ESILSolve: A Symbolic Execution Engine using ESIL

Austin “alkali” Emmitt

$ whoami

- Mobile Security Researcher at Nowsecure where I work on automating security tests for mobile applications (with Pancake, Edu, Grant, others!)
- Studied Math and Physics at University but sold out and did infosec instead
- Background of doing CTFs and game hacking for fun, as well as some iOS tweak dev
- Have worked as both a Reverse Engineer and boring Software Developer in previous roles
- Twitter: `@alkalinesec`

Introduction to ESILSolve

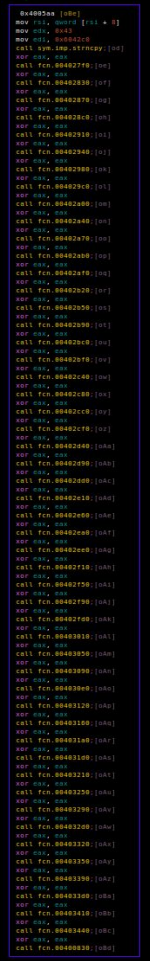
```
[0x400590]
 ;-- section..text:
409: int main (uint32_t argc, char **src);
; arg uint32_t argc @ rdi
; arg char **src @ rsi
0x00400590 4883ec08      sub rsp, 8
0x00400594 83ff02        cmp edi, 2
0x00400597 7411          je 0x4005aa
```

f t

```
 0x400599 [oc]
0x00400599 be58354000      mov esi, str._unbreakable_enterprise_product_activation_product_key
0x0040059e bf01000000      mov edi, 1
0x004005a3 31c0            xor eax, eax
0x004005a5 e886ffffff      call sym.imp.errx;[ob]
```

```
 0x4005aa [oBe]
0x004005aa 488b7608        mov rsi, qword [rsi + 8]
0x004005ae ba43000000      mov edx, 0x43
0x004005b3 bfc0426000      mov edi, 0x6042c0
0x004005b8 e853ffffff      call sym.imp.strncpy;[od]
0x004005bd 31c0            xor eax, eax
0x004005bf e82c220000      call fcn.004027f0;[oe]
0x004005c4 31c0            xor eax, eax
0x004005c6 e865220000      call fcn.00402830;[of]
0x004005cb 31c0            xor eax, eax
0x004005cd e89e220000      call fcn.00402870;[og]
0x004005d2 31c0            xor eax, eax
0x004005d4 e8e7220000      call fcn.004028c0;[oh]
0x004005d9 31c0            xor eax, eax
0x004005db e830230000      call fcn.00402910;[oi]
0x004005e0 31c0            xor eax, eax
0x004005e2 e859230000      call fcn.00402940;[oj]
0x004005e7 31c0            xor eax, eax
0x004005e9 e892230000      call fcn.00402980;[ok]
0x004005ee 31c0            xor eax, eax
0x004005f0 e8cb230000      call fcn.004029c0;[ol]
0x004005f5 31c0            xor eax, eax
0x004005f7 e804240000      call fcn.00402a00;[om]
0x004005fc 31c0            xor eax, eax
0x004005fe e83d240000      call fcn.00402a40;[on]
0x00400603 31c0            xor eax, eax
0x00400605 e866240000      call fcn.00402a70;[oo]
0x0040060a 31c0            xor eax, eax
0x0040060c e89f240000      call fcn.00402ab0;[op]
0x00400611 31c0            xor eax, eax
0x00400613 e8d8240000      call fcn.00402af0;[oq]
0x00400618 31c0            xor eax, eax
0x0040061a e801250000      call fcn.00402b20;[or]
0x0040061f 31c0            xor eax, eax
0x00400621 e82a250000      call fcn.00402b50;[os]
0x00400626 31c0            xor eax, eax
0x00400628 e863250000      call fcn.00402b90;[ot]
0x0040062d 31c0            xor eax, eax
0x0040062f e88c250000      call fcn.00402bc0;[ou]
```

Too long !



I don't want to manually reverse this.

Why did you make this? Why should I use it?

- ???
- `ESILSolve` will likely never have the stability or sophistication of established tools. But...
- Tight integration with r2 makes it usable with all different kinds of IO plugins in situations that might be awkward for more monolithic tools
- Support for architectures not covered by other tools
- It is a great way to learn about symbolic execution if you are familiar with r2 and `ESIL`, and is a simple but solid framework to test out novel ideas and strategies
- Good for CTFs :)

Symbolic Execution - A very short review

- Symbolic Execution is the emulation of a program with the ability to make some data, memory or register values, symbolic
- Using symbolic values allows exploration of code paths without needing to know concrete values that actually reach them
- Taking branches constrains the symbolic values and an SMT solver (`ESILSolve` uses `Z3`) is used to evaluate the resulting expressions to get concrete values
- Some SymEx tools tools you may have heard of and used: angr, KLEE, MAYHEM, more recently SymCC, Sys

Z3 simplified beyond usefulness

```
In [1]: import z3

In [2]: x = z3.BitVec("x", 32)

In [3]: y = z3.BitVec("y", 32)

In [4]: 3*x + y
Out[4]: 3*x + y

In [5]: z3.simplify(3*x + x + y)
Out[5]: 4*x + y

In [6]: z = z3.If(x/2 == 27, 4*x, x*y)

In [7]: z
Out[7]: If(x/2 == 27, 4*x, x*y)

In [8]: solver = z3.Solver()

In [9]: solver.add(x == 54)

In [10]: solver.check()
Out[10]: sat

In [11]: model = solver.model()
```

```
In [13]: model.eval(z)
Out[13]: 216

In [14]: 54*4
Out[14]: 216
```

- Z3 can solve expressions involving Ints, Floats, and most importantly `Bit Vectors`
- Symbolic values are initialized and used in arithmetic and logical expressions
- Solvers (or Optimizers) can create models that satisfy the added constraints

Simple Branch

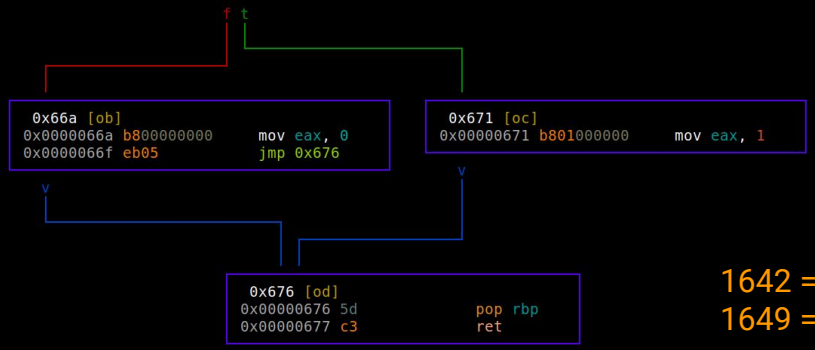

```
expr: rbp,8,rsp,-,=[8],8,rsp,-=
00000000000005fa: push rbp

expr: rsp,rbp,=
00000000000005fb: mov rbp, rsp

expr: edi,0x4,rbp,-,=[4]
00000000000005fe: mov dword [rbp - 4], edi

expr: rsi,0x10,rbp,-,=[8]
0000000000000601: mov qword [rbp - 0x10], rsi

expr: 2,0x4,rbp,-,[4],==,$z,zf,:=,32,$b,cf,:=,$p,pf,:=,31,$s,sf,:=,2
0000000000000605: cmp dword [rbp - 4], 2

expr: zf,!,?{,1554,rip,=,}
0000000000000609: jne 0x612
condition val: If(Extract(31, 0, rdi) == 2, 0, 1)
symbolic pc: If(If(Extract(31, 0, rdi) == 2, 0, 1) == 0, 1547, 1554)
```

1547 == 0x60b

1554 == 0x612

Simplish Branch

```
[0x63a]
62: sym.check (int64_t arg1);
; var int64_t var_4h @ rbp-0x4
; arg int64_t arg1 @ rdi
0x0000063a 55              push rbp
0x0000063b 4889e5          mov rbp, rsp
0x0000063e 897dfc          mov dword [var_4h], edi
0x00000641 8b45fc          mov eax, dword [var_4h]
0x00000644 35bebafeca      xor eax, 0xcafebabe
0x00000649 89c1            mov ecx, eax
0x0000064b babdbcc8ef      mov edx, 0xefc8bcbd
0x00000650 89c8            mov eax, ecx
0x00000652 f7e2            mul edx
0x00000654 89d0            mov eax, edx
0x00000656 c1e80e          shr eax, 0xe
0x00000659 69c054440000    imul eax, eax, 0x4454
0x0000065f 29c1            sub ecx, eax
0x00000661 89c8            mov eax, ecx
0x00000663 3de7110000      cmp eax, 0x11e7
0x00000668 7507            jne 0x671
```



1642 == 0x66a
1649 == 0x671

```
symbolic pc: If(If(Concat(~Extract(31, 30, rdi),
                          Extract(29, 28, rdi),
                          ~Extract(27, 27, rdi),
                          Extract(26, 26, rdi),
                          ~Extract(25, 25, rdi),
                          Extract(24, 24, rdi),
                          ~Extract(23, 17, rdi),
                          Extract(16, 16, rdi),
                          ~Extract(15, 15, rdi),
                          Extract(14, 14, rdi),
                          ~Extract(13, 11, rdi),
                          Extract(10, 10, rdi),
                          ~Extract(9, 9, rdi),
                          Extract(8, 8, rdi),
                          ~Extract(7, 7, rdi),
                          Extract(6, 6, rdi),
                          ~Extract(5, 1, rdi),
                          Extract(0, 0, rdi)) ==
                   4583 +
                   17492*
                   Concat(0,
                          Extract(63,
                                  46,
                                  4022910141*
                                  Concat(0,
                                         ~Extract(31, 30, rdi),
                                         Extract(29, 28, rdi),
                                         ~Extract(27, 27, rdi),
                                         Extract(26, 26, rdi),
                                         ~Extract(25, 25, rdi),
                                         Extract(24, 24, rdi),
                                         ~Extract(23, 17, rdi),
                                         Extract(16, 16, rdi),
                                         ~Extract(15, 15, rdi),
                                         Extract(14, 14, rdi),
                                         ~Extract(13, 11, rdi),
                                         Extract(10, 10, rdi),
                                         ~Extract(9, 9, rdi),
                                         Extract(8, 8, rdi),
                                         ~Extract(7, 7, rdi),
                                         Extract(6, 6, rdi),
                                         ~Extract(5, 1, rdi),
                                         Extract(0, 0, rdi)))),
                   0,
                   1) ==
             0,
             1642,
             1649)
```

ESIL - Another even shorter review

- ESIL is radare2's built in intermediate language
- It uses a simple, stack based system to express any instruction as a string of values and operators
- Example: `add x0, x1, x2` in AArch64 becomes "`x2,x1,+,x0,=`"
- It's simple, readable, and great
- It is an interesting IR for symbolic execution
- There is a really great in depth talk on ESIL from r2con 2019 by @arnaugamez

```
        ; CALL XREF from main @ 0x6d8
┌ 117: sym.check (int64_t arg1);
        ; var int64_t var_4h @ rbp-0x4
        ; arg int64_t arg1 @ rdi
        0x0000063a      55              push rbp                    ; rbp,8,rsp,-,=[8],8,rsp,-=
        0x0000063b      4889e5          mov rbp, rsp                ; rsp,rbp,=
        0x0000063e      897dfc          mov dword [var_4h], edi     ; edi,0x4,rbp,-,=[4] ; arg1
        0x00000641      8b45fc          mov eax, dword [var_4h]     ; 0x4,rbp,-,[4],rax,=
        0x00000644      35feca0000      xor eax, 0xcafe             ; 51966,rax,^,0xffffffff,&,rax,=,$z,zf,:=,$p,pf,:=,31,$s,sf,:=,0,cf,:=,0,of,:=
        0x00000649      89c1            mov ecx, eax                ; eax,rcx,=
        0x0000064b      ba5f5ee477      mov edx, 0x77e45e5f         ; 2011455071,rdx,=
        0x00000650      89c8            mov eax, ecx                ; ecx,rax,=
        0x00000652      f7ea            imul edx                    ; 32,32,edx,~,32,eax,~,*,>>,edx,=,edx,eax,*=,32,32,eax,~,>>,edx,-,?{,1,1,}{,0,0,},cf,:=,of,:=
        0x00000654      c1fa0d          sar edx, 0xd                ; 0,cf,:=,1,13,-,1,<<,edx,&,?{,1,cf,:=,},13,edx,>>>>,edx,=,$z,zf,:=,$p,pf,:=,31,$s,sf,:=
        0x00000657      89c8            mov eax, ecx                ; ecx,rax,=
        0x00000659      c1f81f          sar eax, 0x1f               ; 0,cf,:=,1,31,-,1,<<,eax,&,?{,1,cf,:=,},31,eax,>>>>,eax,=,$z,zf,:=,$p,pf,:=,31,$s,sf,:=
        0x0000065c      29c2            sub edx, eax                ; eax,edx,-=,eax,0x80000000,-,!,31,$o,^,of,:=,31,$s,sf,:=,$z,zf,:=,$p,pf,:=,32,$b,cf,:=,3,$b,af,:=
        0x0000065e      89d0            mov eax, edx                ; edx,rax,=
        0x00000660      69c054440000    imul eax, eax, 0x4454       ; 32,17492,~,32,eax,~,*,DUP,eax,=,eax,-,?{,1,1,}{,0,0,},cf,:=,of,:=
        0x00000666      29c1            sub ecx, eax                ; eax,ecx,-=,eax,0x80000000,-,!,31,$o,^,of,:=,31,$s,sf,:=,$z,zf,:=,$p,pf,:=,32,$b,cf,:=,3,$b,af,:=
        0x00000668      89c8            mov eax, ecx                ; ecx,rax,=
        0x0000066a      3d53140000      cmp eax, 0x1453             ; 5203,eax,==,$z,zf,:=,32,$b,cf,:=,$p,pf,:=,31,$s,sf,:=,5203,0x80000000,-,!,31,$o,^,of,:=,3,$b,af,:=
    ┌<  0x0000066f      7537            jne 0x6a8                   ; zf,!,?{,1704,rip,=,}
    │   0x00000671      8b45fc          mov eax, dword [var_4h]     ; 0x4,rbp,-,[4],rax,=
    │   0x00000674      35beba0000      xor eax, 0xbabe             ; 47806,rax,^,0xffffffff,&,rax,=,$z,zf,:=,$p,pf,:=,31,$s,sf,:=,0,cf,:=,0,of,:=
    │   0x00000679      89c1            mov ecx, eax                ; eax,rcx,=
    │   0x0000067b      ba5f5ee477      mov edx, 0x77e45e5f         ; 2011455071,rdx,=
    │   0x00000680      89c8            mov eax, ecx                ; ecx,rax,=
    │   0x00000682      f7ea            imul edx                    ; 32,32,edx,~,32,eax,~,*,>>,edx,=,edx,eax,*=,32,32,eax,~,>>,edx,-,?{,1,1,}{,0,0,},cf,:=,of,:=
    │   0x00000684      c1fa0d          sar edx, 0xd                ; 0,cf,:=,1,13,-,1,<<,edx,&,?{,1,cf,:=,},13,edx,>>>>,edx,=,$z,zf,:=,$p,pf,:=,31,$s,sf,:=
    │   0x00000687      89c8            mov eax, ecx                ; ecx,rax,=
    │   0x00000689      c1f81f          sar eax, 0x1f               ; 0,cf,:=,1,31,-,1,<<,eax,&,?{,1,cf,:=,},31,eax,>>>>,eax,=,$z,zf,:=,$p,pf,:=,31,$s,sf,:=
    │   0x0000068c      29c2            sub edx, eax                ; eax,edx,-=,eax,0x80000000,-,!,31,$o,^,of,:=,31,$s,sf,:=,$z,zf,:=,$p,pf,:=,32,$b,cf,:=,3,$b,af,:=
    │   0x0000068e      89d0            mov eax, edx                ; edx,rax,=
    │   0x00000690      69c054440000    imul eax, eax, 0x4454       ; 32,17492,~,32,eax,~,*,DUP,eax,=,eax,-,?{,1,1,}{,0,0,},cf,:=,of,:=
    │   0x00000696      29c1            sub ecx, eax                ; eax,ecx,-=,eax,0x80000000,-,!,31,$o,^,of,:=,31,$s,sf,:=,$z,zf,:=,$p,pf,:=,32,$b,cf,:=,3,$b,af,:=
    │   0x00000698      89c8            mov eax, ecx                ; ecx,rax,=
    │   0x0000069a      3dbf1f0000      cmp eax, 0x1fbf             ; 8127,eax,==,$z,zf,:=,32,$b,cf,:=,$p,pf,:=,31,$s,sf,:=,8127,0x80000000,-,!,31,$o,^,of,:=,3,$b,af,:=
    │┌<  0x0000069f      7507            jne 0x6a8                   ; zf,!,?{,1704,rip,=,}
    ││   0x000006a1      b800000000      mov eax, 0                  ; 0,rax,=
    ┌──<  0x000006a6      eb05            jmp 0x6ad                   ; 0x6ad,rip,=
    │└└   ; CODE XREFS from sym.check @ 0x66f, 0x69f
    │└─>  0x000006a8      b801000000      mov eax, 1                  ; 1,rax,=
    │    ; CODE XREF from sym.check @ 0x6a6
    └───>  0x000006ad      5d              pop rbp                     ; rsp,[8],rbp,=,8,rsp,+=
           0x000006ae      c3              ret                         ; rsp,[8],rip,=,8,rsp,+=
[0x0000063a]>
```

```
push rbp                        ; rbp,8,rsp,-,=[8],8,rsp,-=
mov rbp, rsp                    ; rsp,rbp,=
mov dword [var_4h], edi         ; edi,0x4,rbp,-,=[4] ; arg1
mov eax, dword [var_4h]         ; 0x4,rbp,-,[4],rax,=
```

Challenges of ESIL based SymEx

- ESIL is a lot less verbose / precise than other IR
- “add x0, x1, x2” in VEX ->
- Every value and operation has an explicit size, type, and signedness in other IR
- Turns out they are being overly cautious dorks

```
IRSB {
   t0:Ity_I64 t1:Ity_I64 t2:Ity_I64
t3:Ity_I64 t4:Ity_I64

   00 | ------ IMark(0x400400, 4, 0) ------
   01 | t0 = GET:I64(x1)
   02 | t1 = GET:I64(x2)
   03 | t3 = Add64(t0,t1)
   04 | PUT(x0) = t3
   NEXT: PUT(pc) = 0x0000000000400404;
Ijk_Boring
}
```

Challenges of `ESIL` based SymEx Part II

- <show main parse_expression method>
- Other IR have data only in temporary registers and variables
- `ESILSolve` needs to have a *symbolic stack*, where for every conditional both IF and ELSE expressions are executed, their stacks are unwound, and a new stack with `If(CONDITION, IF_VALUE, ELSE_VALUE)` values is pushed
- The GOTO operator essentially uses arbitrarily nested conditionals which creates a host of issues, most of which are soluble
- These operations happen to be relatively efficient ones for python

The ESILSolve API

- ESILSolver
  - `call_state`, `init_state`, `blank_state` - create an initial state to begin running
  - `register_hook`, `register_sim` - register hooks or simulated methods to manipulate state
  - `run`, `terminate` - start symbolically executing, finding `find`, avoiding `avoid`

```
from esilsolve import ESILSolver
import r2pipe
import z3

esilsolver = ESILSolver("ais3_crackme")
state = esilsolver.call_state("sym.verify")

addr = 0x1000000
state.registers["rdi"] = addr
flag = z3.BitVec("flag", 24*8)
state.memory[addr] = flag

def check(state):
    state.constrain(state.registers["zf"] == 1)
    if state.is_sat():
        flag_str = state.evaluate_string(flag)
        print("FLAG: %s " % flag_str)
        esilsolver.terminate()

esilsolver.register_hook(0x004005bd, check)
state = esilsolver.run()
```

The ESILSolve API

- ESILState
  - registers - an instance of ESILRegisters with the register values for this state
  - memory - an instance of ESILMemory for this state
  - solver - and instance of a Z3 solver class (Solver, SimpleSolver, Optimizer)
  - constrain - an alias for solver.add
  - constrain_bytes - method to constrain the bytes of a buffer to match a regex range
  - evaluate - model and eval a symbolic expression if it is satisfiable
  - evaluate_buffer, state.evaluate_string - convenience methods for evaluating a BV and casting it to bytes / string
  - clone - make a (COW) copy of the state
  - step - single step the state, executing the current instruction

Demo ESILSolve

Show some cool stuff now

ESILSolve plugin for r2

```
[0x00400729]> aesx?
Usage: aesx[iscxrebda] # Core plugin for ESILSolve
| aesxi [debug] [lazy]               Initialize the ESILSolve instance and VM
| aesxs[bc] reg|addr [name] [length] Set symbolic value in register or memory
| aesxv reg|addr value               Set concrete value in register or memory
| aesxc sym value                    Constrain symbol to be value, min, max, regex
| aesxx[ec] expr value               Execute ESIL expression and evaluate/constrain the result
| aesxr[ac] target [avoid x,y,z]     Run symbolic execution until target address, avoiding x,y,z
| aesxe[j] sym1 [sym2] [...]         Evaluate symbol in current state
| aesxb[j] sym1 [sym2] [...]         Evaluate buffer in current state
| aesxd[j] [reg1] [reg2] [...]       Dump register values / ASTs
| aesxa                              Apply the current state, setting registers and memory
[0x00400729]>
```

ESILSolve plugin II

```
undefined8 fcn.004006fd(int64_t arg1)
{
    int64_t var_38h;
    char *var_24h;
    char *var_18h;
    char *var_10h;

    stack0xffffffffffffffd8 = "Dufhbmf";
    var_18h = "pG`imos";
    var_10h = "ewUglpt";
    var_24h._0_4_ = 0;
    while( true ) {
        if (0xb < (int32_t)var_24h) {
    // target: flag = b'Code_Talkers'
            return 0;
        }
        if ((int32_t)*(char *)((int64_t)(((int32_t)var_24h / 3) * 2) +
                               *(int64_t *)((int64_t)&var_24h + (int64_t)((int32_t)var_24h % 3) * 8 + 4)) -
            (int32_t)*(char *)(arg1 + (int32_t)var_24h) != 1) break;
        var_24h._0_4_ = (int32_t)var_24h + 1;
    }
    return 1;
}
```

Prove long strings of arithmetical instructions equivalent to simpler ones

Automatically add comments that show concrete values that reach any point in code, for every symbol

```
[0x00000610]> s sym.rotateLeft
[0x000007f2]> pdf
            ; CALL XREF from main @ 0x78d
┌ 89: sym.rotateLeft (int64_t arg1, int64_t arg2);
            ; var int64_t var_18h @ rbp-0x18
            ; var int64_t var_14h @ rbp-0x14
            ; var int64_t var_4h @ rbp-0x4
            ; arg int64_t arg1 @ rdi
            ; arg int64_t arg2 @ rsi
            0x000007f2      55              push rbp
            0x000007f3      4889e5          mov rbp, rsp
            0x000007f6      897dec          mov dword [var_14h], edi
            0x000007f9      8975e8          mov dword [var_18h], esi
            0x000007fc      8b4de8          mov ecx, dword [var_18h]
            0x000007ff      ba85104208      mov edx, 0x8421085
            0x00000804      89c8            mov eax, ecx
            0x00000806      f7e2            mul edx
            0x00000808      89c8            mov eax, ecx
            0x0000080a      29d0            sub eax, edx
            0x0000080c      d1e8            shr eax, 1
            0x0000080e      01d0            add eax, edx
            0x00000810      c1e804          shr eax, 4
            0x00000813      89c2            mov edx, eax
            0x00000815      89d0            mov eax, edx
            0x00000817      c1e005          shl eax, 5
            0x0000081a      29d0            sub eax, edx
            0x0000081c      29c1            sub ecx, eax
            0x0000081e      89c8            mov eax, ecx
            0x00000820      8945e8          mov dword [var_18h], eax
        ┌─< 0x00000823      eb14            jmp 0x839
        │   ; CODE XREF from sym.rotateLeft @ 0x844
        │  > 0x00000825      8b45ec          mov eax, dword [var_14h]
        │  │ 0x00000828      c1e81f          shr eax, 0x1f
        │  │ 0x0000082b      8945fc          mov dword [var_4h], eax
        │  │ 0x0000082e      8b45ec          mov eax, dword [var_14h]
        │  │ 0x00000831      01c0            add eax, eax
        │  │ 0x00000833      0b45fc          or eax, dword [var_4h]
        │  │ 0x00000836      8945ec          mov dword [var_14h], eax
        │  │ ; CODE XREF from sym.rotateLeft @ 0x823
        │  └> 0x00000839      8b45e8          mov eax, dword [var_18h]
        │     0x0000083c      8d50ff          lea edx, [rax - 1]
        │     0x0000083f      8955e8          mov dword [var_18h], edx
        │     0x00000842      85c0            test eax, eax
        └───< 0x00000844      75df            jne 0x825
              0x00000846      8b45ec          mov eax, dword [var_14h]
              0x00000849      5d              pop rbp
              0x0000084a      c3              ret
[0x000007f2]> aeim
[0x000007f2]> aei
[0x000007f2]> aesxi
[0x000007f2]> aesxs rdi num 8
[0x000007f2]> aesxv rsi 8
[0x000007f2]> aesxr 0x00000849
[0x000007f2]> aesxxc "esi,edi,<<<,eax,-,!" 0
[0x000007f2]> aesxe eax
error: state has unsatisfiable constraints
```

Using ESILSolve to improve ESIL

- Symbolic execution of ESIL allows us to convert the string expression to an AST which can be compared to the ASTs generated by other symex tools
- If expressions are not equivalent the SMT solver can give example values that produce different results in the two systems
- ESILCheck is a tool which compares the ASTs generated from ESIL expressions and the ones produced by angr's VEX execution engine
- The claripy backend objects provide conversion to Z3 enabling easy testing
- Other engines, like KLEE, could provide additional checks and coverage, any tool supporting the same SMT solvers can be used.
- A tool called VEX2ESIL also comes with ESILSolve

Future Goals

- Use `ESILSolve` to prove the equality of `ESIL` expressions and other IR expressions, or fix them if necessary, to cover more architectures more faithfully
- Make a modest Sim Procedure library so that common libc functions do not cause state explosion
- Create Sims for syscalls and hopefully have syscalls handled natively in concrete emulation in r2
- Create a fuzzer leveraging `ESILSolve`, concrete `ESIL` emulation in r2, and maybe r2frida to generate amazing coverage guided fuzzing that is snapshot emulated for speed but in-memory fuzzed for accuracy

Thanks To:

Pancake and everyone who has worked on ESIL and radare2 !

Thank You