The ROPER 2 Engine Loader

May 6, 2018

Contents

1 The Engine Structure

```
<<bring the engine loader's dependencies into scope>>
  pub struct Engine {
      pub uc: Box<unicorn::Unicorn>,
      pub arch: Arch,
      pub regids: Vec<i32>,
      mem: MemImage,
      writeable_bak: Option<MemImage>,
      default_uc_mode: unicorn::Mode,
      saved_context: unicorn::Context,
  }
```

```
impl Engine {
    pub fn new(arch: Arch) -> Self {
        let (_uc_arch, uc_mode) = arch.as_uc();
        let mut mem: MemImage = mem_image_deep_copy();
        let emu = init_emulator(arch, &mut mem, false).unwrap();
        let regids = match arch {
            Arch::Arm(_) => regids(&ARM_REGISTERS),
            Arch::Mips(_) => regids(&MIPS_REGISTERS),
            Arch::X86(Mode::Bits64) => regids(&X86_64_REGISTERS),
            _ => unreachable!("Not implemented"),
        };
        let mut emu = Engine {
            uc: emu,
            arch: arch,
            mem: mem,
            regids: regids,
            default_uc_mode: uc_mode,
            saved_context: unicorn::Context::new(),
            writeable_bak: None,
        };
        emu.save_state();
        emu
    }

    /// Saves the register context.
    pub fn save_context(&mut self) -> Result<(), unicorn::Error> {
        match self.uc.context_save() {
            Ok(c) => {
                self.saved_context = c;
                Ok(())
            }
            Err(e) => Err(e),
        }
    }

    /// Saves both the register context and the state of writeable memory.
    pub fn save_state(&mut self) -> Result<(), unicorn::Error> {
        self.writeable_bak = Some(self.writeable_memory());
        self.save_context()
    }

    /// Restores the register context and the state of writeable memory to
    /// their state at the last save_state() event.
    pub fn restore_state(&mut self) -> Result<(), unicorn::Error> {
        for seg in self.writeable_bak.as_ref().unwrap() {
            self.uc.mem_write(seg.aligned_start(), &seg.data);
        }
        self.restore_context()
```

```
}

/// Restores the register context.
pub fn restore_context(&mut self) -> Result<(), unicorn::Error> {
    self.uc.context_restore(&self.saved_context)
}
<<workaround for an ARM emulator bug with a hard reset method>>

pub fn mem_write(&mut self, addr: u64, data: &Vec<u8>) -> Result<(), unicorn::Error> {
    self.uc.mem_write(addr, data)
}

pub fn uc_mode(&self) -> unicorn::Mode {
    let q = self.uc.query(unicorn::Query::MODE);
    match q {
        Ok(n) => umode_from_usize(n),
        Err(_) => self.default_uc_mode,
    }
}

pub fn mode(&self) -> Mode {
    match self.uc_mode() {
        unicorn::Mode::LITTLE_ENDIAN => Mode::Arm,
        unicorn::Mode::THUMB => Mode::Thumb,
        unicorn::Mode::MODE_64 => Mode::Bits64,
        unicorn::Mode::MODE_32 => Mode::Bits32,
        unicorn::Mode::MODE_16 => Mode::Bits16,
        _ => panic!("***** UNIMPLEMENTED! ****"),
    }
}

pub fn risc_width(&self) -> usize {
    if self.uc_mode() == unicorn::Mode::THUMB {
        2
    } else {
        4
    }
}

pub fn find_stack(&self) -> (u64, usize) {
    let regions = self.uc.mem_regions().unwrap();
    let mut bottom: Option<u64> = None;
    let mut stack: Option<MemRegion> = None;
    for region in regions.iter() {
        if region.perms.intersects(PROT_READ | PROT_WRITE)
            && region.begin >= bottom.unwrap_or(0)
        {
            bottom = Some(region.begin);
            stack = Some(region.clone());
```

```
        };
    }
    let stack = stack.expect(&format!(
        "[!] Could not find stack bottom! Regions: {:?}",
        regions
    ));
    (stack.begin, (stack.end - stack.begin) as usize)
}

pub fn set_sp(&mut self, val: u64) -> Result<(), Error> {
    match self.arch {
        Arch::Arm(_) => {
            let sp = RegisterARM::SP as i32;
            self.uc.reg_write(sp, val)
        }
        Arch::Mips(_) => {
            let sp = RegisterMIPS::SP as i32;
            self.uc.reg_write(sp, val)
        }
        Arch::X86(Mode::Bits64) => {
            let sp = RegisterX86::RSP as i32;
            self.uc.reg_write(sp, val)
        }
        _ => unreachable!("Not implemented"),
    }
}

pub fn writeable_memory(&self) -> MemImage {
    let mut wmem = Vec::new();
    for rgn in self.uc
        .mem_regions()
        .unwrap()
        .iter()
        .filter(|r| r.perms.intersects(PROT_WRITE))
    {
        let data: Vec<u8> = self.uc
            .mem_read(rgn.begin, (rgn.end - rgn.begin) as usize)
            .unwrap();
        wmem.push(Seg {
            addr: rgn.begin,
            perm: rgn.perms,
            memsz: (rgn.end - rgn.begin) as usize,
            data: data,
            segtype: SegType::Load,
        });
    }
    wmem
}
```

```
pub fn start(
    &mut self,
    begin: u64,
    until: u64,
    timeout: u64,
    count: usize,
) -> Result<(), Error> {
    self.uc.emu_start(begin, until, timeout, count)
}

pub fn remove_hook(&mut self, uc_hook: unicorn::uc_hook) -> Result<(), Error> {
    self.uc.remove_hook(uc_hook)
}

pub fn add_code_hook<F>(
    &mut self,
    hooktype: unicorn::CodeHookType,
    start_addr: u64,
    stop_addr: u64,
    callback: F,
) -> Result<unicorn::uc_hook, Error>
where
    F: Fn(&Unicorn, u64, u32) -> () + 'static,
{
    self.uc
        .add_code_hook(hooktype, start_addr, stop_addr, callback)
}

pub fn read_general_registers(&self) -> Result<Vec<u64>, Error> {
    Ok(self.regids
        .iter()
        .map(|&x| self.uc.reg_read(x).expect("Error reading registers"))
        .collect::<Vec<u64>>())
}

/// Limitation: Only returns the bounds of the largest executable
/// segment.
pub fn exec_mem_range(&self) -> (Option<u64>, Option<u64>) {
    let regions = self.uc.mem_regions().unwrap();
    let mut exec_start = None;
    let mut exec_stop = None;
    for region in regions {
        if !region.perms.intersects(PROT_EXEC) {
            continue;
        }
        if exec_start == None || region.begin < exec_start.unwrap() {
            exec_start = Some(region.begin)
        };
        if exec_stop == None || region.end > exec_stop.unwrap() {
```

```
            exec_stop = Some(region.end)
        };
    }
    (exec_start, exec_stop)
}

pub fn hook_exec_mem<F>(&mut self, callback: F) -> Result<unicorn::uc_hook, Error>
where
    F: Fn(&Unicorn, u64, u32) -> () + 'static,
{
    let (exec_start, exec_stop) = self.exec_mem_range();
    if exec_start == None || exec_stop == None {
        Err(unicorn::Error::ARG)
    } else {
        //println!("> exec_start: {:08x}, exec_stop: {:08x}", exec_start.unwrap(), exec_stop.unwrap(
        self.uc.add_code_hook(
            unicorn::CodeHookType::CODE,
            exec_start.unwrap(),
            exec_stop.unwrap(),
            callback,
        )
    }
}

pub fn hook_rets<F>(&mut self, callback: F) -> Result<unicorn::uc_hook, unicorn::Error>
where
    F: Fn(&Unicorn, u64, u32) -> () + 'static,
{
    //let (exec_start, exec_stop) = self.exec_mem_range();
    let arch = ARCHITECTURE.with_mode(self.mode());
    match arch {
        Arch::X86(_) => {
            /* KLUDGE -- not sure why instruction hooking won't work here. */
            let _callback = move |uc: &Unicorn, addr, size| {
                let pc = addr;
                if size != 1 {
                    return;
                };
                let bytecode = uc.mem_read(pc, 1); /* ret on x86 is C3 */
                match bytecode {
                    Ok(v) => if v[0] == X86_RET {
                        callback(uc, addr, size)
                    } else {
                        ()
                    },
                    _ => (),
                }
            };
            self.hook_exec_mem(_callback)
```

```
            /*
            self.uc.add_x86_insn_hook(
            unicorn::x86_const::InsnX86::RET,
            exec_start.unwrap(),
            exec_stop.unwrap(),
            callback),
            */
        }
        Arch::Arm(Mode::Arm) => {
            let _callback = move |uc: &Unicorn, addr, size| {
                let pc = addr; //read_pc(uc).unwrap();
                let bytecode = uc.mem_read(pc, 4);
                match bytecode {
                    Ok(v) => if arm_ret(&v) {
                        callback(uc, addr, size)
                    } else {
                        ()
                    },
                    Err(_) => panic!("Failed to read instruction"),
                }
            };
            self.hook_exec_mem(_callback)
        }
        Arch::Arm(Mode::Thumb) => {
            let _callback = move |uc: &Unicorn, addr, size| {
                let pc = addr; //read_pc(uc).unwrap();
                let bytecode = uc.mem_read(pc, 2);
                match bytecode {
                    Ok(v) => if thumb_ret(&v) {
                        callback(uc, addr, size)
                    } else {
                        ()
                    },
                    Err(_) => panic!("Failed to read instruction"),
                }
            };
            self.hook_exec_mem(_callback)
        }
        _ => panic!("Unimplemented. Will need to tinker with unicorn-rs a bit."),
    }
}

pub fn hook_indirect_jumps<F>(
    &mut self,
    callback: F,
) -> Result<unicorn::uc_hook, unicorn::Error>
where
    F: Fn(&Unicorn, u64, u32) -> () + 'static,
{
```

```
        let arch = ARCHITECTURE.with_mode(self.mode());
        match arch {
            Arch::X86(_) => {
                let _callback = move |uc: &Unicorn, addr: u64, size: u32| {
                    let size = u32::min(size, 15);
                    let bytecode = uc.mem_read(addr, size as usize);
                    match bytecode {
                        /* TODO Better indirect jump detector! */
                        Ok(v) => if v[0] == 0xFF {
                            callback(uc, addr, size)
                        } else {
                            ()
                        },
                        Err(_) => println!("Failed to read instruction! {:?}", bytecode),
                    }
                };
                self.hook_exec_mem(_callback)
            }
            _ => panic!("hook_jumps not yet implemented for this architecture"),
        }
    }

    pub fn hook_writeable_mem<F>(&mut self, callback: F) -> Result<unicorn::uc_hook, Error>
    where
        F: Fn(&Unicorn, unicorn::MemType, u64, usize, i64) -> bool + 'static,
    {
        let writeable = self.writeable_memory();
        let mut begin = None;
        let mut end = None;
        for seg in &writeable {
            let b = seg.aligned_start();
            let e = seg.aligned_end();
            if begin == None || b < begin.unwrap() {
                begin = Some(b)
            };
            if end == None || e > end.unwrap() {
                end = Some(e)
            };
        }
        assert!(begin != None && end != None);
        self.uc.add_mem_hook(
            unicorn::MemHookType::MEM_WRITE,
            begin.unwrap(),
            end.unwrap(),
            callback,
        )
    }
}
```

2 A Few Unicorn Utilities

What follows are a handful of convenience functions for interacting with the Unicorn emulator. They are not defined, here, as methods on the `Engine` trait, because we will often have reason to use them in the callbacks we hook into the engine, and those callbacks only have access to the naked `Unicorn` structure, and not to our `Engine` wrapper.

```
/// Returns the regid for the program counter, on the
/// current ARCHITECTURE (wrt static variable)
pub fn whats_pc() -> i32 {
    match *ARCHITECTURE {
        Arch::Arm(_) => RegisterARM::PC.to_i32(),
        Arch::Mips(_) => RegisterMIPS::PC.to_i32(),
        Arch::X86(Mode::Bits64) => RegisterX86::RIP.to_i32(),
        Arch::X86(Mode::Bits32) => RegisterX86::EIP.to_i32(),
        Arch::X86(Mode::Bits16) => RegisterX86::IP.to_i32(),
        _ => panic!("unimplemented"),
    }
}

/// Reads the program counter. Architecture independent. Raw Unicorn needed.
/// Suited for callbacks.
pub fn read_pc(uc: &Unicorn) -> Result<u64, unicorn::Error> {
    uc.reg_read(whats_pc())
}

/// Returns the default accumulator register's identifier.
/// For ARM, this is R0, and for x86_64, this is RAX.
pub fn whats_accum() -> i32 {
    match *ARCHITECTURE {
        Arch::Arm(_) => RegisterARM::R0.to_i32(),
        Arch::X86(Mode::Bits64) => RegisterX86::RAX.to_i32(),
        Arch::X86(Mode::Bits32) => RegisterX86::EAX.to_i32(),
        Arch::X86(Mode::Bits16) => RegisterX86::AX.to_i32(),
        _ => panic!("not yet implemented"),
    }
}

pub fn uc_general_registers(uc: &Unicorn) -> Result<Vec<u64>, unicorn::Error> {
    /* FIXME: optimize away this match, refer to a static instead */
    let regids = match *ARCHITECTURE {
        Arch::Arm(_) => regids(&ARM_REGISTERS),
        Arch::Mips(_) => regids(&MIPS_REGISTERS),
        Arch::X86(Mode::Bits64) => regids(&X86_64_REGISTERS),
        _ => unreachable!("Not implemented"),
    };
    Ok(regids
```

```
        .iter()
        .map(|&x| uc.reg_read(x).expect("Error reading registers"))
        .collect::<Vec<u64>>())
}
// TODO /// Converts a unicorn register id to a capstone one
// pub fn uc2cs_reg(

pub fn regids<T>(regs: &'static [T]) -> Vec<i32>
where
    T: Register,
{
    regs.iter().map(|x| x.to_i32()).collect::<Vec<i32>>()
}

pub fn mem_image_deep_copy() -> MemImage {
    let mut mi = Vec::new();
    for seg in MEM_IMAGE.to_vec() {
        mi.push(seg.deep_copy())
    }
    mi
}

fn mem_image_to_mem_table() -> Vec<(u64, usize, unicorn::Protection, *mut u8)> {
    let mut table = Vec::new();
    for seg in &*MEM_IMAGE {
        let mut data = seg.data.clone();
        let mut data_ptr = data.as_mut_ptr();
        table.push((seg.aligned_start(), seg.aligned_size(), seg.perm, data_ptr));
    }
    table
}

fn uc_mem_table(emu: &Unicorn) -> Vec<(u64, usize, unicorn::Protection, Vec<u8>)> {
    let mut table = Vec::new();
    for region in emu.mem_regions().unwrap() {
        let begin = region.begin;
        let size = (region.end - region.begin) as usize + 1;
        let perms = region.perms;
        let mut data = emu.mem_read(begin, size).unwrap();
        let mut ptr = data;
        table.push((begin, size, perms, ptr));
    }
    table
}/* from raw Unicorn instance. Useful inside callbacks, for disassembling */
pub fn get_mode(uc: &Unicorn) -> Mode {
    /* TODO keep a global static architecture variable, for reference
     * in situations like these. for now, we're just assuming ARM, but
     * plan to extend the system to cover, at least, MIPS, too.
     */
```

```
    let raw = uc.query(unicorn::Query::MODE);

    match raw {
        Ok(0b00000) => Mode::Arm,
        Ok(0b10000) => Mode::Thumb,
        Ok(0b01000) => Mode::Bits64,
        Ok(0b00100) => Mode::Bits32,
        Ok(0b00010) => Mode::Bits16,
        Err(_) => ARCHITECTURE.mode(), /* global default */
        _ => panic!("Mode not recognized"),
    }
}

pub fn init_emulator(
    archmode: Arch,
    mem: &mut MemImage,
    unsafely: bool,
) -> Result<Box<Unicorn>, unicorn::Error> {
    let (arch, mode) = archmode.as_uc();

    let uc = Unicorn::new(arch, mode)?;

    for seg in mem {
        if unsafely {
            unsafe {
                uc.mem_map_ptr(
                    seg.aligned_start(),
                    seg.aligned_size(),
                    seg.perm,
                    seg.data.as_mut_ptr(),
                )
            }?;
        } else {
            uc.mem_map(seg.aligned_start(), seg.aligned_size(), seg.perm)?;
            uc.mem_write(seg.aligned_start(), &seg.data);
        }
    }
    Ok(uc)
}

pub fn align_inst_addr(addr: u64, mode: Mode) -> u64 {
    match mode {
        Mode::Arm | Mode::Le | Mode::Be => addr & 0xFFFFFFFC,
        Mode::Thumb => (addr & 0xFFFFFFFE) | 1,
        Mode::Bits16 => addr & 0xFFFF,
        Mode::Bits32 => addr & 0xFFFFFFFF,
        Mode::Bits64 => addr & 0xFFFFFFFFFFFFFFFF,
    }
}
```

```
pub fn calc_sp_delta(addr: u64, mode: Mode) -> usize {
    let arch_mode = ARCHITECTURE.with_mode(mode);
    /* TODO ! */
    match arch_mode {
        Arch::X86(Mode::Bits64) => x86_64_calc_sp_delta(addr),
        Arch::Arm(Mode::Arm) => arm_calc_sp_delta(addr),
        Arch::Arm(Mode::Thumb) => thumb_calc_sp_delta(addr),
        _ => panic!("unimplemented sp_delta arch/mode"),
    }
}

fn x86_64_calc_sp_delta(_addr: u64) -> usize {
    /* use capstone to disasm_count 1 instruction from addr */
    /* inspect operands and implicit writes, to gauge effect on RSP */
    0
}

fn arm_calc_sp_delta(_addr: u64) -> usize {
    0
}

fn thumb_calc_sp_delta(_addr: u64) -> usize {
    0
}
```

3 Architectures and Hardware Modes

```
#[derive(FromValue, IntoValue, ForeignValue, Clone, Copy, Debug, PartialEq, Eq)]
pub enum Mode {
    Arm,
    Thumb,
    Be,
    Le,
    Bits16,
    Bits32,
    Bits64,
}

impl Mode {
    pub fn as_uc(&self) -> unicorn::Mode {
        match self {
            &Mode::Arm => unicorn::Mode::LITTLE_ENDIAN,
            &Mode::Thumb => unicorn::Mode::THUMB,
            &Mode::Be => unicorn::Mode::BIG_ENDIAN,
            &Mode::Le => unicorn::Mode::LITTLE_ENDIAN,
            &Mode::Bits16 => unicorn::Mode::MODE_16,
```

```
            &Mode::Bits32 => unicorn::Mode::MODE_32,
            &Mode::Bits64 => unicorn::Mode::MODE_64,
        }
    }
}

#[derive(FromValue, IntoValue, ForeignValue, Clone, Copy, PartialEq, Eq, Debug)]
pub enum Arch {
    Arm(Mode),
    Mips(Mode),
    X86(Mode),
}

impl Arch {
    pub fn as_uc(&self) -> (unicorn::Arch, unicorn::Mode) {
        match self {
            &Arch::Arm(ref m) => (unicorn::Arch::ARM, m.as_uc()),
            &Arch::Mips(ref m) => (unicorn::Arch::MIPS, m.as_uc()),
            &Arch::X86(ref m) => (unicorn::Arch::X86, m.as_uc()),
        }
    }
    pub fn mode(&self) -> Mode {
        match self {
            &Arch::Arm(ref m) => m.clone(),
            &Arch::Mips(ref m) => m.clone(),
            &Arch::X86(ref m) => m.clone(),
        }
    }
    /// Returns a new Arch enum with specified mode
    pub fn with_mode(&self, mode: Mode) -> Arch {
        match self {
            &Arch::Arm(_) => Arch::Arm(mode),
            &Arch::Mips(_) => Arch::Mips(mode),
            &Arch::X86(_) => Arch::X86(mode),
        }
    }
    //pub fn as_cs(&self) -> capstone::
}
```

The `unicorn-rs` library of bindings for the *Unicorn* emulator has the peculiarity that, while it defines a `Mode` enum, just as we do here, its mode query method returns an integer (`usize`) value, which the library declines to decode for us. This little function handles that end of things.

```
pub fn umode_from_usize(x: usize) -> unicorn::Mode {
    match x {
        0 => unicorn::Mode::LITTLE_ENDIAN,
        2 => unicorn::Mode::MODE_16,
        4 => unicorn::Mode::MODE_32,
```

```
        8 => unicorn::Mode::MODE_64,
        16 => unicorn::Mode::THUMB,
        32 => unicorn::Mode::MCLASS,
        64 => unicorn::Mode::V8,
        0x40000000 => unicorn::Mode::BIG_ENDIAN,
        _ => unicorn::Mode::LITTLE_ENDIAN,
    }
}
```

Note that `unicorn-rs` uses `Mode::LITTLE_ENDIAN` to signify ARM mode, which can sometimes be a bit confusing.

4 Segments and Memory Images

4.1 The Segment Type

```
#[derive(FromValue, IntoValue, ForeignValue, Copy, Clone, PartialEq, Eq, Debug)]
pub enum SegType {
    Null,
    Load,
    Dynamic,
    Interp,
    Note,
    ShLib,
    PHdr,
    Tls,
    GnuEhFrame,
    GnuStack,
    GnuRelRo,
    Other, /* KLUDGE: a temporary catchall */
}

impl SegType {
    fn new(raw: u32) -> Self {
        match raw {
            0 => SegType::Null,
            1 => SegType::Load,
            2 => SegType::Dynamic,
            3 => SegType::Interp,
            4 => SegType::Note,
            5 => SegType::ShLib,
            6 => SegType::PHdr,
            7 => SegType::Tls,
            0x6474e550 => SegType::GnuEhFrame,
            0x6474e551 => SegType::GnuStack,
            0x6474e552 => SegType::GnuRelRo,
            _ => SegType::Other,
        }
    }
```

```
    pub fn loadable(&self) -> bool {
        match self {
            &SegType::Load => true,
            _ => false,
        }
    }
}

pub type Perm = unicorn::Protection;

#[derive(FromValue, IntoValue, ForeignValue, PartialEq, Eq, Debug, Clone)]
pub struct Seg {
    pub addr: u64,
    pub memsz: usize,
    pub perm: Perm,
    pub segtype: SegType,
    pub data: Vec<u8>,
}

impl Seg {
    pub fn deep_copy(&self) -> Seg {
        Seg {
            addr: self.addr,
            memsz: self.memsz,
            perm: self.perm,
            segtype: self.segtype,
            data: self.data.clone(),
        }
    }

    pub fn from_phdr(phdr: &elf::ProgramHeader) -> Self {
        let mut uc_perm = PROT_NONE;
        if phdr.is_executable() {
            uc_perm |= PROT_EXEC
        };
        if phdr.is_write() {
            uc_perm |= PROT_WRITE
        };
        if phdr.is_read() {
            uc_perm |= PROT_READ
        };
        let mut s = Seg {
            addr: phdr.vm_range().start as u64,
            memsz: (phdr.vm_range().end - phdr.vm_range().start) as usize,
            perm: uc_perm,
            segtype: SegType::new(phdr.p_type),
            data: Vec::new(),
        };
        let size = (s.aligned_end() - s.aligned_start()) as usize;
```

```
        s.data = vec![UNINITIALIZED_BYTE; size];
        s
    }

    pub fn is_executable(&self) -> bool {
        self.perm.intersects(PROT_EXEC)
    }

    pub fn is_writeable(&self) -> bool {
        self.perm.intersects(PROT_WRITE)
    }

    pub fn is_readable(&self) -> bool {
        self.perm.intersects(PROT_READ)
    }

    pub fn aligned_start(&self) -> u64 {
        self.addr & 0xFFFFF000
    }
    pub fn aligned_end(&self) -> u64 {
        (self.addr + (self.memsz as u64) + 0x1000) & 0xFFFFF000
    }
    pub fn aligned_size(&self) -> usize {
        ((self.addr as usize & 0x0FFF) + self.memsz as usize + 0x1000) & 0xFFFFF000
    }
    pub fn loadable(&self) -> bool {
        self.segtype.loadable()
    }
}

impl Display for Seg {
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        write!(
            f,
            "[aligned {:08x} -- {:08x}: {:?}]",
            self.aligned_start(),
            self.aligned_end(),
            self.perm
        )
    }
}
```

4.2 The Static Memory Map

One of the design decisions that separates Return-Oriented Programming with ROPER: The Secret of the Ooze (ROPER2) from Return-Oriented Programming with ROPER (ROPER) is to make generous use of *immutable* global variables, rather than clumsily passing references to large, unwieldy

parameter structs. When information that should, conceptually, be universally accessible within the program has to be passed around by weaving it in and out of function arguments and return values, the result is frequently a tangled mess that becomes increasingly difficult to modify as the number of would-be-globals increases. So, I've adopted the relatively unpopular strategy of just baptising those would-be-globals as actual globals. Some of these globals need to be calculated before being assigned, or read from configuration files or environment variables, and so we can't use ordinary `static` variables. The `lazy_static` macro, fortunately, supplies what we need here.

The most important global variable defined here is the `MEM_IMAGE`, which holds a map of the target executable from which the population of chains will be drawn, and which will be loaded into the emulator engine's memory, where our creatures will be hatched. Keeping this data in a static global variable means that we only need to parse it once, and that we can refer back to it whenever we like, when we need to dereference a pointer, search for a value, or generate a new individual genotype.

```
pub type MemImage = Vec<Seg>;

lazy_static! {
    pub static ref MEM_IMAGE: MemImage
        = {
            let obj = Object::parse(&CODE_BUFFER).unwrap();
            let mut segs: Vec<Seg> = Vec::new();
            match obj {
                Object::Elf(e) => {
                    let mut page_one = false;
                    let shdrs = &e.section_headers;
                    let phdrs = &e.program_headers;
                    for phdr in phdrs {
                        let seg = Seg::from_phdr(&phdr);
                        if seg.loadable() {
                            let start = seg.aligned_start() as usize;
                            if start == 0 { page_one = true };
                            segs.push(seg);
                        }
                    }
                    /* Low memory */
                    if !page_one {
                        segs.push(Seg { addr: 0,
                                        memsz: 0x1000,
                                        perm: PROT_READ,
                                        segtype: SegType::Load,
                                        data: vec![0; 0x1000],
                        });
                    };
```

```
                    for shdr in shdrs {
                        let (i,j) = (shdr.sh_offset as usize,
                                     (shdr.sh_offset+shdr.sh_size) as usize);
                        let aj = usize::min(j, CODE_BUFFER.len());
                        let sdata = CODE_BUFFER[i..aj].to_vec();
                        /* find the appropriate segment */

                        for seg in segs.iter_mut() {
                            if shdr.sh_addr >= seg.aligned_start()
                                && shdr.sh_addr < seg.aligned_end() {
                                let mut v_off
                                    = (shdr.sh_addr - seg.aligned_start()) as usize;
                                for byte in sdata {
                                    if v_off >= seg.data.len() {
                                        println!("[x] v_off 0x{:x} > seg.data.len() 0x{:x}. Look into this
                                        break;
                                    };
                                    seg.data[v_off] = byte;
                                    v_off += 1;
                                }
                                break;
                            }
                        }
                    }
                    /* now allocate the stack */
                    let mut bottom = 0;
                    for seg in &segs {
                        let b = seg.aligned_end();
                        if b > bottom { bottom = b };
                    }
                    segs.push(Seg { addr: bottom,
                                    perm: PROT_READ|PROT_WRITE,
                                    segtype: SegType::Load,
                                    memsz: STACK_SIZE,
                                    data: vec![0; STACK_SIZE]
                    });

                },
                _ => panic!("Not yet implemented."),
            }
            for seg in &segs {
                println!("{}, data len: {:x}", seg, seg.data.len());
            }
            segs
        };
}
```

4.3 Some utilities for working with the MEM_IMAGE

```
fn find_static_seg(addr: u64) -> Option<&'static Seg> {
    let mut this_seg = None;
    for seg in MEM_IMAGE.iter() {
        if seg.aligned_start() <= addr && addr < seg.aligned_end() {
            this_seg = Some(seg);
        };
    }
    this_seg
}

pub fn read_static_mem(addr: u64, size: usize) -> Option<Vec<u8>> {
    if let Some(seg) = find_static_seg(addr) {
        let offset = (addr - seg.aligned_start()) as usize;
        let offend = usize::min(offset + size, seg.data.len());
        if offend < offset {
            return None;
        };
        if offend > seg.data.len() {
            println!("ERROR: addr: {:x}, size: {:x}, offset = {:x}, offend = {:x}, seg.data.len() = {:x}",
                     addr, size, offset, offend, seg.data.len());
            println!("this seg: {}", seg);
            for seg in MEM_IMAGE.iter() {
                println!("{}", seg);
            }
            panic!("Index error!");
        }
        let offend = usize::min(offend, seg.data.len());
        Some(seg.data[offset..offend].to_vec())
    } else {
        None
    }
}
```

5 Some tests

```
#[test]
fn test_engine_new() {
    let emu = Engine::new(*ARCHITECTURE);
}

#[test]
fn test_engine_reset() {
    let mut emu = Engine::new(*ARCHITECTURE);
    let mem1 = emu.writeable_memory();
    let rgn1 = emu.uc.mem_regions().unwrap();
    println!("About to reset...");
```

```
    emu.hard_reset();
    let mem2 = emu.writeable_memory();
    let rgn2 = emu.uc.mem_regions().unwrap();
    assert_eq!(mem1, mem2);
    for (r1, r2) in rgn1.iter().zip(&rgn2) {
        assert_eq!(r1.perms, r2.perms);
        assert_eq!(r1.begin, r2.begin);
        assert_eq!(r1.end, r2.end);
    }
}
```

Some early testing revealed an upstream bug in the `tcg.c` (Tiny Code Generator) module in the Unicorn library, where it generates Advanced RISC Machine (ARM) machine code. This bug would not appear unless the engine was put under considerable strain (as it is, invariably, when working for ROPER2). If the same ARM emulator instance is mapped with more than a few bytes of memory (as it tends to be in the unit and integration tests included in the `unicorn` and `unicorn-rs` libraries), and then reused for multiple executions (i.e., multiple calls to `emu_start()`), it will generate what appears to be a stack overflow, and throw a segfault. At the time of writing (at which the current release of Unicorn is Version 1.0.1), Unicorn still suffers from this bug, though an issue has been raised with the developer. The following test is enough to trigger the bug:

```
#[test]
fn stress_test_unicorn_cpu_arm() {
    if let Arch::Arm(_) = *ARCHITECTURE {
        let mode = unicorn::Mode::LITTLE_ENDIAN;
        let mut uc = CpuARM::new(mode).expect("Failed to create CpuARM");
        let mem_image: MemImage = MEM_IMAGE.to_vec();
        for seg in mem_image {
            uc.mem_map(seg.aligned_start(), seg.aligned_size(), seg.perm)
                .unwrap();
            uc.mem_write(seg.aligned_start(), &seg.data).unwrap();
        }
        let mut rng = thread_rng();
        for i in 0..1000000 {
            //println!("{}",i);
            uc.emu_start(0x8000 + rng.gen::<u64>() % 0x30000, 0, 0, 1024);
        }
    }
}
```

Incidentally, this is why we have the otherwise unnecessary `hard_reset()` method implemented for our `Engine` struct. As a clumsy workaround, we can just perform a hard reset on the engine – throwing out its current Unicorn instance, and generating a fresh one, while transferring over the necessary

context – every execution or two. This lets us do all the things we want to with the ARM emulator, bug notwithstanding, albeit at the cost of an order of magnitude in runtime.

```
/* method for the Engine trait */
  pub fn hard_reset(&mut self) -> () {
      self.save_state();
      let (uc_arch, uc_mode) = self.arch.as_uc();
      let uc = unicorn::Unicorn::new(uc_arch, uc_mode).unwrap();
      for seg in &self.mem {
          uc.mem_map(seg.aligned_start(), seg.aligned_size(), seg.perm)
              .unwrap();
          uc.mem_write(seg.aligned_start(), &seg.data).unwrap();
      }
      self.uc = uc;
      self.restore_state();
  }

#[test]
fn stress_test_unicorn_cpu_x86_64() {
    if let Arch::X86(_) = *ARCHITECTURE {
        let mode = unicorn::Mode::MODE_64;
        let mut uc = CpuX86::new(mode).expect("Failed to create CpuX86");
        let mem_image: MemImage = MEM_IMAGE.to_vec();
        for seg in mem_image {
            uc.mem_map(seg.aligned_start(), seg.aligned_size(), seg.perm)
                .unwrap();
            uc.mem_write(seg.aligned_start(), &seg.data).unwrap();
        }
        let mut rng = thread_rng();
        for i in 0..1000000 {
            //println!("{}",i);
            uc.emu_start(0x8000 + rng.gen::<u64>() % 0x30000, 0, 0, 1024);
        }
    }
}
```

6 Some Useful Constants and Predicates

6.1 General Registers for the Architectures Supported

```
pub static MIPS_REGISTERS: [RegisterMIPS; 33] = [
    RegisterMIPS::PC,
    RegisterMIPS::ZERO,
    RegisterMIPS::AT,
    RegisterMIPS::V0,
    RegisterMIPS::V1,
    RegisterMIPS::A0,
```

```
    RegisterMIPS::A1,
    RegisterMIPS::A2,
    RegisterMIPS::A3,
    RegisterMIPS::T0,
    RegisterMIPS::T1,
    RegisterMIPS::T2,
    RegisterMIPS::T3,
    RegisterMIPS::T4,
    RegisterMIPS::T5,
    RegisterMIPS::T6,
    RegisterMIPS::T7,
    RegisterMIPS::S0,
    RegisterMIPS::S1,
    RegisterMIPS::S2,
    RegisterMIPS::S3,
    RegisterMIPS::S4,
    RegisterMIPS::S5,
    RegisterMIPS::S6,
    RegisterMIPS::S7,
    RegisterMIPS::T8,
    RegisterMIPS::T9,
    RegisterMIPS::K0,
    RegisterMIPS::K1,
    RegisterMIPS::GP,
    RegisterMIPS::SP,
    RegisterMIPS::FP,
    RegisterMIPS::RA,
];

pub static ARM_REGISTERS: [RegisterARM; 16] = [
    RegisterARM::R0,
    RegisterARM::R1,
    RegisterARM::R2,
    RegisterARM::R3,
    RegisterARM::R4,
    RegisterARM::R5,
    RegisterARM::R6,
    RegisterARM::R7,
    /****************/ RegisterARM::R8,
    /****************/ RegisterARM::SB,
    /* Not used in  */ RegisterARM::SL,
    /* Thumb Mode   */ RegisterARM::FP,
    /****************/ RegisterARM::IP,
    /****************/ RegisterARM::SP,
    RegisterARM::LR,
    RegisterARM::PC,
];

pub static X86_64_REGISTERS: [RegisterX86; 17] = [
```

```
    RegisterX86::RAX,
    RegisterX86::RBX,
    RegisterX86::RCX,
    RegisterX86::RDX,
    RegisterX86::RDI,
    RegisterX86::RSI,
    RegisterX86::R9,
    RegisterX86::R10,
    RegisterX86::R11,
    RegisterX86::R12,
    RegisterX86::R13,
    RegisterX86::R14,
    RegisterX86::R15,
    RegisterX86::RBP,
    RegisterX86::RSP,
    RegisterX86::RIP,
    RegisterX86::EFLAGS,
];

pub const ARM_ARM: Arch = Arch::Arm(Mode::Arm);
pub const ARM_THUMB: Arch = Arch::Arm(Mode::Thumb);
pub const STACK_SIZE: usize = 0x1000;
pub const UNINITIALIZED_BYTE: u8 = 0x00;

pub const PROT_READ: Perm = unicorn::PROT_READ;
pub const PROT_EXEC: Perm = unicorn::PROT_EXEC;
pub const PROT_WRITE: Perm = unicorn::PROT_WRITE;
pub const X86_RET: u8 = 0xC3;

fn x86_ret(b: &Vec<u8>) -> bool {
    b[0] == X86_RET
}
/* An ARM return is a pop with PC as one of the destination registers */
fn arm_ret(w: &Vec<u8>) -> bool {
    w[3] & 0x0E == 0x06 &&
    w[0] & 0x10 == 0x10 && /* The instruction is a pop instruction, */
    w[1] & 0x80 == 0x80 /* and R15 is a destination register     */
}
fn thumb_ret(w: &Vec<u8>) -> bool {
    w[0] & 0xF6 == 0xB4 && w[0] & 1 == 1
}
```

7 Dependencies

The two most important external crates used in this module are the **goblin** crate, which handles the executable file parsing (in various formats, though we restrict ourselves to Executable and Linkable Format (ELF) binaries for now), and the **unicorn** crate, which provides bindings to the *Unicorn* CPU

emulation engine. **Capstone** is brought into scope to facilitate instruction analysis, where this can't easily be done with a trivial and efficient bitmask operation.

```
extern crate capstone;
extern crate goblin;
extern crate rand;
extern crate unicorn;

use std::fmt::{Display, Formatter};
use std::fmt;
use self::goblin::{elf, Object};
use self::unicorn::*;
use par::statics::*;
```