radius: *fast* symbolic execution with radare2

Austin “alkali” Emmitt

$ whoami

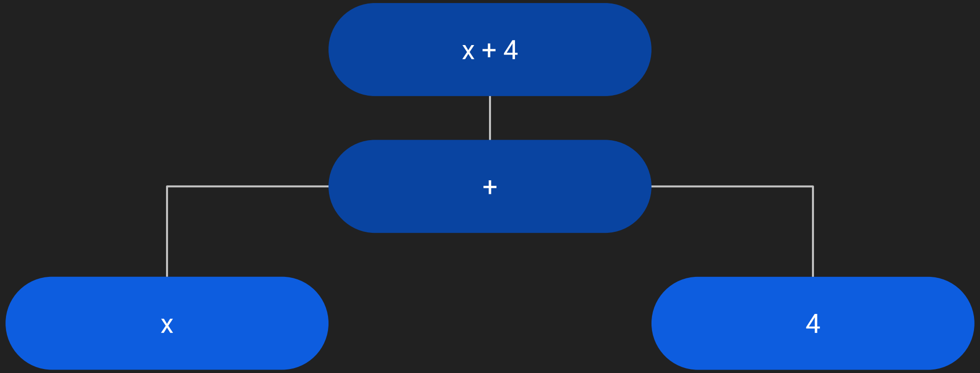
- Mobile Security Researcher at NowSecure
- Went to school for Physics and Mathematics
- Fan of fancy shmancy techniques that are rarely useful

Appetizer Demo

<do some cool stuff>

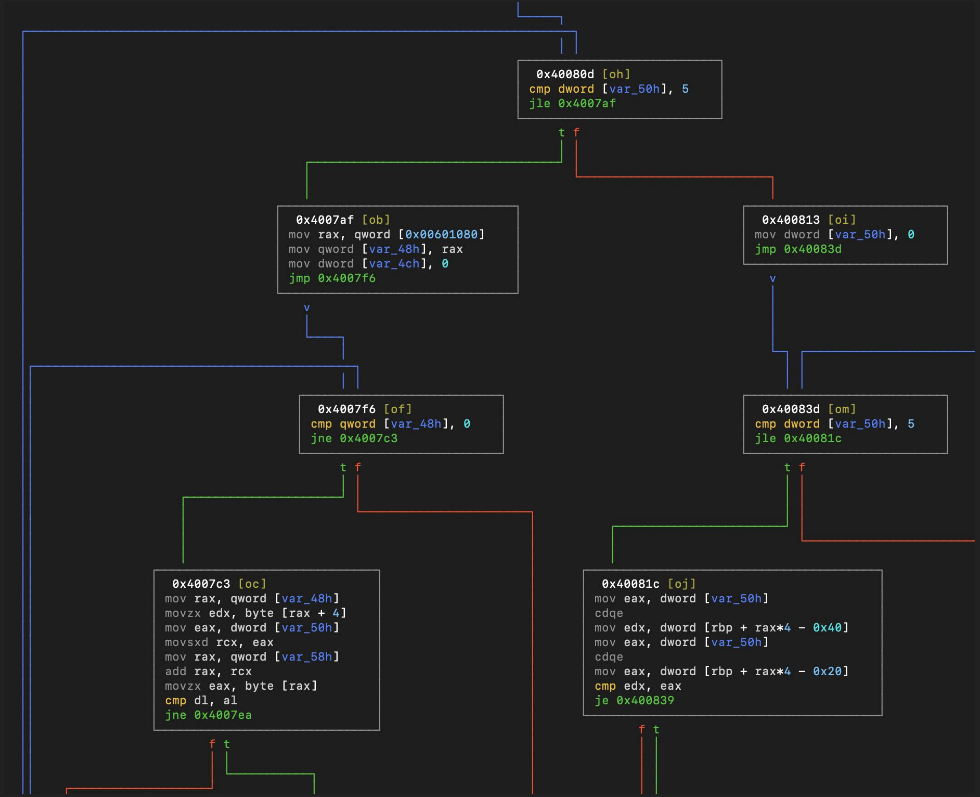
Symbolic Execution “explained”

- Symbolic Execution refers to emulation of a program with symbolic values
- Instead of having a register **rax** set to 0x8 it is set to *x* and operations on this value are simply kept track of in a structure called an AST
- Ex. the instruction **add rax, 4** would result in *x + 4* stored in a tree like the one here



Symbolic Execution “explained”

- Each conditional branch involving a symbolic value results in a symbolic PC address. The corresponding AST is evaluated and the state is split, with each resulting state being constrained by one evaluation of PC
- Ex. on the right the jne and jle instructions would cause a split if [var_48h] and [var_50h] are symbolic



```
 0x40080d [oh]
cmp dword [var_50h], 5
jle 0x4007af
```

```
 0x4007af [ob]
mov rax, qword [0x00601080]
mov qword [var_48h], rax
mov dword [var_4ch], 0
jmp 0x4007f6
```

```
 0x400813 [oi]
mov dword [var_50h], 0
jmp 0x40083d
```

```
 0x4007f6 [of]
cmp qword [var_48h], 0
jne 0x4007c3
```

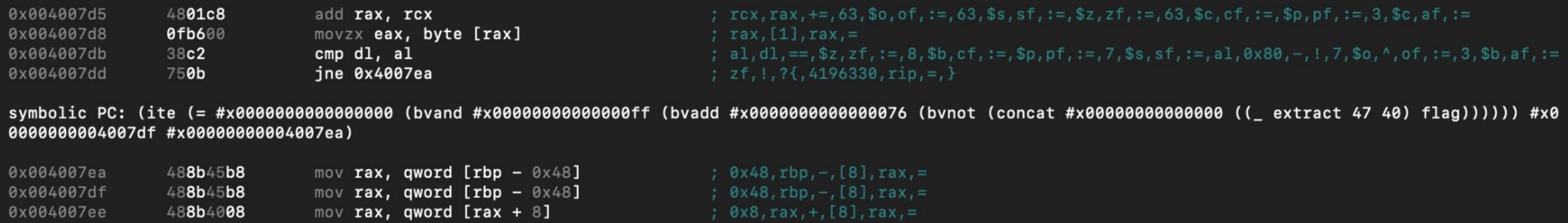
```
 0x40083d [om]
cmp dword [var_50h], 5
jle 0x40081c
```

```
 0x4007c3 [oc]
mov rax, qword [var_48h]
movzx edx, byte [rax + 4]
mov eax, dword [var_50h]
movsxd rcx, eax
mov rax, qword [var_58h]
add rax, rcx
movzx eax, byte [rax]
cmp dl, al
jne 0x4007ea
```

```
 0x40081c [oj]
mov eax, dword [var_50h]
cdqe
mov edx, dword [rbp + rax*4 - 0x40]
mov eax, dword [var_50h]
cdqe
mov eax, dword [rbp + rax*4 - 0x20]
cmp edx, eax
je 0x400839
```

Symbolic Execution “explained”

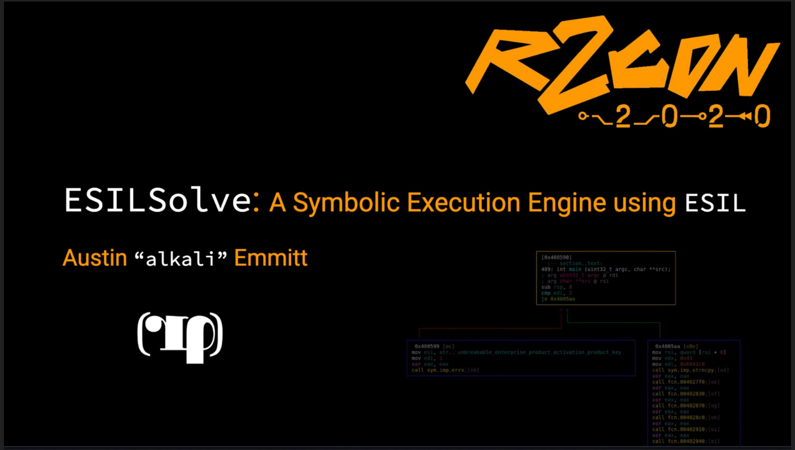
- Below is an example of debugging output from radius
- This shows the symbolic PC register value. This will be evaluated and execution will resume at the possible resulting addresses
- These possibilities can be seen at the end of the output representing the bitvector expression



```
0x004007d5      4801c8        add rax, rcx                    ; rcx,rax,+=,63,$o,of,:=,63,$s,sf,:=,$z,zf,:=,63,$c,cf,:=,$p,pf,:=,3,$c,af,:=
0x004007d8      0fb600        movzx eax, byte [rax]           ; rax,[1],rax,=
0x004007db      38c2          cmp dl, al                      ; al,dl,==,$z,zf,:=,8,$b,cf,:=,$p,pf,:=,7,$s,sf,:=,al,0x80,-,!,7,$o,^,of,:=,3,$b,af,:=
0x004007dd      750b          jne 0x4007ea                    ; zf,!,?{,4196330,rip,=,}

symbolic PC: (ite (= #x0000000000000000 (bvand #x00000000000000ff (bvadd #x0000000000000076 (bvnot (concat #x00000000000000 ((_ extract 47 40) flag)))))) #x0
0000000004007df #x00000000004007ea)

0x004007ea      488b45b8      mov rax, qword [rbp - 0x48]     ; 0x48,rbp,-,[8],rax,=
0x004007df      488b45b8      mov rax, qword [rbp - 0x48]     ; 0x48,rbp,-,[8],rax,=
0x004007ee      488b4008      mov rax, qword [rax + 8]        ; 0x8,rax,+,[8],rax,=
```

ESILSolve Recap

- ESILSolve is a symbolic execution framework written in Python that executes radare2’s intermediate representation ESIL which turns an instruction like add eax, 4 into a string 4,eax,+,eax,=
- It uses Z3 to make symbolic values and solve the generated expressions
- See my talk from r2con2020:



ESILSolve: A Symbolic Execution Engine using ESIL

Austin “alkali” Emmitt

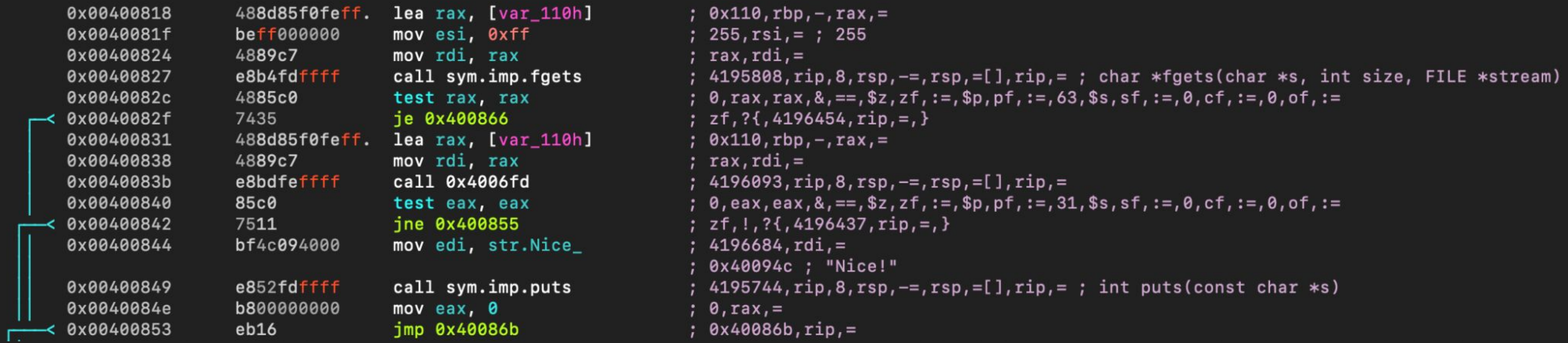
```
[0x400590]
  ;-- section..text:
409: int main (uint32_t argc, char **src);
; arg uint32_t argc @ rdi
; arg char **src @ rsi
sub rsp, 8
cmp edi, 2
je 0x4005aa
```

```
 0x400599 [oc]
mov esi, str.._unbreakable_enterprise_product_activation_product_key
mov edi, 1
xor eax, eax
call sym.imp.errx;[ob]
```

```
 0x4005aa [oBe]
mov rsi, qword [rsi + 8]
mov edx, 0x43
mov edi, 0x6042c0
call sym.imp.strncpy;[od]
xor eax, eax
call fcn.004027f0;[oe]
xor eax, eax
call fcn.00402830;[of]
xor eax, eax
call fcn.00402870;[og]
xor eax, eax
call fcn.004028c0;[oh]
xor eax, eax
call fcn.00402910;[oi]
xor eax, eax
call fcn.00402940;[oj]
```

ESIL Recap

- ESIL is a string of values, registers, and operations that can represent any instruction from any architecture
- Values and registers get pushed onto a stack and are popped off by operators, which push new results to the stack
- Examples from a binary



```
    0x00400818      488d85f0feff.  lea rax, [var_110h]       ; 0x110,rbp,-,rax,=
    0x0040081f      beff000000     mov esi, 0xff             ; 255,rsi,= ; 255
    0x00400824      4889c7         mov rdi, rax              ; rax,rdi,=
    0x00400827      e8b4fdffff     call sym.imp.fgets        ; 4195808,rip,8,rsp,-=,rsp,=[],rip,= ; char *fgets(char *s, int size, FILE *stream)
    0x0040082c      4885c0         test rax, rax             ; 0,rax,rax,&,==,$z,zf,:=,$p,pf,:=,63,$s,sf,:=,0,cf,:=,0,of,:=
 ┌─< 0x0040082f      7435           je 0x400866               ; zf,?{,4196454,rip,=,}
 │  0x00400831      488d85f0feff.  lea rax, [var_110h]       ; 0x110,rbp,-,rax,=
 │  0x00400838      4889c7         mov rdi, rax              ; rax,rdi,=
 │  0x0040083b      e8bdfeffff     call 0x4006fd             ; 4196093,rip,8,rsp,-=,rsp,=[],rip,=
 │  0x00400840      85c0           test eax, eax             ; 0,eax,eax,&,==,$z,zf,:=,$p,pf,:=,31,$s,sf,:=,0,cf,:=,0,of,:=
┌──< 0x00400842      7511           jne 0x400855              ; zf,!,?{,4196437,rip,=,}
││  0x00400844      bf4c094000     mov edi, str.Nice_        ; 4196684,rdi,=
││                                                           ; 0x40094c ; "Nice!"
││  0x00400849      e852fdffff     call sym.imp.puts         ; 4195744,rip,8,rsp,-=,rsp,=[],rip,= ; int puts(const char *s)
││  0x0040084e      b800000000     mov eax, 0                ; 0,rax,=
┌───< 0x00400853      eb16           jmp 0x40086b              ; 0x40086b,rip,=
```

radius: Introduction and Motivation

- radius is essentially a rust rewrite of ESILSolve
- Why did I do this?

  - Python is slow
  - Python is sooooooooooooooooo slow
  - ESILSolve was not designed in a way that made multithreaded execution easy or very worthwhile

  - Rust is fast and makes it easy to write safe multithreaded programs
  - It is very trendy and I want to be cool

radius vs. ESILSolve

- radius executes concrete instructions about 2000x faster than ESILSolve
- Instructions using symbolic values are executed roughly 20x faster
- radius uses Boolector as its SMT solver instead of Z3.

- The Processor is no longer nominally a part of the state. Processors act on states, to execute instructions which change the Register and Memory values
- This allows Processors to step states in parallel
- States contain a user defined list of arbitrary values called context

radius vs ESILSolve cont.

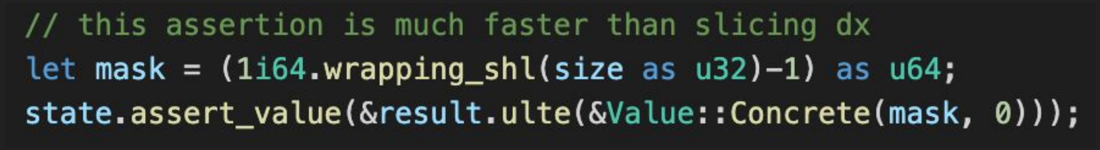
- radius does not just use Bit Vectors, it has 2 types of values
  - Value::Concrete
  - Value::Symbolic
- The concrete values are simply u64 primitives which is all ESIL needs
- This makes radius faster than even other C / C++ / Rust based symbolic execution engines that have high overhead for every operation even on constant bit vector values
- radius does not keep copies of unsatisfiable or inactive states, instead relying on hooking and context to deal with unknown conditions that require examining many states

radius vs The Rest

	radius	esilsolve	angr	angr + unicorn	manticore	triton
short looper*	0.005	3.404	2.181	0.183	-	-
long looper*	0.257	eternity	eternity	0.241	eternity	-
r100	0.186	0.891	4.242 **	-	broken	0.27
r200	0.321	1.42	broken	-	-	-
babyre	4.94	5.97 ***	5.987	5.892	-	310.81
unbreakable	0.418	1.363	3.509	-	broken	8.04
ais3_crackme	0.237	1.06	5.618	-	broken	-

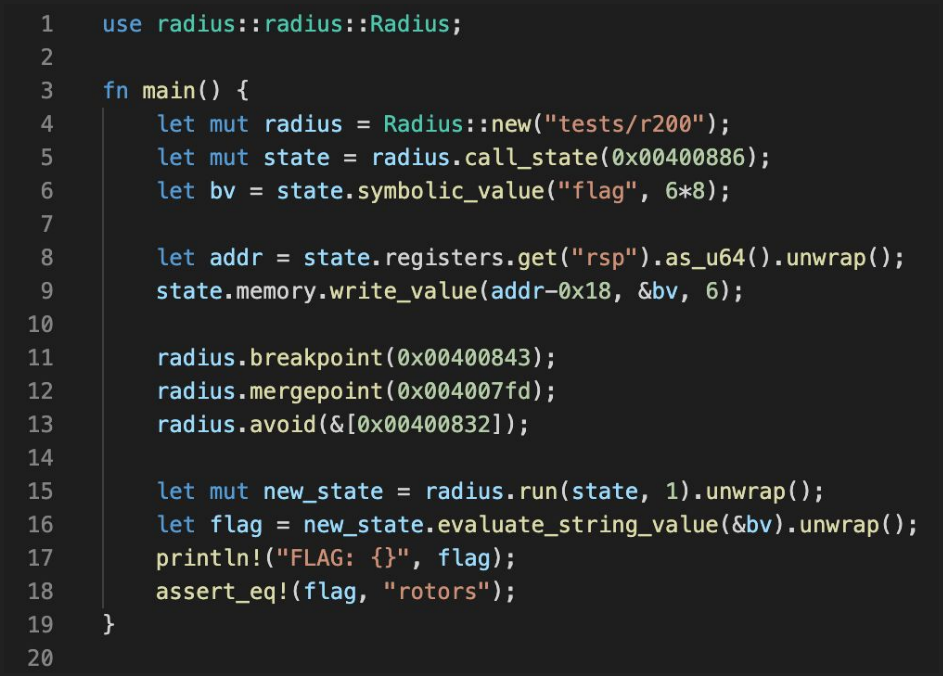
Symbolic Speed

- The speed of a symbolic execution tool is **complicated**
- Essentially there is the speed of initialization, execution, and concretization (solving).
- r2 provides quick initialization, tokenizing ESIL gives quick execution
- For many / most programs concretizing symbolic values will completely dominate the run time
- SMT solvers are weird. Like *real* weird.



```
// this assertion is much faster than slicing dx
let mask = (1i64.wrapping_shl(size as u32)-1) as u64;
state.assert_value(&result.ulte(&Value::Concrete(mask, 0)));
```

radius: fast merging

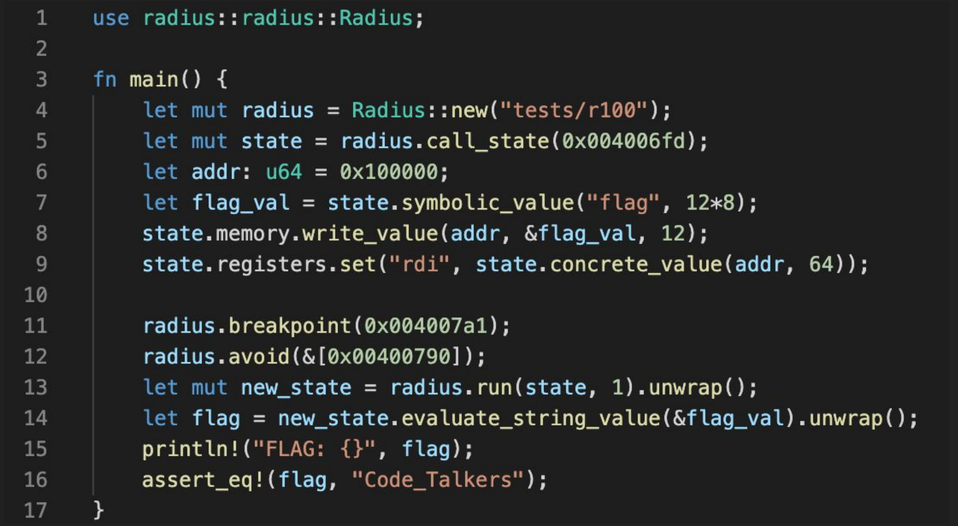


```rust
use radius::radius::Radius;

fn main() {
    let mut radius = Radius::new("tests/r200");
    let mut state = radius.call_state(0x00400886);
    let bv = state.symbolic_value("flag", 6*8);

    let addr = state.registers.get("rsp").as_u64().unwrap();
    state.memory.write_value(addr-0x18, &bv, 6);

    radius.breakpoint(0x00400843);
    radius.mergepoint(0x004007fd);
    radius.avoid(&[0x00400832]);

    let mut new_state = radius.run(state, 1).unwrap();
    let flag = new_state.evaluate_string_value(&bv).unwrap();
    println!("FLAG: {}", flag);
    assert_eq!(flag, "rotors");
}

```

- The fast execution, copying, and merging of states is one of radius' most valuable features
- Use of aggressive automated merging could fix state explosion in most cases, one of symbolic execution's most difficult problems
- Unfortunately it is not possible to merge in multithreaded runs

The radius API: by example

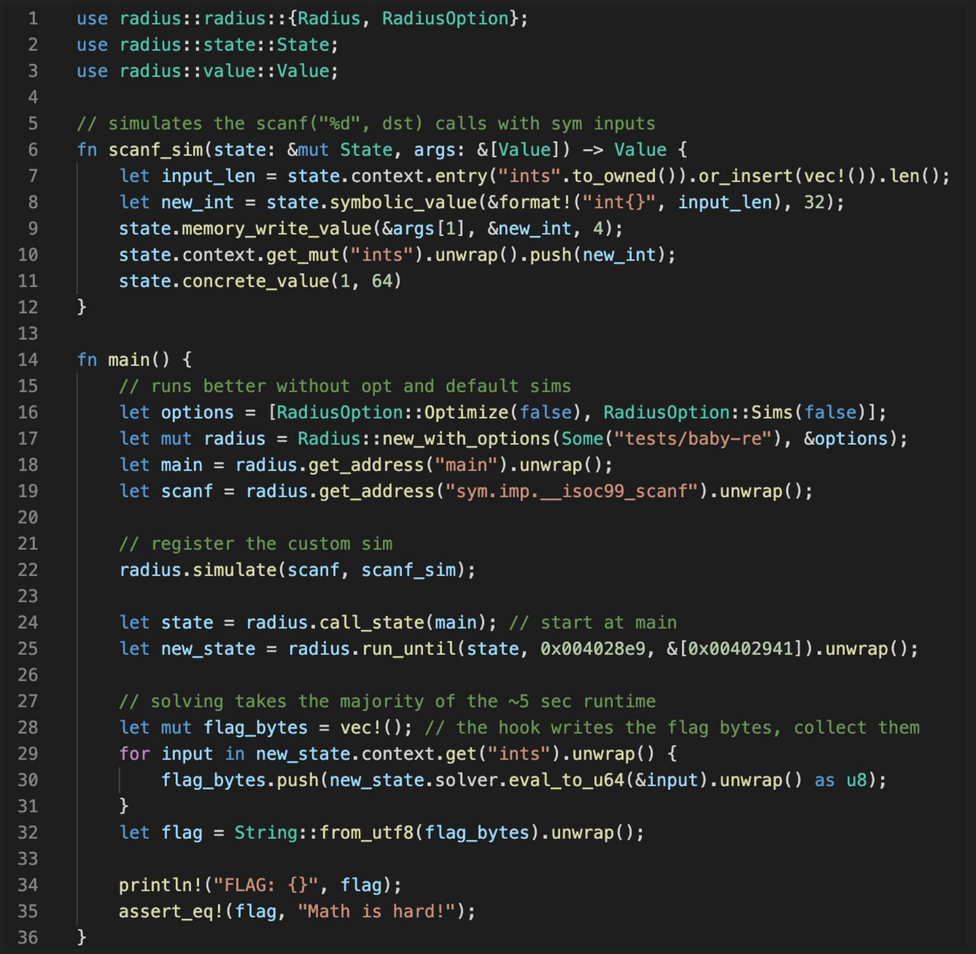
- Radius::new : instantiate a new radius instance with a path to a binary to analyze
- Methods to create a starting state:
  - Radius::call_state
  - Radius::init_state
  - Radius::blank_state
  - Radius::entry_state
- Methods to set addresses to target and avoid
  - Radius::breakpoint
  - Radius::mergepoint
  - Radius::avoid
- Radius::run : start symbolic execution from the provided state



```rust
use radius::radius::Radius;

fn main() {
    let mut radius = Radius::new("tests/r100");
    let mut state = radius.call_state(0x004006fd);
    let addr: u64 = 0x100000;
    let flag_val = state.symbolic_value("flag", 12*8);
    state.memory.write_value(addr, &flag_val, 12);
    state.registers.set("rdi", state.concrete_value(addr, 64));

    radius.breakpoint(0x004007a1);
    radius.avoid(&[0x00400790]);
    let mut new_state = radius.run(state, 1).unwrap();
    let flag = new_state.evaluate_string_value(&flag_val).unwrap();
    println!("FLAG: {}", flag);
    assert_eq!(flag, "Code_Talkers");
}
```

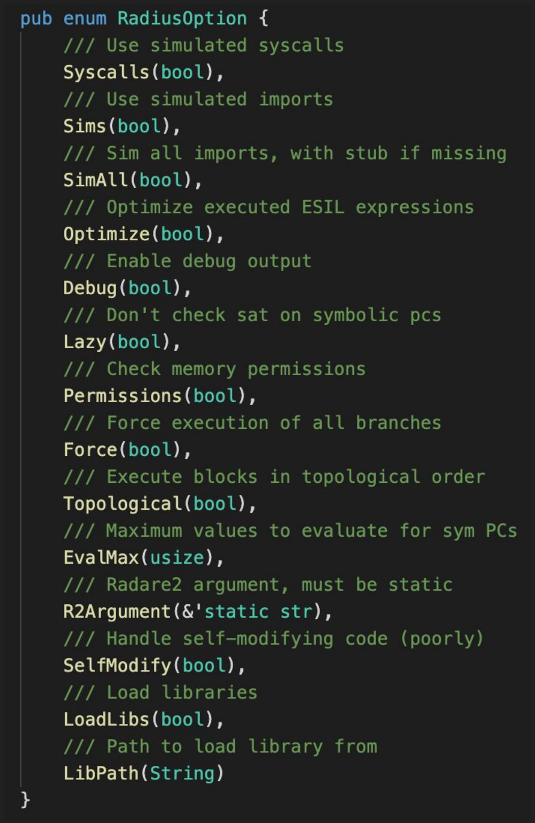
The radius API: by example

- State::symbolic_value
- State::concrete_value
- State::memory_* : interact with state memory, reading, writing, searching and comparing
- Radius::simulate : add a Sim, a replacement for a function (scanf in the example)
- Radius::hook : hook at an address
- State.context : HashMap of strings to Values so the user can keep track of arbitrary values



```rust
use radius::radius::{Radius, RadiusOption};
use radius::state::State;
use radius::value::Value;

// simulates the scanf("%d", dst) calls with sym inputs
fn scanf_sim(state: &mut State, args: &[Value]) -> Value {
    let input_len = state.context.entry("ints".to_owned()).or_insert(vec!()).len();
    let new_int = state.symbolic_value(&format!("int{}", input_len), 32);
    state.memory_write_value(&args[1], &new_int, 4);
    state.context.get_mut("ints").unwrap().push(new_int);
    state.concrete_value(1, 64)
}

fn main() {
    // runs better without opt and default sims
    let options = [RadiusOption::Optimize(false), RadiusOption::Sims(false)];
    let mut radius = Radius::new_with_options(Some("tests/baby-re"), &options);
    let main = radius.get_address("main").unwrap();
    let scanf = radius.get_address("sym.imp.__isoc99_scanf").unwrap();

    // register the custom sim
    radius.simulate(scanf, scanf_sim);

    let state = radius.call_state(main); // start at main
    let new_state = radius.run_until(state, 0x004028e9, &[0x00402941]).unwrap();

    // solving takes the majority of the ~5 sec runtime
    let mut flag_bytes = vec!(); // the hook writes the flag bytes, collect them
    for input in new_state.context.get("ints").unwrap() {
        flag_bytes.push(new_state.solver.eval_to_u64(&input).unwrap() as u8);
    }
    let flag = String::from_utf8(flag_bytes).unwrap();

    println!("FLAG: {}", flag);
    assert_eq!(flag, "Math is hard!");
}
```

radius Options



```
pub enum RadiusOption {
    /// Use simulated syscalls
    Syscalls(bool),
    /// Use simulated imports
    Sims(bool),
    /// Sim all imports, with stub if missing
    SimAll(bool),
    /// Optimize executed ESIL expressions
    Optimize(bool),
    /// Enable debug output
    Debug(bool),
    /// Don't check sat on symbolic pcs
    Lazy(bool),
    /// Check memory permissions
    Permissions(bool),
    /// Force execution of all branches
    Force(bool),
    /// Execute blocks in topological order
    Topological(bool),
    /// Maximum values to evaluate for sym PCs
    EvalMax(usize),
    /// Radare2 argument, must be static
    R2Argument(&'static str),
    /// Handle self-modifying code (poorly)
    SelfModify(bool),
    /// Load libraries
    LoadLibs(bool),
    /// Path to load library from
    LibPath(String)
}
```

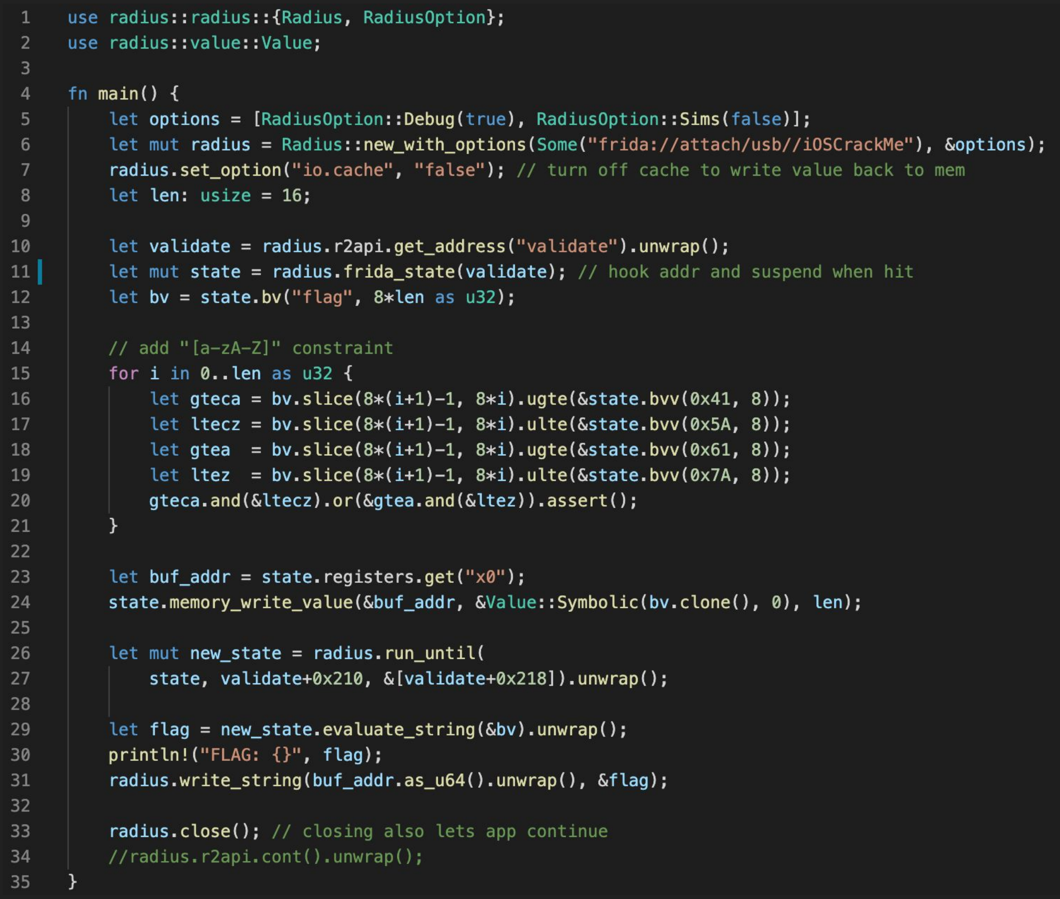
- Another difference between radius and ESILSolve is that radius is Lazy by default
- This is because executing instructions is a lot faster than checking satisfiability
- A properly defined set of merge and avoid points will lead to rapid execution

Taint Analysis

- radius also provides the ability to do simple taint analysis
- Taint analysis is much faster that full symbolic execution but is much less reliable as it may often give false positives when values are mixed in complicated ways and taints are not properly cleared
- Symbolic execution is cooler

Frida Support

- radius can be initialized from a real state using frida or a debugger
- Simply use the right URI scheme in the file name
- Then to initialize the state from frida simply call Radius::frida_state(addr)
- The app will be suspended during the execution



```
use radius::radius::{Radius, RadiusOption};
use radius::value::Value;

fn main() {
    let options = [RadiusOption::Debug(true), RadiusOption::Sims(false)];
    let mut radius = Radius::new_with_options(Some("frida://attach/usb//iOSCrackMe"), &options);
    radius.set_option("io.cache", "false"); // turn off cache to write value back to mem
    let len: usize = 16;

    let validate = radius.r2api.get_address("validate").unwrap();
    let mut state = radius.frida_state(validate); // hook addr and suspend when hit
    let bv = state.bv("flag", 8*len as u32);

    // add "[a-zA-Z]" constraint
    for i in 0..len as u32 {
        let gteca = bv.slice(8*(i+1)-1, 8*i).ugte(&state.bvv(0x41, 8));
        let ltecz = bv.slice(8*(i+1)-1, 8*i).ulte(&state.bvv(0x5A, 8));
        let gtea  = bv.slice(8*(i+1)-1, 8*i).ugte(&state.bvv(0x61, 8));
        let ltez  = bv.slice(8*(i+1)-1, 8*i).ulte(&state.bvv(0x7A, 8));
        gteca.and(&ltecz).or(&gtea.and(&ltez)).assert();
    }

    let buf_addr = state.registers.get("x0");
    state.memory_write_value(&buf_addr, &Value::Symbolic(bv.clone(), 0), len);

    let mut new_state = radius.run_until(
        state, validate+0x210, &[validate+0x218]).unwrap();

    let flag = new_state.evaluate_string(&bv).unwrap();
    println!("FLAG: {}", flag);
    radius.write_string(buf_addr.as_u64().unwrap(), &flag);

    radius.close(); // closing also lets app continue
    //radius.r2api.cont().unwrap();
}
```

Frida demo

<do some cooler stuff>

Plans for the Future

- Improve architecture coverage and fidelity
- More sims to better execute standard library functions in a performant way
- Integrate radius with LibAFL to make a binary-only fuzzer guided by symbolic execution
- Better DEX support to fully symbolically execute Android apps and seamlessly transition from Java to native code.

Thanks

- Thanks to pancake and the whole radare2 community!