The ROPER 2 Hatchery

May 4, 2018

Contents

1 The Hatchery

The hatchery module of Return-Oriented Programming with ROPER: The Sequel (ROPER2) consists of two logical components: a mechanism for performing "return-oriented programming (ROP) chain embryogenesis", which maps genotypes to phenotypes so as to prepare them for fitness evaluation and selection (§1.1), and a mechanism to handle the concurrency plumbing for the system – setting up multiple `unicorn` emulator instances on separate, looping threads, with which the rest the system can communicate through a network of `channels`.

1.1 ROP-chain embryogenesis

```
/* I think some of this data cloning could be optimized away. FIXME */
#[inline]
pub fn hatch_cases(creature: &mut gen::Creature, emu: &mut Engine) -> gen::Phenome {
    let mut map = gen::Phenome::new();
    {
        let mut inputs: Vec<gen::Input> = creature.phenome.keys().map(|x| x.clone()).collect();
        while inputs.len() > 0 {
            let input = inputs.pop().unwrap();
            /* This can't really be threaded, due to the unsendability of emu */
            let pod = hatch(creature, &input, emu);
```

```
            map.insert(input.to_vec(), Some(pod));
        }
    }
    map
}

#[inline]
pub fn hatch(creature: &mut gen::Creature, input: &gen::Input, emu: &mut Engine) -> gen::Pod {
    let mut payload = creature.genome.pack(input);
    let start_addr = creature.genome.entry().unwrap();
    /* A missing entry point should be considered an error,
     * since we try to guard against this in our generation
     * functions.
     */
    let (stack_addr, stack_size) = emu.find_stack();
    payload.truncate(stack_size / 2);
    let _payload_len = payload.len();
    let stack_entry = stack_addr + (stack_size / 2) as u64;
    /* save writeable regions **/

    /* load payload **/
    emu.restore_state();
    emu.mem_write(stack_entry, &payload)
        .expect("mem_write fail in hatch");
    emu.set_sp(stack_entry + *ADDR_WIDTH as u64);

    let visitor: Rc<RefCell<Vec<VisitRecord>>> = Rc::new(RefCell::new(Vec::new()));
    let writelog = Rc::new(RefCell::new(Vec::new()));
    let retlog = Rc::new(RefCell::new(Vec::new()));
    let jmplog = Rc::new(RefCell::new(Vec::new()));

    let mem_write_hook = {
        let writelog = writelog.clone();
        let callback = move |uc: &unicorn::Unicorn,
                             _memtype: unicorn::MemType,
                             addr: u64,
                             size: usize,
                             val: i64| {
            let mut wmut = writelog.borrow_mut();
            let pc = read_pc(uc).unwrap();
            let write_record = WriteRecord {
                pc: pc,
                dest_addr: addr,
                value: val as u64,
                size: size,
            };
            wmut.push(write_record);
            true
        };
```

```
        emu.hook_writeable_mem(callback)
    };

    let visit_hook = {
        let visitor = visitor.clone();
        let callback = move |uc: &unicorn::Unicorn, addr: u64, size: u32| {
            let mut vmut = visitor.borrow_mut();
            let mode = get_mode(&uc);
            let size: usize = (size & 0xF) as usize;
            let registers = uc_general_registers(&uc).unwrap();
            let visit_record = VisitRecord {
                pc: addr,
                mode: mode,
                inst_size: size,
                registers: registers,
            };
            vmut.push(visit_record);
        };
        emu.hook_exec_mem(callback)
    };

    let ret_hook = {
        let retlog = retlog.clone();
        let callback = move |_uc: &unicorn::Unicorn, addr: u64, _size: u32| {
            let mut retlog = retlog.borrow_mut();
            let pc = addr;
            retlog.push(pc);
        };
        emu.hook_rets(callback)
    };

    let indirect_jump_hook = {
        let jmplog = jmplog.clone();
        let callback = move |_uc: &unicorn::Unicorn, addr: u64, _size: u32| {
            let mut jmplog = jmplog.borrow_mut();
            jmplog.push(addr);
        };
        emu.hook_indirect_jumps(callback)
    };

    /* Hatch! **/
/* FIXME don't hardcode these params */
    let _res = emu.start(start_addr, 0, 0, 1024);

    /* Now, clean up the hooks */
    match visit_hook {
        Ok(h) => {
            emu.remove_hook(h).unwrap();
        }
```

```
        Err(e) => {
            println!("visit_hook didn't take {:?}", e);
        }
    }
    match mem_write_hook {
        Ok(h) => {
            emu.remove_hook(h).unwrap();
        }
        Err(e) => {
            println!("mem_write_hook didn't take {:?}", e);
        }
    }
    match ret_hook {
        Ok(h) => {
            emu.remove_hook(h).unwrap();
        }
        Err(e) => {
            println!("ret_hook didn't take: {:?}", e);
        }
    }
    match indirect_jump_hook {
        Ok(h) => {
            emu.remove_hook(h).unwrap();
        }
        Err(e) => {
            println!("indirect_jmp_hook didn't take: {:?}", e);
        }
    }

    /* Now, get the resulting CPU context (the "phenotype"), and
     * encase it in a Pod structure.
     */
    let registers = emu.read_general_registers().unwrap();
    let vtmp = visitor.clone();
    let visited = vtmp.borrow().to_vec().clone();
    let wtmp = writelog.clone();
    let writelog = wtmp.borrow().to_vec().clone();
    let rtmp = retlog.clone();
    let retlog = rtmp.borrow().to_vec().clone();
    drop(vtmp);
    drop(wtmp);

    let pod = gen::Pod::new(registers, visited, writelog, retlog);
    pod
}
```

1.2 Concurrency plumbing

```
fn spawn_coop(rx: Receiver<gen::Creature>, tx: Sender<gen::Creature>) -> () {
```

```
    /* a thread-local emulator */
    let mut emu = Engine::new(*ARCHITECTURE);

    /* Hatch each incoming creature as it arrives, and send the creature
     * back to the caller of spawn_hatchery. */
    for incoming in rx {
        let mut creature = incoming;
        let phenome = hatch_cases(&mut creature, &mut emu);
        creature.phenome = phenome;
        tx.send(creature); /* goes back to the thread that called spawn_hatchery */
    }
}

/* An expect of 0 will cause this loop to run indefinitely */
pub fn spawn_hatchery(
    num_engines: usize,
    expect: usize,
) -> (
    Sender<gen::Creature>,
    Receiver<gen::Creature>,
    JoinHandle<()>,
) {
    let (from_hatch_tx, from_hatch_rx) = channel();
    let (into_hatch_tx, into_hatch_rx) = channel();

    /* think of ways to dynamically scale the workload, using a more
     * sophisticated data structure than a circular buffer for carousel */
    let handle = spawn(move || {
        let mut carousel = Vec::new();

        for _ in 0..num_engines {
            let (eve_tx, eve_rx) = channel();
            let from_hatch_tx = from_hatch_tx.clone();
            let h = spawn(move || {
                spawn_coop(eve_rx, from_hatch_tx);
            });
            carousel.push((eve_tx, h));
        }

        let mut coop = 0;
        let mut counter = 0;
        for incoming in into_hatch_rx {
            let &(ref tx, _) = &carousel[coop];
            let tx = tx.clone();
            tx.send(incoming);
            coop = (coop + 1) % carousel.len();
            counter += 1;
            if counter == expect {
                break;
```

```
            };
        }
        /* clean up the carousel */
        while carousel.len() > 0 {
            if let Some((tx, h)) = carousel.pop() {
                drop(tx); /* there we go. that stops the hanging */
                h.join();
            };
        }
    });

    (into_hatch_tx, from_hatch_rx, handle)
}
```

1.3 Hatchery dependencies

```
// #![feature(fnbox)]
extern crate capstone;
extern crate hexdump;
extern crate rand;
extern crate rayon;
extern crate unicorn;

//use std::boxed::FnBox;
use std::thread::{sleep, spawn, JoinHandle};
use std::sync::mpsc::{channel, Receiver, Sender};
use std::rc::Rc;
use std::cell::RefCell;
use std::time::Duration;
//use self::rayon::prelude::*;

use emu::loader::{get_mode, read_pc, uc_general_registers, Engine};
use par::statics::*;
use gen;
use gen::phenotype::{VisitRecord, WriteRecord};
// use log;
```

1.4 Putting things together

```
<<hatchery dependencies>>
<<concurrency plumbing>>
<<ROP-chain embryogenesis>>
```