Triton and Symbolic execution on GDB

bananaappletw @ DEF CON China

2018/05/11

$whoami

- Wei-Bo Chen(@bananaappletw)
- MS major in CSE, Chiao Tung University, Hsinchu
- Organizations:
  - Software Quality Laboratory
  - Co-founder of NCTUCSC
  - Bamboofox member
- Specialize in:
  - symbolic execution
  - binary exploitation
- Talks:
  - HITCON CMT 2015
    HITCON CMT 2017

Outline

- Why symbolic execution?
- What is symbolic execution?
- Triton
- SymGDB
- Conclusion
- Drawbacks of Triton
- Comparison between other symbolic execution framework

Why symbolic execution?

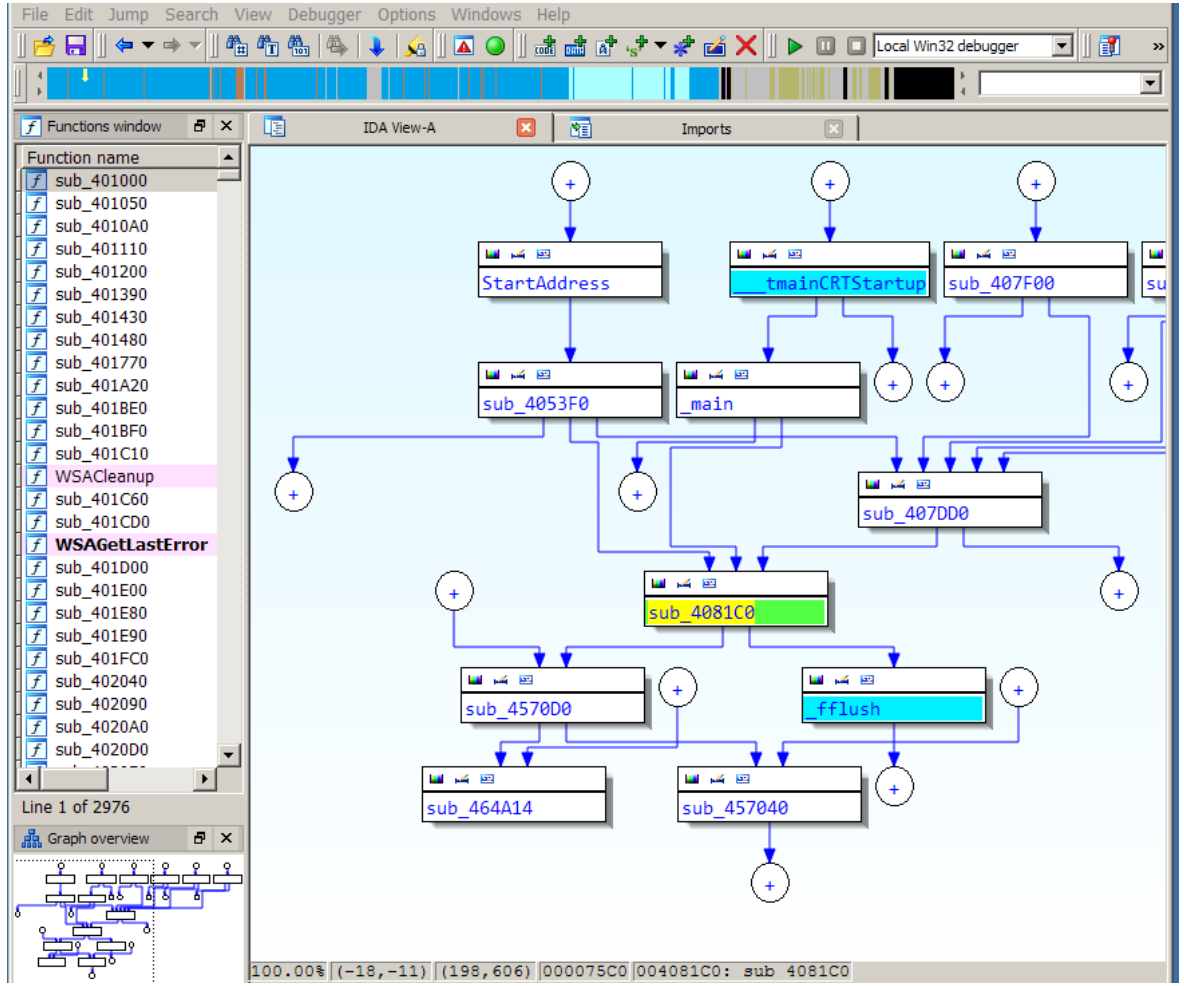
In the old days

- Static analysis
- Dynamic analysis

Static analysis

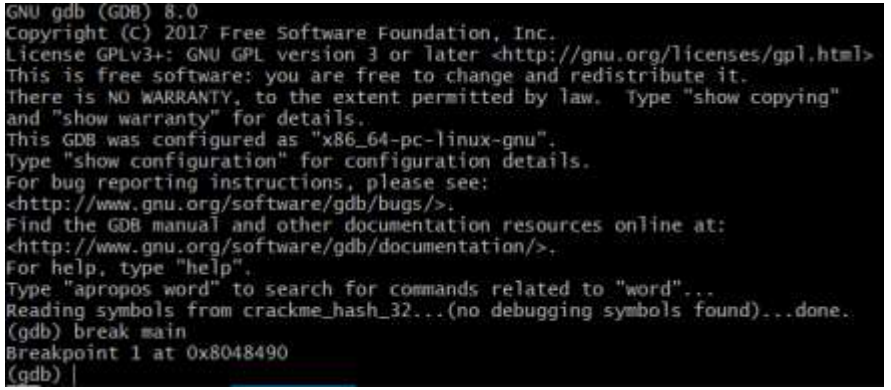
- objdump
- IDA PRO

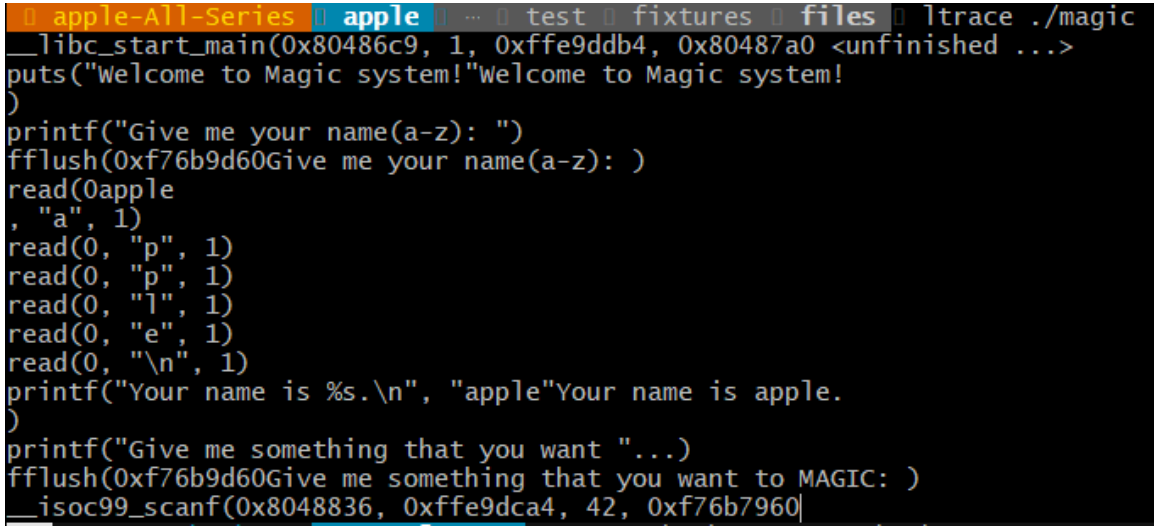
```
08048482 <main>:
 8048482:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048486:       83 e4 f0                and    $0xfffffff0,%esp
 8048489:       ff 71 fc                pushl  -0x4(%ecx)
 804848c:       55                      push   %ebp
 804848d:       89 e5                   mov    %esp,%ebp
 804848f:       51                      push   %ecx
 8048490:       83 ec 14                sub    $0x14,%esp
 8048493:       89 c8                   mov    %ecx,%eax
 8048495:       83 38 02                cmpl   $0x2,(%eax)
 8048498:       74 07                   je     80484a1 <main+0x1f>
 804849a:       b8 ff ff ff ff          mov    $0xffffffff,%eax
 804849f:       eb 44                   jmp    80484e5 <main+0x63>
 80484a1:       8b 40 04                mov    0x4(%eax),%eax
 80484a4:       83 c0 04                add    $0x4,%eax
 80484a7:       8b 00                   mov    (%eax),%eax
 80484a9:       50                      push   %eax
 80484aa:       e8 5c ff ff ff          call   804840b <_Z5checkPc>
 80484af:       83 c4 04                add    $0x4,%esp
 80484b2:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484b5:       81 7d f4 6d ad 00 00    cmpl   $0xad6d,-0xc(%ebp)
 80484bc:       75 12                   jne    80484d0 <main+0x4e>
 80484be:       83 ec 0c                sub    $0xc,%esp
 80484c1:       68 76 85 04 08          push   $0x8048576
 80484c6:       e8 15 fe ff ff          call   80482e0 <puts@plt>
 80484cb:       83 c4 10                add    $0x10,%esp
 80484ce:       eb 10                   jmp    80484e0 <main+0x5e>
```

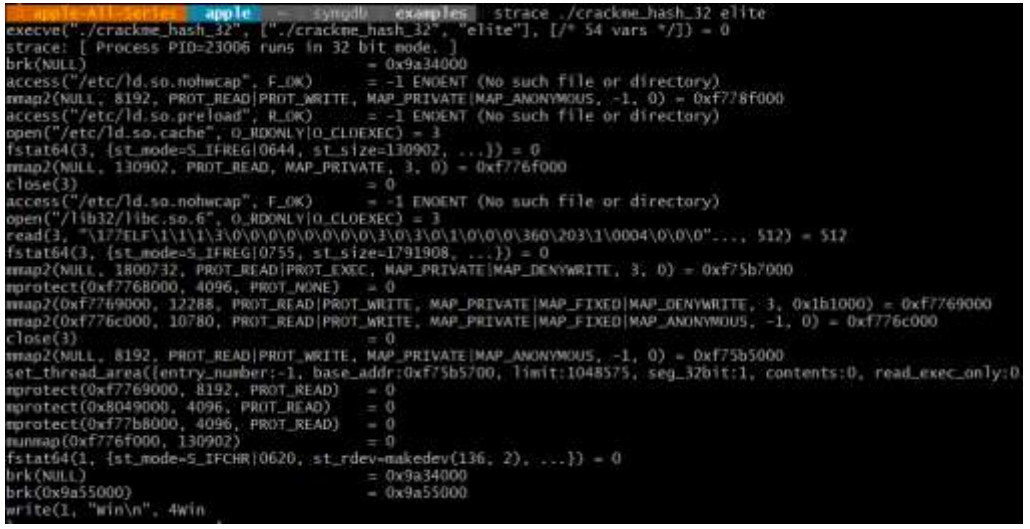


Dynamic analysis

- GDB
- ltrace
- strace







My brain is going to explode

Symbolic execution!!!

```
[+] Asking for a model, please wait...
[+] Symbolic variable 00 = 43 (C)
[+] Symbolic variable 01 = 6f (o)
[+] Symbolic variable 02 = 64 (d)
[+] Symbolic variable 03 = 65 (e)
[+] Symbolic variable 04 = 5f (_)
[+] Symbolic variable 05 = 54 (T)
[+] Symbolic variable 06 = 61 (a)
[+] Symbolic variable 07 = 6c (l)
[+] Symbolic variable 08 = 6b (k)
[+] Symbolic variable 09 = 65 (e)
[+] Symbolic variable 10 = 72 (r)
[+] Symbolic variable 11 = 73 (s)
0x40078e: je 0x400797
0x400797: add dword ptr [rbp - 0x24], 1
0x40079b: cmp dword ptr [rbp - 0x24], 0xb
0x40079f: jle 0x40072d
0x4007a1: mov eax, 0
0x4007a6: pop rbp
0x4007a7: ret
[+] Emulation done.
```

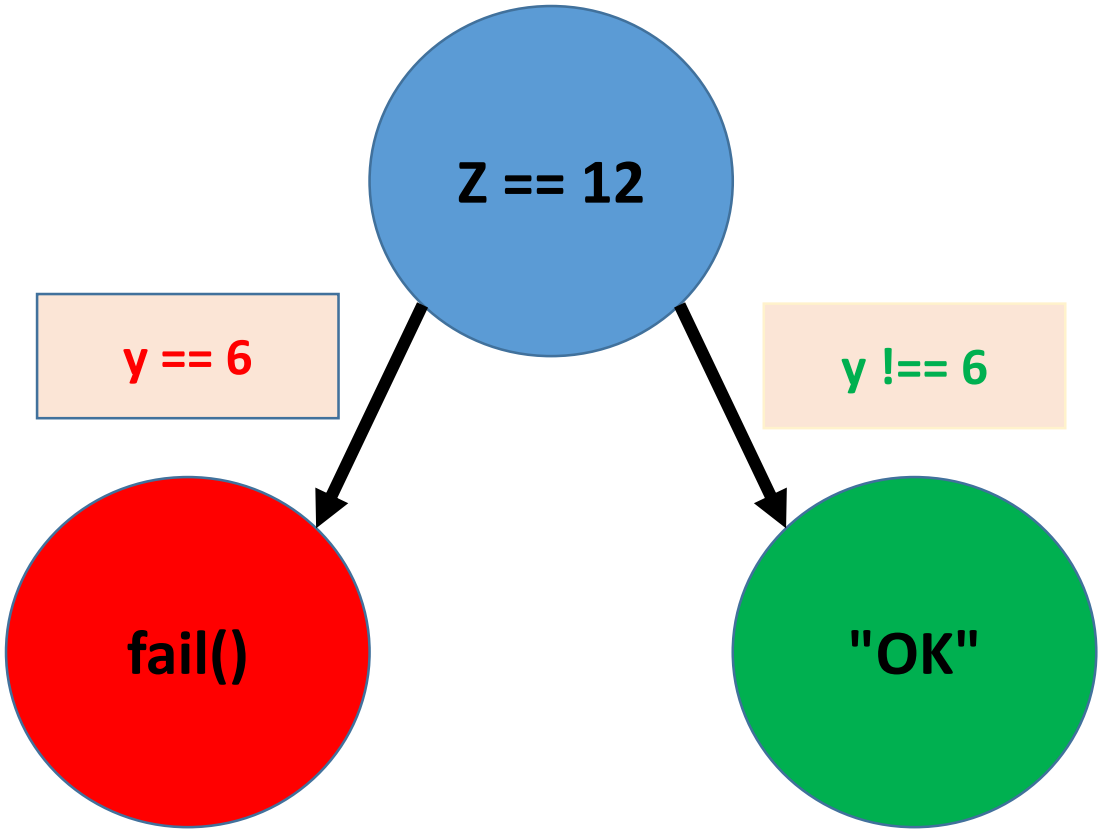
What is symbolic execution?

Symbolic execution

- Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute.
- System-level
  - S2e(https://github.com/dslab-epfl/s2e)
- User-level
  - Angr(http://angr.io/)
  - Triton(https://triton.quarkslab.com/)
- Code-based
  - klee(http://klee.github.io/)

Symbolic execution

```
 1 int f() {
 2    ...
 3    y = read();
 4    z = y * 2;
 5    if (z == 12) {
 6       fail();
 7    } else {
 8       printf("OK");
 9    }
10 }
```



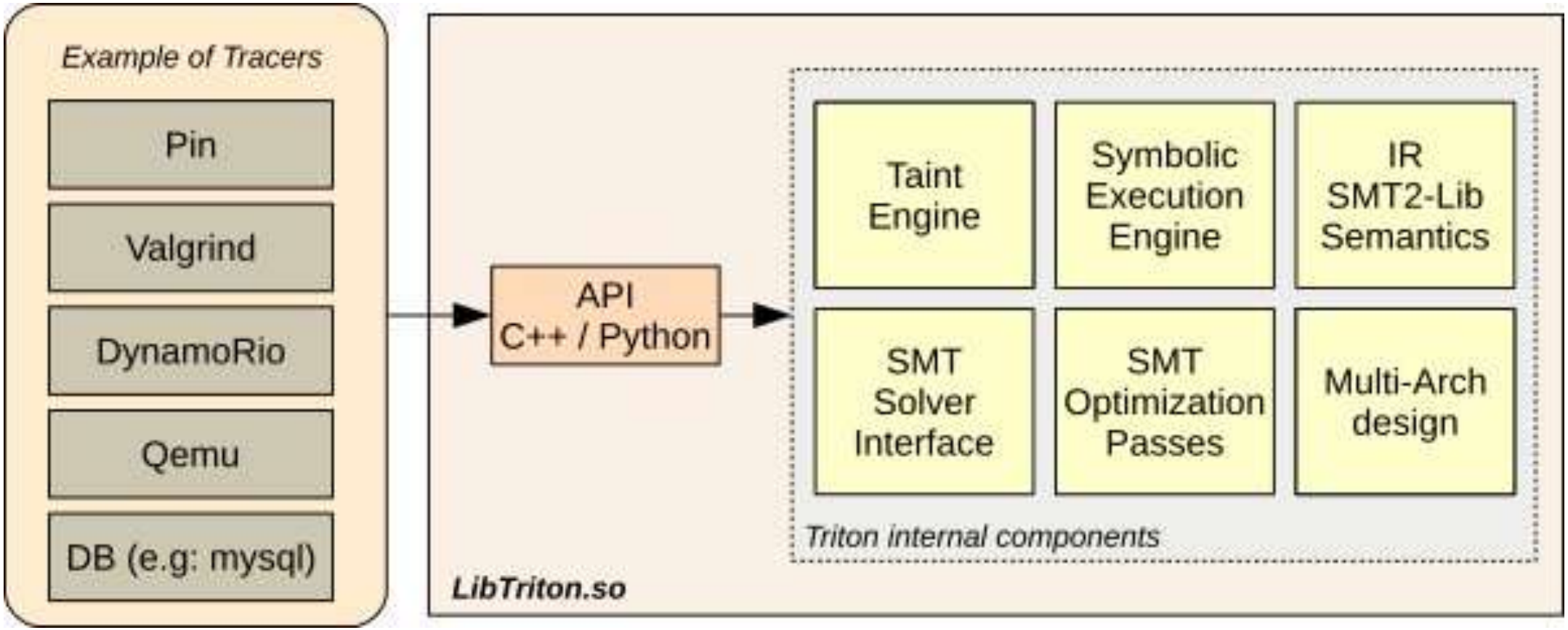
Triton

- Website: https://triton.quarkslab.com/
- A dynamic binary analysis framework written in C++.
  - developed by Jonathan Salwan
- Python bindings
- Triton components:
  - Symbolic execution engine
  - Tracer
  - AST representations
  - SMT solver Interface

Triton

- Structure
- Symbolic execution engine
- Triton Tracer
- AST representations
- Static single assignment form(SSA form)
- Symbolic variables
- SMT solver Interface
- Example

Structure


Example of Tracers
Pin
Valgrind
DynamoRio
Qemu
DB (e.g: mysql)
API
C++ / Python
Taint Engine
Symbolic Execution Engine
IR SMT2-Lib Semantics
SMT Solver Interface
SMT Optimization Passes
Multi-Arch design
Triton internal components
LibTriton.so

Symbolic execution engine

- The symbolic engine maintains:
  - a table of symbolic registers states
  - a map of symbolic memory states
  - a global set of all symbolic references

Step	Register	Instruction	Set of symbolic expressions
init	eax = UNSET	None	$\perp$
1	eax = ϕ1	mov eax, 0	{ϕ1=0}
2	eax = ϕ2	inc eax	{ϕ1=0,ϕ2=ϕ1+1}
3	eax = ϕ3	add eax, 5	{ϕ1=0,ϕ2=ϕ1+1,ϕ3=ϕ2+5}

Triton Tracer

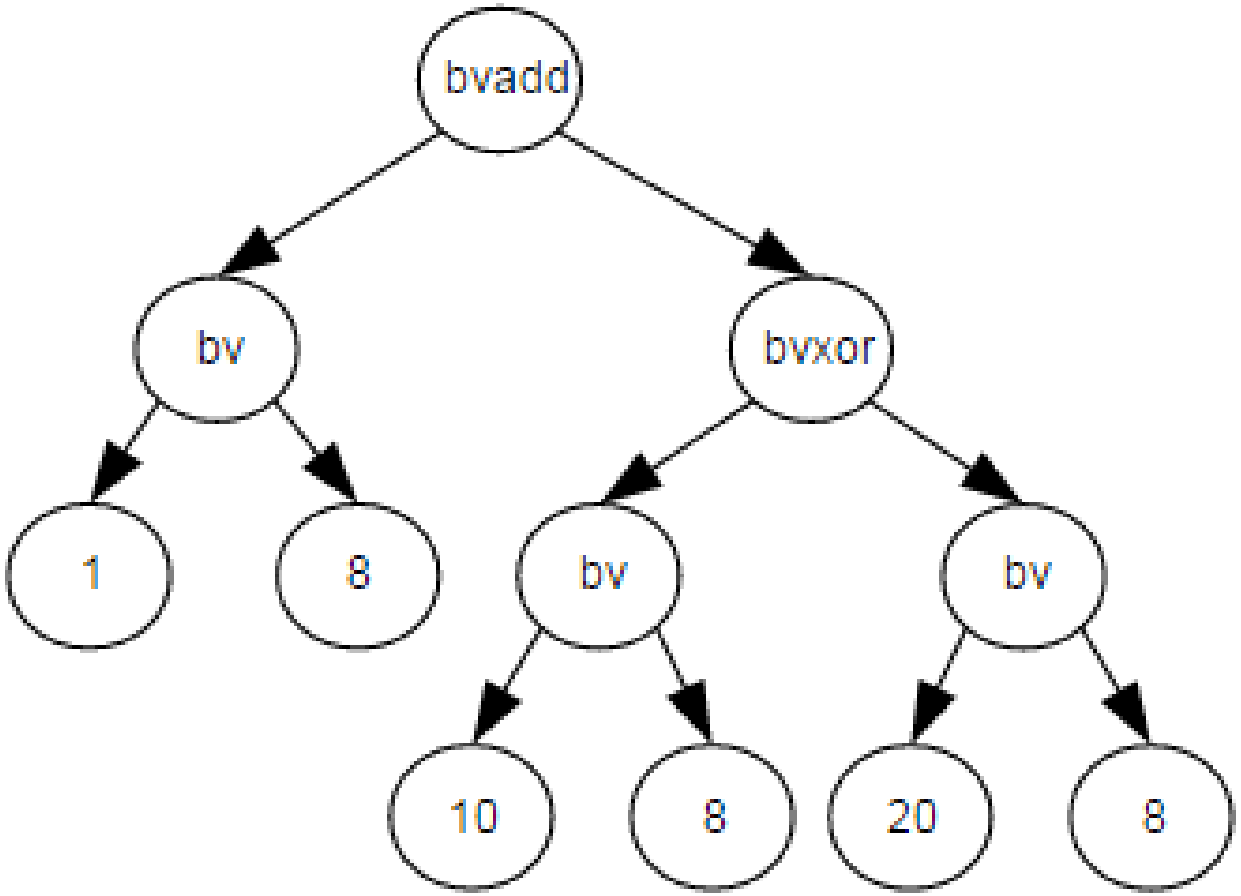
- Tracer provides:
  - Current opcode executed
  - State context (register and memory)
- Translate the control flow into **AST Representations**
- Pin tracer support

AST representations

- Triton converts the x86 and the x86-64 instruction set semantics into AST representations.
- Triton's expressions are on **SSA form**.
- Instruction: add rax, rdx
- Expression: **ref!41** = (bvadd ((_ extract 63 0) **ref!40**) ((_ extract 63 0) **ref!39**))
- **ref!41** is the new expression of the RAX register.
- **ref!40** is the previous expression of the RAX register.
- **ref!39** is the previous expression of the RDX register.

AST representations

- mov al, 1
- mov cl, 10
- mov dl, 20
- xor cl, dl
- add al, cl



Static single assignment form(SSA form)

Each variable is assigned exactly **once**

- y := 1
- y := 2
- x := y

Turns into

- y1 := 1
- y2 := 2
- x1 := y2

Static single assignment form(SSA form)

~~y1 := 1~~ (This assignment is not necessary)

y2 := 2

x1 := y2

- When Triton process instructions, it could ignore some unnecessary instructions.

Symbolic variables

- Imagine **symbolic** as a infection. If one of the operand of a instruction is symbolic, the register or memory which the instruction infect will be symbolic.
- In Triton, we could use the following method to manipulate it.
  - convertRegisterToSymbolicVariable(const triton::arch::Register &reg)
  - isRegisterSymbolized(const triton::arch::Register &reg)

Symbolic variables

1. Make **ecx** as symbolic variable

- convertRegisterToSymbolicVariable(Triton.registers.ecx)
- isRegisterSymbolized(Triton.registers.ecx) == True

Symbolic variables

1. Make **ecx** as symbolic variable
2. test **ecx**, **ecx**

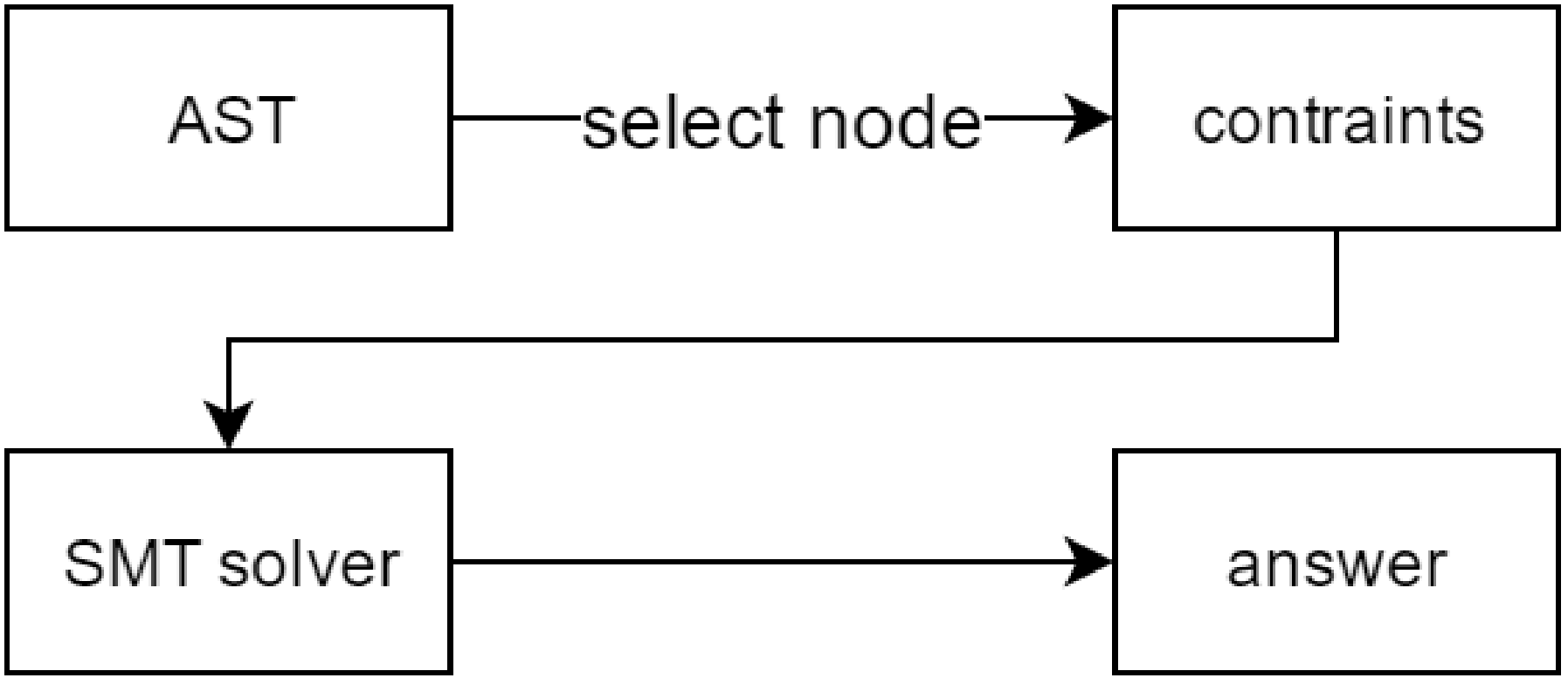
- ZF = AND(ecx, ecx) == 0
- If ecx == 0:
  - Set ZF to 1
- Else:
  - Set ZF to 0

Symbolic variables

1. Make **ecx** as symbolic variable
2. test **ecx**, **ecx**
3. je +7 (**eip**)
4. mov edx, 0x64
5. nop

- If ZF == 1:
  - Jump to nop
- Else:
  - Execute next instruction
- isRegisterSymbolized(Triton.registers.eip) == True

SMT solver Interface



Example

- Defcamp 2015 r100
- Program require to input the password
- Password length could up to 255 characters
- Then do the serial operations to check password is correct

Defcamp 2015 r100

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax@3
  __int64 v4; // rcx@6
  char s; // [sp+0h] [bp-110h]@1
  __int64 v6; // [sp+108h] [bp-8h]@1

  v6 = *MK_FP(__FS__, 40LL);
  printf("Enter the password: ", argv, envp);
  if ( fgets(&s, 255, stdin) )
  {
    if ( (unsigned int)sub_4006FD((__int64)&s) )
    {
      puts("Incorrect password!");
      result = 1;
    }
    else
    {
      puts("Nice!");
      result = 0;
    }
  }
  else
  {
    result = 0;
  }
  v4 = *MK_FP(__FS__, 40LL) ^ v6;
  return result;
}
```

Defcamp 2015 r100

```
signed __int64 __fastcall sub_4006FD(char *a1)
{
  signed int i; // [sp+14h] [bp-24h]@1
  char v3[8]; // [sp+18h] [bp-20h]@1
  char v4[8]; // [sp+20h] [bp-18h]@1
  char v5[8]; // [sp+28h] [bp-10h]@1

  *(_QWORD *)v3 = "Dufhbmf";
  *(_QWORD *)v4 = "pG`imos";
  *(_QWORD *)v5 = "ewUglpt";
  for ( i = 0; i <= 11; ++i )
  {
    if ( *(_BYTE *)(*(_QWORD *)&v3[8 * (i % 3)] + 2 * (i / 3)) - a1[i] != 1 )
      return 1LL;
  }
  return 0LL;
}
```

Defcamp 2015 r100

- Import Triton and initialize Triton context
- Set Architecture
- Load segments into triton
- Define fake stack (RBP and RSP)
- Symbolize user input
- Start to processing opcodes
- Set constraint on specific point of program
- Get symbolic expression and solve it
- Answer

Import Triton and initialize Triton context

```
1  from triton import ARCH, TritonContext, Instruction, MODE, MemoryAccess, CPUSIZE
2
3  Triton = TritonContext()
```

Set Architecture

```
1    setArchitecture(ARCH.X86_64)
```

Load segments into triton

```
1   def loadBinary(path):
2       import lief
3       binary = lief.parse(path)
4       phdrs  = binary.segments
5       for phdr in phdrs:
6           size   = phdr.physical_size
7           vaddr  = phdr.virtual_address
8           print '[+] Loading 0x%06x - 0x%06x' %(vaddr, vaddr+size)
9           Triton.setConcreteMemoryAreaValue(vaddr, phdr.content)
10      return
```

Define fake stack (RBP and RSP)

```
1   # Define a fake stack
2   Triton.setConcreteRegisterValue(Triton.registers.rbp, 0x7fffffff)
3   Triton.setConcreteRegisterValue(Triton.registers.rsp, 0x6fffffff)
```

Symbolize user input

```
1  # Define an user input
2  Triton.setConcreteRegisterValue(Triton.registers.rdi, 0x10000000)
3
4  # Symbolize user inputs (30 bytes)
5  for index in range(30):
6      Triton.convertMemoryToSymbolicVariable(MemoryAccess(0x10000000+index, CPUSIZE.BYTE))
```

Start to processing opcodes

```
1   emulate(0x4006FD)
2   def emulate(pc):
3       while pc:
4           # Fetch opcode
5           opcode = Triton.getConcreteMemoryAreaValue(pc, 16)
6
7           # Create the Triton instruction
8           instruction = Instruction()
9           instruction.setOpcode(opcode)
10          instruction.setAddress(pc)
11
12          # Process
13          Triton.processing(instruction)
14
15          # Next
16          pc = Triton.getConcreteRegisterValue(Triton.registers.rip)
```

Get symbolic expression and solve it

```
# 40078B: cmp eax, 1
# eax must be equal to 1 at each round.
if instruction.getAddress() == 0x40078B:
    # Slice expressions
    rax = Triton.getSymbolicExpressionFromId(Triton.getSymbolicRegisterId(Triton.registers.rax))
    eax = astCtxt.extract(31, 0, rax.getAst())

    # Define constraint
    cstr = astCtxt.land([
                Triton.getPathConstraintsAst(),
                astCtxt.equal(eax, astCtxt.bv(1, 32))
            ])

    print '[+] Asking for a model, please wait...'
    model = Triton.getModel(cstr)
    for k, v in model.items():
        value = v.getValue()
        Triton.setConcreteSymbolicVariableValue(Triton.getSymbolicVariableFromId(k), value)
        print '[+] Symbolic variable %02d = %02x (%c)' %(k, value, chr(value))
```

Answer

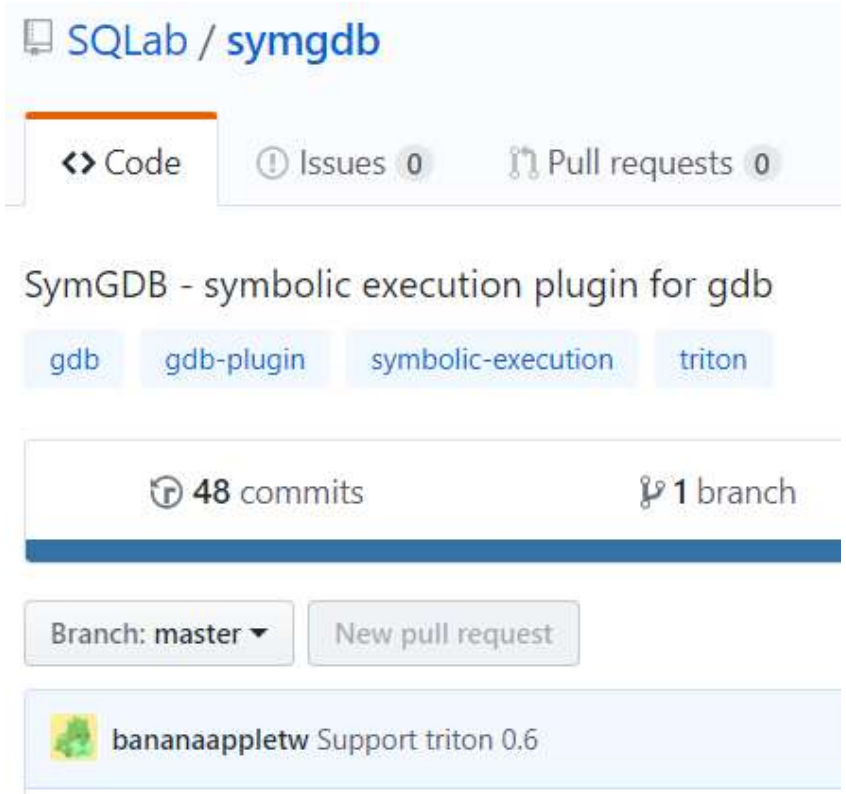
```
[+] Asking for a model, please wait...
[+] Symbolic variable 00 = 43 (C)
[+] Symbolic variable 01 = 6f (o)
[+] Symbolic variable 02 = 64 (d)
[+] Symbolic variable 03 = 65 (e)
[+] Symbolic variable 04 = 5f (_)
[+] Symbolic variable 05 = 54 (T)
[+] Symbolic variable 06 = 61 (a)
[+] Symbolic variable 07 = 6c (l)
[+] Symbolic variable 08 = 6b (k)
[+] Symbolic variable 09 = 65 (e)
[+] Symbolic variable 10 = 72 (r)
[+] Symbolic variable 11 = 73 (s)
0x40078e: je 0x400797
0x400797: add dword ptr [rbp - 0x24], 1
0x40079b: cmp dword ptr [rbp - 0x24], 0xb
0x40079f: jle 0x40072d
0x4007a1: mov eax, 0
0x4007a6: pop rbp
0x4007a7: ret
[+] Emulation done.
```

Some problems of Triton

- The whole procedure is too complicated.
- High learning cost to use Triton.
- With support of debugger, many steps could be simplified.

SymGDB

- Repo: https://github.com/SQLab/symgdb
- Symbolic execution support for GDB
- Combined with:
  - GDB Python API
  - Triton
- Symbolic environment
  - symbolize argv



Design and Implementation

- GDB Python API
- Failed method
- Successful method
- Flow
- SymGDB System Structure
- Implementation of System Internals
- Relationship between SymGDB classes
- Supported Commands
- Symbolic Execution Process in GDB
- Symbolic Environment
  - symbolic argv
- Debug tips
- Demo

GDB Python API

- API: https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html
- Source python script in .gdbinit
- Functionalities:
  - Register GDB command
  - Register event handler (ex: breakpoint)
  - Execute GDB command and get output
  - Read, write, search memory

Register GDB command

```
1  class Triton(gdb.Command):
2      def __init__(self):
3          super(Triton, self).__init__("triton", gdb.COMMAND_DATA)
4
5      def invoke(self, arg, from_tty):
6          Symbolic().run()
7  Triton()
```

Register event handler

```
1   def breakpoint_handler(event):
2       GdbUtil().reset()
3       Arch().reset()
4
5   gdb.events.stop.connect(breakpoint_handler)
```

Execute GDB command and get output

```
1  def get_stack_start_address(self):
2      out = gdb.execute("info proc all", to_string=True)
3      line = out.splitlines()[-1]
4      pattern = re.compile("(0x[0-9a-f]*)")
5      matches = pattern.findall(line)
6      return int(matches[0], 0)
```

Read memory

```
1   def get_memory(self, address, size):
2       """
3       Get memory content from gdb
4       Args:
5           - address: start address of memory
6           - size: address length
7       Returns:
8           - list of memory content
9       """
10      return map(ord, list(gdb.selected_inferior().read_memory(address, size)))
```

Write memory

```
1  def inject_to_gdb(self):
2      for address, size in self.symbolized_memory:
3          self.log("Memory updated: %s-%s" % (hex(address), hex(address + size)))
4          for index in range(size):
5              memory = chr(TritonContext.getSymbolicMemoryValue(MemoryAccess(address + index, CPUSIZE.BYTE)))
6              gdb.selected_inferior().write_memory(address + index, memory, CPUSIZE.BYTE)
```

Failed method

- At first, I try to use Triton callback to get memory and register values
- Register callbacks:
  - needConcreteMemoryValue(const triton::arch::MemoryAccess& mem)
  - needConcreteRegisterValue(const triton::arch::Register& reg)
- Process the following sequence of code
  - mov eax, 5
  - mov ebx,eax (**Trigger needConcreteRegisterValue**)
- We need to set Triton context of eax

Problems

- Values from GDB are out of date
- Consider the following sequence of code

mov eax, 5

- We set breakpoint here, and call Triton's processing()

mov ebx,eax (trigger callback to get eax value, eax = 5)

mov eax, 10

mov ecx, eax (Trigger again, get eax = 5)

- Because context state not up to date

Triton callbacks

```
1  def needConcreteMemoryValue(TritonContext, mem):
2      mem_addr = mem.getAddress()
3      mem_size = mem.getSize()
4      mem_val = TritonContext.getConcreteMemoryValue(MemoryAccess(mem_addr,mem_size))
5      TritonContext.setConcreteMemoryValue(MemoryAccess(mem_addr,mem_size, mem_val))
6
7
8  def needConcreteRegisterValue(TritonContext, reg):
9      reg_name = reg.getName()
10     reg_val = TritonContext.getConcreteRegisterValue(getattr(TritonContext.registers, reg_name))
11     setConcreteRegisterValue(getattr(TritonContext.registers, reg_name), reg_val))
12
13 TritonContext.addCallback(needConcreteMemoryValue, CALLBACK.GET_CONCRETE_MEMORY_VALUE)
14 TritonContext.addCallback(needConcreteRegisterValue, CALLBACK.GET_CONCRETE_REGISTER_VALUE)
```

Tried solutions

- Before needed value derived from GDB, check if it is not in the Triton's context yet

Not working!

Triton will fall into infinite loop

Successful method

- Copy GDB context into Triton
- Load all the segments into Triton context
- Symbolic execution won't affect original GDB state
- User could restart symbolic execution from breakpoint

Flow

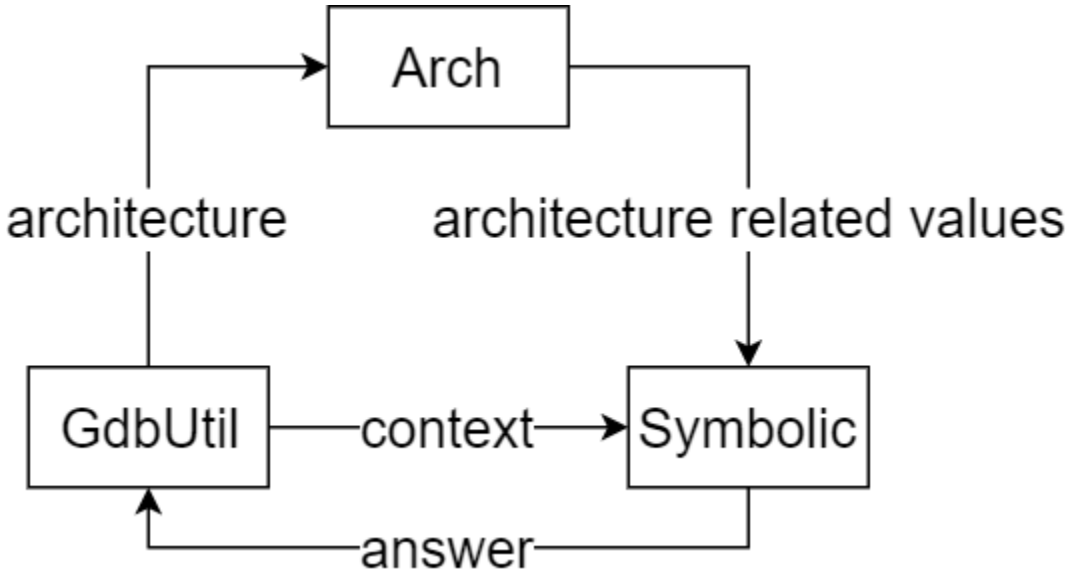
- Get debugged program state by calling GDB Python API
- Get the current program state and yield to triton
- Set symbolic variable
- Set the target address
- Run symbolic execution and get output
- Inject back to debugged program state

SymGDB System Structure

Implementation of System Internals

- Three classes in the symGDB
  - Arch(), GdbUtil(), Symbolic()
- Arch()
  - Provide different pointer size、register name
- GdbUtil()
  - Read write memory、read write register
  - Get memory mapping of program
  - Get filename and detect architecture
  - Get argument list
- Symbolic()
  - Set constraint on pc register
  - Run symbolic execution

Relationship between SymGDB classes



Supported Commands

Command	Option	Functionality
symbolize	argv memory [address][size]	Make symbolic
target	address	Set target address
triton	None	Run symbolic execution
answer	None	Print symbolic variables
debug	symbolic gdb	Show debug messages

Symbolic Execution Process in GDB

- **gdb.execute("info registers", to_string=True)** to get registers
- **gdb.selected_inferior().read_memory(address, length)** to get memory
- **setConcreteMemoryAreaValue** and **setConcreteRegisterValue** to set triton state
- In each instruction, use **isRegisterSymbolized** to check if pc register is symbolized or not
- Set target address as constraint
- Call **getModel** to get answer
- **gdb.selected_inferior().write_memory(address, buf, length)** to inject back to debugged program state

Symbolic Environment: symbolic argv

- Using "info proc all" to get stack start address
- Examining memory content from stack start address
  - argc
  - argv[0]
  - argv[1]
  - ……
  - null
  - env[0]
  - env[1]
  - ……
  - null

argc	argument counter(integer)
argv[0]	program name (pointer)
argv[1]	program args (pointers)
…	
argv[argc-1]	
null	end of args (integer)
env[0]	environment variables (pointers)
env[1]	
…	
env[n]	
null	end of environment (integer)

Debug tips

- Simplify: https://github.com/JonathanSalwan/Triton/blob/master/src/examples/python/simplification.py

```
Expr:  (bvxor (_ bv1 8) (_ bv1 8))
Simp:  (_ bv0 8)

Expr:  (bvor (bvand (_ bv1 8) (bvnot (_ bv2 8))) (bvand (bvnot (_ bv1 8)) (_ bv2 8)))
Simp:  (bvxor (_ bv1 8) (_ bv2 8))

Expr:  (bvor (bvand (bvnot (_ bv2 8)) (_ bv1 8)) (bvand (bvnot (_ bv1 8)) (_ bv2 8)))
Simp:  (bvxor (_ bv1 8) (_ bv2 8))

Expr:  (bvor (bvand (bvnot (_ bv2 8)) (_ bv1 8)) (bvand (_ bv2 8) (bvnot (_ bv1 8))))
Simp:  (bvxor (_ bv1 8) (_ bv2 8))

Expr:  (bvor (bvand (_ bv2 8) (bvnot (_ bv1 8))) (bvand (bvnot (_ bv2 8)) (_ bv1 8)))
Simp:  (bvxor (_ bv2 8) (_ bv1 8))
```

Demo

- Examples
  - crackme hash
  - crackme xor
- GDB commands
- Combined with Peda

crackme hash

- Source: https://github.com/illera88/Ponce/blob/master/examples/crackme_hash.cpp
- Program will pass argv[1] to check function
- In check function, argv[1] xor with serial(fixed string)
- If sum of xored result equals to 0xABCD
  - print "Win"
- else
  - print "fail"

crackme hash

```c
#include <stdio.h>
#include <stdlib.h>
char *serial = "\x31\x3e\x3d\x26\x31";
int check(char *ptr)
{
        int i;
        int hash = 0xABCD;
        for (i = 0; ptr[i]; i++)
                hash += ptr[i] ^ serial[i % 5];
        return hash;
}
int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;
        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

```c
int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;
        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

crackme hash

```
.text:080484A1 loc_80484A1:                            ; CODE XREF: main+16↑j
.text:080484A1                 mov     eax, [eax+4]
.text:080484A4                 add     eax, 4
.text:080484A7                 mov     eax, [eax]
.text:080484A9                 push    eax             ; char *
.text:080484AA                 call    _Z5checkPc      ; check(char *)
.text:080484AF                 add     esp, 4
.text:080484B2                 mov     [ebp+var_C], eax
.text:080484B5                 cmp     [ebp+var_C], 0AD6Dh
.text:080484BC                 jnz     short loc_80484D0
.text:080484BE                 sub     esp, 0Ch
.text:080484C1                 push    offset s        ; "Win"
.text:080484C6                 call    _puts
.text:080484CB                 add     esp, 10h
.text:080484CE                 jmp     short loc_80484E0
```

crackme hash

crackme xor

- Source: https://github.com/illera88/Ponce/blob/master/examples/crackme_xor.cpp
- Program will pass argv[1] to check function
- In check function, argv[1] xor with 0x55
- If xored result not equals to serial(fixed string)
  - return 1
  - print "fail"
- else
  - go to next loop
- If program go through all the loop
  - return 0
  - print "Win"

crackme xor

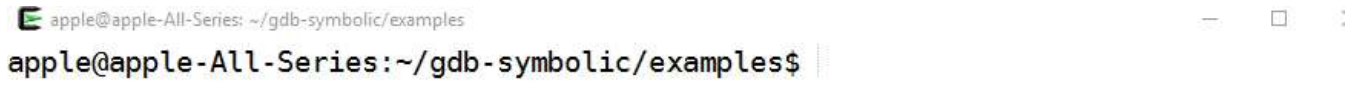
```c
#include <stdio.h>
#include <stdlib.h>
char *serial = "\x31\x3e\x3d\x26\x31";
int check(char *ptr)
{
        int i = 0;
        while (i < 5){
                if (((ptr[i] - 1) ^ 0x55) != serial[i])
                        return 1;
                i++;
        }
        return 0;
}
```

```c
int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;
        ret = check(av[1]);
        if (ret == 0)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

crackme xor

```
.text:08048418 loc_8048418:                            ; CODE XREF: check(char *)+49↓j
.text:08048418                 cmp     [ebp+var_4], 4
.text:0804841C                 jg      short loc_8048456
.text:0804841E                 mov     edx, [ebp+var_4]
.text:08048421                 mov     eax, [ebp+arg_0]
.text:08048424                 add     eax, edx
.text:08048426                 movzx   eax, byte ptr [eax]
.text:08048429                 movsx   eax, al
.text:0804842C                 sub     eax, 1
.text:0804842F                 xor     eax, 55h
.text:08048432                 mov     ecx, eax
.text:08048434                 mov     edx, serial
.text:0804843A                 mov     eax, [ebp+var_4]
.text:0804843D                 add     eax, edx
.text:0804843F                 movzx   eax, byte ptr [eax]
.text:08048442                 movsx   eax, al
.text:08048445                 cmp     ecx, eax
.text:08048447                 jz      short loc_8048450
.text:08048449                 mov     eax, 1
.text:0804844E                 jmp     short locret_804845B
.text:08048450 ; ---------------------------------------------------------------------------
.text:08048450
.text:08048450 loc_8048450:                            ; CODE XREF: check(char *)+3C↑j
.text:08048450                 add     [ebp+var_4], 1
.text:08048454                 jmp     short loc_8048418
```

crackme xor


apple@apple-All-Series: ~/gdb-symbolic/examples
apple@apple-All-Series:~/gdb-symbolic/examples$

GDB commands

```
#!/bin/bash
DIR=$(dirname "$(readlink -f "$0")")
TESTS=(crackme_hash_32 crackme_hash_64 crackme_xor_32 crackme_xor_64)
for program in "${TESTS[@]}"
do
  gdb -x $DIR/$program $DIR/../examples/$program
done
```

```
break main
symbolize argv
target 0x080484be
run aaaaa
triton
continue
```

GDB commands

Combined with Peda

- Same demo video of crackme hash
- Using find(peda command) to find argv[1] address
- Using symbolize memory argv[1]_address argv[1]_length to symbolic argv[1] memory

Combined with Peda

Conclusion

- Using GDB as the debugger to provide the information. Save you the endeavor to do the essential things.
- SymGDB plugin is independent from the debugged program except if you inject answer back to it.
- With the tracer support(i.e. GDB), we could have the concolic execution.

Concolic Execution

- **Concolic** = **Conc**rete + Symb**olic**
- Using both symbolic variables and concrete values
- It is **fast**. Compare to Full Emulation, we don’t need to evaluate memory or register state from SMT formula, directly derived from real CPU context.

Drawbacks of Triton

- Triton doesn't support GNU c library
- Why?
- SMT Semantics Supported: https://triton.quarkslab.com/documentation/doxygen/SMT_Semantics_Supported_page.html
- Triton has to implement system call interface to support GNU c library a.k.a. support "int 0x80"
- You have to do state traversal manually.

Comparison between other symbolic execution framework

- KLEE
- Angr

KLEE

- Symbolic virtual machine built on top of the LLVM compiler infrastructure
- Website: http://klee.github.io/
- Github: https://github.com/klee/klee
- KLEE paper: http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf (Worth reading)
- Main goal of KLEE:
  1. Hit every line of executable code in the program
  2. Detect at each dangerous operation

Introduction

- KLEE is a symbolic machine to generate test cases.
- In order to compiled to LLVM bitcode, source code is needed.
- Steps:
  - Replace input with KLEE function to make memory region symbolic
  - Compile source code to LLVM bitcode
  - Run KLEE
  - Get the test cases and path's information

get_sign.c

```c
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

get_sign.ll

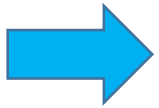
```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  store i32 0, i32* %1
  call void @llvm.dbg.declare(metadata !{i32*
%a}, metadata !25), !dbg !26
  %2 = bitcast i32* %a to i8*, !dbg !27
  call void @klee_make_symbolic(i8* %2, i64 4,
i8* getelementptr inbounds ([2 x i8]* @.str,
i32 0, i32 0)), !dbg !27
  %3 = load i32* %a, align 4, !dbg !28
  %4 = call i32 @get_sign(i32 %3), !dbg !28
  ret i32 %4, !dbg !28
}
```

Result

```
klee@561b436ff126:~/klee_src/examples/get_sign$ klee get_sign.bc
KLEE: output directory is "/home/klee/klee_src/examples/get_sign/klee-out-3"
KLEE: Using STP solver backend

KLEE: done: total instructions = 31
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
klee@561b436ff126:~/klee_src/examples/get_sign$ ktest-tool ./klee-last/*.ktest
ktest file : './klee-last/test000001.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: b'\x00\x00\x00\x00'

ktest file : './klee-last/test000002.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: b'\x01\x01\x01\x01'

ktest file : './klee-last/test000003.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: b'\x00\x00\x00\x80'
```

Diagram

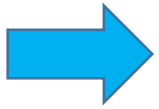
1. Step the program until it meets the branch



```c
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

Diagram

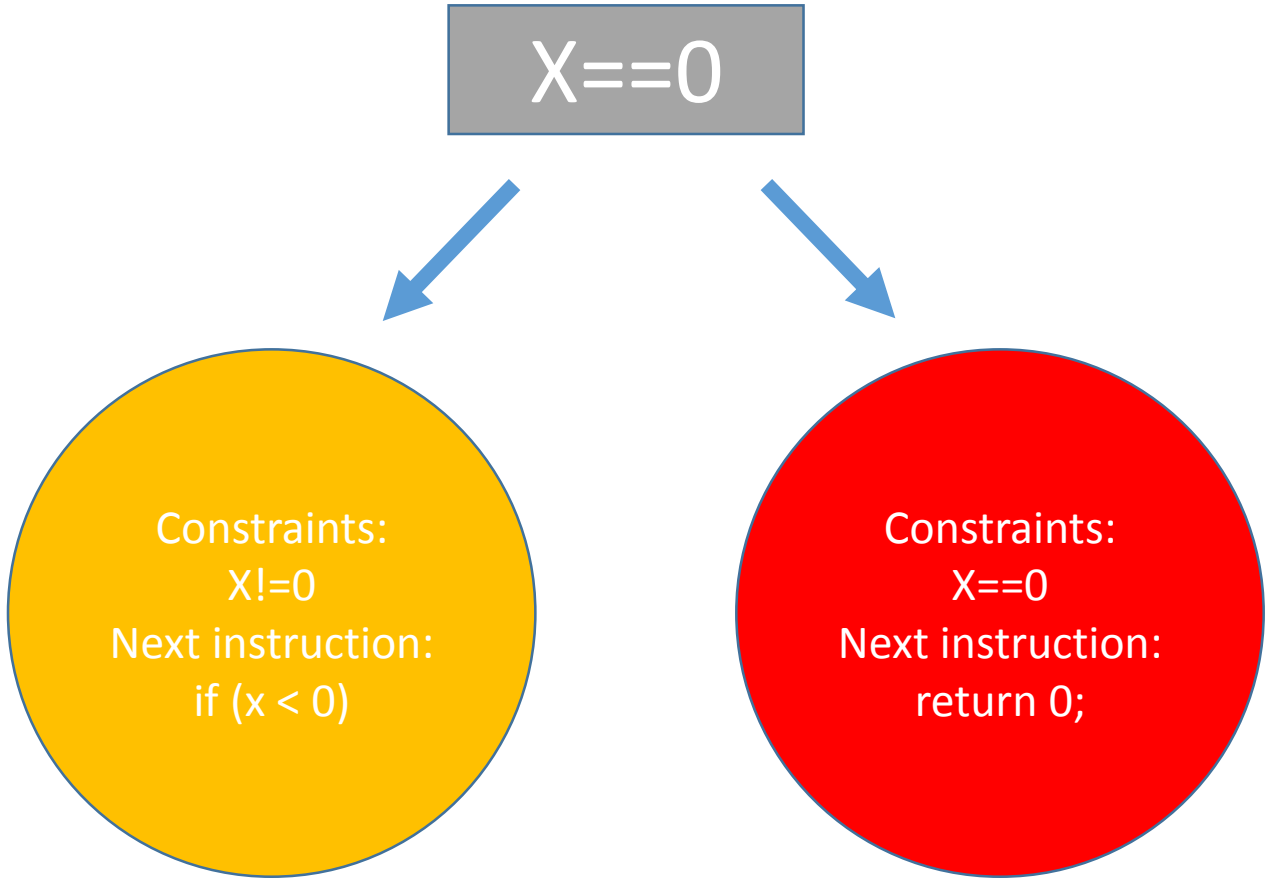
1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state.



```c
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

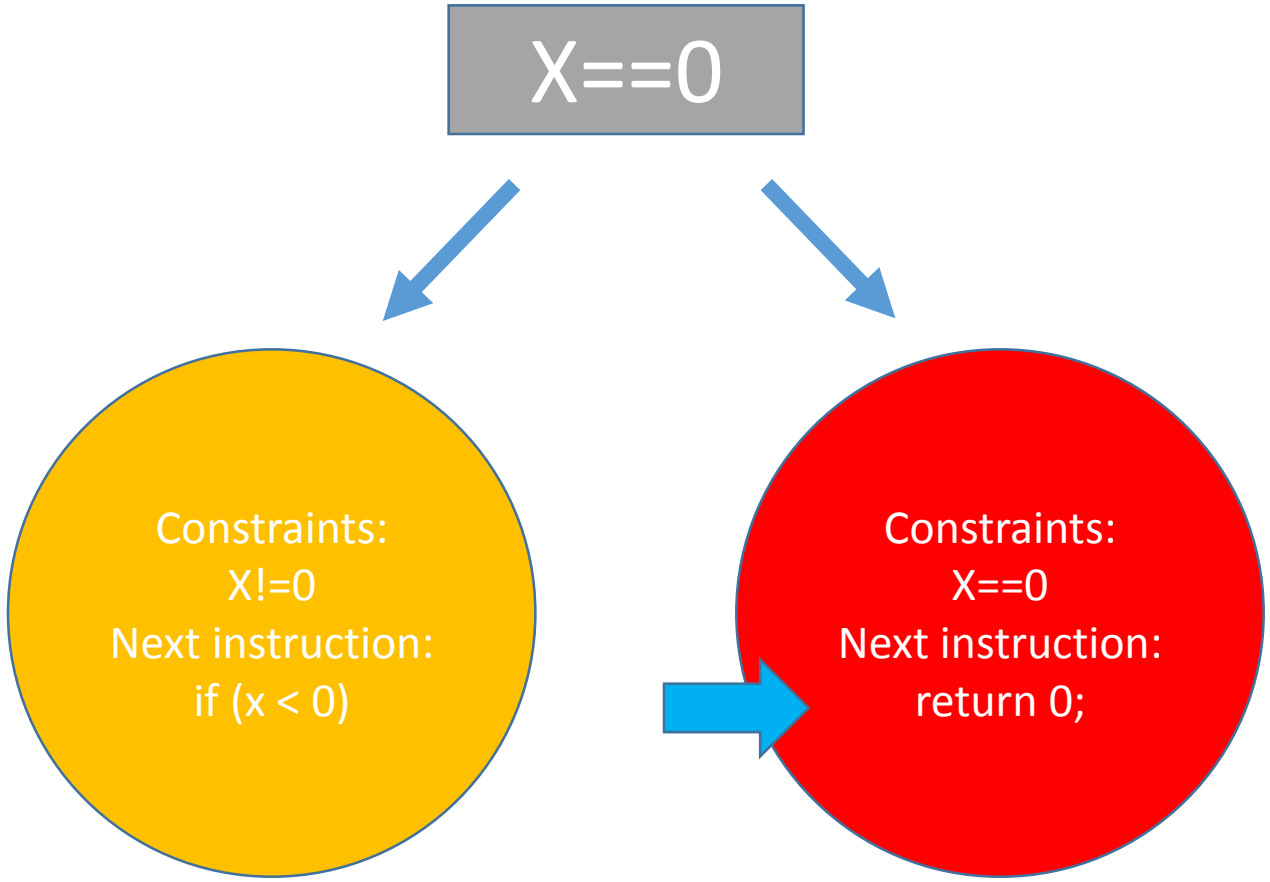
Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state
3. Step the states until they hit exit call or error



Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state
3. Step the states until they hit exit call or error
4. Solve the conditional constraint



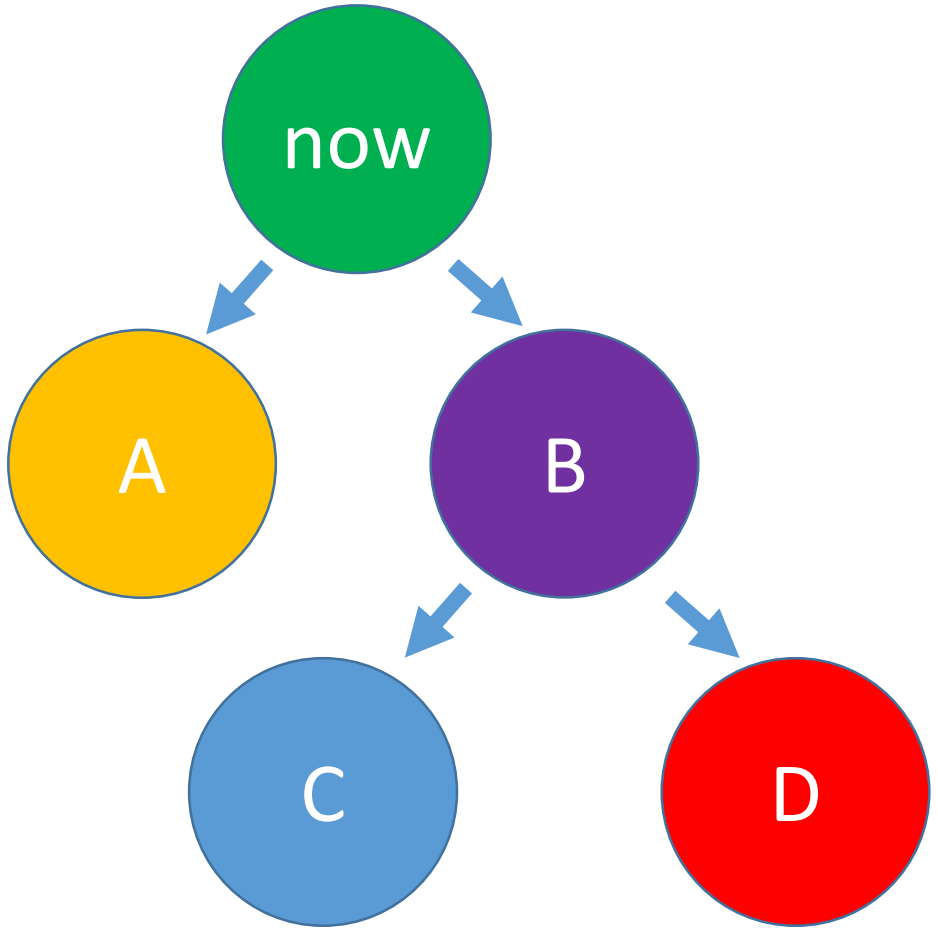
Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state
3. Step the states until they hit exit call or error
4. Solve the conditional constraint
5. Loop until no remaining states or user-defined timeout is reached

What's the difference in KLEE

- Introduce to the concept of state, the deeper path could be reached by stepping the state tree.
- Seems like support GNU c library?

What's the difference in KLEE



- Current state is **now**, our final goal is to reach path **D**.
- In Triton
  - solve the symbolic variable to path **B**
  - Set the concrete value and step to path **B**
  - Solve the symbolic variable to path **D**
- In KLEE
  - Record condition constraints to path **B**
  - Clone the state
  - Solve the symbolic variable to path **D**

What's the difference in KLEE

- When KLEE need to deal with GNU c library, run KLEE with --libc=uclibc --posix-runtime parameters.
- When KLEE detect the analyzed program make the external call to the library, which isn't compiled to LLVM IR instead linked with the program together.
- The library call is only done concretely, which means loosing symbolic information within the library call.

Angr

- Website: http://angr.io/
- Angr is a python framework for analyzing binaries. It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks.
- Support various architectures
- Flow
  - Loading a binary into the analysis program.
  - Translating a binary into an intermediate representation(IR).
  - Performing the actual analysis

Flow

- Import angr

```
import angr
```

- Load the binary and initialize angr project

```
project = angr.Project('./ais3_crackme')
```

- Define argv1 as 100 bytes bitvectors

```
argv1 = claripy.BVS("argv1",100*8)
```

- Initialize the state with argv1

```
state = project.factory.entry_state(args=["./crackme1",argv1])
```

Flow

- Initialize the simulation manager

simgr = p.factory.simgr(state)

- Explore the states that matches the condition

simgr.explore(find= 0x400602)

- Extract one state from found states

found = simgr.found[0]

- Solve the expression with solver

solution = found.solver.eval(argv1, cast_to=str)

ais3 crackme

- Binary could be found in: https://github.com/angr/angr-doc/blob/master/examples/ais3_crackme/
- Run binary with argument
- If argument is correct
  - print "Correct! that is the secret key!"
- else
  - print "I'm sorry, that's the wrong secret key!"

Target address

```
.text:00000000004005EB loc_4005EB:                             ; CODE XREF: main+13↑j
.text:00000000004005EB                 mov     rax, [rbp+var_10]
.text:00000000004005EF                 add     rax, 8
.text:00000000004005F3                 mov     rax, [rax]
.text:00000000004005F6                 mov     rdi, rax
.text:00000000004005F9                 call    verify
.text:00000000004005FE                 test    eax, eax
.text:0000000000400600                 jz      short loc_40060E
.text:0000000000400602                 mov     edi, offset aCorrectThatIsT ; "Correct! that is the secret key!"
.text:0000000000400607                 call    _puts
.text:000000000040060C                 jmp     short loc_400618
.text:000000000040060E ; ---------------------------------------------------------------------------
.text:000000000040060E
.text:000000000040060E loc_40060E:                             ; CODE XREF: main+3B↑j
.text:000000000040060E                 mov     edi, offset aIMSorryThatSTh ; "I'm sorry, that's the wrong secret key!"
```

Solution

```
import angr
import claripy
project = angr.Project("./ais3_crackme")
argv1 = claripy.BVS("argv1",100*8)
state =
project.factory.entry_state(args=["./crackme1",argv1])
simgr = project.factory.simgr(state)
simgr.explore(find=0x400602)
found = simgr.found[0]
solution = found.solver.eval(argv1, cast_to=str)
print(repr(solution))
```

Result

```
apple ~ angr-doc examples ais3_crackme python solve.py
'ais3{I_tak3_g00d_n0t3s}\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00'
```

Intermediate Representation

- In order to be able to analyze and execute machine code from different CPU architectures, Angr performs most of its analysis on an intermediate representation
- Angr's intermediate representation is **VEX**(Valgrind), since the uplifting of binary code into VEX is quite well supported

Intermediate Representation

- IR abstracts away several architecture differences when dealing with different architectures
  - **Register names**: VEX models the registers as a separate memory space, with integer offsets
  - **Memory access**: The IR abstracts difference between architectures access memory in different ways
  - **Memory segmentation**: Some architectures support memory segmentation through the use of special segment registers
  - **Instruction side-effects**: Most instructions have side-effects

Intermediate Representation

- addl %eax, %ebx

- t3 = GET:I32(0)
- # get %eax, a 32-bit integer
- t2 = GET:I32(12)
- # get %ebx, a 32-bit integer
- t1 = Add32(t3,t2)
- # addl
- PUT(0) = t1
- # put %eax

Stash types

active	**This stash contains the states that will be stepped by default, unless an alternate stash is specified.**
deadended	A state goes to the deadended stash when it cannot continue the execution for some reason, including no more valid instructions, unsat state of all of its successors, or an invalid instruction pointer.
pruned	When using LAZY_SOLVES, states are not checked for satisfiability unless absolutely necessary. When a state is found to be unsat in the presence of LAZY_SOLVES, the state hierarchy is traversed to identify when, in its history, it initially became unsat. All states that are descendants of that point (which will also be unsat, since a state cannot become un-unsat) are pruned and put in this stash.
unconstrained	If the save_unconstrained option is provided to the SimulationManager constructor, states that are determined to be unconstrained (i.e., with the instruction pointer controlled by user data or some other source of symbolic data) are placed here.
unsat	If the save_unsat option is provided to the SimulationManager constructor, states that are determined to be unsatisfiable (i.e., they have constraints that are contradictory, like the input having to be both "AAAA" and "BBBB" at the same time) are placed here.

What's difference in Angr

- State concept is more complete, categorized, and more operation we can do upon the state.
- Symbolic function

Symbolic Function

- Project tries to replace external calls to library functions by using symbolic summaries termed **SimProcedures**
- Because SimProcedures are library hooks written in Python, it has inaccuracy
- If you encounter path explosion or inaccuracy, you can do:
  1. Disable the SimProcedure
  2. Replace the SimProcedure with something written directly to the situation in question
  3. Fix the SimProcedure

Symbolic Function(scanf)

- Source code: https://github.com/angr/angr/blob/master/angr/procedures/libc/scanf.py
- Get first argument(pointer to format string)

1. Define function return type by the architecture
2. Parse format string
3. According format string, read input from file descriptor 0(i.e., standard input)
4. Do the read operation

Symbolic Function(scanf)

```
class SimProcedure(object):
    @staticmethod
    def ty_ptr(self, ty):
        return SimTypePointer(self.arch, ty)

class FormatParser(SimProcedure):
    def _parse(self, fmt_idx):
    """
    fmt_idx: The index of the (pointer to the) format string in the arguments list.
    """


def interpret(self, addr, startpos, args, region=None):
    """
    Interpret a format string, reading the data at `addr` in `region` into `args`
starting at `startpos`.
    """
```

Symbolic Function(scanf)

```
from angr.procedures.stubs.format_parser import FormatParser
from angr.sim_type import SimTypeInt, SimTypeString
class scanf(FormatParser):
    def run(self, fmt):
        self.argument_types = {0: self.ty_ptr(SimTypeString())}
        self.return_type = SimTypeInt(self.state.arch.bits, True)
        fmt_str = self._parse(0)
        f = self.state.posix.get_file(0)
        region = f.content
        start = f.pos
        (end, items) = fmt_str.interpret(start, 1, self.arg, region=region)
        # do the read, correcting the internal file position and logging the action
        self.state.posix.read_from(0, end - start)
        return items
```

def _parse(self, fmt_idx):

```
int scanf ( const char * format, ... );
scanf ("%d",&i);

fmt_str = self._parse(0)
```

```
int sscanf ( const char * s, const char
* format, ...);
sscanf (sentence,"%s %*s %d",str,&i);

fmt_str = self._parse(1)
```

def interpret(self, addr, startpos, args, region=None):

```
int scanf ( const char * format, ... );
scanf ("%d",&i);
```

```
f = self.state.posix.get_file(0)
region = f.content
start = f.pos
(end, items) = fmt_str.interpret(start,
1, self.arg, region=region)
```

```
int sscanf ( const char * s, const char
* format, ...);
sscanf (sentence,"%s %*s %d",str,&i);
```

```
_, items =
fmt_str.interpret(self.arg(0), 2,
self.arg, region=self.state.memory)
```

References

- Symbolic execution wiki: https://en.wikipedia.org/wiki/Symbolic_execution
- GDB Python API: https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html
- Triton: https://triton.quarkslab.com/
- Peda: https://github.com/longld/peda
- Ponce: https://github.com/illera88/Ponce
- Angr: http://angr.io/

References

- KLEE: https://klee.github.io/
- Symbolic execution versus Concolic execution: https://github.com/JonathanSalwan/Triton/issues/284
- KLEE library call explained: https://dimjasevic.net/marko/2016/06/03/klee-it-aint-gonna-do-much-without-libraries/

Q & A

bananaappletw@gmail.com

@bananaappletw