) %#x %s' %(instruction.threadId,
                              instruction.address,
                              instruction.assembly)

if __name__ == '__main__':

    # Start the symbolic analysis from the 'check' function
    startAnalysisFromSymbol('check')

    # Add a callback.
    addCallback(my_callback_before, IDREF.CALLBACK.BEFORE)

    # Run the instrumentation - Never returns
    runProgram()
```

Code 6 result

```
TID (0) 0x40056d push rbp
TID (0) 0x40056e mov rbp, rsp
TID (0) 0x400571 mov qword ptr [rbp-0x18], rdi
TID (0) 0x400575 mov dword ptr [rbp-0x4], 0x0
...
TID (0) 0x4005b2 mov eax, 0x1
TID (0) 0x4005b7 jmp 0x4005c8
TID (0) 0x4005c8 pop rbp
```

Instruction class

```
def my_callback(instruction):
    ...
```

- instruction.address
- instruction.assembly
- instruction.imageName - e.g: libc.so
- instruction.isBranch
- instruction.opcode
- instruction.opcodeCategory
- instruction.operands
- instruction.symbolicElements – List of SymbolicElement class
- instruction.routineName - e.g: main
- instruction.sectionName - e.g: .text
- instruction.threadId

SymbolicElement class

```
Instruction: add rax, rdx
SymbolicElement: #41 = (bvadd ((_ extract 63 0) #40) ((_ extract 63 0) #39)) ; blah
```

```
instruction.symbolicElements[0]
```

- symbolicElement.comment → blah
- symbolicElement.destination → #41
- symbolicElement.expression → #41 = (bvadd ((_ extract 63 0) #40) ((_ extract 63 0) #39))
- symbolicElement.id → 41
- symbolicElement.isTainted → True or False
- symbolicElement.source → (bvadd ((_ extract 63 0) #40) ((_ extract 63 0) #39))

Dump the symbolic expressions trace

Code 7: Dump a symbolic expression trace

```
def my_callback_after(instruction):
    print '%#x: %s' %(instruction.address, instruction.assembly)
    for se in instruction.symbolicElements:
        print '\t -> ', se.expression
    print

if __name__ == '__main__':
    startAnalysisFromSymbol('check')
    addCallback(my_callback_after, IDREF.CALLBACK.AFTER)
    runProgram()
```

Code 7 result

```
0x4005ab: movsx eax, al
         -> #70 = ((_ sign_extend 24) ((_ extract 7 0) #68))
         -> #71 = (_ bv4195758 64)

0x4005ae: cmp ecx, eax
         -> #72 = (bvsub ((_ extract 31 0) #52) ((_ extract 31 0) #70))
         ...
         -> #77 = (ite (= ((_ extract 31 31) #72) (_ bv1 1)) (_ bv1 1) (_ bv0 1))
         -> #78 = (ite (= #72 (_ bv0 32)) (_ bv1 1) (_ bv0 1))
         -> #79 = (_ bv4195760 64)

0x4005b0: jz 0x4005b9
         -> #80 = (ite (= #78 (_ bv1 1)) (_ bv4195769 64) (_ bv4195762 64))
```

Play with the Taint engine at runtime

Code 8: Taint memory at runtime

```
# 0x40058b: movzx eax, byte ptr [rax]
def cbeforeSymProc(instruction):
    if instruction.address == 0x40058b:
        rax = getRegValue(IDREF.REG.RAX)
        taintMem(rax)

if __name__ == '__main__':
    startAnalysisFromSymbol('check')
    addCallback(cbeforeSymProc, IDREF.CALLBACK.BEFORE_SYMPROC)
    runProgram()
```

Modifications must be done before the symbolic processing

Code 8 result

```
0x40058b: movzx eax, byte ptr [rax]
         ->  #33 = SymVar_0
         ->  #34 = (_ bv4195726 64)

0x40058e: movsx eax, al
         ->  #35 = ((_ sign_extend 24) ((_ extract 7 0) #33))
         ->  #36 = (_ bv4195729 64)
```

Taint argv[x][x] at the main function

Code 9: Taint all arguments when the main function occurs

```
def mainAnalysis(threadId):

    rdi = getRegValue(IDREF.REG.RDI) # argc
    rsi = getRegValue(IDREF.REG.RSI) # argv

    while rdi != 0:
        argv = getMemValue(rsi + ((rdi-1) * 8), 8)
        offset = 0
        while getMemValue(argv + offset, 1) != 0x00:
            taintMem(argv + offset)
            offset += 1
        print '[+] %03d bytes tainted from the argv[%d] (%#x) pointer'
          %(offset, rdi-1, argv)
        rdi -= 1

    return
```

Code 9 result

```
$ pin -t ./triton.so -script taint_main.py -- ./example.bin64 12 123456 123456789
[+] 009 bytes tainted from the argv[3] (0x7fff802ad116) pointer
[+] 006 bytes tainted from the argv[2] (0x7fff802ad10f) pointer
[+] 002 bytes tainted from the argv[1] (0x7fff802ad10c) pointer
[+] 015 bytes tainted from the argv[0] (0x7fff802ad0ef) pointer
```

Play with the Symbolic engine

Example 10: Assembly code

We know that rax points on a tainted area

```
0x40058b: movzx eax, byte ptr [rax]
...
...
0x4005ae: cmp ecx, eax
```

Symbolic Variable

Code 10: Backtrack symbolic expression

```
def callback_beforeSymProc(instruction):
    if instruction.address == 0x40058b:
        rax = getRegValue(IDREF.REG.RAX)
        taintMem(rax)

def callback_after(instruction):
    if instruction.address == 0x4005ae:
        # Get the symbolic expression ID of ZF
        zfId = getRegSymbolicID(IDREF.FLAG.ZF)
        # Backtrack the symbolic expression ZF
        zfExpr = getBacktrackedSymExpr(zfId)
        # Craft a new expression over the ZF expression : (assert (= zfExpr True))
        expr = smt2lib.smtAssert(smt2lib.equal(zfExpr, smt2lib.bvtrue()))
        print expr
```

Example 10 result

```
(assert (= (ite (= (bvsub ((_ extract 31 0) ((_ extract 31 0) (bvxor ((_ extract 31 0)
(bvsub ((_ extract 31 0) ((_ sign_extend 24) ((_ extract 7 0) SymVar_0))) (_ bv1 32)))
(_ bv85 32)))) ((_ extract 31 0) ((_ sign_extend 24) ((_ extract 7 0) ((_ zero_extend 24)
(_ bv49 8)))))) (_ bv0 32)) (_ bv1 1) (_ bv0 1)) (_ bv1 1)))
```

Play with the Symbolic engine

Extract of the Code 10

```
...
zfExpr = getBacktrackedSymExpr(zfId)

# Craft a new expression over the ZF expression : (assert (= zfExpr True))
expr = smt2lib.smtAssert(smt2lib.equal(zfExpr, smt2lib.bvtrue()))
...
```

- What does it really mean?
  - Triton builds symbolic formulas based on the instructions semantics
  - Triton also exports smt2lib functions which allows you to create your own formula
  - In this example, we want that the ZF expression is equal to 1

Play with the Solver engine

- getModel() returns a dictionary of valid model for each symbolic variable

Extract of the Code 10

```
...
zfExpr = getBacktrackedSymExpr(zfId)

# Craft a new expression over the ZF expression : (assert (= zfExpr True))
expr = smt2lib.smtAssert(smt2lib.equal(zfExpr, smt2lib.bvtrue()))
...
model = getModel(expr)
print model
```

Result

```
{'SymVar_0': 0x65}
```

We know now that the first character must be 0x65 to set the ZF at the compare instruction

Example 10: Assembly code

```
0x40058b: movzx eax, byte ptr [rax]
...
...
0x4005ae: cmp ecx, eax
```

Play with the Solver engine and inject values directly in memory

- Each symbolic variable is assigned to a memory address (SymVar ↔ Address)
  - Possible to get the symbolic variable from a memory address
    - getSymVarFromMemory(addr)
  - Possible to get the memory address from a symbolic variable
    - getMemoryFromSymVar(symVar)

Extract of the Code 10

```
...
model = getModel(expr)
print model
```

Result

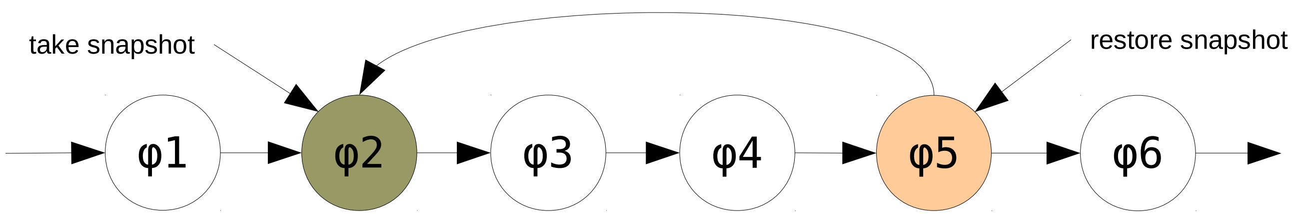
```
{'SymVar_0': 0x65}
```

Inject values given by the solver in memory

```
for k, v in model.items():
     setMemValue(getMemoryFromSymVar(k), getSymVarSize(k), v)
```

Inject values in memory is not enough
Play with the snapshot engine

- Inject values in memory after instructions processing is useless
- That's why Triton offers a snapshot engine



0x40058b: movzx eax, byte ptr [rax]

0x4005ae: cmp ecx, eax

```
def callback_after(instruction):
    if instruction.address == 0x40058b and isSnapshotEnable() == False:
        takeSnapshot()

    if instruction.address == 0x4005ae:
        if getFlagValue(IDREF.FLAG.ZF) == 0:
            zfExpr = getBacktrackedSymExpr(...)    # Described on slide 45
            expr = smt2lib.smtAssert(...zfExpr...) # Described on slide 45
            for k, v in getModel(expr).items():    # Described on slide 48
                setMemValue(...)                   # Described on slide 48
            restoreSnapshot()
```

Stop pasting fucking code
Show me a global vision

Stop pasting fucking code
Show me a global vision

- Full API and Python bindings describes here
  - https://github.com/JonathanSalwan/Triton/wiki/Python-Bindings
  - ~80 functions exported over the Python bindings
- Basically we can:
  - Taint and untaint memory and registers
  - Inject value in memory and registers
  - Add callbacks at each program point, syscalls, routine
  - Assign symbolic expression on registers and bytes of memory
  - Build and customize symbolic expressions
  - Solve symbolic expressions
  - Take and restore snapshots
  - Do all this in Python!

Conclusion

Conclusion

- Triton:
  - is a Pintool which provides others classes for DBA
  - is designed as a concolic execution framework
  - provides an API and Python bindings
  - supports only x86-64 binaries
  - currently supports ~100 semantics but we are working hard on it to increase the semantics support
    - An awesome thanks to Kevin `wisk` Szkudlapski and Francis `gg` Gabriel for the x86.yaml from the Medusa project :)
  - is free and open-source :)
  - is available here : github.com/JonathanSalwan/Triton

Thanks For Your Attention Question(s)?

- Contacts
  - fsaudel@gmail.com
  - jsalwan@quarkslab.com

- Thanks
  - We would like to thank the SSTIC's staff and especially Rolf Rolles, Sean Heelan, Sébastien Bardin, Fred Raynal and Serge Guelton for their proofreading and awesome feedbacks! Then, a big thanks to Quarkslab for the sponsor.

Q&A - Performances

- Host machine configuration
  - Tested with an Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
  - 16 Go DDR3
  - 415 Go SSD Swap

- The targeted binary analyzed was /usr/bin/z3
  - 6,789,610 symbolic expressions created for 1 trace
  - The binary has been analyzed in 180 seconds
    - One trace with SMT2-LIB translation and the taint spread
  - 19 Go of RAM consumed
    - Due to the SMT2-LIB strings manipulation