The ROPER 2 Hatchery

May 5, 2018

Contents

1 The Hatchery **1**

1.1 ROP-chain embryogenesis 1

1.1.1 Hooks for behavioural analysis 3

1.1.2 Dealing with multiple problem cases 6

1.2 Concurrency plumbing 6

1.3 Hatchery dependencies 9

1 The Hatchery

The hatchery module of Return-Oriented Programming with ROPER: The Secret of the Ooze (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` (§1.2).

1.1 ROP-chain embryogenesis

Like its predecessor, Return-Oriented Programming with ROPER (ROPER), ROPER2 maintains distinctions between genotype and phenotype, on the one hand, and between phenotype and fitness, on the other. The *phenotype* of an individual, in this context, is its behaviour during execution. Execution, here, is provided by an emulated central processing unit (CPU), into which an executable binary has already been loaded (the loading is handled by the `emu::loader` module, documented in `loader.org`). The individual's

genotype is serialized into its "natural form" – a stack of addresses and machine words, which either dereference to locations in executable memory (ideally, to *gadgets*) or exist to provide raw numerical material to be used by the instructions dereferenced. It then loaded into the CPU's stack memory, and the first address is popped into the the program counter register (PC), just as it would be in a "stack smashing" attack. The emulator is fired up, and everything proceeds just as it would in a "wild" ROP attack.

```
<<bring the hatchery's dependencies into scope>>
<<spawn the main hatchery loop>>
<<spawn the subsidiary loops to divy up the workload>>
<<develop phenotypic profiles responding to a range of problem cases>>
  #[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;
      emu.restore_state();

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

    <<attach hooks for tracking performance>>

      let _res = emu.start(start_addr, 0, 0, 1024);

    <<clean up the hooks>>

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

1.1.1 Hooks for behavioural analysis

The phenotype – or, rather, that aspect of the phenotype that developed [1] in this particular execution – is then returned in the form of a `Pod` struct. The fields you see being passed to the `Pod` constructor, here, are populated by a series of functions that have been hooked into the emulator, using `Unicorn`'s hook API. (The `Engine` struct you see at work, here, in `hatch()`, is defined in the `emu::loader` module as well. It is more or less just a convenient encapsulation of a `Unicorn` CPU emulator, tailored to ROPER2's needs.)

The hooks are constrained, in type, in two important ways: they must be `'static` closures, implementing the `Fn` trait, and their signature is fixed in advance. They can mutate data when they are called, so long as that data has been suitably massaged into reference-counting, internally mutable cells, but they can't return values.

The list of hooks used is almost certain to grow, but for the time being they collect data pertaining to

- the execution path of the phenotype through memory, in the form of a vector of addresses visited, along with some useful information concerning the size of the instructions executed (to facilitate disassembly, when we come to analyse that path) and the hardware mode (which, in some instruction set architectures (ISAs), such as Advanced RISC Machine (ARM), can change at runtime);

- the *return* instructions hit. This could be gleaned from the execution path, but it's much more efficient to track the information separately. Return-type instructions are of special interest, since they represent the most basic form by which a ROP chain can maintain control over complex execution flows. This is information that we can profitably put to use in the fitness functions.

- the valid *writes* performed by the individual, tracked by instruction address, the destination address of the write, and the size of the data written.

```
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 = {
```

[1] In the sense of an embryo, or a photograph.

```
    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)
};
```

After the execution, we need to clean up the hooks, since they interact with data structures that will soon be falling out of scope, and we don't want that data to be unnecessarily held in memory, or to have an accumulating series of hooks cluttering up and slowing down execution in subsequent runs.

```
/* 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);
    }
}

/* Get the behavioural data from the mutable vectors */
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();
```

1.1.2 Dealing with multiple problem cases

Depending on the task at hand, the phenotypic profile that we're interested in evaluating may need to include the responses of the individual to a variety of inputs, exemplars, environmental states, etc. It's simple enough to treat cases where the problem space *isn't* multiple as a singleton, and so it fits comfortably enough within this scheme.

The `hatch` function is therefore dispatched by another, called `hatch_cases`, which is little more than a `while` loop, iterating over the various problem cases associated with the task or environment of interest.

Since the `Unicorn` emulator is a foreign struct, implemented in **C**, there's no easy way to thread this portion of the program. Forcing an implementation of the `Send` trait on this struct may expose us to various race conditions, and other unsafe hazards.

```
#[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
}
```

1.2 Concurrency plumbing

We can nevertheless make great gains in efficiency by spinning up a set of threads at the beginning of each evaluation phase, and binding an `Engine` instance to each thread's scope. The main loop of each of those threads is

implemented by the function, `spawn_coop`. Rather than collect and return a vector of results from these evaluations, `spawn_coop` maintains a line of communication back to the caller of the function that called it, in the form of a `channel` (specifically, a `Creature channel`).

The concurrency paradigm being used here is more or less "the actor model" of concurrency. There is no shared memory, and when one of our "actors" (hatcheries or coops) takes possession of a `Creature`, it does so uniquely. No mutexes or reference counters are needed to protect the `Creature` from race conditions, since it never needs to be in the hands of two actors at the same time. Instead, we just pass *ownership* of the `Creature` from actor to actor – and thanks to Rust's exquisite ownership system, this is just a matter of transferring a handful of machine words. No copying or cloning is needed.[2]

```
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 */
    }
}
```

The threads are spawned and dispatched by another looping thread, which is spawned, in turn, by the `spawn_hatchery` function.

```
/* 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>,
```

[2]This is the concurrency model used throughout ROPER2. The only actor that takes a clone of a `Creature`, rather than temporarily seizing ownership of it, is the `logger` actor, which performs statistical analysis on the population stream, and logs data to files. This is done so to avoid having the logger's relatively expensive operations block the pipeline, and for this, skimming off a stream of clones is a small price to pay. The upshot, as we'll see, is that the logger needs no return channel. The trip to the agent is one-way, and the clone is dispensed with afterwards.

```
    JoinHandle<()>,
) {
    let (from_hatch_tx, from_hatch_rx) = channel();
    let (into_hatch_tx, into_hatch_rx) = channel();

    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);
                h.join();
            };
        }
    });

    (into_hatch_tx, from_hatch_rx, handle)
}
```

This function returns almost immediately when called, bearing three values to its caller:

- `into_hatch_tx`, which is the `Sender` end of a channel that can be used to transmit individuals (of type `Creature`, which at the time of arrival are little more than genomes in hollow shells, whose phenotypes have not yet been brought to maturity) to the *genome → phenome map*

- `from_hatch_rx`, which is the channel on which the caller (or some thread delegated by the caller) listens for the creatures to return, now developed into mature phenotypes,
- `handle`, the `JoinHandle` of the thread, which will be used to join the main hatchery thread.

digraph { foo -> bar; }

1.3 Hatchery dependencies

```
extern crate unicorn;
use std::thread::{spawn, JoinHandle};
use std::sync::mpsc::{channel, Receiver, Sender};
use std::rc::Rc;
use std::cell::RefCell;
use emu::loader::{get_mode, read_pc, uc_general_registers, Engine};
use par::statics::*;
use gen;
use gen::phenotype::{VisitRecord, WriteRecord};
```