

URSCHLEIM IN SILICON:  
RETURN ORIENTED PROGRAM  
EVOLUTION WITH ROPER

by

Olivia Lucca Fraser

Submitted in partial fulfillment of the requirements  
for the degree of Master of Computer Science

at

Dalhousie University  
Halifax, Nova Scotia  
December 2018

© Copyright by Olivia Lucca Fraser, 2018

*This thesis is dedicated to my children, Kai, Nahní, Sophie, Quin, and  
Faro, and to Andrea Shepard.*

# Table of Contents

<b>List of Tables</b> . . . . .	<b>vi</b>
<b>List of Figures</b> . . . . .	<b>vii</b>
<b>Abstract</b> . . . . .	<b>1</b>
<b>Acknowledgements</b> . . . . .	<b>2</b>
<b>1 Introduction</b> . . . . .	<b>3</b>
1.1 Problem . . . . .	3
1.2 Synopsis . . . . .	4
<b>2 Weird Machines and Return-Oriented Programming</b> .	<b>5</b>
2.1 A Fundamental Problem of Cybersecurity . . . . .	5
2.2 Return and Jump Oriented Programming . . . . .	9
2.2.1 The C Abstract Machine Model . . . . .	10
2.2.2 The ROP Virtual Machine . . . . .	10
2.3 Prior Art: Exploit Engines and Weird Compilers . . . . .	12
2.4 Prospects for Genetic ROP-chain Crafting . . . . .	13
<b>3 On the History and Potential of Genetic Programming in Offensive Security</b> . . . . .	<b>14</b>
3.1 A Brief Overview of Genetic Programming . . . . .	14
3.2 Natural Selection Considered Harmful: Evolutionary Computation in Offensive Security . . . . .	14
3.2.1 Viruses and Evolutionary Computation . . . . .	15

	3.2.2	Genetic Payload Crafting . . . . .	24
	3.2.3	The Road Ahead . . . . .	25
<b>4</b>		<b>On the Design and Implementation of ROPER: Algorithmic Overview . . . . .</b>	<b>27</b>
	4.1	Gadget Extraction . . . . .	33
	4.2	Genotype Representation: Gadgets, Clumps, and Chains . . . . .	34
	4.3	Genetic Operators . . . . .	40
	4.3.1	Mutation . . . . .	40
	4.3.2	Crossover . . . . .	41
	4.3.3	Fragility and Gene Linkage . . . . .	41
	4.4	Ontogenesis and Evaluation . . . . .	44
	4.4.1	From Genotype to Phenotype . . . . .	44
	4.4.2	Ontogenesis of a ROP-chain . . . . .	45
	4.4.3	Fitness Functions . . . . .	46
	4.4.4	Fitness Sharing . . . . .	48
	4.5	Remarks on Implementation . . . . .	52
<b>5</b>		<b>Experimental Studies . . . . .</b>	<b>53</b>
	5.1	Fitness Functions . . . . .	53
	5.1.1	Classification. . . . .	54
	5.1.2	Testing the Extended Gadgetry Conjecture with Explicitly Defined Introns . . . . .	61
	5.2	Evolving 'Intelligent' Payloads . . . . .	61
	5.2.1	Fleurs du Malware . . . . .	61
	5.2.2	Snake . . . . .	63
	5.2.3	Aside: A Plague of Segfaults . . . . .	63
<b>6</b>		<b>Future Work . . . . .</b>	<b>64</b>
	6.1	Current Limitations and Open Problems . . . . .	64
	6.2	A More Robust and Evolvable Intermediate Language (Push) . . . . .	65
	6.2.1	Raising the Level of Abstraction with PUSH . . . . .	65
	6.2.2	Autoconstructive Reproduction . . . . .	66
	6.3	Semantic Analysis and Synthesis (Q) . . . . .	66
	6.4	Utility in the Wild with Blind ROP (Braille) . . . . .	66

7	Conclusions . . . . .	67
	Bibliography . . . . .	68

## List of Tables

4.1	Program Headers of a Typical ELF Executable . . . . .	36
5.1	A Rather Labyrinthine Chain . . . . .	56
5.2	Disassembly of a succesful chain, with ‘extended gadgets’. ** indicates where the pattern is completed. . . . .	57

# List of Figures

4.1	A bird's eye view of ROPER . . . . .	28
-----	--------------------------------------	----

# List of Algorithms

1	Population Initialization . . . . .	29
2	Genotype Evaluation (Ontogenesis) . . . . .	30
3	Evolve Population (Tournament Selection) . . . . .	31
4	Linear sweep algorithm for gadget extraction. . . . .	35
5	Spawning an Initial Individual . . . . .	39
6	Single-Point Crossover, with Fragility . . . . .	42
7	Headless Chicken Patch . . . . .	52




Abstract

abstract!



## ~~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/>.~~

# 1

## Introduction

### 1.1 Problem



This thesis explores the use of evolutionary techniques in return-oriented programming.<sup>1</sup> It details the design and implementation of an engine called ROPER, which employs the methods of genetic programming to generate declaratively specified ROP payloads from scratch, and walks through a series of experiments that establish the feasibility of this approach. Since this is, to the best of my knowledge, the first time that evolutionary techniques have been put to work in the field of return-oriented programming, my intention is only to establish a *proof of concept*, rather than to advance the state of the art in terms of performance and precision<sup>2</sup>

The “crafted input” by means of which a hacker controls the execution of an exploited system is typically best understood as a sequence of instructions for a previously unknown virtual machine, whose supervenience on the intended machine

---

<sup>1</sup>“Return-oriented programming” is defined in Section 2.2.2.

<sup>2</sup>Unless we allow the casting of a null “state of the art” to zero.

is accidental, and often unknown before it is exploited. These payloads tend to be short, highly constrained by contingent pressures, and forged from obscure and irregular materials. These factors, which tend to greatly increase the ratio of difficulty to functionality in payload implementation, for human programmers, also make the problem well suited to evolutionary approaches. This, at least, was the intuition that sparked this project. The hope is that by putting evolutionary techniques to work in this field, we can better explore and understand the algorithmic wilderness that supervenes on our machines, and gain a deeper sense of the possibilities harboured there.

## 1.2 Synopsis

/ rewrite: rearranged chapters

In Chapter 2, I set up some of the conceptual background for this study, explaining the broader problems broached by return-oriented programming (ROP) (Section 2), and genetic programming (GP) (Section 3), before surveying a handful of historical efforts to enlist evolutionary techniques in the domain of offensive security and malware design (Section 3.2).



Chapter 4 introduces my contribution to research in the field of evolutionary offensive security, with an overview of the design (Section 4) and implementation (Section ) of a return-oriented program evolution engine called ROPER.

Chapter 5 goes over a handful of experimental studies with ROPER, and consequent modifications to the design.

Chapter 6 ~~lays~~ <sup>lays</sup> out some directions for future work and study on this ~~topic, and~~ <sup>topic,</sup> and ~~Chapter 7 brings this thesis to a conclusion.~~

*Between the idea  
And the reality  
Between the motion  
And the act  
Falls the Shadow*

T.S. Eliot, "The Hollow Men"

# 2

## Weird Machines and Return-Oriented Programming

### 2.1 A Fundamental Problem of Cybersecurity

The distinction between data and code tends to fade from view as we approach the most elementary strata of computation – whether we are dealing with the austere formalism of the lambda calculus, the ideal Von Neumann machine model, or the various instruction set architectures that concretize it.<sup>1</sup>

But at any level where one computational system interfaces with another, “in the real world”, the problem of imposing and maintaining this distinction is critical – even, I would argue, the *fundamental* problem of cybersecurity. What we call data, generally speaking, is information that one system (A) receives from another (B), or the result of applying any sequence of transformations to that information. Insofar

---

<sup>1</sup>And, as we’ll see, machine models that *appear* to take such a distinction as primitive, such as the Harvard Bus model, often only succeed in draping a thin and permeable veil between the two.

as we are to have any assurances at all about the behaviour of system A, A must, by design, place some constraints on how it lets itself be steered by the data it receives – unless, of course, it is *intended* to be a general programming environment.<sup>2</sup> Data is *just* data only to the extent that such constraints can hold.

Nothing makes this clearer than remote code execution (RCE) attacks, each of which can be seen as a “proof by construction” that what we assumed to be “merely data” was in fact code for a machine that we didn’t understand.<sup>3</sup> In many such cases, the breach occurs when the attacker slips past the *intended* interface and dispatches instructions (perform state transitions) on one or more of the system’s “internal” components. Take the classic SQL injection attack, for example. The attack succeeds when the attacker crafts the input data to the system in such a way that the system interpret some portion of that data as code. In the simplest cases, this may be done by inserting a single quotation mark in the text provided to an input field. If this input is not safely parsed by the frontend, then any text *following* the delimiting quote will be interpreted as additional SQL instructions, and executed by the backend. The injected delimiter plays the role of an unsuspected pivot between data and code, switching the context of the input string to an SQL execution environment. Something similar happens in the classic style of buffer overflow attack described in Aleph One’s famous textfile, “Smashing the Stack for Fun and Profit”. The *pivot*, in that case, is achieved by the attacker supplying an input string that the vulnerable application writes to a buffer that has not been allocated enough space to contain it. In many cases, this gives the attacker the ability to write to stack memory “beneath” the ill-sized buffer. What makes this dangerous is that, according to a certain, widely implemented abstract machine model, the return address of each subroutine is often stored on the stack as well, just a few words below the space where local variables are stored. This lets the attacker control the return address, which can be redirected to *another* region of the input data, where the attacker has encoded a sequence of machine code instructions for the vulnerable system’s CPU. In these cases, and in many, many more, the attacker succeeds in exploiting some oversight in the design or implementation of the input

---

<sup>2</sup>Of course, many programming language environments, usually in hopes of improving the security of the code developers write with them, *do* seek to constrain the freedom and power of the programmer, in ways that, according to taste, range from elegant to irritating.

<sup>3</sup>I owe this formulation to Sergey Bratus.

handler, in such a way that the vulnerable system treats some portion of the input just as it would treat its own code. In each of these cases, it's possible to distinguish two distinct moments:

1. the delivery mechanism, or “pivot”, of the attack, by which the input “data” is transubstantiated into “code” – the aberrant delimiter in the SQL injection, or the corruption of the instruction pointer, in the case of the buffer overflow, for example, and
2. the “payload”, through which the attacker exercises fine-grained control over the vulnerable system. In the case of the buffer overflow attack, this may be a string of shellcode. In the case of the SQL injection, a sequence of one or more SQL expressions or operations.

This is the general outlook that seems to motivate most defensive tactics in computer security. Take, for instance, a tactic that has been widely deployed in an effort to defend against shellcode attacks. These attacks play on the fact that, to the CPU, “code” is wherever it points its program counter<sup>4</sup>. The stack overflow vulnerability detailed by Aleph One is one such delivery mechanism, but the general strategy of feeding the vulnerable system machine code instructions in the form of input data, and then redirecting the program counter so that it points to that data, and executes it as code, has other forms as well – such as use-after-free attacks, which may exploit a lack of coordination in heap memory management to overwrite a virtual function pointer (an object method, for example) with a pointer to the attacker's shellcode. Defensive measures against these attacks, then, could follow two distinct prongs: on the one hand, we could inhibit the *pivot* stage, or on the other inhibit the *payload*.

With respect to the pivot stage, buffer overflow attacks can be prevented, piecemeal, by carefully constraining the data that's written to fixed-length buffers on the stack (use `strncpy()` instead of `strcpy()`, etc.). They can also be mitigated by the compiler, by inserting a random string as a sort of tripwire between the writeable stack buffer and the return address, such that any attempt to overwrite that portion

---

<sup>4</sup>Also called the “instruction pointer”, the program counter is a special register various implementations of the Von Neumann machine, which directs the CPU to the next instruction to fetch from memory and execute. On x86 machines, it is instantiated by the EIP register; on x86<sub>64</sub>, by RIP; on MIPS and ARM, it is called 'PC'.

of the stack would also corrupt this randomized value or “stack canary”. Neither of these mitigations prevent a block of malicious code that the attacker has written to memory from being executed, should some other means of corrupting the instruction pointer become available.

The sort of attack that Aleph One describes could also be prevented by mitigating the attacker’s ability to pass control to the payload, rather than their ability to achieve the initial corruption of the instruction pointer. This is what is achieved, for example, through what Windows natives call “Data Execution Prevention” (DEP), and what Unix dwellers call, a bit less pronounceably, “Write xor Exec” ( $W \oplus X$ ), whereby the memory pages of a running process may be mapped as writeable, or may be mapped as executable, but may no longer be mapped as *both*.<sup>5</sup> With this mitigation in place, the attacker may succeed in corrupting the instruction pointer, and may succeed in loading their attack code into memory, but is unable to pass control to the latter – an instruction pointer dereferenced to a non-executable location in memory will result in a segmentation fault (as Unixers call it) or an access violation error (as it’s known in Windows), which may succeed in crashing the program, and thereby carrying out a non-trivial denial-of-service (DoS) attack, but at no point does the attacker achieve fine-grained control of the process.

There is another way of looking at all of this, which is both more general and more fruitful. As hacker folklore is fond of repeating, what we call a system’s “code” is, in some sense, nothing but *the specification of a state machine driven by the input data*.<sup>6</sup> As Halvar Flake explains, to write a program is to constrain the virtually boundless potential of a general computer so as to have it emulate “a specific finite-state machine that addresses your problem”. “The machine that address the problem,” he go on, o

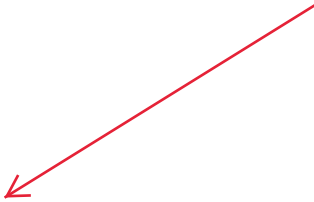
The machine that addresses the problem is the **intended finite state machine** [...]

---

<sup>5</sup>When Microsoft first introduced DEP into their products, with Windows XP Service Pack 2 (<https://support.microsoft.com/en-us/kb/889741>), they advertised it as “Protecting against Buffer Overflows”, confusing a mitigation of the *payload* of the classical buffer-overflow-shellcode-attack (which they delivered) with a mitigation of its *pivot* (which they did not). This was before ROP attacks became well-known. Sergey Bratus points to this confusion as an illustration of the following principle: we never really understand a security feature until we understand how to exploit and subvert it [5].

<sup>6</sup>“Any computing device that accepts input and reacts to this input by executing a different program path can be viewed as a computing device where the *inputs* are the program” ([8]).





The security properties of the IFS are 'what we want to be true' for the IFSM. This is needed to define 'winning' for an attacker: He wins when he defeats the security properties of the IFSM.

Assuming that there has been no trivial misconfiguration of the IFSM, and that it is, on its own, particular, level of abstraction, consistent, the attacker defeats those security properties by ferreting out a leak in the programmer's abstraction, and tapping into a reserve of computational power that the programmer had considered foreclosed by the IFSM. This is done by, *first*, finding a way to access a state from the IFSM that is not accounted for by the IFSM's design. These are what Sergey Bratus [CITE] calls "weird states". An example is the state that the CPU enters when its instruction pointer has been overwritten by input. This is what we have called the *pivot* of the attack.

Once a weird state is entered, many other weird states can be reached by applying the transitions intended for sane states on them. A new computational device emerges, the *weird machine*. The *weird machine* is the computing device that arises from the operation of the emulated transition of the IFSM on weird states.

... Given a method to enter a weird state from a set of particular sane states, *exploitation* is the process of:

1. setup (choosing the right sane state)
2. instantiation (entering the weird state), and
3. programming of the weird machine

so that security properties of the IFSM are violated.

The concept of a *weird machine* opens onto an extremely versatile and general theory of exploitation. [finish thought]

## 2.2 Return and Jump Oriented Programming

It is due to a leaky abstraction of this nature, and an unswerving view of the underlying CPU *from the perspective of application programmers and compilers*, that  $\mathbf{X} \oplus \mathbf{W}$

ultimately fails to prevent remote code execution. It fails because it is built on an insufficiently general concept of *code*.

### 2.2.1 The C Abstract Machine Model

According to this model, computation proceeds by iterating through a buffer of instructions in a designated segment of memory, using a designated register, the “program counter” or “instruction pointer”, to track the location of the next instruction to execute (we’ll call this the IP when referring to its abstract role, but its concretization has different names on different architectures – EIP on x86, RIP on x86<sub>64</sub>, PC on ARM, etc.) Each instruction prompts the processor to mutate its state (its registers, memory, etc.) in some fashion. “Code” is wherever the IP points, and the instruction set is fixed by the architecture.

On this basis is implemented the *procedural* layer of abstraction, which the underlying architecture is largely designed to accommodate. According to this layer, a program is typically broken up into a collection of *subroutines* (or “functions”). A subroutine is characterized by two essential properties:

1. it has a local variable scope, and
2. it can be run, or “called”, as a cohesive unit, with execution *returning* to the place it is run from once it completes. Abstractly, both

of these properties rely on the *stack* data structure. Both the scopes, and the execution flow, of subroutines, is organized in a first-in-last-out fashion.

Interestingly, though they are *conceptually* distinct, the data stack and the execution stack are typically *interleaved* in practice.

This interleaving is orchestrated, on most modern architectures, by means of three abstract registers: the *stack pointer* (SP),

### 2.2.2 The ROP Virtual Machine

A ROP chain can be seen as actualizing a somewhat weirder machine, which just happens to supervene on the same process mobilized by the programmer’s machine model, the process that is *supposed* to be executing a perfectly normal program. Let’s call this a ROPVM.

Like the programmer’s machine model, the ROPVM works by iterating through a sequence of instructions, tracking the location of the next instruction by means of a special register, and in the process mutates the CPU context. But the instruction set used for this machine is *not* the instruction set targetted by its host. It is an emergent instruction set, peculiar to the state of conventionally executable memory at the time of the pivot. These instructions are called “gadgets”, and are composed of chunks of data that is:

1. already mapped to executable memory – on Unix systems, this generally means the `.text` section of the binary;
  2. performs some mutation of CPU context when conventionally executed, and
  3. returns control of execution flow to the attacker-supplied data after executing.
- Trait #3 is typically satisfied by choosing gadgets that

end with a **return** instruction, or some semantic equivalent – any combination of instructions that results in a value from the stack being loaded into the instruction pointer. This can also be accomplished by means of a combination of **load** and **jump** instructions, which gives us “JOP”, or jump-oriented-programming, but the difference between JOP and ROP is not critical here, and for our purposes “ROP” will be used to refer to both varieties.<sup>7</sup>

The ROPVM is, in some sense, an essentially parasitic, or supervenient, creature. Its instruction set is cobbled together from chunks of machine code whose frequency in the victim process is largely a result of the process’s intended code being crafted with the procedural abstraction in mind.

This point is worth dwelling on for a moment, because it illustrates ROP’s ingenuity beautifully.  $W \oplus X$ , after all, *prevents* the data stack, which needs to remain writeable by the process, from being used as a code buffer, the way it is in a shellcode attack. But the schematic idea of *code* that  $W \oplus X$  guards against is code as understood by the programmer’s machine model. The ROPVM *is* able to use the data stack as a code buffer because it represents a change in perspective regarding what counts as code, what counts as an instruction, and what counts as an instruction pointer.

---

<sup>7</sup>Cite the passage on free branches, from that thesis on ARM ROPs.

Even when a strict separation of “data” and “code” is in place (via  $W \oplus X$ , and/or the hardware restrictions imposed by a Harvard Bus architecture), the PMM *expects* an interleaving of the control and data stacks, and so cannot very well ban the presence of code segment pointers from its stack, or prevent the loading of the pointer at the top of its stack into its own designated instruction pointer. But these two factors are all that are needed in order to superimpose the ROPVM on top of the PMM: we don’t need to execute PMM-level instructions from the stack, we just need to be able to use *data* on the stack to *influence* the execution of instructions, in a fine-grained fashion. But this is just what the **return** instruction does, in the PMM: it fetches data from the top of the stack, maps that data to an address in its own code buffer, and then executes the instructions it finds there, until it is instructed to fetch the next pointer from its stack. In this way, the PMM already *implies* the possibility of the ROPVM, which is its shadow. The PMM’s interleaving of control stack and data stack makes the principled separation of the writeable and the executable all but futile, since the latter represents a true separation of **code** and **data** only if the abstract machine model stays fixed.

Between the programmer’s abstract machine model, and the actual behaviour of the CPU, between the specification and the implementation, falls the shadow.

### 2.3 Prior Art: Exploit Engines and Weird Compilers

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 very handy `mona.py`<sup>8</sup> – to tools We are aware of two such projects at the moment: *Q* [14], 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.<sup>9</sup> The latter has since spawned a fork that aims to use

<sup>8</sup><https://github.com/corelan/mona> which approach the problem through the lens of compiler design, running with the insight that the set of gadgets out of which we build ROP chains is, in fact, the instruction set for a virtual machine, which can be treated as just another compiler target.

<sup>9</sup><https://github.com/pakt/ropc>

ROPC’s own intermediate language as an LLVM backend, which, if successful, would let programs written in any language that compiles to LLVM’s intermediate language, compile to ROPC-generated ROP-chains as well.

Another, particularly interesting contribution to the field of automated ROP-chain generation is *Braille*, which automates an attack that its developers term “Blind Return-Oriented Programming”, or BROP [3]. BROP solves the problem of developing ROP-chain attacks against processes where not only the source code but the binary itself is 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 I will outline here.

## 2.4 Prospects for Genetic ROP-chain Crafting

To the best of our knowledge, no attempt has yet been made to bring evolutionary methods to bear on the problem of ROP-chain generation; there is little precedence, in fact, for any use of genetic techniques to craft exploits.

*The biological analogy was obvious; evolution would favor such code, especially if it was designed to use clever methods of hiding itself and using others' energy (computing time) to further its own genetic ends. So I wrote some simple code and sent it along in my next transmission. Just a few lines in Fortran told the computer to attach these lines to programs being transmitted to a certain terminal. Soon enough – just a few hours – the code popped up in other programs, and started propagating.*

Gregory Benford, Afterword to "The Scarred Man"

# 3

## On the History and Potential of Genetic Programming in Offensive Security

### **3.1 A Brief Overview of Genetic Programming**

### **3.2 Natural Selection Considered Harmful: Evolutionary Computation in Offensive Security**

While evolutionary techniques have been more or less frequently employed in the field of *defensive* security – where they are put to work much in the same way as other machine learning algorithms, and built into next-generation firewalls, intrusion-detection systems, and so on – there has been far less exploration of these techniques in the realm of offensive security. This is not to say, however, that the idea has never occurred to anyone – the idea seems to have captured the imagination of hackers, malware engineers, and cyberpunk science fiction authors, ever since there have been such things.

### 3.2.1 Viruses and Evolutionary Computation

#### 3.2.1.1 1969: Benford

The oldest occurrence of the concept of evolving, intrusive code that I was able to excavate dates to sometime around 1969, in an experiment performed – and subsequently extrapolated into fiction – by the astrophysicist and science-fiction author, Gregory Benford, during his time as a postdoctoral fellow at the Lawrence Radiation Laboratory, in Livermore, California. “There was a pernicious problem when programs got sent around for use: ‘bad code’ that arose when researchers included (maybe accidentally) pieces of programming that threw things awry,” Benford recalls of his time at the LRL. `#+BEGINQUOTE` One day [in 1969] I was struck by the thought that one might do so intentionally, making a program that deliberately made copies of itself elsewhere. The biological analogy was obvious; evolution would favor such code, especially if it was designed to use clever methods of hiding itself and using others’ energy (computing time) to further its own genetic ends. So I wrote some simple code and sent it along in my next transmission. Just a few lines in Fortran told the computer to attach these lines to programs being transmitted to a certain terminal. Soon enough – just a few hours – the code popped up in other programs, and started propagating. `#+ENDQUOTE` Benford’s experiments unfolded in relative obscurity, apart from inspiring a short story that he would publish in the following year, entitled “The Scarred Man”. As far as we can tell, however, the invocation of “evolution” remained entirely analogical, and did not signal any rigorous effort to implement Darwinian natural selection in the context of self-reproducing code. It was nevertheless an alluring idea, and one that would reappear with frequency in the young craft of virus programming.

#### 3.2.1.2 1985: Cohen

Though anticipated by over a decade of scattered experiments, the **concept** of “computer virus” made its canonical entrance into computer science in the 1985 dissertation of Fred Cohen, at the University of Southern California, *Computer Viruses*. *Computer Viruses* is a remarkable document. Not only does it provide the first rigorously formulated – and *formalized* – concept of computer virus, which Cohen appears to

have discovered independently of his predecessors (whose work was confined to obscurity and fiction), explore that concept at the highest possible level of generality, in the context of the Turing Machine formalism, develop an elegant order-theoretic framework for plotting contagion and network integrity, leverage language-theoretic insights to subvert hypothetical (because it did not yet exist!) anti-virus software through Godelian diagonalization, and suggest a number of defenses, such as the cryptographic signing of executables, which are still used today, it also hints – elliptically – at the potential for viral evolution. At first glance, what Cohen calls the *evolution* of a virus resembles what would later be called *polymorphism* or even *metamorphism*

– the process of altering the *syntactic* structure of the pathogen in the course of infection, so that the offspring is not simply a copy of the parent. This is indeed enough to expose the virus to a certain amount of differential selective pressure, so long as antiviral software (the virus’s natural predator) pattern matches on the virus’s syntactic structure (the precise sequence of opcodes used), or on some low-level features on which the syntax supervenes (one or more bitwise hashes of the virus, for example). But Cohen goes a step further than this, and considers a far broader range of infection transformations that do *not* preserve semantic invariants. That is to say, he considers reproduction operators – operators embedded in the virus itself, which, following Spector <sup>1</sup> I can call “autoconstructive operators” – which generate semantically dissimilar offspring.

Cohen thus deploys all the essential instruments for an evolutionary treatment of viruses:

1. reproduction with variation (the “genetic operators”)
2. selection (detection by recognizers, or “antivirus” software)
3. differential survival (there is no recognizer that can recognize every potential virus, as a corollary of Rice’s theorem) :CITE:

He goes no further in systematizing this dimension of the problem, unfortunately, and nowhere in this text do we find anything that either draws on or converges with

---

<sup>1</sup>[ add footnote ]



contemporaneous research into evolutionary computation as a mechanism for program discovery or artificial intelligence.

Cohen can hardly be blamed for this, of course. The dissertation as it stands is a work of rare ambition and scope. The casual observer of virus research and development over the past three decades, however, might be surprised by the impression that so little has been done to bridge the distance that lay between it and study of evolutionary computation. While the rhetoric surrounding the study of computer viruses remained replete with references to evolution, to ecology, to natural selection, and so on, [MEMO: CITATIONS ] efforts to actually integrate the two fields appear to have been rare.

This impression is not wholly accurate, however. Closer study shows us that the experimental fringe of the VX scene has indeed retained an interest in exploring the use of genetic methods in their work. If this has gone relatively unnoticed by the security community, this is likely for one or two reasons:

1. the VXers who have implemented genuinely evolutionary methods in their work seem to be motivated primarily by hacker's curiosity and not by monetary gain.  
<sup>2</sup> The viruses they write are intended to be more playful than harmful, and it appears that several of the evolutionary viruses I have found were sent directly by their authors to antivirus researchers, or published, along with source code and documentation, on publically accessible websites and VXer ezines.
2. Of course, we should consider the non-negligible selection effect implied in reason #1: it's not surprising that the viruses *that I was able to find* in the course of writing this chapter are those circulated by the grey-hat VXer community, as opposed to those developed, or contracted, by intelligence agencies and criminal syndicates, who tend to hold somewhat more stringent views on matters of intellectual property. And so a second, plausible-enough explanation presents itself: it is possible that far less playful evolutionary viruses *do* exist in the wild, but that they tend to either go undetected, are used primarily for targeted operations less exposed to the public, or that they are not being properly recognized or reported in the security bulletins released by the major antivirus companies.

### 3.2.1.3 Nonheritable Mutations in Virus Ontogeny

For reasons of stealth, virus writers have explored ways of incorporating variation into their mechanisms of infection and replication. The first trick to surface was simple encryption, employed for the sake of obfuscation rather than confidentiality. This first became widely known with the Cascade virus in [YEAR]. Viruses using this obufscation method would encrypt their contents using variable keys, so that the bitwise contents of their bodies would vary from transmission to transmission. The encryption engine itself, however, would remain unencrypted and exposed, and so antiviral software simply looked for recognizable encryptors instead.

Next came oligomorphic viruses, starting with Whale in [YEAR]. These would use one of a fixed set of encryption engines, adding some variability to the mix. This would make the problem of detection some 60 or 90 times harder, depending on the number of engines, but such distances are easily closed algorithmically.

Next came polymorphic engines, which would scramble and rebuild their own encryption engine with each transmission, while preserving all the necessary semantic invariants. The antivirus developers countered by running suspicious code in emulators, waiting until the body of the virus was decrypted before attempting to classify it.

The last and most interesting development in this (pre-genetic) sequence rests with *metamorphic* viruses, which redirected the combinatorial treatment that polymorphics reserved for the encryption engine onto the virus body as a whole. There was no longer any need for encryption, strictly speaking, since the purpose of encryption in polymorphism is to obfuscate, not to lock down, and this allowed viruses to avoid any reliance on the already somewhat suspicious business of decrypting their own code before running.

In biological terms, what we're seeing with both polymorphic and metamorphic viruses is a capacity for ontogenetic variation. While it is possible for the results of metamorphic transformations to accumulate over generations, in most cases (unless there are bugs in the metamorphic engine), these changes are semantically neutral, and do not affect the functionality of the code (though this raises a subtle point regarding what we are to count as 'functionality', especially when faced with detectors that turn syntactic quirks and timing sidechannels into a life-or-death matter for the

virus). They are also, in general, reversible, forming a group structure. So long as they are not subjected to selective pressure, and complex path-dependencies don't form, the 'evolution' of a metamorphic virus typically has the form of a random walk.

It is nevertheless evident how close we are to an actual evolutionary process.

#### **3.2.1.4 2000: W32.Evol**

#### **3.2.1.5 2002: MetaPHOR (W32/Simile, {W32,Linux}/Simile.D, Etap.D)**

In 2002, Mental Driller developed and released a virus that bridged the gulf between metamorphic viruses and a new variety of viruses that could be called “genetic”. MetaPHOR is a highly sophisticated metamorphic virus, capable of infecting binaries on both Linux and Windows platforms. Written entirely in x86 assembly, it includes its own disassembler, intermediate pseudo-assembly language, and assembler, as well as a complex metamorphic and encryption engines. Its metamorphic engine mutates the code body through instruction permutation, register swapping, 1-1, 1-2, and 2-1 translations of instructions into semantic equivalents, and the injection of 'garbage code', or what we will later call “semantic introns”.

But the final touch, which elevates this program to evolutionary status, is the use of a simple genetic algorithm, which is responsible for weighting the probabilities of each metamorphic transformation type. As Mental Driller comments in the MetaPHOR source code: `#+BEGINQUOTE` I have added a genetic algorithm in certain parts of the code to make it evolve to the best shape (the one that evades more detections, the action more stealthy, etc. etc.). It's a simple algorithm based on weights, so don't expect artificial intelligence :) (well, maybe in the future :P). `#+ENDQUOTE` The way it works is that each instance of the virus carries with it a small gene sequence that represents a vector of weights – one for each boolean decision that the metamorphic engine will make when replicating and transforming the virus, in the process of infection. These are modified

a little with each replication. The hope is that the selective pressure imposed (for free!) by antiviral software will select for strains of the virus that have evolved in such a way as to favour transformations that evade detection, and shun transformations that give the virus away. (Descendents of the virus, for instance, may adapt in such a

way as to never use decryption, if that should turn out to be a tactic that attracts the scanners' attention, in a given ecosystem. Or they may evolve to be less aggressive in infecting files on the same host, or filter their targets more carefully according to filename.

### 3.2.1.6 2004-2005: W32/Zellome

The frequent invocation of ecological and evolutionary tropes in virus literature, combined with the lack of any genuine appearance of evolutionary malware, has led many to speculate as to its impossibility. The most frequently cited reason

for the unfeasibility of viral evolution is *computational brittleness* – the claim being that the machine languages (or even scripting languages) that most viruses are implemented in are relatively intolerant to random mutation. The odds that a few arbitrary bitflips will result in functional, let alone 'fitter', code is astronomically small, these critics reason. This is in contrast to the instruction sets typically used in GP and ALife, which are *designed* to be highly fault-tolerant and evolvable.

This is so far from being an insuperable obstacle that it suggests its own solution: define a more robust meta-grammar to which genetic operators can be more safely applied, and use those higher-level recombinations to steer the generation of low-level machine code.

We can find this idea approximated in a brief article by ValleZ, appearing in the 2004 issue of the VXer ezine, 29A, under the title "Genetic Programming in Virus". The article itself is just a quick note on what the author sees as interesting but in all likelihood impractical ideas. `#+BEGINQUOTE` I wanna comment here some ideas i have had. They are only ideas... these ideas seems very beautiful however this seems fiction more than reality. `#+ENDQUOTE` ValleZ goes on to sketch out the main principles behind genetic programming, and then gets to the crux of the piece: "how genetic programming could be used in the virus world".

As already noted, most of the essential requirements for GP are already present in viral ecology: selective pressure is easy to locate, given the existence of antiviral software, and replication is a given. However, ValleZ notes, the descendant of a virus tends to be (semantically) identical to its parent, and even when polymorphism or metamorphism are used, the core semantics remain unchanged, and there is no

meaningful accumulation of changes down generational lines.

(Conjecture: If we were to picture the distribution of diversity in the genealogy of a metamorphic virus, for instance, we would see a hub-and-spoke or starburst design in the cluster, with no interesting progressions away from the centre. Take a look at the Eigenvirus thesis to see if there’s any corroboration there.)

ValleZ suggests the use of genetic search operators – mutation, and, perhaps, in situations where viruses sharing a genetic protocol encounter one another in the same host, crossover – in virus replication. They would take over the work that is usually assigned to polymorphic engine, with the added, interesting feature of generating enough semantic diversity for selective pressures to act on. But for this to work, they note, it would be necessary to operate not on the level of individual machine instructions (which are, as noted, rather brittle with respect to mutation) but higher-level “blocks”, envisioned as compact, single-purpose routines that the genetic operators would treat as atomic.

The idea is left only barely sketched out, however, and ValleZ concludes by reflecting that it seems more an idea “for a film than for real life, however i think its not a bad idea :-m”.

In 2005, an email arrived in the inbox of the virus researchers Peter Ferrie and Heather Shannon. Attached was a sample of what would go on to be known as the W32/Zallome worm. The code of the worm appeared unweildly and bloated, but its unusual polymorphic engine captured the analysts’ attention.

### **3.2.1.7 2009: Noreen’s experiment on grammatic malware evolution**

At GECCO ’09, Sadia Noreen presented a report on her recent experiments involving the evolution of computer viruses. The approach she adopted was to first collect samples of several varieties of the Beagle worm (CARO name W32/Bagle.{a,b,c,d,e}@mm), and then define a regular grammar that isolated the separable components of each variant, and which could be used to recombine and generate new variants. An initial population of grammatically correct, but randomly generated, individuals would then be spawned.

The fitness function used in these experiments was, curiously, resemblance to the existing samples, as judged by a distance metric and then ratified by an antivirus

scanner. The idea was that if an evolved specimen so closely resembled the original samples they were indiscernible to a scanner, then this would prove that viruses **could** be generated using evolutionary techniques.

This isn't the most compelling use of evolutionary techniques in this realm – that random sets of parameters can be made to approximate or match a training sample, when the fitness function depends precisely on the resemblance of the former to the latter, is not surprising. Genetic algorithms are often introduced through the use of “hello world” exercises posing formally similar problems. But the framework that Noreen developed could, itself, be put to much more interesting and creative ends, and the idea of assuring the evolvability and mutational robustness of viral genotypes by defining and adhering to a strict grammar is promising.

The idea of taking detection as a goal (in an effort to establish the possibility of evolution in this context) rather than as an obstacle is a strange approach, given that several scanners **also** use grammatical analysis to detect the code (often limiting themselves to regular expressions and FSAs), and so it's quite possible that the grammar itself went a long way towards preserving the invariants that resulted in detection.

If the goal were to evolve viruses that had a chance of being viable in the wild, and so had to contend with the selective pressures imposed by detectors, the ideal approach would be to employ a grammar with greater Chomsky complexity, as the virus writer known as “Second Part to Hell” points out in a 2008 post on his website [16].

### 3.2.1.8 2010-2011: Second Part to Hell: Evoris and Evolus

Second Part to Hell's experiments in viral evolution appear to be the most sophisticated yet encountered. SPTH begins by identifying computational fragility as the principal obstacle to the the evolvability of virus code as implemented in x86 assembly. An obvious way to circumvent this problem, SPTH reasons, is to have the genetic operators operate, not on the level of architecture-specific opcodes, but on an intermediate language defined in the virus's code itself.

SPTH designed his IL to be as highly-evolvable as possible, structured in such a way that an arbitrary bit-flip would still result in a valid instruction, so that they

could be permuted or altered with little risk of throwing an exception, and so that there would exist a considerable amount of redundancy in the instruction set: 38 semantically unique instructions are defined in a space of 256, with the remainder being defined as NOPs, affording a plentiful supply of introns, should they be required.

“The mutation algorithm is written within the code (not given by the platform, as it is possible in *Tierre* or *avida*)”, SPTH notes, referring to two well-known Artificial Life engines. [1]

The same is true of the IL syntax. In fact, what’s particularly interesting about this project, and with the problem of viral evolution in general, is that the entire genetic machinery must be contained either in the organism itself, or in features that it can be sure to find in its environment. In *Evoris*, the only mechanism that remains external to the organism is the source of selective pressure – antivirus software and attentive sysadmins. Two types of mutation are permitted with each replication: the first child is susceptible to bit flips in its IL sequence, with a certain probability. With the second, however, the IL instruction set may mutate as well, meaning that the virtual architecture itself may change shape over the course of evolution. Interestingly, the first-order mutation operators in the virus are themselves implemented with the viral IL, and so a mutation to the alphabet – one that changes the `xor` instruction to a `nop`, for instance – may, as a consequence, disable, or otherwise change the functioning of, first-order mutation (as SPTH observed in some early experiments).

*Evolus* extends *Evoris* to include a third type of mutation: “horizontal gene transfer” between the viral code and files that it finds in its environment. Since the bytes taken from those files will be interpreted in a language entirely foreign to their source, there’s no real reason to expect any useful building blocks to be extracted, unless, of course, the *Evolus* has encountered another of its kind, in which case we have something analogous to crossover. (Horizontal gene transfer with an arbitrary file would then be analogous to “headless chicken crossover”, with the random bytes being weighted to reflect what the distribution found in the files from which the bytes are sourced.)

Though SPTH’s results were fairly modest, the underlying idea of having the virus carry with it its own language for genotype representation, and to take cares to ensure the evolvability of that language – and to expose the genetic language itself

to mutation and selective pressure – is inspired, and turns SPTH’s experiments into valuable proofs of concept.

### 3.2.1.9 Summary

Interestingly, even in the virus scene, which is certainly where we find the most prolonged and serious interest in evolutionary computation among black and grey hat hackers, the uses to which evolutionary methods are put tend, for the most part, to be fairly modest, and oriented towards defence (defending the virus from detection). When genetic operators are employed, they tend to serve as part of a polymorphic or metamorphic engine, and the force of selection principally makes itself felt through antivirus and IDS software. There is still a tremendous amount of potential in this direction, and the threat of unpredictable, evolving viral strains emerging from this sort of research is one that has been taken seriously by {{ cite Darwin inside the Machines paper }}.

What haven’t yet been seen are experiments dealing with the wholesale evolution of malware’s functionality, constrained by only the most minimal and declarative behavioural specifications.

Nor, outside of science fiction [9], has there been any visible attempt to put evolutionary techniques in the service of malware that *learns*, in a fashion comparable to what we see with next-generation defence systems.

Nevertheless, at least two major obstacles to the use of evolutionary techniques in the field of offence **have** been addressed and, to some extent, solved by the VX community: the problem of code brittleness, or the viability of genetic operators, and the problem of self-sufficiency (unlike academic experiments in evolutionary computation, the virus must carry an implementation of the relevant genetic operators with it everywhere it goes – “the artificial organisms are not trapped in virtual systems anymore”, SPTH writes, in the conclusion to the first of his series of essays on Evoris and Evolus, “they can finally move freely – they took the redpill” ([1], 18).

### 3.2.2 Genetic Payload Crafting

Gunes Kayacik’s 2009 research brought evolutionary methods – specifically, linear genetic programming (LGP) and grammatical evolution (GE) – to bear on the problem





of automatically generating shellcode payloads for use in the sort of buffer overflow attacks already known to us from Aleph One. The aim of that research is twofold:

1. it aims to evolve payloads that can evade not just rudimentary signature-based detection engines, like Snort's, monitoring inbound packets, but also anomaly-detecting, host-based intrusion detection systems, such as Process Homeostasis (pH). In this respect, it has much in common with the uses of genetic algorithms that we start to see in some of the more experimental corners of the virus scene, in the early years of the millenium. In fact, the principal means of obfuscation that Kayacik saw emerging from his attack population was the proliferation of "introns", or what the virus literature refers to as "garbage code" when discussing an analogous tactic of metamorphic engines.
2. the secondary aim of Kayacik's research, however, is to use these evolving shellcode specimens to better train the same defensive AIs that the population of attacks is struggling to subvert. The ideal, here, is to lock both intelligences into an evolutionary arms race. In practice, however, the attack populations had little difficulty leaving the defenders in the dust.

### 3.2.3 The Road Ahead

Despite the relative dearth of work being done on the intersection of exploit research and evolutionary computation – an intersection which is all but barren, though flanked by thriving research communities on both sides – it is our conviction that this may become extraordinarily fertile terrain for research. Evolutionary methods are naturally well-suited to the exploration of the possibility space inhabited by weird machines. This is not least due to the fact that such machines, whose existence is an emergent and altogether accidental effect, are in no way designed to be hospitable to human programmers. Even the most obtuse and ugly programming language – including the tiramisu of backwards-compatible ruins that makes up the x86<sub>64</sub> – is designed with *some* aspiration of cognitive tractability and elegance in mind. As much as it may *seem* that this or that programming environment cares little for the programmer, this is never truly the case – until you enter the terrain of weird machines. These are landscapes that were never intended to exist in the first place – they're a wilderness

supervening on artifice.

# 4

## On the Design and Implementation of ROPER: Algorithmic Overview

What we will establish in the pages that follow is that it is indeed possible to generate functioning, ROP-chain payloads through purely evolutionary techniques. By “purely evolutionary”, here, we mean that payloads are to be evolved *from scratch*, starting with nothing but a collection of gadget pointers, of which we have virtually no semantic information, and a pool of integer values. This stands in contrast to

most previous experiments in the field of offensive security, where the role of evolutionary techniques is restricted to the fine-tuning or obfuscation of already existing malware specimens

or to the recombination of high-level modules into working programs, following an established pattern.

By “functioning”, we mean only that we are able to generate ROP payloads that reliably perform to specification, for a wide variety of tasks. Some of these tasks

are simple and exact – such as preparing the CPU context for a given system call, with certain parameters – whereas others are complex but vague in nature – tasks concerning the classification of data by implicit properties, or interacting with a dynamic environment. In each case, *all* that is provided to our system by way of instruction are the specifications of the task, translated into selective pressures in the form of a “fitness function”.

It should be emphasized that this system, acronymously named ROPER, is presented as a *proof of concept*, and not as a refinement of evolutionary techniques. ROPER is far from being an impressively efficient compiler or classifier, and no attempt was made to have it be otherwise. What ROPER is, is the first known use of evolutionary computation in return oriented programming, and, more generally, the first time that genetic programming has been put to work at a task for which it seems so obviously suited: the autonomous programming of state machines that emerge entirely by accident, supervening on the systems we designed, without our having ever designed them, and having languages and instruction sets all their own, without having ever been specified, spontaneously coalescing in the cracks of our abstractions.

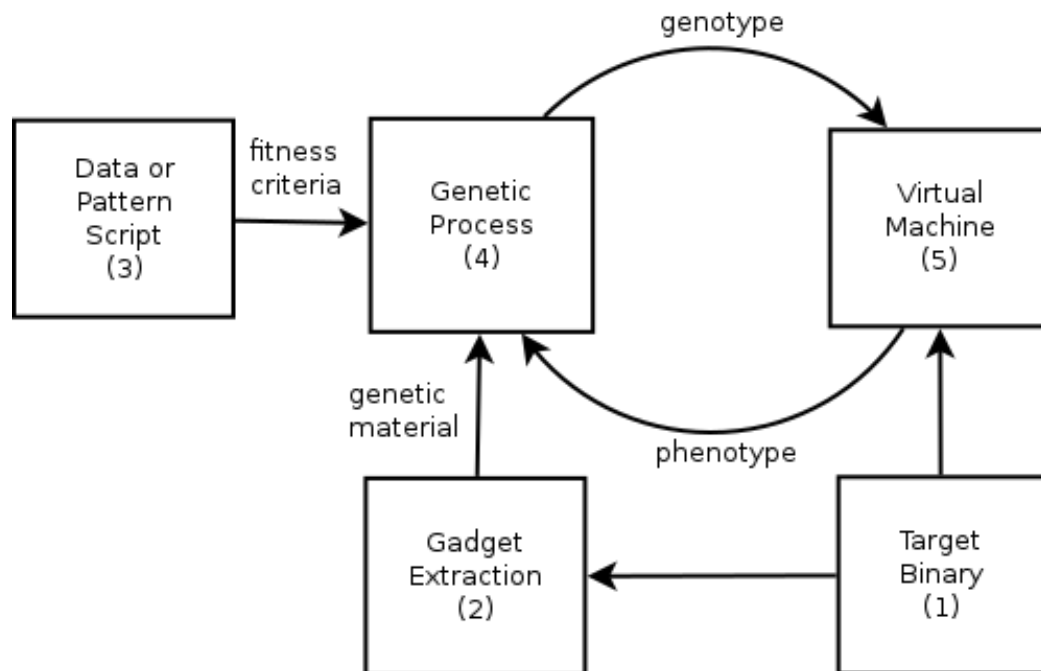


Figure 4.1: A bird’s eye view of ROPER

---

**Algorithm 1** Population Initialization

---

**Require:**  $\varepsilon$ , ELF binary of the process to be attacked

**Require:**  $\pi$ , the problem set specification

**Require:**  $n$ , the desired population size

**Require:**  $\iota$ , a pool of raw integers

**Require:**  $(\mathfrak{R}, s)$ , a pseudo-random number generator and seed

```

1: text, rodata  $\leftarrow$  parse( $\varepsilon$ )
2:  $\iota \leftarrow \iota \cup \text{find-pointers}(\text{rodata}, \iota)$ 
3:  $\gamma \leftarrow \text{harvest-gadgets}(\text{text})$ 
4:  $\Pi \leftarrow \text{empty-vector}(n)$ 
5:  $\mathfrak{R} \leftarrow \text{seed}(\mathfrak{R}, s)$ 
6: for  $x \leftarrow 1$  to  $n$  do
7:    $\mathfrak{R}, \Pi_x \leftarrow \text{spawn-individual}(\mathfrak{R}, \gamma, \iota, \text{rodata})$ 
8: end for
9: return population

```

---

Algorithms 1, 2, and 3 furnish a bird’s eye view of ROPER, abstracting away from questions of implementation, and streamlining away various bits of functionality aimed at optimization, bookkeeping, and fine-tuning.

ROPER begins with the analysis of an executable binary file (either an application or a library). For the time being, we are restricting ourselves to binaries targeting the 32-bit ARMv7 architecture, in ELF format, but there is nothing essential about this restriction, and ROPER could easily be extended to handle a variety of hardware platforms and executable formats, if desired. It harvests as many viable ROP gadgets as it can from the file (within parameterizable limits), by means of a linear sweep search, walking backwards through the file’s executable `.text` section until it hits a return instruction, and then walking further backwards until it reaches an instruction that would prevent the execution flow from reaching the return. This isn’t the most thorough or exacting technique for finding gadgets, and a wider variety of potentially usable gadgets can be uncovered by means of a constraint-solving algorithm, which is able to detect stack-controllable indirect jumps other exploitable control-flow artefacts as well. (We experiment with such an approach in ROPER II, which is still in progress at the time of writing.) A linear sweep nevertheless suffices to provide us with a fairly

---

**Algorithm 2** Genotype Evaluation (Ontogenesis)

---

**Require:**  $E$ , the CPU emulator

**Require:**  $IO : [(in, out, weight)]$ , the input/output rules for the problem set

**Require:**  $\varphi : [\mathbb{N}] \rightarrow \mathbb{F}$ ,

**Require:** SENTINEL: uint, a fixed-width integer constant (0, e.g.)

**Require:**  $\mu : \mathbb{N}$ , the maximum number of execution steps permitted the fitness function, mapping vectors of integers to floats

**Require:**  $\Gamma$ , the genotype to be evaluated

```

1:  $\sigma \leftarrow \text{serialize}(\Gamma) \cup [\text{SENTINEL}]$  {into a stack of bitvectors}
2: accumulator  $\leftarrow ()$ 
3: for all case in  $IO$  do
4:    $E \leftarrow$  prime  $E$  with case.in
5:    $E \leftarrow$  load  $\sigma$  into stack memory of  $E$ 
6:    $E \leftarrow \text{exec}(E, \text{"POP PC, SP"})$  {pop stack into program counter}
7:    $i \leftarrow 0$ 
8:   while  $i < \mu$  and program-counter( $E$ )  $\neq$  SENTINEL and in-legal-state( $E$ ) do
9:      $E \leftarrow \text{step}(E)$  {fetch instruction at PC and execute}
10:     $i \leftarrow i + 1$ 
11:   end while
12:   accumulator  $\leftarrow \text{acc}(\text{accumulator}, \text{case.weight}, \varphi(\text{read-registers}(E), \text{case.out}))$ 
13:    $E \leftarrow \text{reset}(E)$ 
14: end for
15: return accumulator {the 'phenotype'}

```

---

---

**Algorithm 3** Evolve Population (Tournament Selection)

---

**Require:**  $\Pi$ , the population *{as initialized by Algorithm 1}*

**Require:**  $E$ , the CPU emulator

**Require:**  $\Omega : \Pi \rightarrow \mathbb{B}$ , the stop condition (predicate over  $\Pi$ )

**Require:**  $\Sigma$ , the problem set

**Require:**  $(\mathfrak{R}, s)$ , a PRNG and seed

**Require:**  $n$ , the number of individuals competing in each selection tournament

```

1:  $\mathfrak{R} \leftarrow \text{seed}(\mathfrak{R}, s)$ 
2: repeat
3:    $\mathfrak{R}, \text{candidates} \leftarrow \text{using } \mathfrak{R}, \text{pick } n \text{ from } \Pi$ 
4:    $\Phi \leftarrow \text{empty list of (float, genotype) pairs}$ 
5:   for  $\Gamma$  in candidates do
6:      $\Gamma_{fitness} \leftarrow \text{evaluate-genotype}(\Gamma, \Sigma, E) \text{ \{Algorithm 2\}}$ 
7:   end for
8:    $\Phi \leftarrow \text{sort}(\Phi, \text{by } fitness)$ 
9:   victors  $\leftarrow \text{take } m \text{ from } \Phi$ 
10:  vanquished  $\leftarrow \text{take } k \text{ of reverse}(\Phi)$ 
11:   $\mathfrak{R}, \text{offspring}_{1...k} \leftarrow \text{breed}(\mathfrak{R}, k, \text{victors}) \text{ ??}$ 
12:   $\Pi \leftarrow [\text{vanquished}/\text{offspring}]\Pi \text{ \{replace vanquished with offspring\}}$ 
13: until  $\Omega(\Pi) = \text{true}$ 
14: champion  $\leftarrow \text{head}(\text{sort}(\Pi, \text{by } fitness))$ 
15: return champion

```

---

generous number of gadgets for our purposes, and has the advantage of being both simple and efficient.

The addresses of these gadgets, together with a pool of potentially useful immediate integer values and data pointers, which can be supplied by the user, or inferred from the specification of the problem set, supply us with the primitive genetic units from which the first genotypes in the population will be composed. With no more abuse of terminology than is customary in evolutionary computation, we can call this the “gene pool” of the population. It should nevertheless be noted that the biological concept of *gene* presupposes many structural constraints that have, as of yet, no parallel in our system.

The initial population, as yielded by Algorithm 1, is little more than an array of variable-length vectors of fixed-width integers (32-bits, so long as we are restricting ourselves to the ARMv7, but, again, this restriction matters little so far as the system’s algorithmic structure is concerned). The length of the initial individuals is left parameterizable, but is upper-bounded by the amount of stack memory that will be available in the target process for our attacks to write to. We will complicate this structure somewhat, in Section 4.2, but it remains a useful simplification.

The main loop, outlined in Algorithm 3, is built around a well-known and widely used genetic programming algorithm called “tournament selection”. On each iteration of the loop,  $n$  (typically 4) distinct candidate genotypes are chosen from the population, with equal probability. Each is then mapped to its phenotype (its behavioural profile in the emulated CPU), and its fitness evaluated (by applying the fitness function to that profile). The  $m$  (typically 2) candidates with the best fitness are selected for reproduction, while the least-fit  $k$  candidates are culled from the population.

The genotypes selected for reproduction are then passed to our genetic operators, which will return  $k$  offspring, who will replace the least-fit  $k$  candidates in the tournament. In the genetic programming literature, these operators are often referred to as the “search operators”, as they “define the manner in which the system moves through the space of possible solutions” ([2], 144). In ROPER, our genetic operators comprise a single-point crossover function, which maps a pair of parents into a pair of offspring, and a mutation operator, which maps a single genotype into a variant thereof. The internals of these operators are detailed in Section 4.3.



This loop continues until the halting conditions are satisfied. These are most often set either to a maximum number of iterations, or the attainment of a set degree of fitness by the population's fittest specimen.

In the following sections, we will explain the finer-grained design decisions involved in implementing the algorithms specified above.

In the following sections, I will unfold and justify the decisions that went into implementing the algorithms surveyed in Section 4. We can begin with the representation of the genotypes constructed by the **spawn-individual()** algorithm, called on line 7 of Algorithm 1.

#### 4.1 Gadget Extraction

Since the aim of ROPER is to foster the evolution of ROP chains, we must begin by supplying the engine with a sufficient pool of gadgets, harvested from the target executable.<sup>1</sup>

There are several ways that this can be done, but the simplest is just to scan the executable for a subset of easily recognizable 'gadgets' using a linear sweep algorithm, shown in Algorithm 4. Since we are dealing only with a RISC instruction set architecture here, we can avoid several complexities in our gadget search that we would need to grapple with were we adapting ROPER to handle CISC instruction sets (such as the x86 and its ilk) as well. The instructions of a RISC ISA are all of equal length (with a certain exceptions, and assuming the mode fixed), and so if a sequence of bytes beginning at address  $i$  is parsed as instruction  $X$  when beginning the parse *from*  $i$ , then it will also be parsed as  $X$  when beginning the parse from some  $j < i$ . To put it another way, the list of RISC instructions parsed from bytevector  $C$ , beginning at address  $i$ , extends *monotonically* with each decrement of  $i$ . In practical terms, this means that an instruction that looks like a return from far away will still look like a return by the time you've parsed your way up to it. This is very different from what we encounter with CISC ISAs, where the length of instructions is variable, and instructions are not aligned. Suppose we had the string "aabbcc" of bytes. Suppose that **aa** parses to  $\alpha$ , **ab** parses to  $\beta$ , **bb** parses to  $\tau$ , **cc** parses to  $\delta$  and **bcc** parses

---

<sup>1</sup>See Section 2.2 for a sustained explanation of how return-oriented programming works, and an explanation of the concept of 'gadget'.

to  $\gamma$ . If we begin the parse from the beginning of the string, we get  $\alpha\tau\delta$ . But if we increment our cursor one byte forward before parsing, then our parse yields  $\beta\gamma$ , with  $\delta$  nowhere to be seen. In order to adapt our gadget harvesting algorithm to CISC ISAs, therefore, we would have to continually check to ensure that the **return** instruction spotted at line 5 of Algorithm 4 is still parseable as a return, and still reachable, from the address indicated by **i** on line 7. This would increase the complexity of the algorithm substantially.

Fortunately, for the time being, we are concerned only with the two main instruction sets of the ARMv7: the *arm* instruction set, which is aligned to four-byte intervals, and the *thumb* instruction set, which is aligned to two-byte intervals. A sufficient supply of gadgets can usually be found by passing our extraction algorithm twice over the executable segments of our target binary, gathering a pool of both *arm* and *thumb* gadgets. Since the least significant bit of an instruction address is invariably 0, for this ISA, the ARM CPU uses this bit to distinguish between *arm* mode and *thumb* mode. We therefore increment the address of each of our freshly harvested thumb gadgets by 1.

## 4.2 Genotype Representation: Gadgets, Clumps, and Chains

From a certain perspective – that of the evaluation engine – the individual genotypes of the population are little more than bare ROP-chain payloads: vectors of 32-bit words, each of which is either a pointer into the executable memory of the host process, or raw data (the former being a subtype of the latter, of course). The view afforded to the genetic operators, and to the initial spawning algorithm, exposes slightly more structural complexity, which is introduced in response to the following problem:

In the set of 32-bit integers (`0x100000000` in all), the subset representing the set of pointers into the executable memory segments of a given ELF file tends to be rather small: in the case of `tomato-RT-N18U-httpd`, an HTTP server that ships with a version of the Tomato firmware for certain ARM routers, which we will be using for a few of the experiments that follow, we can see that only `0x1873c + 0xc0 = 0x187ec` bytes are mapped to executable memory. Now, the ARMv7 CPU is capable of running in two different modes, each with their own instruction set: *arm* mode,

---

**Algorithm 4** Linear sweep algorithm for gadget extraction.

---

**Require:**  $\mathbf{C}$ : a contiguous vector of bytes representing instructions

**Require:**  $\lceil \mathbf{X}_j \rceil : [\text{byte}] \rightarrow \mathbb{N} \rightarrow \text{inst} | \Lambda$ , a parsing function, from byte-vectors  $\mathbf{X}$  and indices  $j$  to instructions, or  $\Lambda$  in case of unparseable bytes.

**Require:**  $\rho : \text{inst} \rightarrow \mathbb{B}$ , predicate to recognize returns

**Require:**  $\varphi : \text{inst} \rightarrow \mathbb{B}$ , predicate to recognize control instructions, with  $\forall(\mathbf{x}) \rho(\mathbf{x}) \Rightarrow \varphi(\mathbf{x})$ , but not necessarily the converse.  $\varphi$  should also return **true** for  $\Lambda$  (signalling unparseable bytes).

**Require:**  $\delta$ : positive integer, offset of base virtual address for  $\mathbf{C}$

```

1:  $\Gamma \leftarrow$  empty stack of integers
2:  $i \leftarrow \text{length}(\mathbf{C})$ 
3: while  $i > 0$  do
4:    $i \leftarrow i - 1$ 
5:   if  $\rho(\lceil \mathbf{C}_{i+1} \rceil)$  then
6:     while  $\neg \varphi(\lceil \mathbf{C}_i \rceil)$  and  $i > 0$  do
7:       push  $i$  onto  $\Gamma$ 
8:        $i \leftarrow i - \text{length}(\lceil \mathbf{C}_i \rceil)$ 
9:     end while
10:  end if
11: end while
12:  $\Gamma^* \leftarrow \text{map } (\lambda \mathbf{x}. \delta + \mathbf{x})$  over  $\Gamma$ 
13: return  $\Gamma^*$ 

```

---

Table 4.1: Program Headers of a Typical ELF Executable

---

```
$ readelf --program-headers tomato-RT-N18U-httpd

Elf file type is EXEC (Executable file)
Entry point 0xa998
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00008034 0x00008034 0x000c0 0x000c0 R E 0x4
  INTERP        0x0000f4 0x000080f4 0x000080f4 0x00014 0x00014 R   0x1
      [Requesting program interpreter: /lib/ld-uClibc.so.0]
  LOAD          0x000000 0x00008000 0x00008000 0x1873c 0x1873c R E 0x8000
  LOAD          0x01873c 0x0002873c 0x0002873c 0x0040c 0x005c8 RW 0x8000
  DYNAMIC       0x018748 0x00028748 0x00028748 0x00118 0x00118 RW 0x4
  GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4

Section to Segment mapping:
Segment Sections...
00
01      .interp
02      .interp .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn
      .rel.plt .init .plt .text .fini .rodata .eh_frame
03      .init_array .fini_array .jcr .dynamic .got .data .bss
04      .dynamic
05
```

---

which requires the instructions to be aligned to 4-byte units, and *thumb* mode, which demands only a 2-byte alignment of instructions. Since the least significant bit in a dword can therefore not be used to differentiate between instruction addresses, the ARMv7 CPU uses it to distinguish between the two modes: any address  $a$  whose least significant bit is 1 (i.e., any odd-valued address) is dereferenced to a thumb instruction at address  $a \oplus 1$  (rounding down to the nearest even address). This gives us a total of  $\frac{0x187ec}{4} + \frac{0x187ec}{2} = 0x125f1$  valid executable pointers – which, roughly, means that only one in fifty-thousand of integers between  $0x00000000$  and  $0xFFFFFFFF$  can be dereferenced to executable memory in a the ELF executable in question – a ratio that is seldom increased by more than one or two orders of magnitude, even when dealing with large, statically linked ELF binaries.<sup>2</sup>

This means that if we allow the integers composing the genotypes of our initial population to be randomly selected from the entire 32-bit range, only a tiny fraction of those integers will dereference to any meaningful executable addresses in the code

---

<sup>2</sup>There’s a fair bit of handwaving, here, when referring to a ‘typical’ ELF executable – obviously the size of the executable can vary. We’re also restricting ourselves to the executable memory mapped in the file of a *dynamically linked* executable here, ignoring the addresses that may dereference to executable addresses where dynamically loaded libraries might be mapped.

– let alone useful gadget addresses. Restricting the pool of integers sampled to the set of valid executable pointers, let alone potentially useful gadget points, however, may deprive the population of useful numerical values.

The execution of these individuals, after all, will be driven by return instructions, and these, in ARM machine code, are most often implemented as multi-pops, which pop an address from the stack into the program counter, while simultaneously popping a variable number of dwords into other, general-purpose registers. This means that each `return` – each “tick” of the ROP state-machine – not only steers the control flow of our machine, sending it to a new gadget, but the data flow as well, furnishing each gadget with a handful (between zero and a dozen) of numerical values, to use internally. We don’t necessarily want to restrict these numerical resources to the range of gadget pointers – it would be better, in fact, if we could tailor the pool of “potentially useful” numerical values to a set of integers (including, perhaps, data pointers) that seems suited to the problem set at hand.

This suggests a potentially useful structural constraint that we can impose on the genotypes, to increase the likelihood that they will be found useful for the problem space at hand, and greatly increase the probability that `.text` pointers will be popped into the `pc` register, while other integers land predominantly in general-purpose registers. To do this, we calculate the distance the stack pointer will shift when each gadget executes,  $\Delta_{\text{SP}}(\mathbf{g})$ , and then clump together each gadget pointer  $\mathbf{g}$  with a vector of  $\Delta_{\text{SP}}(\mathbf{g}) - 1$  non-gadget values. Consider, for example, the instruction,

—————→ **LDMIA!** **sp,** {r0,r7,r9,pc}

which pops the stack into registers `r0`, `r7`, `r9`, and `pc`, in sequence, “returning” the program counter to the address represented by the fourth dword on the stack, while at the same time populating three general purpose registers with the stack’s first three dwords. This instruction has a  $\Delta_{\text{SP}}$  of 4. For a gadget  $\mathbf{g}$ , we define  $\Delta_{\text{SP}}(\mathbf{g})$  as

$$\Delta_{\text{SP}}(\pi) = \sum_{\mathbf{i} \in \pi} \Delta_{\text{SP}} \mathbf{i}$$

for some control path  $\pi$  in  $\mathbf{g}$  that reaches the return. In practice, we choose our initial pool of gadgets in such a way that each contains only a basic block of code, with control flow entirely in the hands of the return instruction that terminates it, so that

the choice of  $\pi$  is unique for each  $\mathbf{g}$ . If this condition is relaxed, we suggest generating  $n$  distinct clumps for each distinct member of  $\{\Delta_{\text{SP}}(\pi) | \pi \text{ is a control path in } \mathbf{g}\}$ . Exactitude isn't strictly necessary, here, however – as we'll see, the evolutionary process that follows is robust enough to handle a fairly large number of gadgets with miscalculated  $\Delta_{\text{SP}}$  values. A good rule of thumb, here, is that when the approximation of  $\Delta_{\text{SP}}(\mathbf{g})$  is left inexact, in the interest of efficiency, dump several options into the pool, and let natural selection sort them out.

Given a gadget entry point address  $\hat{\mathbf{g}}$ , a “clump” around  $\mathbf{g}$  can now be assembled by taking a stack of  $\Delta_{\text{SP}}(\mathbf{g}) - 1$  arbitrary values, and pushing  $\hat{\mathbf{g}}$  on top of it. By the time  $\mathbf{g}$  has run to completion, it will have popped  $\Delta_{\text{SP}}(\mathbf{g})$  values from the process stack. The first  $\Delta_{\text{SP}} - 1$  of these will populate the general purpose registers of the machine, and the  $\Delta_{\text{SP}}^{\text{th}}$  will pop the entry point of the *next* gadget,  $\mathbf{g}'$ , into pc. That entry point,  $\hat{\mathbf{g}}'$  will be found at the top of the next clump in the sequence that makes up the genotype.<sup>3</sup>

As explained in Section 2.2.2, it is often helpful to think of each gadget as an instruction in a virtual machine – an emergent machine, supervening on the host's native instruction set architecture. What we're calling a clump here maps onto this concept of “instruction”, but with a slight displacement: the gadget address can be seen as something like an “opcode” for the ROP-VM, and the immediate values in each clump can be seen as operands – *but operands of the next instruction*, not of the instruction whose opcode is represented by their own clump's gadget pointer.

When the initial population is generated, we take a pool of gadget pointers, harvested from the target binary (see Section 4.1), and a pool of integers and data pointers, supplied by the user as part of the problem specification. We then form clumps, as described above, using randomly chosen elements of these two pools, as needed. The clumps are then assembled into variable length chains (with the minimum and maximum allowed lengths being parameterized by the user), which gives us our genotype representation. The internals of this algorithm are details in Algorithm [ref{alg:spawn}](#).

---

<sup>3</sup>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,  $\Delta_{\text{SP}}(\mathbf{g})$  and the offset of the next gadget from  $\mathbf{g}$  are distinct. But this is a complication that we don't need to dwell on here.

---

**Algorithm 5** Spawning an Initial Individual

---

**Require:**  $\mathbf{G} : \llbracket \mathbb{N}^{32} \rrbracket$ , a set of gadget pointers

**Require:**  $\mathbf{P} : \llbracket \mathbb{N}^{32} \rrbracket$ , a set of integers and data pointers

**Require:**  $(\mathfrak{R}, \mathbf{s})$ : a PRNG and seed

**Require:**  $(\min, \max) : (\mathbb{N}, \mathbb{N})$ , minimum and maximum genotype lengths

```

1:  $\Gamma \leftarrow$  empty stack of clumps  $\{\textit{the genotype representation}\}$ 
2:  $\mathfrak{R} \leftarrow \text{seed}(\mathfrak{R}, \mathbf{s})$ 
3:  $\mathbf{n}, \mathfrak{R} \leftarrow \text{random-int}(\mathfrak{R}, \min, \max)$ 
4: for  $i \leftarrow 0$  to  $\mathbf{n}$  do
5:    $\hat{\mathbf{g}}, \mathfrak{R} \leftarrow \text{choose}(\mathfrak{R}, \mathbf{G})$ 
6:    $\mathbf{C} \leftarrow$  empty stack of  $\mathbb{N}^{32}$ 
7:    $\delta \leftarrow \Delta_{\text{SP}}(\mathbf{g})$   $\{\textit{cf. sec. 4.2 for def. of } \Delta_{\text{SP}}\}$ 
8:   for  $j \leftarrow 0$  to  $\delta$  do
9:      $\mathbf{p}, \mathfrak{R} \leftarrow \text{choose}(\mathfrak{R}, \mathbf{P})$ 
10:    push  $\mathbf{p}$  onto  $\mathbf{C}$ 
11:   end for
12:   push  $\hat{\mathbf{g}}$  onto  $\mathbf{C}$ 
13:   push  $\mathbf{C}$  onto  $\Gamma$ 
14: end for
15: return  $\Gamma$ 

```

---

### 4.3 Genetic Operators

In order for our population of loosely structured but otherwise random ROP chains to explore the vast and uncharted space of possible combinations and (on the side of phenotypes) their associated behaviours, we need a means of moving from a given subset of our population to “similar” genotypes in the neighbourhood of that subset, which may not yet belong to the population. This is accomplished by the genetic operators, which allow our population to search the genotype space through reproduction and variation.

ROPER makes use of two such operators: a crossover operator, which operates on genotypes as lists of clumps, and a mutation operator, which operates on clumps internally.

#### 4.3.1 Mutation

The mutation operator applies one of several bitwise operations to one or more words contained in one or more randomly selected clumps. The choice of operation is constrained by the word slot being operated on: the word that is (probabilistically) fated to be loaded into the instruction pointer isn’t subject to the same range of modifications that the other words in the clump are. The reason for this is that the performance of an individual will, in general, be more sensitive to modifications to its gadget pointers than to its immediate values, and so it makes sense to afford the mutation operator a greater degree of freedom when dealing with a value that is unlikely to be used to directly control the instruction pointer. It is relatively safe to increment or decrement a pointer by a word size or two, but almost always hazardous to negate or shift it, for example.

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).



### 4.3.2 Crossover

At a slightly higher structural level, the reproduction algorithm may apply a crossover operation to the list of clumps, taking the clumps as opaque units.

I chose single-point crossover over two-point or uniform crossover to favour what I judged to be the most likely form for gene linkage to take in this context: A single gadget can transform the CPU context in fairly complex ways, since it may include any number of architectural instructions. The prevalence of multipop returns in ARM code further increase the odds that the work performed by a gadget  $g$  will be clobbered by a subsequent gadget  $g'$ , and this risk increases monotonically as we move down the chain from  $g$ . This means that adjacent gadgets are more likely to achieve a combined, fitness-relevant effect, than non-adjacent gadgets. Lacking any reason to complicate things further, we restricted the number of parents involved in each mating event to two.<sup>4</sup>

In single-point crossover between two genotypes,  $\mu$  and  $\varphi$ , we randomly select a link index  $\mu_i$  where  $\mu_i < |\mu|$ , and  $\varphi_i$  where  $\varphi_i < |\varphi|$ . We then form one child whose first  $\mu_i$  genes are taken from the beginning of  $\mu$ , and whose remaining genes are taken from the end of  $\varphi_{\mu_i\dots}$ , and another child using the complementary choice of genes. The only modification I make to this well-known algorithm, in ROPER, is to weight the choices of  $\mu_i$  and  $\varphi_i$ , using a parameter we call “fragility”, whose calculation I explain in Section 4.3.3. The details of the algorithm itself can be found in 6.

### 4.3.3 Fragility and Gene Linkage

As a way to encourage 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  $\mu_i$  and  $\varphi_i$ , instead of letting them be simply uniform. With each adjacent pair of nodes is associated a “fragility” value, representing the likelihood of that pair being split by a crossover operation. The fragility of each link in  $A$  is derived from the running average of fitness scores exhibited by the sequence of ancestors of  $\mu$  who shared the same linked pair. Links that have a

---

<sup>4</sup>One of the limitations of ROPER is that the mating algorithm, and the genetic search operators in general, are assumed fixed. In ROPER II, we will experiment with a technique for opening this, too, to genetic exploration and selective pressure, which Lee Spector calls “autoconstructive evolution”.

---

**Algorithm 6** Single-Point Crossover, with Fragility

---

**Require:**  $(\tilde{\mu}, \tilde{\varphi})$ : ( $\llbracket \text{clump} \rrbracket$ ,  $\llbracket \text{clump} \rrbracket$ ), the parental genotypes

**Require:** *Fragility*:  $\llbracket \text{clump} \rrbracket \times \llbracket \text{clump} \rrbracket \times \text{lineage} \rightarrow \mathbb{F}$

**Require:**  $(\mathfrak{R}, \mathbf{s})$ : a PRNG and seed

**Require:**  $n$ :  $\mathbb{N}$ , brood size

```

1:  $\mathfrak{R} \leftarrow \text{seed}(\mathfrak{R}, \mathbf{s})$ 
2: splice-points  $\leftarrow ()$  {the indices at which the parental genes will be recombined}
3: for  $\tilde{\alpha} \in (\tilde{\mu}, \tilde{\varphi})$  do
4:    $\mathbf{t} \leftarrow \sum_{\alpha \in \tilde{\alpha}} 1.0 - \text{Fragility}(\alpha)$ 
5:    $\mathbf{p}, \mathfrak{R} \leftarrow \text{random-float}(\mathfrak{R}, \mathbf{t})$  {p is more likely to fall on a highly fragile link}
6:    $\mathbf{x} \leftarrow 0$ 
7:    $\mathbf{i} \leftarrow 0$ 
8:   while  $\mathbf{x} < \mathbf{p}$  do
9:      $\mathbf{x} \leftarrow \mathbf{x} + \text{Fragility}(\alpha_{\mathbf{i}})$ 
10:     $\mathbf{i} \leftarrow \mathbf{i} + 1$ 
11:   end while
12:   push  $(\tilde{\alpha}, \mathbf{i})$  onto splice-points
13: end for
14: let  $\mu^{\mathbf{a}}, \mu^{\mathbf{b}} = \text{split } \mu \text{ at splice-points.lookup}(\mu)$ 
15: let  $\varphi^{\mathbf{b}}, \varphi^{\mathbf{a}} = \text{split } \varphi \text{ at splice-points.lookup}(\varphi)$ 
16: let  $\chi^{\mathbf{a}} = \mu^{\mathbf{a}} \cup \varphi^{\mathbf{a}}$ 
17: let  $\chi^{\mathbf{b}} = \varphi^{\mathbf{b}} \cup \mu^{\mathbf{b}}$ 
18: return  $\chi^{\mathbf{a}}, \chi^{\mathbf{b}}$  {the offspring}

```

---

genealogical track record of appearing in relatively fit ancestors (i.e., ancestors with anumerically *low* fitness rank) will have a correspondingly low fragility score, while links from weaker genealogical lines will have a respectively greater fragility.

Following a fitness evaluation of  $\mu$ , the link-fitness of each clump  $\mathbf{f}(\mu_i)$  (implicitly, between each clump and its successor) is calculated on the basis of the fitness of  $\mu$ ,  $F(\mu)$ :

$$\mathbf{f}(\mu_i) = F(\mu)$$

if the prior link fitness  $\mathbf{f}'(\mu_i)$  of  $\mu_i$  is **None**, and

$$\mathbf{f}(\mu_i) = \alpha F(A) + (1 - \alpha) \mathbf{f}'(\mu_i)$$

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

Fragility is calculated from link-fitness simply by substituting a default value (50%) for **None**, and taking the link-fitness score, otherwise.

In the event of a crash – where the emulation of a specimen terminates prematurely, due to a CPU exception, such as a segmentation fault or division by zero – the link-fitness of the clump prior to the one responsible for the crash-event is severely worsened (raised) and the fragility adjusted accordingly. Attribution of responsibility is approximate at best – all we do is lay the blame at the feet of the last clump to execute before the crash event – but the penalty is ultimately probabilistic. A clump whose successful execution is highly dependent on the existing CPU context should be seen as a liability, in any case, regardless of whether or not that same clump may have behaved normally in other circumstances. (An example of such a clump would be one that reads from a memory location specified by a register that it does not, itself, set.) This penalty in link-fitness makes connections to the crash-labile clump highly fragile, and so the weighted crossover employed here becomes much more likely to set a splice point just prior to that clump. This has the effect of weeding particularly hazardous genes out of the genepool fairly quickly, as we will see.

## 4.4 Ontogenesis and Evaluation

The algorithms explained above all depend, either directly or in the way they hang together, on having a way to evaluate the “fitness” of arbitrary genotypes.

The genetic programming literature often enlists the biological distinction between *genotype* and *phenotype*.

### 4.4.1 From Genotype to Phenotype

“Genotype” is used to refer to the immediate representations of the individuals in the population, as sequences of semantically uninterpreted instructions. It is, in a sense, a purely *syntactic* concept. The genotype is the genetic syntax of an individual in the population, and belongs to the domain of the genetic operators – crossover, mutation, and so on, all of which operate on syntax alone, at least in principle.<sup>5</sup>

*Selection*, however, does not directly operate on genotypes but *phenotypes*. In the context of genetic programming, “phenotype” is the name given to the semantic interpretation of an individual’s genetic code. If the genotype is a sequence of instructions, then the phenotype is the behaviour expressed when that sequence is *executed*. Some theorists, such as Wolfgang Bahnzaf, have argued that the notion of phenotype should be constrained further still, to refer not just to the semantic interpretation of the genome, but to *the result of applying the fitness function to that interpretation*.

While this distinction does bring some clarity to the issue, and give the engineer a better view of *what*, exactly, is the subject of selection, it does deprive us of a nice term for the *intermediate representation*, between genotype and fitness value. In ROPER, in particular, the semantic image of the genotype is complex enough that it’s worth distinguishing from its later collapse into a fitness value, for some purposes. We have, moreover, set things up in such a way that it is possible to vary the *fitness function* while keeping the semantic image – what we call the phenotype – constant. It is simpler, in this case, to “carve nature at the joints”, and define the fitness function as a function *from phenotypes to floats*, rather than as much more complex function from genotypes to floats. The floats, in this case, will be called “fitness values”, rather

---

<sup>5</sup>It could be argued that the fragility mechanism described above leaks some amount of semantic/phenotypic information into our genetic operations, but this is no cause for concern – the distinction is simply descriptive, and carries no prescriptive force.

than phenotypes, as Bahnzaf would have it.

As for the function from genotypes to phenotypes – the semantic evaluation function – we might as well keep on pilfering biology textbooks for our terminology, and refer to it as *ontogenesis*.

#### 4.4.2 Ontogenesis of a ROP-chain

Our definition of ontogenesis in ROPER should be no surprise: it is simply the execution of the ROP-chain payload encoded in the genotype in the “womb” of the host process.

If we strip away the clump structure, and associated metadata, such as fragility ratings, with which we saddled our genotypes in order to provide better traction to our genetic operators, what remains is just a stack of fixed-width integers. Some of these integers index “gadgets” in the host process, while others are there only to provide raw numeric material to register and memory operations. If we take this stack, pack it down to an array of bytes, and write it to the stack memory of the host process, we should be able to evaluate it simply by popping the first item on the stack into the instruction pointer – which is precisely what would happen when a `pop {ip}` return instruction is executed.

From that point on, we only need to sit back and watch as the ensuing cascade of returns executes our payload. This is no different from what takes place in a ROP-chain attack in the wild – aside from a few simplifications: for the time being, we are abstracting away from any particular attack vector or preexisting machine state. The registers of the virtual machine are all initialized to arbitrary, constant values, and we don’t bother to ask *how* the ROP payload happened to get written to the stack. The stack is of fixed size, and restricted to the region of memory that the ELF program headers prescribe for it – thereby placing an upper bound on the effective size of individuals in our population – but the exact address of the stack pointer at the moment of inception is not based on any observed process state, just set, conveniently, to the centre of the available stack segment. No consideration, as of yet, has been given to avoiding “bad characters” in our payloads, though introducing this restriction would be fairly trivial. Execution is terminated as soon as any of the following conditions obtain:

1. the value of the instruction pointer is 0;
2. the CPU has thrown an exception (a segmentation fault, a bad instruction, division by zero, etc.);
3. some fixed number `n` of instructions has been executed.

The first outcome is treated as a “well-behaved” termination, as though the payload had reached its proper conclusion. Null bytes are written to the stack just beneath each payload, with the intention of having `0x00000000` popped into the instruction pointer by the final return statement. This condition, of course, can easily be gamed by an individual that finds another means of zeroing out its instruction pointer, with something like

```
xor r3, r3, r3
mov ip, r3
```

for example.

The second and, to a lesser extent, the third outcome both result in a variable penalty to fitness, the details of which will be discussed in Section 4.4.3.

The execution of the ROP chain payload is, in the context of ROPER, our ontogenesis function: it gives us the phenotype, the behavioural, semantic profile of the genotype. It is to this structure that the fitness functions are applied.

#### 4.4.3 Fitness Functions

Each of the fitness functions with which we’ve experimented begin with a partial sampling of the individual’s behavioural profile, generally restricted to just a few features:

1. the state of the CPU’s registers at the end of the individual’s execution;
2. the number of gadgets executed, as determined by the number of `return` instructions evaluated;
3. whether or not a CPU exception has been thrown.

This behavioural synopsis is then passed to a task-specific fitness function. We experimented with three types of task : a. reproduction of an specific register state, such as we might try to achieve in order to prepare the CPU for a specific system call, for example; b. classification of a simple data set, using supervised learning techniques; c. participation in an interactive game, where the evaluation of the payload makes up the body of the game’s main loop.

The task-specific function maps the behavioural synopsis onto a double-width float, between 1.0 and 0.0, with better performance corresponding to lower values. The exact nature of the tasks and performance of the system will be discussed in detail in Chapter 5. For the time being, the matter of CPU exceptions deserves closer comment.

#### 4.4.3.1 Failure modes and crash rates

Our population of random ROP-chains begins its life as an extraordinarily noisy and error-prone species. The old problem of *computational brittleness* [MEMO: reference to earlier section] resurfaces here in full force: the odds of a randomly generated chain of gadgets executing without crashing is extremely small [MEMO: collect some solid figures here] – under 5%, on average, at the beginning of a run. [MEMO: discover the robustness of the genetic operators, run the necessary experiments] If we were to let each crash count as unconditionally lethal, this would impose such a tremendous selective pressure on the population as to make it virtually unevolvable. What few islands of stability exist in the initial population would be cut off from one another by an inhospitable ocean of segfaults, leaving little room for exploration.

Fortunately, our chains have the luxury of being raised in the safety of a virtual nursery, and nothing obliges us to make crashes unconditionally fatal. We have at least two alternative possibilities:

1. apply a fixed penalty to fitness in the event of a crash,
2. make the crash penalty proportionate to the ratio of the chain that executed prior to the exception, measured in gadgets

We decided to implement follow the second tactic, which we implemented by trapping the return instructions in the Unicorn emulator. This lets us smooth an

abrupt cliff in the fitness landscape down to a gentle slope, incentivizing adaptations that minimize the likelihood of crashing while at the same time leaving room to reward specimens that do a failure good job of solving the problems posed to them, even if they botch the landing. This prevents us from sacrificing a number of useful genes, and gives them a chance to decouple from their pathological counterparts, through crossover, or to be repaired through mutation.

With this modification to the fitness function in place, the percentage of chains that crash before completing execution has a tendency to drop to less than 10% within a few hundred generations. [MEMO: need exact figures] What’s particularly interesting is what happens when the average fitness of the population hits a plateau: the crash rate begins to rise again, until the plateau breaks, and the error rates begin to drop again. A plausible explanation for this behaviour is that we are seeing the genetic search start to explore riskier behaviours as the competition between combatants in each tournament slackens (we will soon examine some examples in detail). As soon as a new breakthrough is discovered in the problem space, the competition once again hardens, and crash-prone behaviour becomes a more severe liability. In this way, the fitness landscape, as a whole, becomes elastic.

#### 4.4.4 Fitness Sharing

The most serious problem that ROPER’s populations appear to encounter, particularly when dealing with relatively complex problem spaces – classification problems or interactive games – is the depletion of diversity.

As a population becomes increasingly homogenous, the exploratory potential of the genetic operations becomes more and more constricted. There are two distinct, but closely related, forms under which diversity should be considered here: genotypic diversity and phenotypic diversity. At the beginning of the evolutionary process, when the population consists entirely of randomly-initialized specimens, genotypic diversity is likely at its historic peak: the sum of genetic differences between each specimen and every other is maximal, with no discernible “family resemblance” between them, beyond those afforded by chance. Behavioural, or phenotypic, diversity, however, is typically rather meager at this point. Unless the problem is extremely simple, and likely to be solved by random search, the odds are that almost every specimen



behaves in an effectively similar fashion: near-total failure. Nevertheless, if sufficient genetic material exists, however, and if the fitness function is sufficiently subtle, *some* phenotypic gradients will distinguish themselves from the white noise of failure, and it is these minor differences that selection will accentuate. As a result, the population will often experience a “Cambrian Explosion” of some form in the early phases of the evolutionary process: a tremendous flowering of phenotypic diversity, paid for by a reduction in genotypic diversity (at least insofar as we can measure genotypic diversity in terms of raw hamming distances or bitstring similarity, without giving any consideration to structure). The danger is that some particular family of phenotypes will be so strongly favoured by selection that its corresponding genotypes *consistently* replicate faster than any others, squeezing their rivals out of the population altogether. This can lead us to a point where the exploratory power of recombination is nearly exhausted: the only remaining sources of novelty, now, is the slow trickle of random mutation or the creation of new, random individuals *ex nihilo*. The likelihood of this situation being disrupted by sheer randomness, however, is as small as that of discovering competitive solutions to the problem set through random search. The result is that evolution stagnates, if not eternally, at least for much longer than we, as experimenters and engineers, would care to wait.

When the problem set we are dealing with is plural – as it is in the second and third types of fitness function, listed in Section 4.4.3 – one way that diversity depletion often occurs is through *hypertelia*, or an adaptive fixation on low-hanging fruit.<sup>6</sup> It is common for some subset of the problem set to be considerably simpler than the rest, or for distinctions between certain classes in a classification problem to be more computationally tractable than distinctions between other, more ambiguous or complexly defined classes. It is consequently likely that the population will produce specimens that are capable of handling those simpler problems and clearer distinctions before anything exhibits comparable skill in handling the “harder” problems. So long as the fitness function remains static, selection will magnify this discrepancy, and the simple-problem-solvers will enjoy a persistent reproductive advantage over any

---

<sup>6</sup>The notion of hypertelia used here has been borrowed from Gilbert Simondon. See, for example, the discussion in Chapter II, Section I, of *On the Mode of Existence of Technical Objects*, which begins, “The evolution of technical objects manifests certain hypertelic phenomena which endow each technical object with specialization, which causes it to adapt badly to changes, however slight, in the conditions of its operation or manufacture.”

specimens that may be still fumbling their way through the more complex regions of the fitness landscape. Once the bottomfeeders reach such numerical dominance that they start to appear in the majority of tournaments, there remains very little selective advantage in tackling any other aspect of the problem space, and the population suffers a rapid loss of phenotypic diversity. Whatever tacit grasp on the problem space’s more challenging terrain may have emerged in the population up to that point is quickly eclipsed and snuffed out. In the evolutionary computation literature, this dynamic is referred to as “premature convergence”.

What guards natural ecosystems against this development are the merciless pressures of crowding, scarcity, competition, which introduce a dynamic selective pressure for phenotypic diversity. The fitness rewards provided by low-hanging fruit are no longer boundless, but diminish in proportion to the number of individuals that reap them. At a certain point, the selective advantage no longer lies with those individuals that exploit the same, simple regions of the problem space, but with those who discover a niche that hasn’t yet been picked thin by crowds of competitors.

A similar tactic can be adopted in evolutionary computation, where it goes by the name of “fitness sharing”. At least two implementations of this strategy have become canonical in the literature: *explicit* fitness sharing, introduced in [7], and *implicit* fitness sharing, introduced in [15].

The underlying idea in both is that *selective advantage should be diluted by non-diversity*. Explicit fitness sharing “relies on a distance metric to cluster population members,” writes R.I. McKay in [10]. “Implicit fitness sharing,” by contrast, “differs from the explicit form in that no explicit distance metric is required. Instead, all population members which correctly predict a particular input/output pair share the payoff for that pair.” In ROPER we adopt a variation on the latter approach. The implementation is as follows:

1. each exemplar is initialized with a baseline **difficulty** score. It doesn’t much matter which value is used for this, but setting it to the inverse of the probability of solving the problem by random guess works well, when dealing with classification problems;
2. each problem is also allocated a **predifficulty** score, which is initialized to 1. Every time an individual responds to it correctly, the problem’s **predifficulty**

is incremented by 1; if the problem is such that a successful solution is a matter of degree (with 0.0 being a perfect solution and 1.0 being an utter failure), the predifficulty is incremented by  $1.0 - \text{score}$ );

3. after a  $N$  tournaments, where

$$N \leftarrow \frac{\text{population\_size}}{\text{tournament\_size} * (1 - x)}$$

and  $x$  is the probability of “headless chicken crossover” (cf. Algorithm 7), we iterate through the problem set. The problem  $e$ ’s `difficulty` field is set to

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

The higher, the harder, since `difficulty(e)` approximates 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  $1.0 - \text{difficulty}(e)$  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.)

#### 4.4.4.1 Mechanisms of Selection

This brings us back to where our algorithmic overview began: to the tournament algorithm used to select mating pairs. In the interest of bolstering the diversity of the population, and staving off premature convergence, we incorporated two fairly well-known modifications into the steady-state, tournament selection scheme described in Algorithm 3: the partitioning of the population into “islands” or “demes”, with rarefied points of contact, and the occasional use of “headless chicken crossover” as ongoing supply of novelty to the gene pool.

**4.4.4.1.1 Islands in the Bitstream** The mechanism used to isolate ROPER’s subpopulation or “demes” is extremely simple: when we go to select our candidates for each tournament, we do so by choosing  $n$  random indices  $\tilde{i}$  into the general population array, but each time we choose, we restrict ourselves to choosing integer

between 0 and some constant, `island_size`, decided in advance. The index  $j$  of the candidate is then set to  $j \leftarrow i * \text{island\_size} + \text{island\_id}$ . So long as this restriction is in place, each individual will only directly compete with its compatriots, throttling the speed at which the population is likely to converge on a single dominant genetic strain. This throttle is modulated by allowing the selection of every  $m^{\text{th}}$  candidate to be chosen from the general population, without any regard given to island of origin. The migration rate,  $m$ , can be easily adjusted to experiment with more and less genealogically interconnected populations.

**4.4.4.1.2 Headless Chicken Crossover** As a means of supplying the gene pool with an additional spring of novelty, we also make use of a simple technique called “headless chicken crossover”, which amounts to a small patch to Algorithm 3: we replace line 3 with Algorithm 7.

---

**Algorithm 7** Headless Chicken Patch

---

**Require:**  $H$ : float, with  $0.0 < H < 1.0$

---

**Require:**  $G, P, \min, \max$ : the parameters needed for Algorithm 5: the gadget pool, the integer pool, and the minimum and maximum length of new individuals

```

1:  $\mathfrak{R}, i \leftarrow \mathfrak{R}$ , pick a random float  $0 < i < 1$ 
2: if  $i < \text{headless\_chicken\_rate}$  then
3:    $\mathfrak{R}, \text{candidates} \leftarrow$  using  $\mathfrak{R}$ , pick  $n - 1$  from  $\Pi$ 
4:    $\mathfrak{R}, \text{candidates} \leftarrow \text{candidates} \cup \text{spawn}(\mathfrak{R}, G, P, \min, \max)$  {Using Algorithm 5}
5: else
6:    $\mathfrak{R}, \text{candidates} \leftarrow$  using  $\mathfrak{R}$ , pick  $n$  from  $\Pi$  {As before}
7: end if
```

---

## 4.5 Remarks on Implementation

The system described above has been implemented using the Rust programming language, and the Unicorn emulation engine [12] [11].

# 5

## Experimental Studies

### 5.1 Fitness Functions

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

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
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.


### 5.1.1 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.

#### 5.1.1.1 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.

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`<sup>1</sup>. 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~ is one such result: a 31<sup>st</sup>-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.

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

It's an extraordinarily labyrinthine chain, by human standards, and there's little in its genotype to hint at the path it charts 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 [6].

Table 5.1.1.1 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`.

---

<sup>1</sup>Available at <https://advancedtomato.com/downloads/router/rt-n18u>.

---

```

000100fc 0002bc3e 0002bc3e 0002bc3e
00012780 0000000b 0000000b 0000000b 0000000b 0002bc3e
00016884 0002bc3e 00012780 0002bc3e 0002bc3e 0002bc3e 0002bc3e 0000000b
000155ec 00000000 0000000b 0002bc3e
000100fc 0002bc3e 0000000b 00000000
0000b49c 0002bc3e 0000000b 0002bc3e 0000000b 0002bc3e
0000b48c 0002bc3e 00000000 0002bc3e 0002bc3e 0002bc3e
0000b48c 0002bc3e 0002bc3e 0002bc3e 0002bc3e 00000000
00016918 0002bc3e 0000000b 0002bc3e 0002bc3e 0000000b
00015d24 0002bc3e 00000000 00000000
00012a78 0000000b 00000000 0000e0f8 00000000
000109b4 0002bc3e 0000000b
0000b48c 0002bc3e 0002bc3e 0002bc3e 0000000b 0002bc3e
000100fc 0002bc3e 00000000 00000000
000109b4 0002bc3e 0002bc3e
00016758 0000000b 0000e0f8 0002bc3e
000100fc 0002bc3e 00000000 0000000b
00012a78 0002bc3e 0002bc3e
0001569c 0000000b 0002bc3e 0002bc3e
0000bfc4 0002bc3e 0002bc3e
00013760 0000000b 0002bc3e 0000000b 0002bc3e 0000000b
0000bfc4 0002bc3e 0002bc3e
0000b49c 0000000b 00000000 0000000b 0000000b 0002bc3e
00016884 0002bc3e
00012a78 00000000 0000000b
00011fd8 0000000b
00016758 0002bc3e
0000e0f8 0002bc3e
00013760 00000000 0000000b 0002bc3e 0002bc3e 0002bc3e

```

---

Table 5.1: A Rather Labyrinthine Chain



;; Gadget 0			[00016890] str r0,[r4,#0x1c]	R0: 0000000b R1: 00000000
[000100fc]	mov r0,r6		[00016894] mov r0,r4	R2: 00000000 R7: 0002bc3e
[00010100]	ldrb r4,[r6],#1			
[00010104]	cmp r4,#0		[00016898] pop {r4,lr}	;; Extended Gadget 1
			[0001689c] b #4294966744	[00012780] bne #0x18
[00010108]	bne #4294967224		[00016674] push {r4,lr}	[00012784] add r5,r5,r7
[0001010c]	rsb r5,r5,r0			[00012788] rsb r4,r7,r4
[00010110]	cmp r5,#0x40		[00016678] mov r4,r0	[0001278c] cmp r4,#0
			[0001667c] ldr r0,[r0,#0x18]	[00012790] bgt #4294967240
[00010114]	movgt r0,#0		[00016680] ldr r3,[r4,#0x1c]	[00012794] b #8
[00010118]	movle r0,#1			
[0001011c]	pop {r4,r5,r6,pc}		[00016684] cmp r0,#0	[0001279c] mov r0,r7
R0: 00000001 R1: 00000001			[00016688] ldrne r1,[r0,#0x20]	[000127a0] pop {r3,r4,r5,r6,r7,pc}
R2: 00000001 R7: 0002bc3e			[0001668c] moveq r1,r0	
				R0: 0002bc3e R1: 00000000
			[00016690] cmp r3,#0	R2: 00000000 R7: 0000000b
;; Gadget 1			[00016694] ldrne r2,[r3,#0x20]	
[00012780]	bne #0x18		[00016698] moveq r2,r3	;; Extended Gadget 2
[00012798]	mvn r7,#0			[000155ec] b #0x1c
[0001279c]	mov r0,r7		[0001669c] rsb r2,r2,r1	[00015608] add sp,sp,#0x58
[000127a0]	pop {r3,r4,r5,r6,r7,pc}		[000166a0] cmn r2,#1	[0001560c]
			[000166a4] bge #0x48	pop {r4,r5,r6,pc}
R0: ffffffff R1: 00000001			[000166ec] cmp r2,#1	
R2: 00000001 R7: ffffffff			[000166f0] ble #0x44	R0: 0002bc3e R1: 00000000
			[00016734] mov r2,#0	R2: 00000000 R7: 0000000b
;; Gadget 2			[00016738] cmp r0,r2	
[00016884]	beq #0x1c		[0001673c] str r2,[r4,#0x20]	;; Extended Gadget 3
[00016888]	ldr r0,[r4,#0x1c]		[00016740] beq #0x10	[00016918] mov r1,r5 **
			[00016750] cmp r3,#0	[0001691c] mov r2,r6
[0001688c]	b1 #4294967280		[00016754] beq #0x14	[00016920] bl #4294967176
[0001687c]	push r4,lr			[000168a8] push {r4,r5,r6,r7,r8,lr}
			[00016758] ldr r3,[r3,#0x20]	[000168ac] subs r4,r0,#0
[00016880]	subs r4,r0,#0		[0001675c] ldr r2,[r4,#0x20]	[000168b0] mov r5,r1
[00016884]	beq #0x1c		[00016760] cmp r3,r2	[000168b4] mov r6,r2
[000168a0]	mov r0,r1			[000168b8] beq #0x7c
[000168a4]			[00016764] strgt r3,[r4,#0x20]	
pop {r4,pc}			[00016768] ldr r3,[r4,#0x20]	[000168bc] mov r0,r1
			[0001676c] mov r0,r4	[000168c0] mov r1,r4
R0: 00000001 R1: 00000001				[000168c4] blx r2
R2 00000001 R7: 0002bc3e			[00016770] add r3,r3,#1	
			[00016774] str r3,[r4,#0x20]	R0: 0002bc3e R1: 0002bc3e
;; Extended Gadget 0			[00016778] pop {r4,pc}	R2: 00000000 R7: 0000000b

Table 5.2: Disassembly of a succesful chain, with ‘extended gadgets’. \*\* indicates where the pattern is completed.

### 5.1.1.2 Intron Pressure, Self-Modifying Payloads, and Extended Gadgetry

What pressures could possibly be driving the evolution of such strange specimens? The canonical set of gadgets that the population inherits as its primordial gene pool is noisy and brittle enough, but at least those gadgets are selected for stability – first, prior to each run, by our gadget harvesting routines, which look for code fragments that are at least *likely* to preserve control flow, and then, throughout the run, by fitness pressures that penalize the loss of execution control (chains which crash before completion, or which do not reach the designated termination address within a fixed number of steps), and genetic operators that will tendentially drop unreliable gadgets from the gene pool. And yet we find a tendency for the population to occasionally favour gadgets that overwrite the individual’s own code stack, and branch to uncharted regions of executable memory that have no direct representation in the set of gadgets making up the gene pool.

This type of behaviour appears to emerge only at a certain phase of the evolutionary trajectory, which is no doubt significant: it has a tendency to be favoured by periods during which the average fitness of the population more or less plateaus, and its standard deviation narrows.

As Bahnzaf and others have shown [cite:], these are typically the conditions under which we should expect to see signs of an accelerating accumulation of *introns*, or non-coding genes, in the population. The reason for this, Bahnzaf conjectures, is that as dramatic improvements in the performance of the specimens, with respect to their explicit fitness function, become increasingly difficult to attain, and as specimens more and more find themselves competing against relative equals, the immediate selective pressures imposed by the fitness function become less decisive in steering the course of the evolution. The greatest differential threat to our specimens – or, rather, to their genetic lineages – during such plateaus, is no longer the performance of their immediate rivals, but the destructive potential of the genetic operators themselves. There is very little, after all, to prevent crossover or even mutation from mangling the genome beyond repair, and yielding dysfunctional offspring. Unlike animals, plants, or any advanced life forms familiar to us from nature, our creatures lack any sophisticated mechanism for ensuring the homological transfer of genes in sexual reproduction.

There is very little to predispose crossover operations to preserve adaptive groupings of genes, or to replace the genes of one parent with semantically similar genes from the other. The only structural constraint that we have explicitly afforded to those operations is a fairly lighthanded “fragility” mechanism, that, over time, decreases the chance that crossover will break apart adjacent pairs of genes that have historically (in terms of the individual’s own genealogy) performed well together. But this is a very mild constraint.

The gene lines best protected against such threats are those that are structured in such a way that crossover is least likely to do damage, or to break apart genes that are best kept together. A relatively simple way to achieve such protection is to pad the genome with semantically meaningless, or “non-coding”, sequences. So long as the probability that any gene sequence will be affected by the action of a genetic operator is inversely proportionate to the length of the genome, increasing the genome’s length by adding otherwise ineffectual sequences makes it less likely for those operators to mangle it in a semantically meaningful – and a fortiori, semantically maladaptive – way.

Introns are therefore a valuable resource for the gene pool, and are favoured by selection as soon as the threat posed by the genetic operators outweighs the threat posed by immediate rivals. A particularly common form that introns may take, and which we see in a variety of genetic programming systems, is a NOP instruction, an instruction that does nothing, or some sequence of instructions that semantically cancel one another out. In order to exploit that resource, however, we need both a base language in which NOPs or NOP sequences are relatively common, and latitude in the maximum length of the individuals, so that introns can be freely padded onto the genome.

In the context of ROP chains, a NOP is just a gadget that returns without performing any other operations. If we were dealing with gadgets defined over the Intel instruction set, we could find these just by taking the address of RET instructions. When it comes to ARM, however, such gadgets are significantly rarer. We rarely find a pop instruction that *only* pops into the program counter, without tainting the other registers as well. For reasons of efficiency, most compilers favour multi-pop instructions. Longer gadgets are even less likely to execute without inducing

side-effects. As we have already noted, we simply do not have the luxury of a sleek, minimal, more or less orthogonal instruction set, where each instruction performs a single, well-defined, semantically atomic operation. Our instruction set will almost always be a noisy assemblage of irregular odds and ends, in which the sort of introns we typically encounter in genetic programming systems is rather uncommon.

Gadgets that overwrite or leap out of their own ROP stack, on the other hand, are relatively easy to come by. Though they pose a tremendous risk to the gene line, when it comes to first-order fitness, they offer access to an otherwise scarce resource: they protect against damaging crossover operations, by rendering the entire, unused sequence of gadgets that will be either overwritten or avoided, an unbroken sequence of introns. Crossover and mutation can do whatever they will to the lower regions of these aberrant genome without inflicting any damage on adaptive clusters of genes.

It’s interesting to observe that this particular strategy for intron padding resembles the use of resources found in the individuals’ “environment”.

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 {shellpattern-graph}, about halfway back along the champion’s phylogenic tree, the percentage of crashes in the population peaked to levels unseen since 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.<sup>2</sup>

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.

---

<sup>2</sup>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 [4]

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.

### 5.1.2 Testing the Extended Gadgetry Conjecture with Explicitly Defined Introns

It should be possible to test the merits of this explanation by setting up a few simple experiments. If, as I have conjectured, the peculiar behaviour appears because it represents a risky source of introns, which our system has, in various ways, made a rare resource, then we should expect to see it decrease in frequency as a consequence of introducing a risk-free supply of **explicit** introns into the gene pool. This can be easily done. All we need to do is to define a new type of **clump** that doesn't code for any gadgets or immediates in the actual payload, but which can still be manipulated by the genetic operators. This should increase the number of genotypes that map to identical phenotypes, affording a greater margin of independence of the latter from the former.

## 5.2 Evolving 'Intelligent' Payloads

### 5.2.1 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 (§??).

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

<sup>3</sup>Available at <https://archive.ics.uci.edu/ml/datasets/Iris>} 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/g 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 ???. A dip in average length coincides with the peak in the crash rate, around phylogenic 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 ??? 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 ??.

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 ?? shows the course the 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. ?. Note the tight interbraiding of problem difficulties in fig. ?, as compared to their gaping – but still, slowly, fluctuating – disparity in fig. ?. The ballooning standard deviation of difficulty by class in fig. ? also suggests a dramatic increase in behavioural diversity in the population, which is precisely what we aimed for with fitness sharing.

### 5.2.2 Snake

### 5.2.3 Aside: A Plague of Segfaults

# 6

## Future Work



### 6.1 Current Limitations and Open Problems

ROPER I faces at three serious limitations in its design:

1. the programming interface that it exposes to the evolutionary algorithm is brittle and uneven, and in no way optimized for evolvability;
2. the evolutionary process has little means of gaining traction on the genotypes' program semantics – in themselves, the genotypes are little more than vectors of integers, and there is no way of acquiring any information of how those vectors will behave, except for executing them – and this is something at which each individual only ever gets a single attempt;
3. the reproduction algorithms are fixed, and, as we have seen, most frequently destructive. There is nothing inherent in crossover, or in our mutation operations, that makes them well suited to the problem of recombining ROP chains, or



exploring the uneven, and largely uncharted, semantic space that the execution of those chains represents. We may not *know* a better algorithm, but perhaps we could at least let ROPER explore other possibilities, itself, and expose the reproduction algorithms themselves to evolutionary pressure.

## 6.2 A More Robust and Evolvable Intermediate Language (Push)

The design for the second iteration of ROPER owes a great deal to Lee Spector, who made the suggestion (at GECCO '17) that the issues with semantic opacity and brittleness that I was grappling with in ROPER 1 might become more tractable if, instead of having the individuals of my population be more or less direct representations of ROP-chain payloads, I instead evolved populations of ROP-chain builders – programs that would compose ROP-chains from the available materials, but which may, themselves, have a very different structure.

The ontogenetic map from genotype to phenotype would then consist of two phases, rather than just one:

1. a mapping from the builder’s code to a constructed payload, implemented by executing the builder,
2. the mapping we’re already familiar with, from ROPER, which maps the constructed payload (ROP chain) to the behaviour of the attack in the emulated host.

The language in which the builders are defined could then be tailored to fit the situation as well as possible – pursuing a strategy similar to the one that SPTH used in the design of Evolis and Evoris.

I decided to experiment with style of language that Spector had, himself, introduced into genetic programming, and write a dialect of PUSH for this purpose.

### 6.2.1 Raising the Level of Abstraction with PUSH

#### 6.2.1.1 The PUSH Idiom

PUSH is a statically typed FORTH-like language that is designed with an eye towards evolutionary methods rather than use by human programmers. Unhandled

exceptions, for instance, are effectively absent from the language, optimizing it for mutational robustness rather than debugging and predictability.

#### **6.2.1.2 BNF Grammar for ROPUSH**

#### **6.2.2 Autoconstructive Reproduction**

#### **6.3 Semantic Analysis and Synthesis (Q)**

#### **6.4 Utility in the Wild with Blind ROP (Braille)**



**7**

~~Conclusions~~

# Bibliography

- [1] Sperl Thomas (aka Second Part to Hell). Artificial evolution in native x86 systems. Retrieved from the author’s website., 2010.
- [2] W. Banzhaf. *Genetic Programming*. Morgan Kaufmann, 1998.
- [3] A. Bittau, A. Belay, A. Mashtizadah, D. Mazieres, and D. Boneh. Hacking blind. In *IEEE Security and Privacy*, pages 277–242, 2014.
- [4] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.
- [5] Sergey Bratus. Offense and defense: Notes on the shape of the beast. presented at BSides Knoxville, June 2016.
- [6] R. Dawkins. *The Extended Phenotype: The Long Reach of th Gene*. Oxford University Press, 1999.
- [7] K. Deb and D.E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 42,50, 1989.
- [8] Halvar Flake. Understanding the fundamentals of attacks: What is happening when someone writes an exploit? presented as the keynote talk at the Cyber Security Alliance, September 2016.
- [9] William Gibson and Tom Maddox. Kill switch, Feb 1998.
- [10] R. I. McKay. Fitness sharing in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 435–442. Morgan Kaufmann, 2000.
- [11] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn: Next generation cpu emulator framework. <http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf>, 2015.
- [12] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn cpu emulator framework. <http://www.unicorn-engine.org/>, 2015–2018.

- [13] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.
- [14] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In *USENIX Security Symposium*, 2011.
- [15] R.E Smith, S. Forrest, and A.S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1:127–149, 1992.
- [16] Second Part to Hell. Chomsky hierarchy and the word problem in code mutation. Retrieved from the author’s website., 2008.