The Evolutionary Exploration of Emergent Execution: AIMEE Progress Report

Olivia Lucca Fraser & Daniel Kuehn of Special Circumstances

October 9, 2020

Contents

1 Design of ROPER II (Berbalang)

ROPER (Return Oriented Program Evolution with ROPER) is a system for the evolutionary exploration of a certain species of weird machine, a species

which was first discovered and exploited by hackers through the technique known as "return-oriented programming". The first iteration of ROPER was developed by Special Circumstances researcher, Lucca Fraser, as part of her 2017 MCS dissertation at Dalhousie University [2]. There, Fraser showed that it was possible to evolve populations of ROP payloads on arbitrary executables, which could accomplish tasks varying from simple system call preparation to the accurate classication of small, linearly inseparable data sets, or even controlling an agent in a simple video game.

The second and current iteration of ROPER builds on Fraser's earlier techniques, while extending it to cover other architectures – chiefly the x86 and $x86_{64}$ chips most commonly encountered on servers and personal computers. A particular focus, this time around, has been given to *exploratory* tasks, where the objective is less the achievement of a determinate goal than (partially) mapping the potential of a particular kind of emergent execution on a given target.

1.1 A General Framework for Genetic Programming

We decided to design a more or less general framework for the study of genetic programming that we could use to house both ROPER II and various other experiments in the *evolutionary exploration of emergent execution* (EEEE). We have called this framework *Berbalang*, and have made its source code available under GNU Public License, here.

1.2 Design Decisions

1.2.1 Genotype and Phenotype Representations

1. Genotype

   (a) Bare Integer Sequences

   Unless otherwise stated, the genotypes circulating in ROPER's populations are represented as simple vectors of 32 or 64-bit integers, depending on the word size of the target architecture. In viable specimens, these can be understood as the machine words making up a ROP chain payload. Some of these integers will be interpreted as addresses, pointing to "gadgets" of code in the target process's executable memory regions, or pointing to data that may be manipulated by the target machine when the payload is executed. Some of them will be interpreted as immediate integer values, and used in computations. But all of this concerns

the phenotype rather than the genotype. As far as the genotype itself is concerned, these are mere sequences of tokens, which are susceptible to being manipulated and recombined by the system's genetic operators.

(b) Payload Builder Instructions

Following a suggestion that Lee Spector made to Lucca Fraser at the 2017 GECCO conference, where she presented her initial findings with ROPER, we decided to see what might happen if, instead of evolving bare ROP payloads (treated as mere vectors of integers), we evolve a population of programs that, when given access to the target memory image, along with a basic semantic analysis of that image (provided by the falcon binary analysis library), *build* ROP payloads.

We designed a stack-based virtual machine to execute these builder programs, closely following Lee Spector's designs for what he calls the Push VM [4].

2. Phenotype

Genetic programming turns on a distinction between an individual's genotype – the object of its genetic operators – and its phenotype – the object of selection. In ROPER's case, what we are calling the "phenotype" is the observed behaviour of an emulated CPU when that payload is executed. (In the case of our Push VM genomes, there is an intermediary step, whereby a genotype consisting of Push instructions is first mapped to a ROP payload, which is then sent to the CPU emulator.) The resulting observations, or "execution profile", furnish the domain for one of a variety of *fitnes functions*, which map execution profiles to a table of scalar attributes. ROPER then applies an easily-configurable *weighting formula* to the resulting table of attributes, mapping the attribute table to a single floating point number, which we call the individual's *scalar fitness value*. This value is used to rank the contestants in a tournament and select parents for the next generation of genotypes.

(a) "Commitment points" and composability

Something that arguably distinguishes ROPER from most other genetic programming environments is that the *composability* of the instructions that comprise its genetic material is a feat to achieve and not a given. Not every "instruction" in an arbitrary

genome is necessarily capable of passing execution on to a successor instruction, and so swapping subsequences of genetic material between individuals will only *occasionally* have phenotypic consequences. The expectation in a purely random population is that most genetic material, particularly as we move past the *head* of the genome, which is always (if executable) executed, will remain phenotypically inactive – will have no impact on execution behaviour.

The programmatic exploration of return-oriented machine space depends, therefore, on the discovery of *composable* instructions or genes: elements whose phenotypic effects depend on, and condition, those which come before or after in the sequence, and whose effects combine into new effects that are not achieved separately.

We made a significant change to the system, in this iteration, in order to better foster the evolutionary discovery of composable components – changes which, we learned, would also help to avoid various dead-ends and sticky local optima. Whereas, previously, we recorded the execution behaviour of each specimen on an instruction-by-instruction basis, from the beginning of the chain's execution up until its termination (whether by crash, interrupt, or arrival at the address 0), we now maintain a temporary execution log that is commited to the specimen's execution profile *only* when a return instruction is reached – which is to say, when another address is about to be popped from the attacker-controlled stack into the instruction pointer. We further restrict this "commitment points" to return instructions that are executed when the call stack is empty, and, in some runs, explore the option of halting execution when *any* function call is dispatched, so that every return is a return to a potentially (perhaps indirectly) attacker-controlled address. (This method could be generalized to treat jumps to addresses in stack-controlled registers as commitment points, too, with a bit of tinkering.)

Any instructions that are executed without eventually reaching such a "commitment point", for all intents and purposes, leave no trace. This is crucial. A sequence of instructions that partially, or even fully, satisfies one of our objectives, but which then crashes, or times out in an endless loop, is of no use to the population, because it cannot be *composed* with other sequences.

1.2.2 Tournament Selection

After some early experimentation with forms of fitness-proportional selection (the "roulette" and "Pareto front" selection methods), and lexicase selection, which we found poorly-suited to our problem domain, we settled on the widely-used technique of *tournament selection*, with an optional geographical constaint (detailed below, under 2). Each iteration, n (typically 5, in our experiments) contestants are drawn from the population and evaluated. The p (typically 2) best performers are selected for breeding. The p offspring thereby produced (by applying the 1.2.4 to the winners) are then inserted into the population, displacing the p worst performers.

This process is repeated until a termination condition is reached.

1.2.3 Geographical Constraints

1. Islands with Migration

   This tournament process churns along on several subpopulations, or "islands", in parallel, a well-established technique for parallelizing genetic algorithms while fostering diversity [6]. Occasionally (at a rate that can be set in the configuration file), an individual may emigrate from an island onto a structure called the "pier" (implemented as a non-locking, threadsafe queue), and occasionally an island may attempt to absorb immigrants from the pier into its population. This allows the island populations to evolve in concert, drawing the benefits of a single, large population, while making room for genetic diversity by slowing evolutionary convergence.

2. Linear Geographies

   Drawing on Lee Spector and Jon Klein's work on "trivial geographies" [5], we impose a secondary geographical structure on each subpopulation. On each island, the subpopulation is structured as a one-dimensional circular buffer, outfitted with a constraint called *radius*. The first contestant for each tournament is drawn with uniform probability from the subpopulation as a whole, but each subsequent contestant is drawn only from among the first contestant's neighbours – those dwelling within *radius* slots of the first. Clearly, setting *radius* to the size of the entire subpopulation captures unrestricted tournament selection as a special case (and this can be enabled by setting the `migration_radius` setting in the configuration file to 0).

1.2.4 Genetic Operators

1. Crossover (Alternating and Single-Point)

   We apply a *crossover* operator to our parental genomes (with a probability set by the configuration file, but which is typically set to 1.0, with the exception of the experiments for which it is set to 0.0), to produce offspring. This mimicks, to some modest extent, the process of *sexual reproduction* in nature. In our earlier experiments, we implemented an algorithm for *alternating crossover*, which composed a child genome by stitching together alternating patches, of lengths drawn from an exponential distribution, from the two parents. This method reliably produced offspring with genomes no longer than the longest parental genome, thereby preventing genetic bloat.

   We later added an implementation for the simple *single-point crossover* algorithm, which composes a child genome simply by snipping the two parents at random indices, and gluing the head of the first to the tail of the second. We will later see the dramatic effects that this difference in crossover algorithm has on the genetic makeup of the population.

2. Memory-aware Mutation Functions

   If a genotype is selected for mutation, we choose n alleles to mutate using a Levy-flight distribution (following suggestions sketched out in [1]), and then a mutation operator is selected to apply to that allele with uniform probability. The set of available mutation operators, for bare payload genomes, includes numerical and bitwise manipulations – incrementing, decrementing, masking, and bitshifting the allele – as well as a pair of memory-aware operations: searching for the allele's numerical value in the target process's memory, and replacing it with its address if found, or treating the allele as an address, and replacing it with whatever lies at that address in memory, if anything.

1.3 Technical Obstacles

1.3.1 A Race Condition Bug in the Unicorn Emulator Library

In order to map ROPER's genotypes to their execution-profile phenotypes, we have relied heavily on the Unicorn Emulation Library, which exposes QEMU's CPU emulation modules through a convenient API, allowing callbacks to be hooked into various processor events. This makes it an ideal instrument for the kind of microscopic attention we wish to bring to ROP-chain execution. To better adapt Unicorn to ROPER's needs, we have made

numerous adjustments to ekse's Rust bindings for Unicorn. Unfortunately, relying heavily on Unicorn's C codebase means that Rust's virtues of thread safety do not extend to this mission critical component, and when we started running ROPER experiments at scale, we soon triggered a segmentation fault in the Unicorn library.

An inspection of the core dumps from these crashes showed that the segmentation faults were due to an attempt to write to a field of a null `cpu` struct (see figure 1).

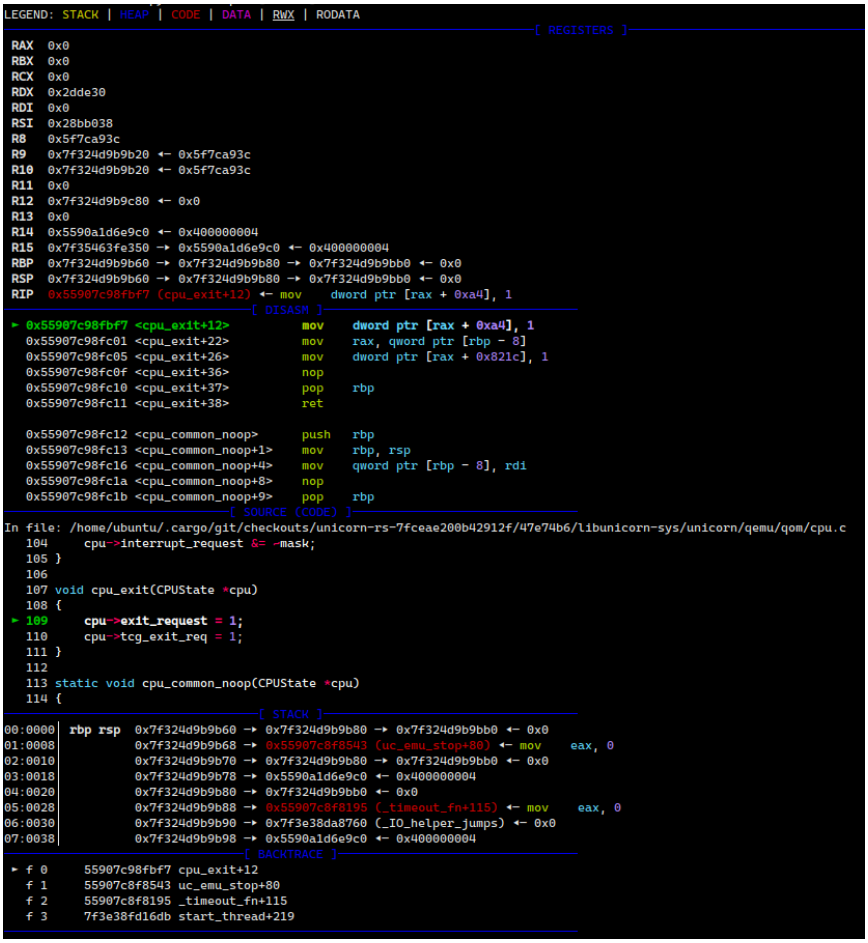


Figure 1: Segmentation fault in the Unicorn emulation library

It appeared that these faults were only being triggered when Unicorn's timeout callback called the `uc_emu_stop()` function, from a watchdog thread

separate from the main emulation thread. This function checks to ensure that `uc->current_cpu` is not null, and *then* calls `cpu_exit(uc->current_cpu)`. This led us to suspect a race condition, whereby, after the check but before the call, `uc->current_cpu` was made null by events unfolding on another thread. The solution to this problem, of course, was just to wrap this critical section of code in a mutex lock:

```
pthread_mutex_lock(&EMU_STOP_MUTEX);
if (uc->current_cpu) {
  // exit the current TB
  cpu_exit(uc->current_cpu);
}
pthread_mutex_unlock(&EMU_STOP_MUTEX);
```

Once we made this patch to the library, the segfaults disappeared.

1.3.2 The Travails and Follies of Cloud Computing

Our AWS setup consisted of three supporting servers:

- Log server
- Storage server
- Jumphost

The log server ran elasticsearch and kibana for gathering data from the compute host and visualising it in a dashboard. Metricbeat was used on the compute nodes to gather running data of the compute nodes load average, %cpu and memory usage amongs others. The storage server hosted a NFS server that was used to collect the logs from the compute nodes experiments in a central place. The jumphost was simply a point of entry to the subnet that hosted the rest of the servers and compute nodes. An AMI was created to make it easy to setup more compute nodes, it was pre-configured with lxd, automatic mount of the log NFS share and had both the berbalang and berbalang-test-runner repositories cloned. Three compute nodes were used, although the third compute node was added quite late, mainly to test a theory shortly. For benchmark purposes, the initial two nodes were one AMD c5a.16xlarge and one Intel c5n.18xlarge. They were chosen because they were the closest to eachother in core count and a main reason to do some testing on AWS was to see how berbalang scaled with more resources and to collect data on what type of resources berbalang used the most.

We had initially planned to run a 24-hour benchmark trial, running the exact same tests on both the AMD and Intel compute node, to get a baseline to compare the performance of the nodes with. The test-runner was started with three simple tests, the experiment was `sshd_x86_mempat_ropgadget_ignore_stack_all` (where ROP populations are bred for heap control) with three different population sizes (0x250, 0x500 and 0x1000). However we ran into segmentation faults quite early, that put the test runs to a halt while we investigated the causes of crash, as discussed in the previous section. We decided to put the benchmark on hold while we manually ran some tests to gather data for analysis, while still gathering data with metricbeat.

What the metricbeat data showed was that the load from running berbalang is choppy and uneven, and does not seem capable of to saturating a large number of cores with pure computations as shown in figure 2.

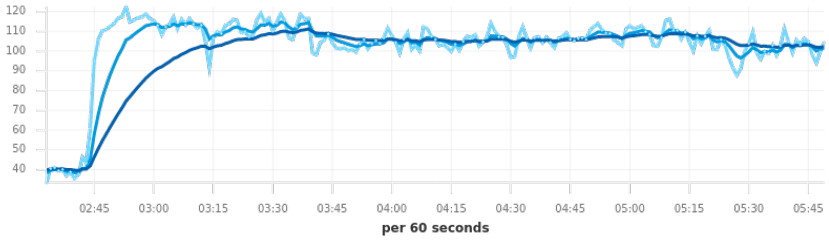


Figure 2: Zoomed in section of load average on the AMD compute node

We can see in the graph for %cpu (fig. 3) taken from the same time period, the red is %sys and the green is %user, showing that there is a fairly static amount of CPU time that %sys uses for the running of berbalang, while %user uses the rest of the cores, in the best case scenario. These small dips in the %cpu are more prominent when watching the graph that plots the entire lifetime of the AMD compute node 4. What these graphs show is that berbalang does not necessarily scale linearly with amount of cores, as we had initially hoped.

Another interesting thing in the data is that the load average graph 5 doesn't entirely correlate with the %cpu graph which shows that the load was usually not from computation, but I/O and memory preassure on top of the computational load. This is shown by the load average being well above the 64 threads (32c/64t) that the AMD compute node had.

The Intel compute nodes showed what could be a hint to what resource that berbalang is actually constrained by, because whereas the AMD compute node was a single socket node, that had 64 threads, the Intel compute

nodes consisted of a dual socket system that had 2x36 threads (2x18c/36t) for a total amount of 72 threads. The Intel compute nodes were a lot better at keeping the threads working as shown in figures 6 and 7. However the load average graphs of the Intel compute nodes (figs. 8 and 9) also showed that it wasn't mainly heavy usage of computation that was causing load. Because the combination that the load average was constantly above the nominal load based on the %cpu utilization and that the load average was very choppy hints that its I/O and memory preassure that is the main bottlenecks for berbalang, with memory preassure being the larger of the two.

One of the probable reasons that the Intel compute nodes could saturate the threads better is that the dual socket system had access to two memory controllers and NUMA zones, which means there are two memory controllers that can fetch/dump data into RAM and the computation that berbalang runs does a lot of memory operations. It doesn't seem to be constraint by memory bandwidth or amount of RAM either, because the memory usage is very modest, and increases in a slow, but fairly linear matter as shown in figures 10 and 11. Thus it would seem like memory latency is a larger issue for berbalang, to be able to keep the CPUs threads saturated. Another factor that could explain part of the fact that the Intel compute nodes could saturate the threads better was that the CPUs that they were equipped with had twice as large L2 caches (1024kB vs 512 kB on the AMD CPU) and ~50% larger L3 caches (24.75MB vs 16MB for the AMD CPU).

We will investigate this peculiarity further, to see what type of compute node is best to run large scale berbalang tests. The initial hunch that one big box, with a lot of threads and RAM, seems to have been incorrect. A large collection of more modest nodes would likely do a better job of things, and we should take advantage of the ease with which a system like Berbalang can be distributed across many nodes.

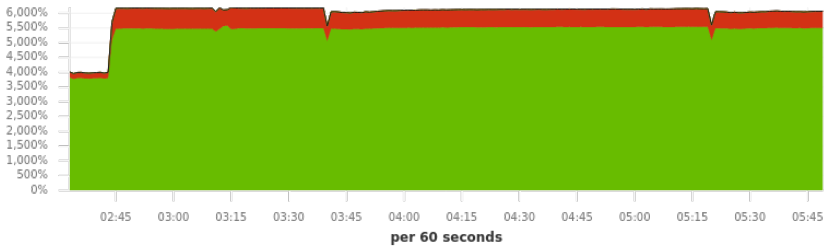


Figure 3: Zoomed in section of %cpu on the AMD compute node

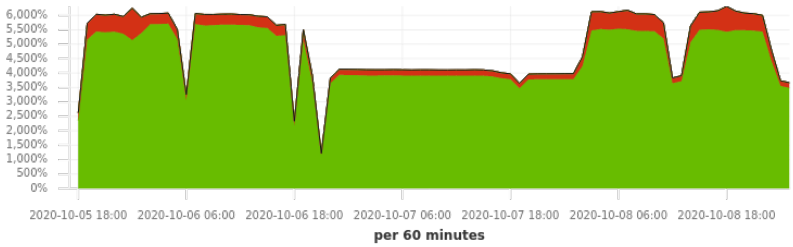


Figure 4: %cpu during the entire run of the AMD compute node

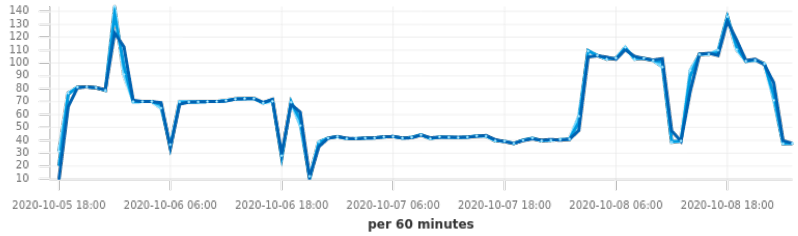


Figure 5: Load average over the entire run of the AMD compute node

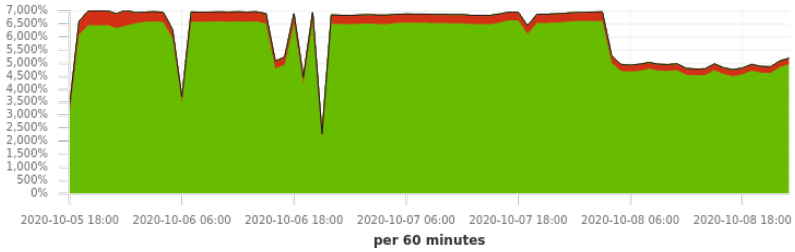


Figure 6: %cpu during entire run of the Intel compute node 0

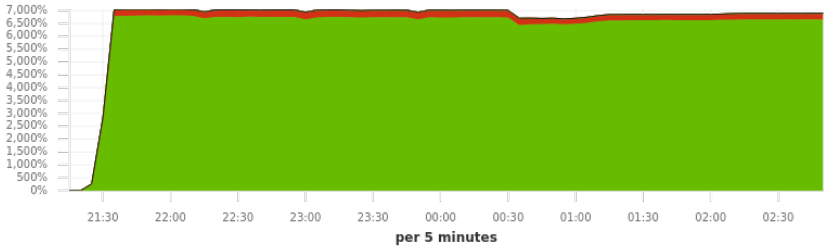


Figure 7: %cpu during entire run of the Intel compute node 1

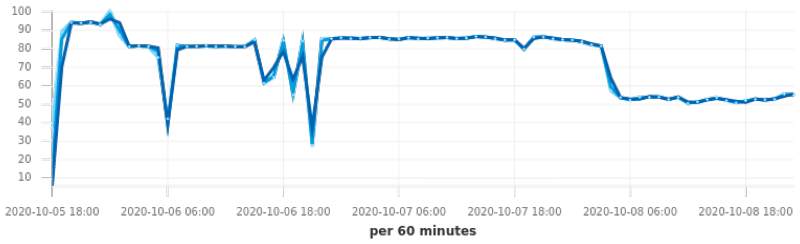


Figure 8: Load average during entire run of the Intel compute node 0

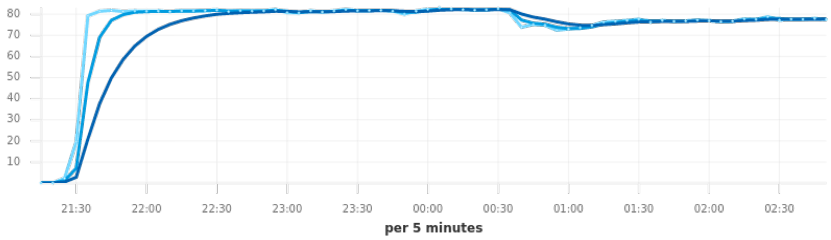


Figure 9: Load average during entire run of the Intel compute node 1

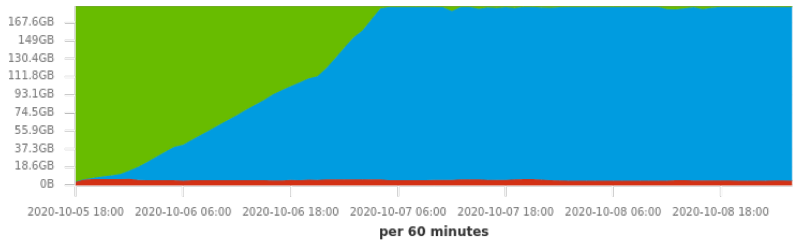


Figure 10: Memory usage during the entire run of the Intel compute node 0 (red is memory in use, blue is memory used as cache and green is free memory)

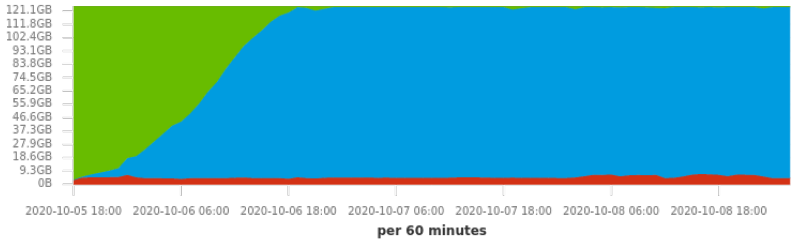


Figure 11: Memory usage during the entire run of the AMD compute node 0 (red is memory in use, blue is memory used as cache and green is free memory)

2 Experiments

2.1 A Note on Terminology

In the experiments discussed below, we make frequent reference to units of time we call "epochs". In this context, an epoch is equal to the number of tournaments after which we can expect every member of the population to have *possibly* been replaced. We define this number to be the population size divided by the number of offspring generated per tournament.

2.2 Sexual Reproduction and Composability

In "A Mixability Theory for the Role of Sex in Evolution," Adi Livnat et al. [3] ask what selective pressures might account for the ubiquity of sexual reproduction in nature:

> We develop a measure, [mixability], which represents the genome-wide ability of alleles to perform well across different combinations. Using numerical iterations within a classical population-genetic framework, we find that sex favors the increase in [mixability] in a highly robust manner. Furthermore, we expose the mechanism underlying this effect and find that it operates during the evolutionary transient, which has been studied relatively little. We also find that the breaking down of highly favourable gene combinations is an integral part of this mechanism. Therefore, if the roles of sex involves selection not for the best combinations of genes, as would be registered by [fitness], but for genes that are favourable in many different combinations, as

> is registered by [mixability], then the breaking down of highly favourable combinations does not necessarily pose a problem.

We expect that the domain of ROP chain evolution might prove to be an interesting case by which to test Livnat's theory, particularly given that the evolution of ROP chains from a soup of random addresses places the problem of composability and mixability front and centre. In traditional genetic programming environments, the composability of instructions is more or less assured *a priori*. Here, by contrast, maintaining control over the flow of execution is an achievement to be won.

A simple, somewhat crude measure of how composable the alleles circulating in a population are can be found in the number of return instructions each specimen executes on average, since these mark the points at which various strings of alleles can be composed. (This measure can be deceived by specimens which create return-loops for themselves, whereby a gadget pushes its own address onto the stack before executing `ret`. But there is no prima facie reason to expect looping behaviour to be more common in sexual populations than asexual ones.)

2.2.1 Comparing Crossover and Asexual Reproduction with a Code-Coverage Fitness Function

We conjecture that crossover, whether single-point or alternating, induces an implicit selection for highly composable genetic sequences, which is to say, genetic sequences that can be easily combined with others to achieve various complex phenotypic phenomena (execution behaviours). We believe that this should result, among other things, in a higher number of executed `ret` instructions in sexually-reproductive populations. This is because *returns* are the simplest way to maintain control over the flow of execution, from one gadget to another. A pressure for the selection of composable units, which can potentially contribute to the fulfillment of the objective function no matter where they appear in an individual's genetic sequence, should therefore steer us towards `ret`-terminated gadgets.

We focussed, here, on populations subjected to the code coverage fitness function, where an individual's fitness is simply proportionate to the number of unique addresses it visits during its execution. This coverage ratio can be a little misleading, when taken in isolation. It's nothing more than the size of the set of bytes executed divided by the total number of executable bytes, but there's no guarantee that all of the bytes in memory flagged with an executable permission are indeed executable in fact. The

score also neglects to take into consideration the step and time limits placed on the emulator, which set an implicit upper bound on the code coverage score that's even possible for a given run. It nevertheless serves as a point of comparison between specimens in the same batch, and places an easily understood selective pressure on the evolving population.

2.2.2 Parameters

The following settings were common to every trial in this experiment:

Setting	Value
number of islands	8
max initial length	500
min initial length	450
island population size	1024
tournament size	5
number of parents	2
number of offspring	2
geographic radius	10
migration rate	0.01
initial soup size	0x40000
binary	OpenSSH 6.8p1 sshd for i386
max emulator steps	0x2000
max emulator time	5 milliseconds
emulator stack size	0x1000
allow function calls	no
fitness function	code coverage
weighting	1.0 - code-coverage
number of epochs	250

In the asexual trials, we have the following settings:

Setting	Value
crossover rate	0.0
mutation rate	1.0

And in the alternating and single-point crossover trials, we have:

Setting	Value
crossover rate	1.0
mutation rate	0.03

As a secondary axis of variation, we seeded *half* the populations with gadgets harvested by the popular tool, ROPgadget, and seeded the other half with randomly generated addresses, with no prior check to ensure that those addresses resolved to composable gadgets.

This gave us six different configurations, and we ran three trials for each, giving us a total of 18 trials total. In the discussion below, we will present plots from the first of each of these triplets of trials, which we judged to be representative of the patterns observed. The remaining plots can be found in our github repository.

The build of berbalang used was compiled from commit `4f59161` of the `master` branch.

2.2.3 Results

1. Return Count

   These experiments bore out our hypothesis on return counts, in part. The mean count of returns per individual execution in the asexual, randomly-seeded (fig. 12) *and* the ROPgadget-seeded populations (fig. 13), over the course of 250 epochs, rarely exceeded 2 or 3. For randomly-seeded populations equipped with single-point crossover (fig. 16), the mean return count was frequently double that, ranging between 4 and 7 across the three trials. The single-point crossover populations seeded with ROPgadget-harvested addresses (fig. 17) showed mean return counts as high as 81, in one case, and between 12 and 15 in the other two. It's interesting to reflect that our asexual populations were unable to extract much benefit at all from these ROPgadget harvest initializations – it seems likely that the high mutation rate in those populations had something to do with this.

   It may be interesting to conduct another series of experiments in which crossover is replaced with some form of permutating, rather than point, mutation, which would rearrange (and perhaps even duplicate or delete) alleles, but which would not lead to a higher degree of allele damage than we already get in sexual populations.

   We were surprised by how weakly the populations equipped with alternating crossover performed. In most respects, they differed very little from the asexual populations: a maximum mean return count between 2 and 3, after 250 epochs, in the randomly-seeded populations (fig. 14), and between 4 and 5.5 in the ROPgadget-seeded populations (fig. 15).

   Plots illustrating mean return counts, along with standard deviations,

for each of these six configurations are shown below, grouped by reproductive type. Additional plots can be found in our github repository.

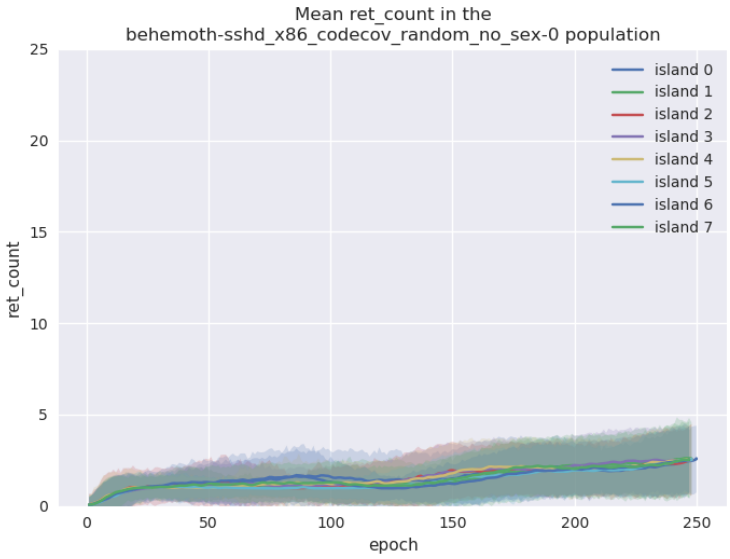


Figure 12: Return count in a population reproducing asexually, seeded with random addresses

The trends we see here are only amplified when we scale up from 250 epochs to 1000, as seen in figures 18. There, we see the single-point crossover population achieve a mean return count of upwards of 40, while neither asexual reproduction nor alternating crossover barely bring the mean higher than 7.

2. Code Coverage

   We see a similar distribution of values when it comes to mean code coverage, in these populations. Single-point crossover (figs. 25, 26) outperformed both alternating crossover (figs. 23, 24) and asexual (figs. 21, 22) populations by a factor of 3. This is more or less what we would expect, given the mean return count measurements.

3. Allele Circulation

   If we turn our attention to the circulation of alleles through the population, and ask how common it is, under each of these configurations, for certain alleles to reappear in a variety of genetic contexts. The

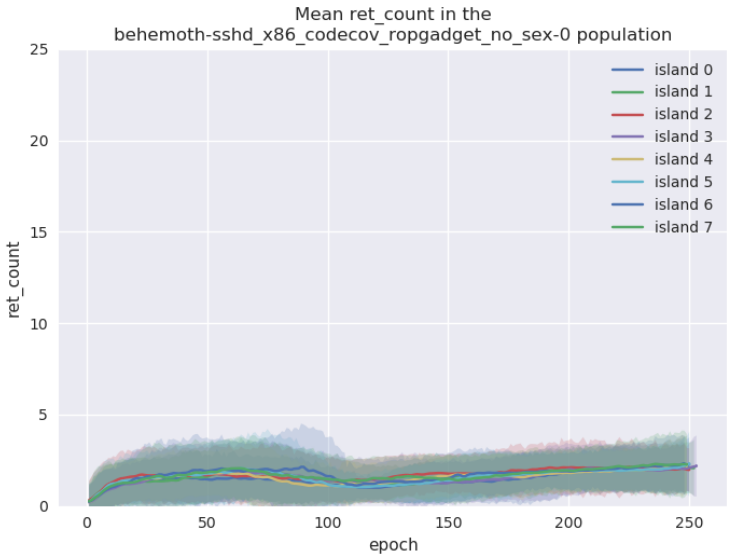


Figure 13: Return count in a population reproducing asexually, seeded with harvested addresses

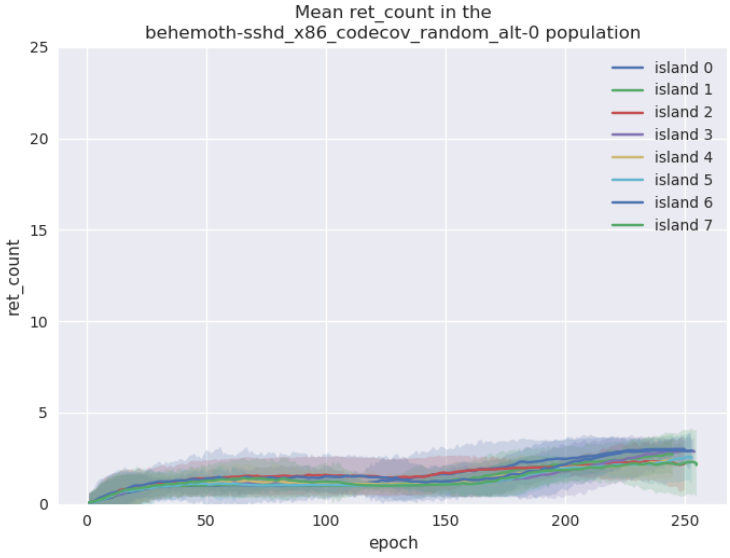


Figure 14: Return count in a population reproducing by alternating crossover, seeded with random addresses

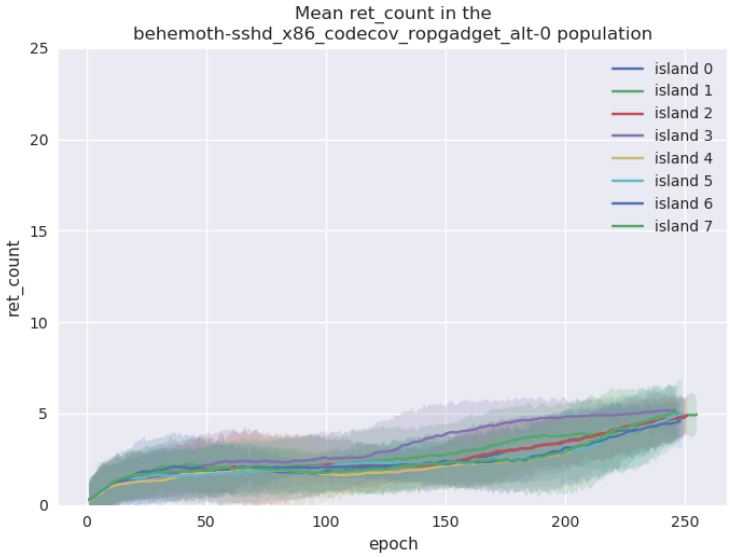


Figure 15: Return count in a population reproducing by alternating crossover, seeded with harvested addresses

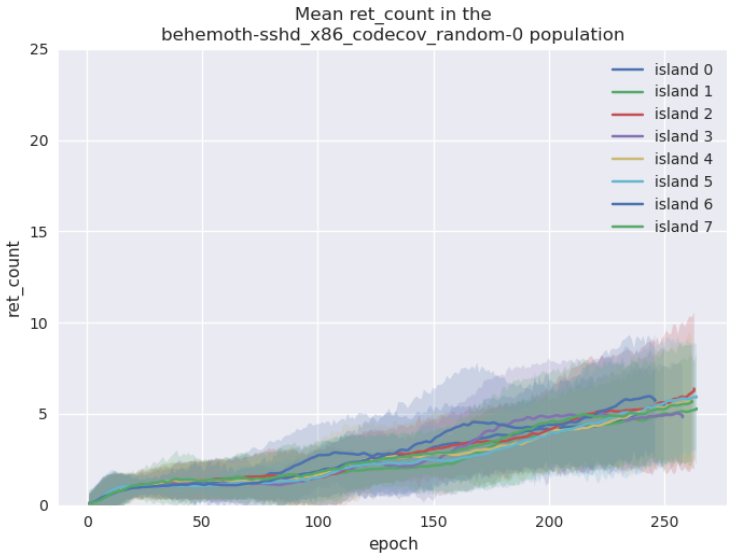


Figure 16: Return count in a population reproducing by single-point crossover, seeded with random addresses

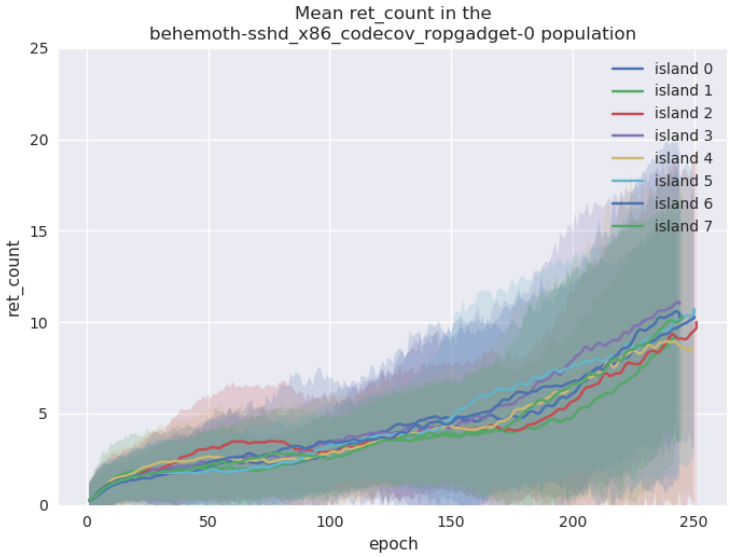


Figure 17: Return count in a population reproducing by single-point crossover, seeded with harvested addresses

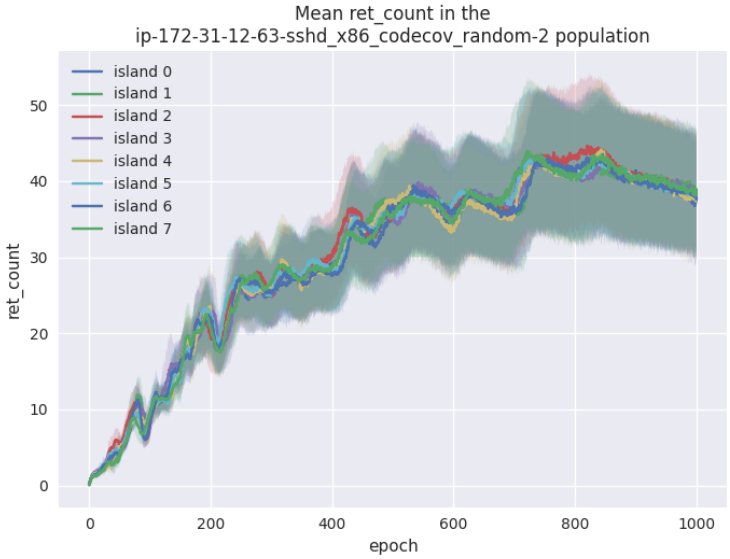


Figure 18: Return count in a population reproducing by single-point crossover, seeded with random addresses, over 1000 epochs.

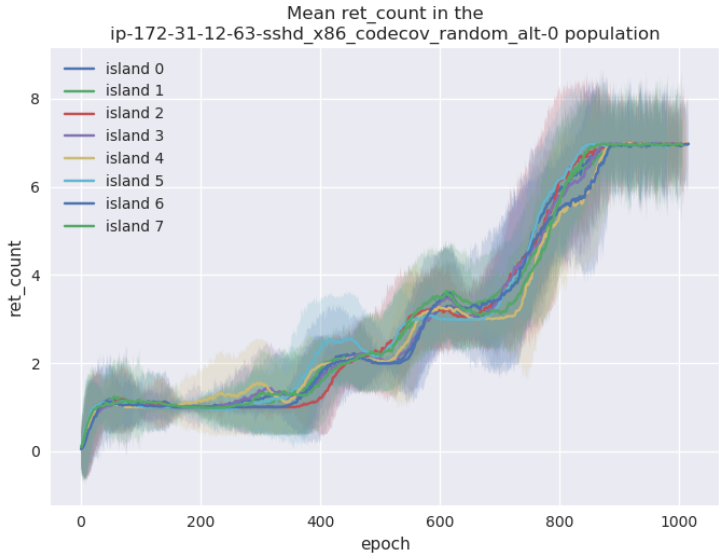


Figure 19: Return count in a population reproducing by alternating crossover, seeded with random addresses, over 1000 epochs.

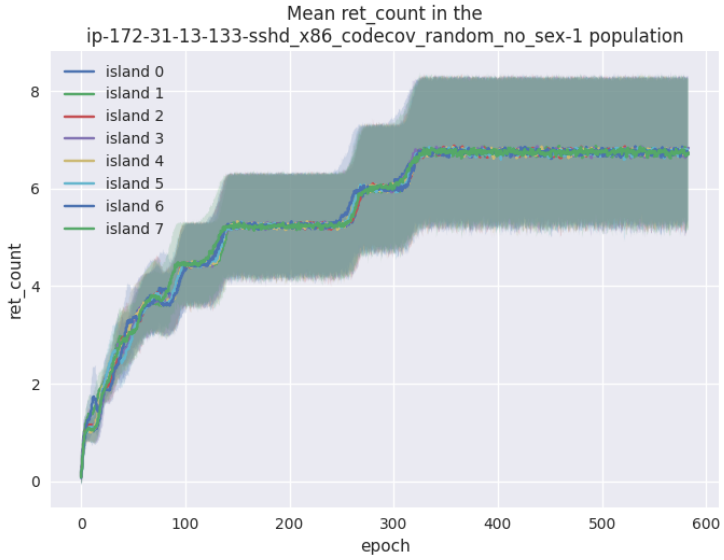


Figure 20: Return count in a population reproducing asexually, seeded with random addresses, over 1000 epochs.

Figure 21: Code coverage in a population reproducing asexually, seeded with random addresses

Figure 22: Code coverage in a population reproducing asexually, seeded with harvested addresses

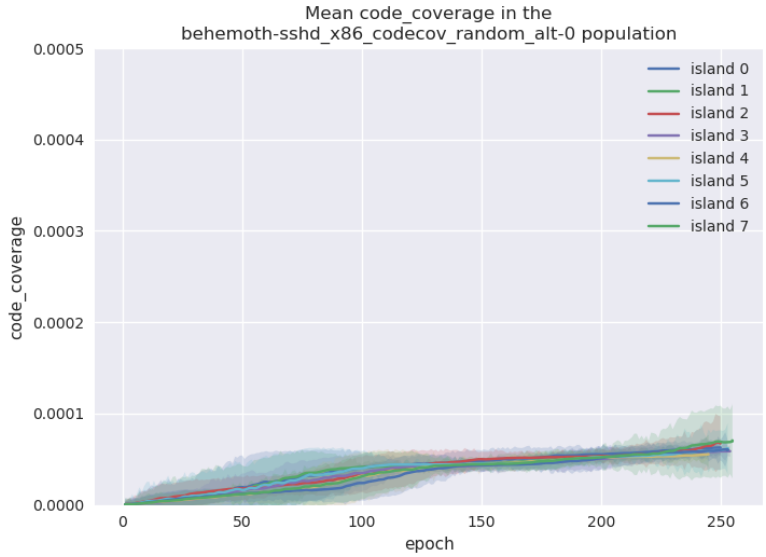


Figure 23: Code coverage in a population reproducing by alternating crossover, seeded with random addresses

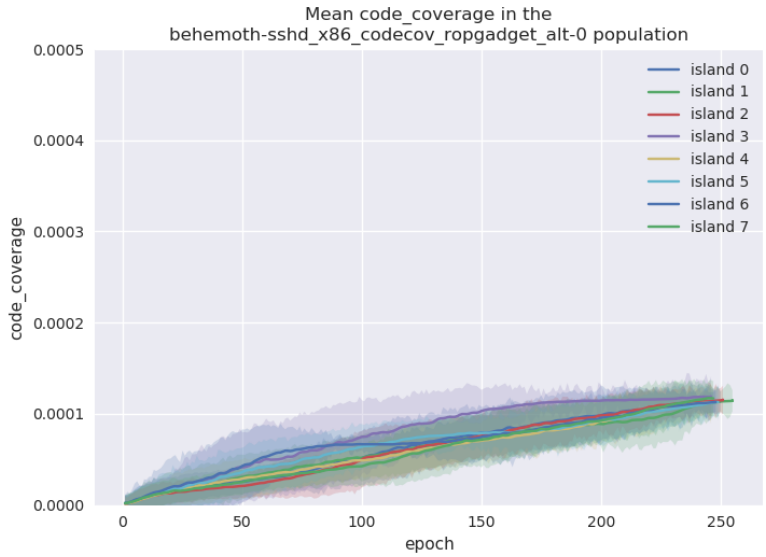


Figure 24: Code coverage in a population reproducing by alternating crossover, seeded with harvested addresses

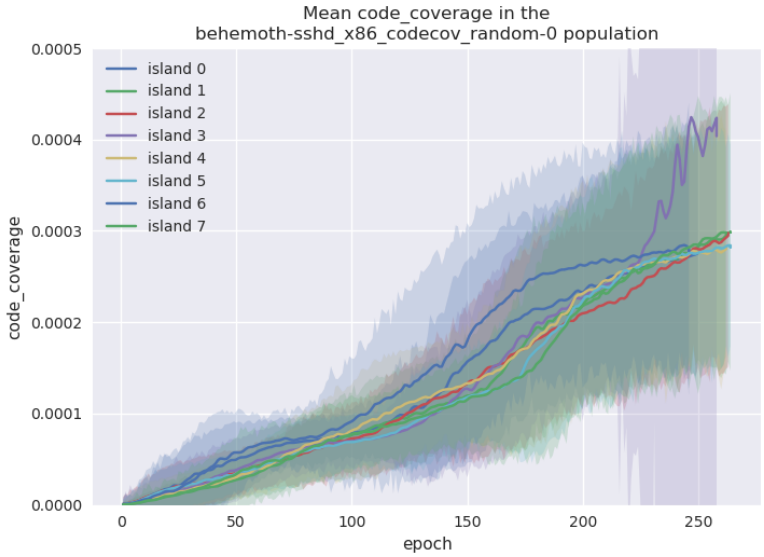


Figure 25: Code coverage in a population reproducing by single-point crossover, seeded with random addresses

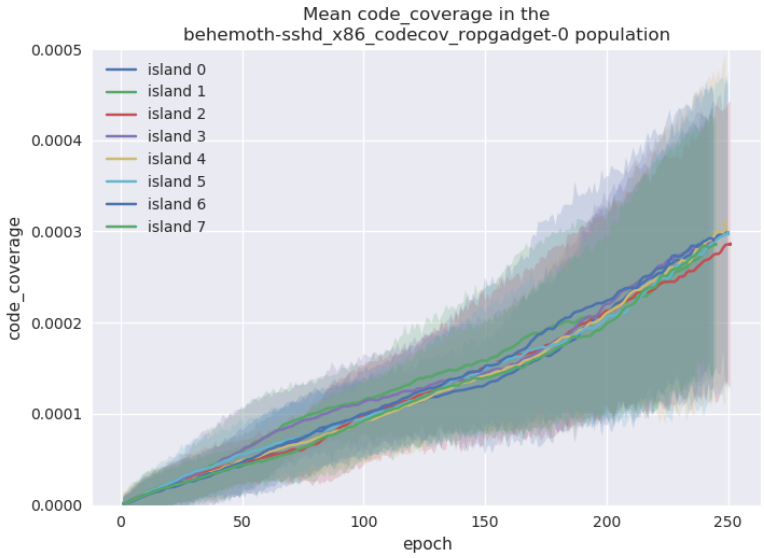


Figure 26: Code coverage in a population reproducing by single-point crossover, seeded with harvested addresses

following plots display a curve for each allele that appears 100 or more times in the individuals contained in a sliding window over the population, across 250 epochs. In these plots, we focus our attention on a single island subpopulation at a time, to avoid cluttering things more than we need to.

In the asexual populations (see figures 27 and 28), we occasionally see a handful of alleles achieve prominent fixation in the population, their trajectories wisping out from the baseline churn of genetic material – a handful, but not many.

In the populations reproducing through alternating crossover (figs. 29, 3), we, perhaps surprisingly, see even fewer alleles *dramatically* separate themselves from the low-frequency genetic churn, but we see many more hovering at the 500-copy level.

The single-point crossover populations (figs. 30, 31) stand out dramatically. Enormous waves of high-frequency alleles circulate through the population, achieving prominent fixation for upwards of 100 epochs before ebbing back into the sea of variation.

It is striking that the difference between randomly- and ROPgadget-seeded populations appears to *make* so little difference in this aspect of the genetic landscape. This may have something to do with the fact that we are looking at a property of the evolutionary system that becomes prominent only 50 or so epochs into the process, whereas the differences between ROPgadget- and randomly-seeded populations tend to be most pronounced early in the evolutionary process.

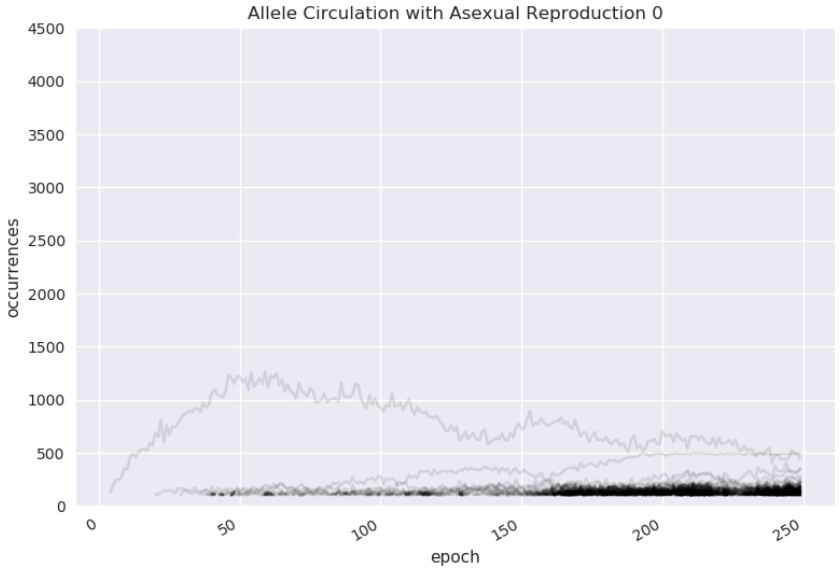


Figure 27: Allele circulation in an asexual population, seeded with random addresses

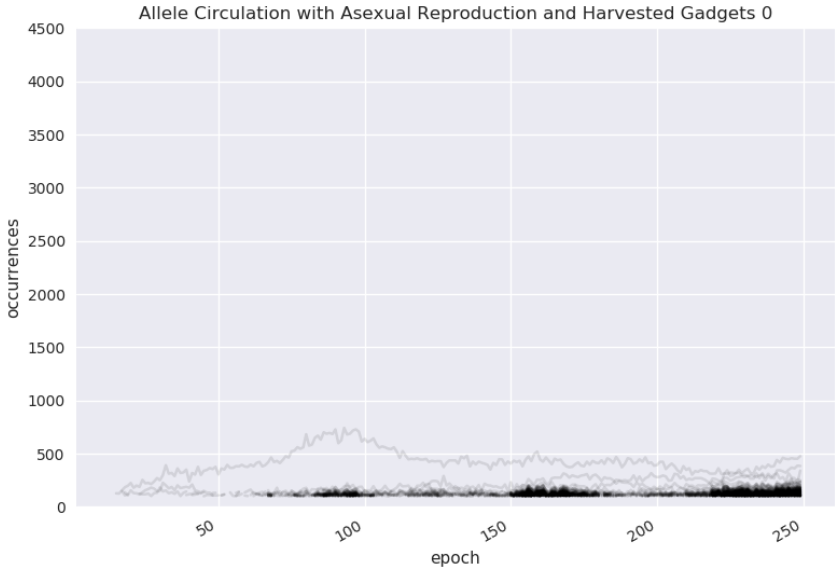


Figure 28: Allele circulation in an asexual population, seeded with harvested addresses

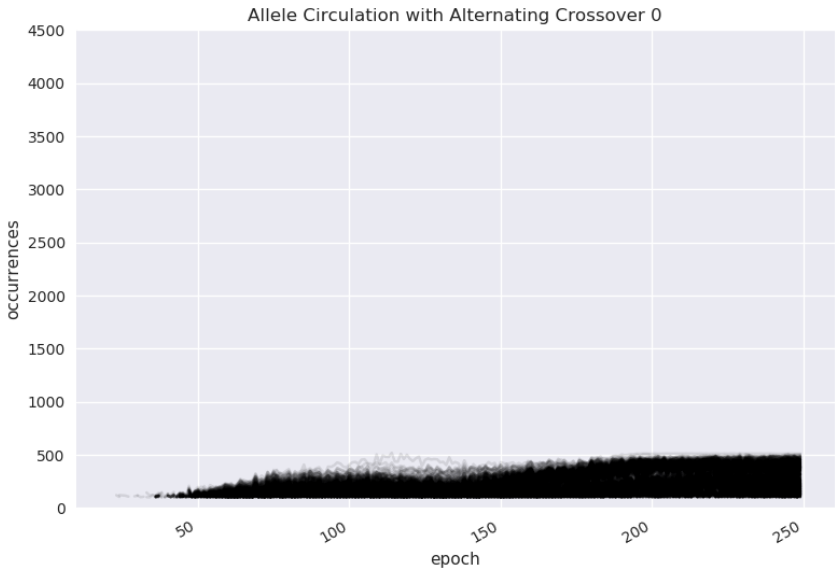
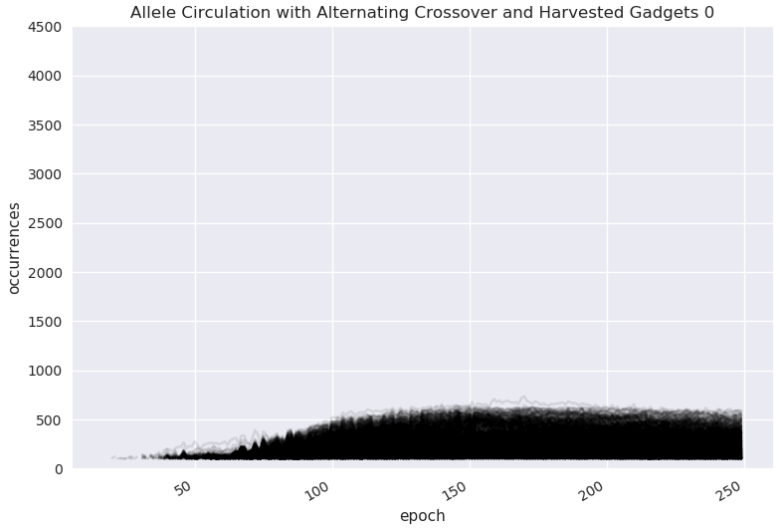


Figure 29: Allele circulation in a population reproducing through alternating crossover, seeded with random addresses



4. Generational distribution

   Another perspective on the effects of reproductive technique on the

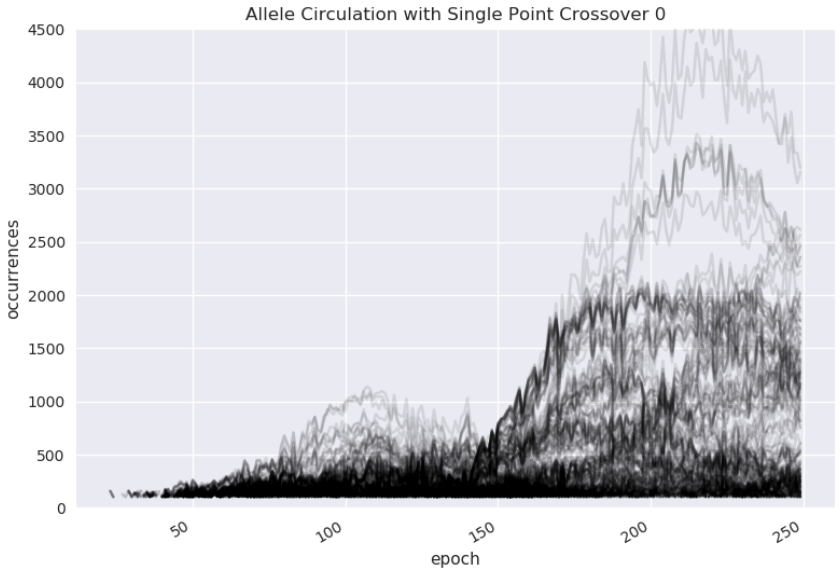


Figure 30: Allele circulation in a population reproducing through single-point crossover, seeded with random addresses

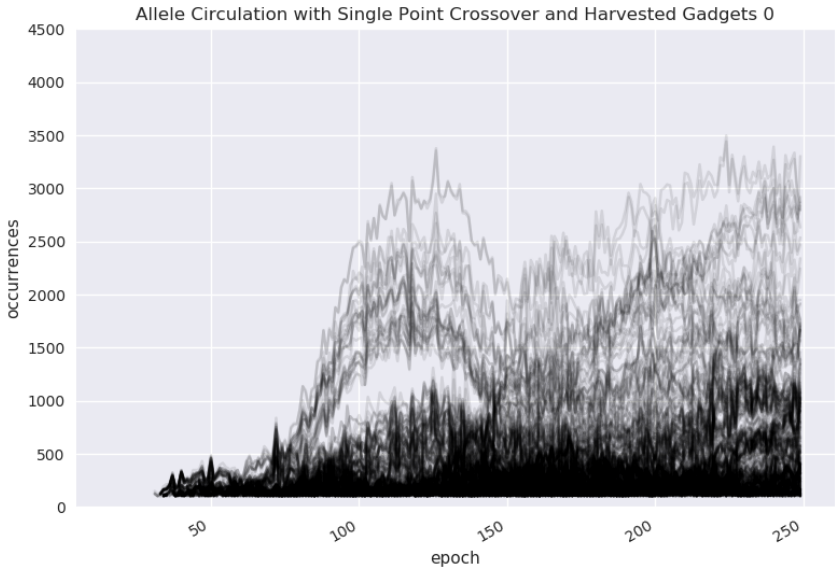


Figure 31: Allele circulation in a population reproducing through single-point crossover, seeded with harvested addresses

genetic makeup of our ROPER populations is provided in the following plots, in which every individual in the population, over the course of 250 epochs, is represented by a dot of varying hue, size, and position along the x and y axes.

The difference between these plots is immediately visible, though some features appear to be more significant than others. The tendency of the points in the asexual populations (figs. 32, 33) to tend to be of lower generation is easily explained: when two genomes of generation n and m produce an offspring through crossover, that offspring is assigned the generation $max(m, n) + 1$. When an asexual parent of generation n spawns a child, that child's generation is just $n + 1$.

The patterns we observed in the line plots are clearly visible here as well, and more. In the single-point crossover populations (figs. 36, 37) we see that the high-ret-count individuals are also those which have tend to be fitter (achieving high code coverage scores) and which, therefore, tend to have the greatest number of offspring.

A curious feature of the alternating-crossover populations (figs. 34, 35) is the preponderence of exceptionally heavy breeders early in the evolutionary process – 25th-generation individuals that have spawned upwards of 200 offspring, for instance.

2.3 Register Control

The task of evolving ROP payloads to set the register state to a determinate pattern was, naturally, one of the first problems we considered in this project. This was, in fact, one of the three problem domains tackled in the first iteration of ROPER, in the course of Lucca Fraser's graduate research. It was found to be a surprisingly difficult problem at the time, and continues to be so, today. Evolutionary computation, like many forms of stochastically-driven machine learning, truly shines in domains where problems and solutions have a bit of vagueness to them, but it has a hard time with exactitude.

The difficulty is compounded by the difficulty inherent in defining a reasonable distance metric between register states. What does it mean to be "near to" or "far from" a specified register pattern?

An *ideal* solution to this problem might be the following: let G be a graph whose vertices are CPU states and whose edges are the state transitions that can be effected by "gadgets" (composable sequences of instructions) in the target binary. Let each edge be weighted, perhaps, according to the

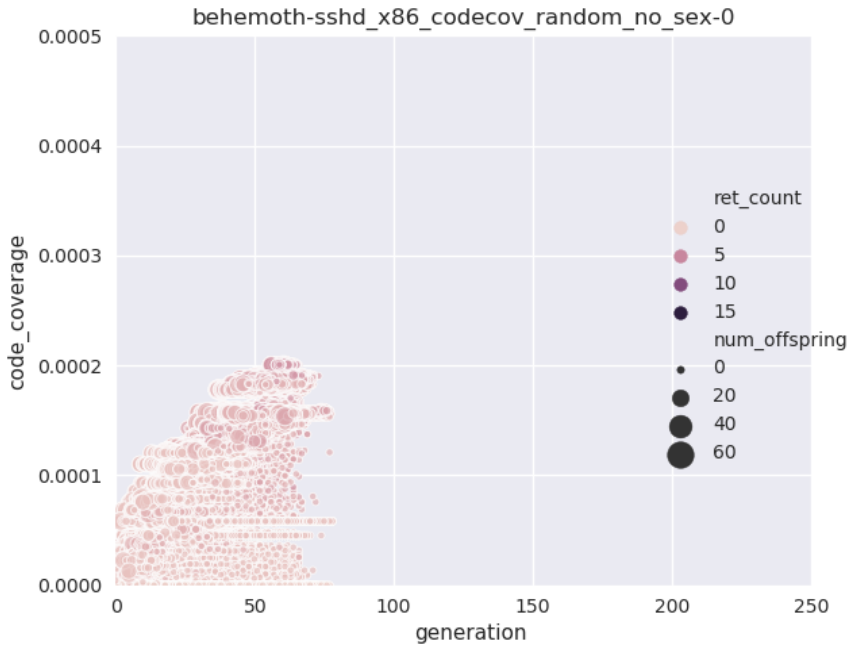


Figure 32: Generational distribution of asexually reproducing population, seeded with random addresses

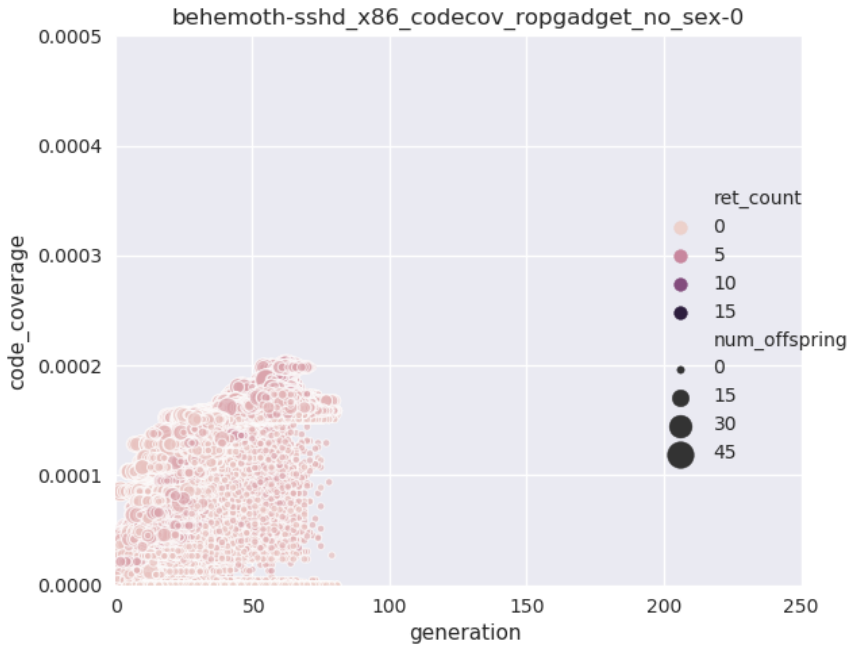


Figure 33: Generational distribution of asexually reproducing population, seeded with harvested addresses

Figure 34: Generational distribution of population reproducing through alternating crossover, seeded with random addresses

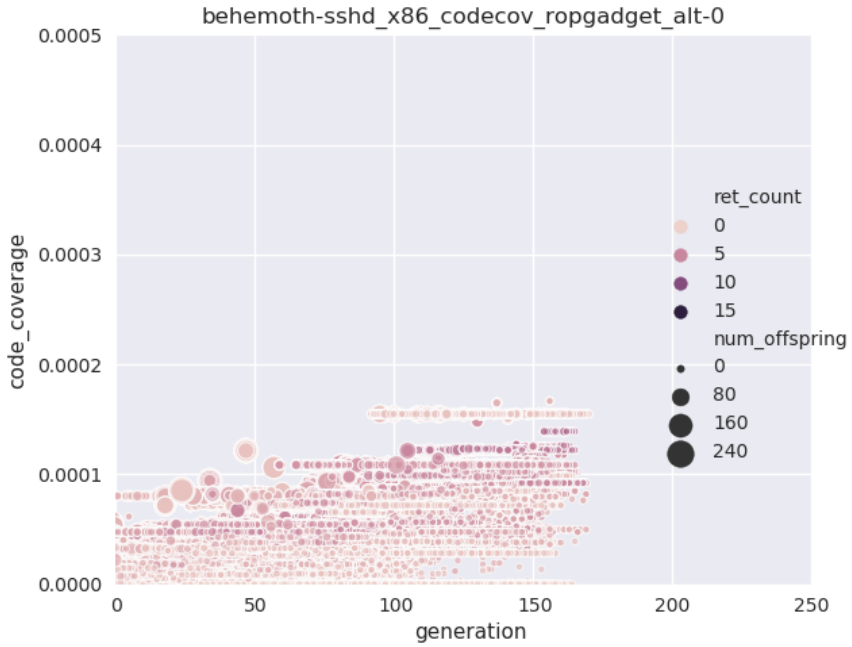


Figure 35: Generational distribution of population reproducing through alternating crossover, seeded with harvested addresses

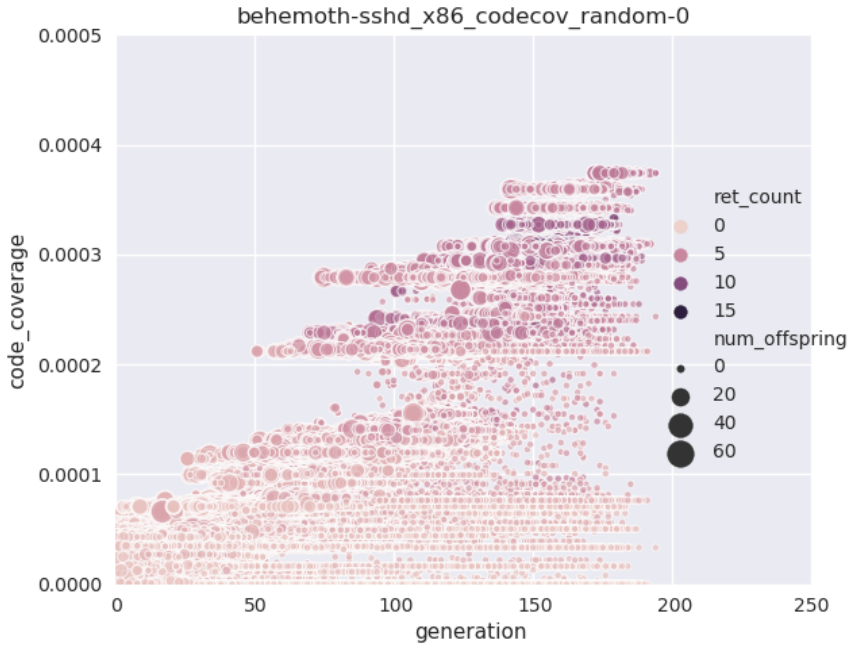


Figure 36: Generational distribution of population reproducing through single-point crossover, seeded with random addresses

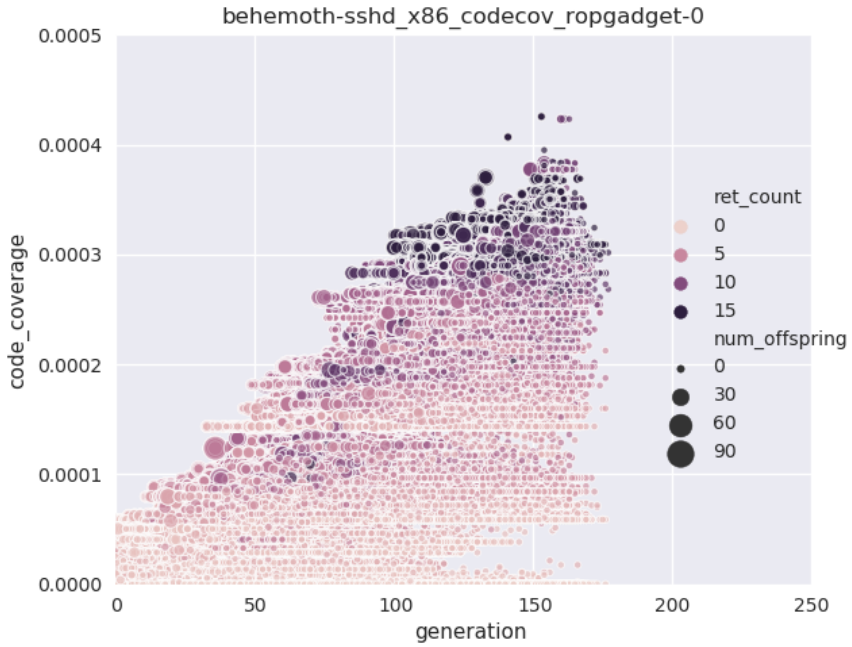


Figure 37: Generational distribution of population reproducing through single-point crossover, seeded with harvested addresses

frequency or genetic accessibility of those gadgets. Then let the distance between a given vertex n and the target state t be the shortest path between n and t in G.

This is unfeasible for a number of reasons. To begin with, the number of vertices, alone, of G is astronomically large. Even if we just count the register states on a 32-bit architecture, and restrict ourselves to, say, 4 registers, we're left with 2^{34} vertices! The number of possible transitions between these vertices is at least as large, and enumerating *those* would require, in addition, a complete semantic analysis of the binary in question. Storing such a monstrous graph, let alone computing its shortest paths `(O(|edges| + |vertices| log |vertices|)` in the worst case, if we use Dijkstra's algorithm), is simply beyond our meagre computational resources.

Once we accept that we cannot get what we want, in this case, we might still ask if we can get what we need: can a more or less reasonable, more or less informative, and, importantly, cheap distance metric be defined?

Two options present themselves:

1. if we restrict our attention to register states, we could treat a state as a vector of integers, and interpret that as the coordinates of a point in Euclidean space. We could then treat the distance between the current state and our target as Euclidean distance.

2. we could treat a register state as a vector of bits, and then take the *hamming distance* between the current state and our target.

Neither conception of distance maps very neatly onto the program space our populations are actually traversing, but this gives us a place to start.

One complication presents itself when we come to consider *indirect* values. If `EBX` needs to point to the value 0x44434241 (a little-endian representation of "ABCD" in ASCII), for example, how should we handle this? We could treat indirect or referenced variables as additional dimensions, if we add a special value to denote invalid references, or we could replace indirect target values with sets of pointers to that value which already reside in memory. Mutability raises a further complication. Should we count a pointer to, say, 0x44434200 to be "close" to the target, if the value resides in a writeable segment of memory?

The approach we took is a somewhat unhappy compromise with these various complications. We employed a *weighted hamming distance* measure for each value: for each register occurring in the target pattern, disagreeing with the *nth* least significant bit of its counterpart in the actual register state

adds $n+1$ to its distance from the target. If there are multiple potential targets, only distance from the nearest counts. This measurement is repeated for all registers and the first m nodes in the chain of references beginning from each register. A constant location penalty is applied to comparisons where there is a difference in location – if the value that we hope to get in `EAX` shows up in `EBX`, for example – but there is no sense in which some registers are nearer to one another than others (an analysis of the target binary's data flow graph could, theoretically, be used to establish a workable notion of register proximity, but we have not yet attempted to implement this).

The sum of these measures gives us the "distance" between the target register and memory state, and the CPU context effected by any given specimen's execution.

2.3.1 Parameters

Setting	Value
number of islands	8
max initial length	500
min initial length	450
island population size	1024
tournament size	5
number of parents	2
number of offspring	2
geographic radius	10
migration rate	0.01
initial soup size	0x40000
binary	OpenSSH 6.8p1 sshd for i386
max emulator steps	0x2000
max emulator time	5 milliseconds
emulator stack size	0x1000
allow function calls	no
fitness function	register pattern
weighting	register-error + 10 * register-freq
number of epochs	1000

The register pattern used for these experiments was:

Register	Value
EAX	0xb
EBX	&"/bin"
ECX	&0
EDX	0

2.3.2 Perfect solutions

Full solutions to the register pattern problem have been somewhat rare. In our run of 10 trials, each for up to 1000 epochs, only 2 arrived at perfect solutions. In both instances, these were in populations where a secondary novelty pressure was applied to the selective process: a count-min-sketch structure was used to log every *incorrect* register state (ignoring correctly set registers), and the sketch was then queried to obtain a frequency score for each particular error. This frequency score was then added to the fitness value, using the weighting formula shown above. The idea, here, is that *new* errors are preferable to old errors, and should be shown greater leniency.

In a disappointing turn, *none* of our trials using the Push VM as an ontogenic intermediary yielded consistently better results than bare ROP chain evolution. The silver lining is that the computationally cheaper technique appears to be just as good as its more elaborate and expensive rival, as far as we have seen.

Plots for these experiments can be found on our github repository.

1. First solution

```
Name: wiles-flied-nooks-whipt, from island 0
Generation: 2736

Trace:
----
80b5dfa:  89 f0                                           mov eax, esi
80b5dfc:  8b 4c 24 54                                     mov ecx, dword ptr [es
80b5e00:  25 00 00 00 c0                                  and eax, 0xc0000000
80b5e05:  01 c1                                           add ecx, eax
80b5e07:  03 44 24 58                                     add eax, dword ptr [es
80b5e0b:  81 e6 ff ff ff 3f                               and esi, 0x3fffffff
80b5e11:  89 c2                                           mov edx, eax
80b5e13:  74 39                                           je 0x80b5e4e
```

```
----
80b5e4e:  83 c4 3c                                        add esp, 0x3c
80b5e51:  b8 01 00 00 00                                  mov eax, 1
80b5e56:  5b                                              pop ebx
80b5e57:  5e                                              pop esi
80b5e58:  5f                                              pop edi
80b5e59:  5d                                              pop ebp
80b5e5a:  c3                                              ret
----
8075df7:  52                                              push edx
8075df8:  1c f6                                           sbb al, 0xf6
8075dfa:  c2 02 74                                        ret 0x7402


Spidered register state:
EAX: 0xb
EBP: 0x81606d5 RX -> 0x312e2520 " %.1"
EBX: 0x81606a8 RX -> 0x6e69622f "/bin"
ECX: 0x8049633 RX -> 0x0
EDX: 0x0
EIP: 0x8075dfa RX -> 0xe7402c2
ESP: 0x8218150 RW (stack) -> 0x0
```

2. Second solution

```
Name: corms-taxis-magma-wefts, from island 4
Generation: 951

Trace:
----
80badf1:  83 fb ff                                        cmp ebx, -
1
80badf4:  74 0f                                           je 0x80bae05
80badf6:  8d 4b 10                                        lea ecx, [ebx + 0x10]
80badf9:  31 d2                                           xor edx, edx
80badfb:  39 4c 24 5c                                     cmp dword ptr [esp + 0
80badff:  0f 85 99 01 00 00                               jne 0x80baf9e
80baf9e:  83 c4 3c                                        add esp, 0x3c
80bafa1:  89 d0                                           mov eax, edx
80bafa3:  5b                                              pop ebx
```

```
80bafa4:  5e                                              pop esi
80bafa5:  5f                                              pop edi
80bafa6:  5d                                              pop ebp
80bafa7:  c3                                              ret
----
80badf1:  83 fb ff                                        cmp ebx, -
1
80badf4:  74 0f                                           je 0x80bae05
80badf6:  8d 4b 10                                        lea ecx, [ebx + 0x10]
80badf9:  31 d2                                           xor edx, edx
80badfb:  39 4c 24 5c                                     cmp dword ptr [esp + 0
80badff:  0f 85 99 01 00 00                               jne 0x80baf9e
80baf9e:  83 c4 3c                                        add esp, 0x3c
80bafa1:  89 d0                                           mov eax, edx
80bafa3:  5b                                              pop ebx
80bafa4:  5e                                              pop esi
80bafa5:  5f                                              pop edi
80bafa6:  5d                                              pop ebp
80bafa7:  c3                                              ret
----
80badf1:  83 fb ff                                        cmp ebx, -
1
80badf4:  74 0f                                           je 0x80bae05
80badf6:  8d 4b 10                                        lea ecx, [ebx + 0x10]
80badf9:  31 d2                                           xor edx, edx
80badfb:  39 4c 24 5c                                     cmp dword ptr [esp + 0
80badff:  0f 85 99 01 00 00                               jne 0x80baf9e
80baf9e:  83 c4 3c                                        add esp, 0x3c
80bafa1:  89 d0                                           mov eax, edx
80bafa3:  5b                                              pop ebx
80bafa4:  5e                                              pop esi
80bafa5:  5f                                              pop edi
80bafa6:  5d                                              pop ebp
80bafa7:  c3                                              ret
----
8088fa1:  7d 94                                           jge 0x8088f37
8088f37:  ac                                              lodsb al, byte ptr [es
8088f38:  c3                                              ret
```

```
Spidered register state:
EAX: 0xb
EBP: 0x81e9182 RX -> 0xe00c0002
EBX: 0x8189e76 RX -> 0x6e69622f "/bin"
ECX: 0x80482dd RX -> 0x0
EDX: 0x0
EIP: 0x8088f38 RX -> 0xf13101c3
ESP: 0x82181f4 RW (stack) -> 0x80ad3ae RX -> 0x8b097400
```

References

[1] Christian Darabos, Mario Giacobini, Ting Hu, and Jason H. Moore. Lévy-flight genetic programming: Towards a new mutation paradigm. In Mario Giacobini, Leonardo Vanneschi, and William S. Bush, editors, *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, pages 38–49, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Olivia Lucca Fraser. Urschleim in silicon: Return-oriented program evolution with roper. 2018.

[3] Adi Livnat, Christos Papadimitriou, Jonathan Dushoff, and Marcus W. Feldman. A mixability theory for the role of sex in evolution. *Proceedings of the National Academy of Sciences*, 105(50):19803–19808, 2008.

[4] Lee Spector. Autoconstructive evolution: Push, pushgp, and pushpop. In A. Wu W.B. Langdon H.-M. Voigt M. Gen S. Sen M. Dorigo S. Pezeshk M. Garzon L. Spector, E. Goodman and E. Burke, editors, *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2001.

[5] Lee Spector and Jon Klein. *Trivial Geography in Genetic Programming*, pages 109–123. Springer US, Boston, MA, 2006.

[6] Reiko Tanese. Distributed genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, page 434439, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.