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

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

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abstract

This thesis lays out the design and implementation of a system for evolving populations of return-oriented programs, and then proceeds to study their behaviour in a series of experiments.

Acknowledgements

I would like to thank my supervisors, Nur Zincir-Heywood and Malcolm Heywood, whose support, encouragement, and expertise have been indispensable to me. My thanks also go to to John Jacobs at Raytheon SAS, who saw enough interest and potential in this project to grant it his agency's financial backing, when it was still in its early stages. And thanks to all the friends and colleagues who helped the ideas presented in these pages come to life in conversation. Without any hope of being exhaustive, these include Andrea Shepard, Meredith Patterson, Pete Wolfendale, Giancarlo Sandoval, Neha Spellfish, Pompolic, Petra Kendall, Peli Grietzer, Alice Maz, 0xdeadbabe, and others in the Greytribe and Special Circumstances chat, Sig Cox at LambdaConf, Chris Watts, Corrie Watts, Raphael Bronfman-Nadas, Deepthi Rajashekar, Aimee Burrows, Rob Curry, Robert Smith, and Stephen Kelly in the NIMS laboratory, past and present, Ed Prevost, Gurjeet Clair, Katie Sexton, and Nick Miles at Tenable (again, past and present), and Rob Piece at 2Keys. And thank you to my long-suffering parents Marion and Zachary Fraser, for the innumerable hours of childcare they freely gave my brood, so that I could finish this thesis while they're still young.

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/`

List of Abbreviations and Symbols Used

SP_Δ	stack pointer shift. 40, 41, 88
$w \oplus x$	write xor execute. *see* DEP, 8, 9, 12, 59
ARM	Advanced RISC Machine. viii, 10, 37, 40, 45, 59, 61, 63, 72, 78, 104
ARMv7	Advanced RISC Machine, Version 7. 32, 35, 37
BAP	the Binary Analysis Platform. 58, 116
CISC	complex instruction set computer. 59
CPU	central processing unit. 6, 7, 9, 11, 12, 13, 30, 35, 37, 45, 48, 51, 52, 58, 63, 66, 67, 72, 74, 75, 78, 80, 107
DEP	Data Execution Prevention. *see* $w \oplus x$, 8
EDI	explicitly defined introns. 105, 106, 107, 108

ELF Executable and Linkable Format. vii, viii, 59, 61, 77, 78, 112

FP the frame pointer register. *see* ARM, 10

GNU GNU's Not Unix. *see* GNU

GP genetic programming. 1, 4

GPL GNU Public License. *see* GNU, 3

HC-128 the HC-128 stream cipher. 59

HTTP Hypertext Transfer Protocol. 72

I/O input/output. 58

IDS intrusion detection system. 2

IFSM intended finite state machine. 9

IP the instruction pointer. *see* PC & ARM, 7, 10

ISA instruction set architecture. 58, 59

JOP jump-oriented programming. 12

LR the link register register. *see* ARM, 10

Mach-O Mach object file format. 59

PC the program counter register. *see* IP & ARM, 10, 40

PE Portable Executable. 59

PRNG pseudo-random number generator. 59

QEMU Quick Emulator. 57, 58, 114

RCE remote code execution. 6, 114

RISC reduced instruction set computer. 59

ROP return-oriented programming. vii, 1, 4, 11, 12, 26, 30, 57, 58, 63, 66, 68, 75, 78, 79, 92, 98, 106, 111, 112, 116

ROPER Return-Oriented Programming with ROPER. *see* ROPER, vii, 1, 2, 4, 26, 31, 32, 35, 36, 41, 43, 46, 45, 50, 51, 53, 56, 57, 58, 59, 61, 63, 64, 66, 67, 68, 69, 70, 72, 74, 75, 77, 78, 79, 80, 84, 86, 88, 89, 92, 95, 97, 101, 105, 107, 111, 112, 113, 114, 115, 116

ROVM return-oriented virtual machine. 11, 12, 41, 111, 116

SP the stack pointer register. *see* ARM, 10

SPMM structured programming machine model. 6, 12

SQL Structured Query Language. 6, 7

SQLi SQL injection. *see* SQL

TTL time-to-live. v, 88, 113

VM virtual machine. 57, 114

VX virus writing. 18, 24, 27

VXer virus writer. 18, 21

1

Introduction

1.1 What is the aim of this research?

This thesis explores the use of evolutionary techniques in ROP. It details the design and implementation of an engine called ROPER, which employs the methods of genetic programming (GP) 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.[1]

[1] Unless we cast a null "state of the art" to zero.

1.2 Why is this interesting?

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 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.

1.3 Where can this work be applied?

The techniques developed here can quite viably be put to work in the field of offensive cybersecurity, and be used to generate swarms of attack payloads whose diversity is, for all intents and purposes, unbounded. The technology developed here could, with minor adaptations,[2] used to test and train intrusion detection systems (IDSs), or provide one more instrument in the penetration tester's toolbox.

1.4 Who is this for?

I hope that the work presented here may be of interest to newcomers to both low-level exploit development and genetic programming, and to those who may have a solid background in one but not the other. The work presented here shows how problems drawn from the field of application security provide an extremely fertile ground for evolutionary experimentation, which I believe is of interest in its own right, independent of applications.

Of course, there's nothing preventing the use of this technology by malicious actors, and in this respect ROPER is in the same boat as any other product of security research – the only defence against use by blackhats, after all, would be to ensure

[2] Discussed in Section 6.

that the research is useless. This isn't just an unavoidable aspect of security research, it's one of its essential motors. Without the endless arms race between attacker and defender, between whitehat and blackhat, it's unlikely we'd have even an *sliver* of the understanding of our own abstractions – and of all their leaky concretizations – that the concern for security demands. The harsh reality that any worthwhile development in security can be picked up and studied by blackhats seeking to use, abuse, apply, subvert, and exploit it, isn't something we should shy away from or apologise for. It's the crucible in which our ideas and their implementations are tested, and a tireless generator of new ideas in its own right. The economic and political fates of attackers and defenders may rise and fall in the arena of applied cybersecurity, but the science ratchets on, day and night. [3]

That said, we should nevertheless take a moment to consider the risks posed by the introduction of evolutionary malware, or any technology that could facilitate its development, into the existing information security ecosystem.

I have decided to make the source code for this project available to the public, warts and all, and place it under the GNU Public License (GPL). It can be accessed on

[3] In the immortal words of Pastor Manul Laphroaig:

> I must warn you to ignore this Black Hat/White Hat nonsense. As a Straw Hat, I tell you that it is not the color of the hat that counts; rather, it is the weave. We know damned well that patching a million bugs won't keep the bad guys out, just as we know that the vendor who covers up a bug caused by his own incompetence is hardly a good guy. We see righteousness in cleverness, and we study exploits because they are so damnably clever! It is a heroic act to build a debugger or a disassembler, and the knowledge of how to do so ought to be spread far and wide. First, consider the White Hats. Black Hats are quick to judge these poor fellows as do-gooders who kill bugs. They ask, "Who would want to kill such a lovely bug, one which gives us such clever exploits?" Verily I tell you that death is a necessary part of the ecosystem. Without neighbours squashing old bugs, what incentive would there be to find more clever bugs, or to write more clever exploits? Truly I say to the Black Hats, you have recouped every dollar you've lost on bugfixes to the selective pressure that makes your exploits valuable enough to sustain a market! Next, consider the Black Hats. White Hat neighbors are so quick to judge these poor fellows, not so much for selling their exploits as for hoarding their knowledge. A neighbor once said to me, "Look at these sinners! They hide their knowledge like a candle beneath a basket, such that none can learn from it." But don't be so quick to judge! While it's true that the Black Hats publish more slowly, do not mistake this for not publishing. For does not a candle, when hidden beneath a basket, soon set the basket alight and burn ten times as bright? And is not self-replicated malware just a self-replicating whitepaper, written in machine language for the edification of those who read it? Verily I tell you, even the Black Hats have neighborliness to them. So please, shut up about hats and get back to the code.

Github, at `https://github.com/oblivia-simplex/roper`, and freely experimented with.

1.5 Synopsis

In Chapters 2 and 3, I set up some of the conceptual background for this study, exploring the broader problems broached by ROP and GP, respectively, before surveying a handful of historical efforts to enlist evolutionary techniques in the domain of offensive security and malware design, in Section 3.1.

Chapter 4 introduces my contribution to research in the field of evolutionary offensive security, with an overview of the design and implementation of a ROP evolution engine called ROPER.

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

Chapter 6 lays out some directions for future work and study on this topic, and 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

At 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 – the distinction between data and code, on which so much of practical computing is founded, tends to fade from view.[1]

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

[1] 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.

the result of applying any sequence of transformations to that information. "Data", in other words, is just what flows from one system to another. Insofar as those systems are meant to be *distinct* – with different capabilities, different access rights, and so on – the notion of data is immediately bound up with those of security and trust. If we are to have any assurances at all about the behaviour of system A, after all, 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.[2] Data is *just* data, as opposed to "code", only to the extent that such constraints 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.[3] In many such cases, the breach occurs when the attacker slips past the *intended* interface and dispatches instructions (performs state transitions) on one or more of the system's "internal" components. Take the classic Structured Query Language (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 interprets 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" [25]. 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-measured buffer. What makes this dangerous is that, according to a certain, widely implemented abstract machine model, which for lack of a better name,[4] we could call the "structured programming

[2]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.

[3]I owe this formulation to Sergey Bratus.

[4]I've seen it called "the C abstract machine model", which comes close but which is overly specific.

machine model (SPMM)", 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 central processing unit (CPU). In these cases, and in many, many more, the attacker succeeds in exploiting some oversight in the design or implementation of the input 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, where the input "data" is transubstantiated into "code" of some sort – the aberrant delimiter in the SQL injection, and the corruption of the instruction pointer, in the case of the buffer overflow, are both instances of this;

2. the "payload", through which the hacker exercises fine-grained control over the vulnerable system. In the case of the buffer overflow attack, this might 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 the the instruction pointer (IP) is pointing. 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 typically follow one of two prongs: either they inhibit the *pivot* stage, or they inhibit the *payload*.

I'm trying to point to something more concrete than an idealization like "the Von Neumann machine model", and more abstract than, say, "the System V ABI".

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.). The onus, in this case, falls on the developer, or her linter. 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 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 blocked by obstructing 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 execute ($w \oplus x$)", 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*.[5] 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). This 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*.[6]

[5]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 [7].

[6]"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" ([13]).

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,

> is the *intended finite state machine* [...]
>
> The security properties of the intended finite state machine (IFSM) 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 that 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 design. These are what Sergey Bratus [6] 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", Flake continues,

> 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, which will remain the backdrop for much of what follows.

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*, blinkered by what Meredith Patterson has called "boundaries of competence" [24], that $w \oplus x$ ultimately fails to prevent remote code execution. It fails because it is built on an insufficiently general concept of *code*.

2.2.1 The Structured Programming 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_64, the program counter register (PC) on ARM, etc.) Each instruction prompts the processor to mutate its state (its registers, memory, etc.) in some fashion. "Code" is wherever IP points, and he 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 *interleafed* in practice.

This interleafing is orchestrated, on most modern architectures, by means of three abstract registers: the stack pointer register (SP), the frame pointer register (FP), and IP. On x64_64, these are implemented by `RSP`, `RBP`, and `RIP`, respectively. On ARM,

by `SP/R13`, `FP/R12`, and `SP/R15`. When a subroutine is called, the address of the next instruction address in the calling routine is typically pushed onto the stack. (In some cases a special register is used to hold the most recent return address – the top of the abstract calling stack – as an optimization. This is the role of the link register register (LR) on ARM. For nested subroutine calls, however, it's necessary to fall back to a stack structure. The FP is then used to mark the base of the scope of stack memory that belongs to the subroutine. Any memory beyond FP is the subroutine's own to make use of, though this claim is abandoned when the subroutine returns. Returning from a subroutine, in most cases, is just a matter of popping the return address from the control stack, and loading it into the instruction pointer. On x64_64, this is accomplished by the `ret` instruction, on ARM, by `pop {pc}`, and on MIPS by first loading a register from the top of the stack, and then jumping to that register.

This is, of course, why an attacker can "smash the stack for fun and profit". Even if they must tailor their attack for a specific architecture, they are attacking a vulnerability in the C virtual machine: that improperly handled writes to the data stack can corrupt the control stack with which it is interleafed. The interleafing makes accessible to the attacker the critical kind of *weird states* on which their attack pivots. In executing this attack, the attacker violates the conceptual separation of schematically interleafed control and data stacks, but otherwise remains within the same basic abstract machine model. An elegant shellcode payload will even take care to restore any corrupted registers, clear its own local stack, and return control to the caller, as if nothing out of the ordinary had happened. The attacker is descending to a lower level of abstraction, but not an entirely foreign one. It is a level already implicit, and (leaks notwithstanding) encapsulated in the victim process.

2.2.2 The ROP Virtual Machine

A ROP chain can be seen as a program written to run on a weird 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 return-oriented virtual machine (ROVM).

Like the programmer's machine model, the ROVM works by iterating through a sequence of instructions, tracking the location of the next instruction by means of a

special registerr, 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 chosing 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 "jump-oriented programming (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. In general terms,

> To be able to build a program from gadgets, they must be combinable. Gadgets are combinable if they end in an instruction that, controlled by the user, alters the control flow. Instructions which end gadgets are named 'free branch' instructions. A 'free branch' instruction must satisfy the following properties:
>
> - The control flow must change at this instruction.
> - The target of the control flow must be controllable (free) such that the input from a register or stack defines the target. [21]

The ROVM 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 beautifully illustrates the ingenuity of ROP. $w \oplus x$, 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 ROVM *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. 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 SPMM *expects* an interleafing 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 ROVM on top of the SPMM: we don't need to execute SPMM 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 SPMM: 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 SPMM already *implies* the possibility of the ROVM, which is its shadow. The SPMM's interleafing 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.

To paraphrase Eliot: 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` [7] – to tools We are aware of two such projects at the moment: *Q* [28], 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. [8] The latter has since spawned a fork that aims to use 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 [4]. BROP solves the problem of developing ROP-chain attacks against processes where not only the source code but the binary itself in unknown. *Braille* first uses a stack-reading technique to probe a vulnerable process (one that is subject to a buffer overflow and which automatically restarts after crashing), to find enough gadgets, through trial and error, for a simple ROP chain whose purpose will be to write the process's executable memory segment to a socket, sending that segment's data back to the attacker – data that is then used, in conjunction with address information obtained through stack-reading, to construct a more elaborate ROP-chain the old-fashioned way. It is an extremely interesting and clever technique, which could, perhaps, be fruitfully combined with the genetic techniques 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.

[7] `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.

[8] `https://github.com/pakt/ropc`

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 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.

3.1.1 Viruses and Evolutionary Computation

3.1.1.1 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.

> 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.

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.

3.1.1.2 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* [9]. *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 then-hypothetical anti-virus software through Gödelian 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 [1] 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 corrolary of Rice's theorem)

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

[1] [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, 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 virus writing (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 virus writers (VXers) who have implemented genuinely evolutionary methods in their work seem to be motivated primarily by hacker's curiosity and not by monetary gain. 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 publicly 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 targetted 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.

3.1.1.3 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 bytewise 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.

3.1.1.4 2002: MetaPHOR[2]

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:

> 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).

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 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. (Descendants of the

[2] W32/Simile, {W32,Linux}/Simile.D, Etap.D

virus, for instance, may adapt in such a way as to never use decryption, if that should turn out to the 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.

3.1.1.5 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:

> 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.

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.

3.1.1.6 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 indiscernibly to a scanner, than 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 [30].

3.1.1.7 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. [3]

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. With them, 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" ([3], 18).

3.1.1.8 Concluding remarks on the history of evolutionary techniques in virus programming

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. Outside of science fiction [16], however, we have not seen any discernable 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. There is nevertheless 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 hasn't failed to capture the imagination.

In a paper presented at the 2008 *Virus Bulletin* conference, two artificial life researchers, Dimitris Iliopoulos and Christoph Adami, together with malware analyst Péter Ször of Symantec, outline the threat that such technology may pose and the extent to which it would be feasible to produce [20]. The greatest risk, it seems, concerns

the possibility of detecting such malware. Existing obfuscation techniques, they note, all share the same theoretical limit: though polymorphic and metamorphic variants of a malware strain may evade literal signature detection, and and syntactic/structural detection, they do tend to share common *semantic* invariants, and remain vulnerable to detection by means of a well-tuned behavioural profile. "Simply put," they write,

> biological viruses are constantly testing new ways of exploiting environmental resources via the process of mutation. In contrast, computer viruses do not exhibit such traits, relying instead on changing their appearance to avoid detection. *Functional* (as opposed to cryptic) variation, such as the discovery of a new exploit or the mimicry of non-malicious behaviour masking malicious actions, is not part of the arsenal of current malware.

Evolutionary techniques, by contrast, could allow for the generation of malware instances whose semantic variation is bounded in extremely minimal, abstract, and subtle fashions, as demanded by the task at hand, offering little to no foothold for existing detection technologies. If allowed to develop more freely, moreover, with no selective pressures beyond replication, survival, and the subversion of the systems intended to stop them – and if they could incubate in environments where those particular pressures are gentle enough to allow for relatively "neutral" (non-advantageous, but non-deleterious) exploration of their environment – then "the emergence of complex adaptive behaviors becomes an expected result rather than an improbability, as long as exploitable opportunities exist within the malware's environment" [20].

Ször, Iliopoulos, and Adami, here, are discussing the use of evolutionary techniques in virus generation, rather than payload generation, as examined in this thesis – and, indeed, as we'll see, despite the relative dearth of concrete advancements, the theme of evolutionary computation has been a preoccupation of virus writers ever since the first computer virus was crafted, a phenomenon we don't see paralleled in other fields of offensive/counter-security. There are challenges facing the deployment of evolutionary malware "in the wild" that we don't encounter when developing it "in vitro" – that is to say, in a virtual laboratory, where selective pressures can be fine-tuned with care, rather than left to external circumstance. Where the research

presented here rejoins Szöor, Iliopoulos, and Adami's anticipations is in examining the results of relatively free and unconstrained exploration of a host environment by evolutionary malware, where the tether to semantic invariance is intentionally kept as loose as possible and the specimens have the ability to salvage and recombine whatever functional code they can from their hosts. This is, after all, the very nature of a "code reuse" or "data-only" attack – terms often given to ROP in the literature – a quality that makes them an especially appealing subject for evolutionary study. It should nevertheless be emphasized that we are *not* engineering *viral* malware here – ROPER's populations are not capable of *self*-replication, and do not encapsulate their own genetic operators. For that, they rely on the ROPER engine. Once generated, they can certainly be *deployed* in the wild, but we do not expect any such specimens to be capable of reproduction, there, and so their evolutionary history ends as soon as they exit the incubator.

The historical lineage that comes closest to what we are doing here, then, is what we could call *evolutionary payload generation.*

This lineage is considerably shorter – we see no comparable fascination with evolutionary techniques in the exploit-writing world, as compared to the VX scene – but the achievements that have been made in this direction tend to be considerably more robust, in terms of evolutionary computation. The most likely reason for this is simply that payload evolution – where the malware is *produced* using genetic techniques, but is not expected to *continue* evolving once "released" – is amenable to laboratory study, and to rapid iterations of the evolutionary cycle, in a way that virus crafting is not.

3.1.2 Genetic Payload Crafting

3.1.2.1 Gunes Kayacik and the evolution of buffer overflow attacks

Gunes Kayacik's 2005-2011 research (see [18], [**?**], and [19], for instance) 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 shell-code 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.

Kayacik's research was one of the initial inspirations for the current project, and remains one of the very few serious attempts to put evolutionary methods to work in the domain of offensive cybersecurity. I was quite surprised to find – or, rather, fail to find – any significant research by others, continuing in this vein, after 2011.

3.1.3 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 x64_64 – 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.

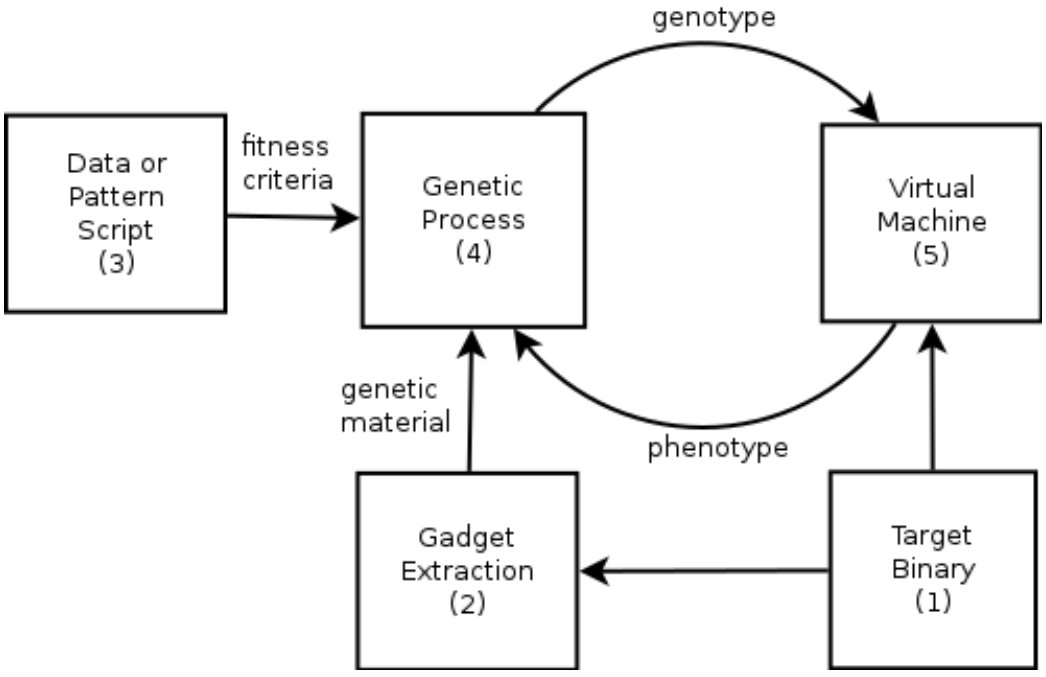

Figure 4.1: A bird's eye view of ROPER

Algorithm 1 Population Initialization

Require: ε, ELF binary of the process to be attacked

Require: π, the problem set specification

Require: n, the desired population size

Require: ι, a pool of raw integers

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

1: text, rodata $\leftarrow$ parse(ε)
2: $\iota \leftarrow \iota \cup$ find-pointers(rodata, ι)
3: $\gamma \leftarrow$ harvest-gadgets(text)
4: $\Pi \leftarrow$ empty-vector(n)
5: $\mathfrak{R} \leftarrow$ seed($\mathfrak{R}, s$)
6: **for** $x \leftarrow 1$ **to** n **do**
7: $\quad \mathfrak{R}, \Pi_x \leftarrow$ spawn-individual($\mathfrak{R}$, γ, ι, 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 targetting the 32-bit Advanced RISC Machine, Version 7 (ARMv7) architecture, in ELF format, but there is nothing essential about this restriction, and ROPER could easily be extended to handed 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

Algorithm 2 Genotype Evaluation (Ontogenesis)

Require: E, the CPU emulator

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

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

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

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

Require: Γ, the genotype to be evaluated

1: $\sigma \leftarrow$ serialize(Γ) $\cup$ [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 σ into stack memory of E
6: $E \leftarrow$ exec(E, `"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: $E \leftarrow$ step(E) {*fetch instruction at* `PC` *and execute*}
10: $i \leftarrow i + 1$
11: **end while**
12: accumulator $\leftarrow$ acc(accumulator, case.weight, φ(read-registers(E), case.out))
13: $E \leftarrow$ reset(E)
14: **end for**
15: **return** accumulator {*the 'phenotype'*}

Algorithm 3 Evolve Population (Tournament Selection)

Require: Π, the population {*as initialized by Algorithm 1*}

Require: E, the CPU emulator

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

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

sweep nevertheless suffices to provide us with a fairly 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 primtive 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" ([31], 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. [1]

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 assumping 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 $\mathbf{C}$, 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 α, `ab` parses to β, `bb` parses to τ, `cc` parses to

[1] See Section 2.2 for a sustained explanation of how return-oriented programming works, and an explanation of the concept of 'gadget'.

δ and `bcc` parses to γ. 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 δ 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: **C**: a contiguous vector of bytes representing instructions

Require: $\ulcorner X_j \urcorner : [\textbf{byte}] \to \mathbb{N} \to \text{inst}|\Lambda$, a parsing function, from byte-vectors X and indices j to instructions, or Λ in case of unparseable bytes.

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

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

Require: δ: positive integer, offset of base virtual address for **C**

$\Gamma \leftarrow$ empty stack of integers
$i \leftarrow \text{length}(\mathbf{C})$
while $i > 0$ **do**
  $i \leftarrow i - 1$
  if $\rho(\ulcorner \mathbf{C}_{i+1} \urcorner)$ **then**
    while $\neg\varphi(\ulcorner \mathbf{C}_i \urcorner)$ **and** $i > 0$ **do**
      push i onto Γ
      $i \leftarrow i - \text{length}(\ulcorner \mathbf{C}_i \urcorner)$
    end while
  end if
end while
$\Gamma^\star \leftarrow$ map $(\lambda x.\delta + x)$ over Γ
return $\Gamma^\star$

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 $\frac{\texttt{0x187ec}}{4} + \frac{\texttt{0x187ec}}{2} = \texttt{0x125f1}$ 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. [2]

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

[2]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 PC, while other integers integers land predominantly in general-purpose registers. To do this, we calculate the distance the stack pointer will shift when each gadget executes, the stack pointer shift ($\mathtt{SP}_\Delta$) of g or $\mathtt{SP}_\Delta(g)$, and then clump together each gadget pointer g with a vector of $\mathtt{SP}_\Delta(g)-1$ non-gadget values. Consider, for example, the instruction,

$$\mathtt{LDMIA!} \quad \mathtt{SP}, \quad \{\mathtt{R0}, \mathtt{R7}, \mathtt{R9}, \mathtt{PC}\}$$

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 $\mathtt{SP}_\Delta$ of 4. For a gadget g, we define $\mathtt{SP}_\Delta(g)$ as

$$\mathtt{SP}_\Delta(\pi) = \sum_{i \in \pi} \mathtt{SP}_\Delta i$$

for some control path π in 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 π is unique for each g. If this condition is relaxed, we suggest generating n distinct clumps for each distinct member of $\{\mathsf{SP}_\Delta(\pi)|\pi \textit{ is a control path in } g\}$. 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 SP_Δ values. A good rule of thumb, here, is that when the approximation of $\mathsf{SP}_\Delta(g)$ 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 $\widehat{g}$, a "clump" around g can now be assembled by taking a stack of $\mathsf{SP}_\Delta(g) - 1$ arbitrary values, and pushing $\widehat{g}$ on top of it. By the time g has run to completion, it will have popped $\mathsf{SP}_\Delta(g)$ values from the process stack. The first $\mathsf{SP}_\Delta - 1$ of these will populate the general purpose registers of the machine, and the ${\mathsf{SP}_\Delta}^{th}$ will pop the entry point of the *next* gadget, g', into `pc`. That entry point, $\widehat{g'}$ will be found at the top of the next clump in the sequence that makes up the genotype. [3]

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 ROVM, 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 detailed in Algorithm 5.

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

Algorithm 5 Spawning an Initial Individual

Require: $\mathbf{G} : [\![\mathbb{N}^{32}]\!]$, a set of gadget pointers

Require: $\mathbf{P} : [\![\mathbb{N}^{32}]\!]$, a set of integers and data pointers

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

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

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

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 selects, randomly, from a set of transformations, which it then applies 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. The operations currently available include:

1. arithmetically, by applying either numerical operations such as addition and subtraction;

2. bitwise operations, such as shifts, rotations, sums, and products by a randomly selected constant value;

3. the pointer operations of dereference (interpreting a value as a pointer, when possible, and replacing it with the value found at the corresponding address in the process's memory) and indirection (the somewhat more costly ($O(n)$ over the size n of the memory space) operation of searching through memory for an instance of the value, and replacing it with a pointer that dereferences to it). When a value cannot be dereferenced as a pointer, or a pointer to a word cannot be found in memory, the operation collapses to the identity function;[fn:: Putting & for raw indirection and $*$ for raw dereference, as in C, our pointer operators $\&'$ and $*'$ are defined as endomorphisms in $\not\models^{32}$, where $\&'(x) = \&x$ when $(\exists y) * y = \&x$, and $\&'(x) = x$ otherwise. $*'$ is the dual of $\&'$. Unlike the familiar C operators, our pointer mutations therefore have algebraic closure.

4. a permuation operation, by which two randomly selected machine words in a clump exchange places;

5. a combination of 3 and 4, where two randomly selected words in a clump are chosen, and one is replaced with their bitwise sum, the other with their bitwise product;

What the four mutation operations have in common is that they share a certain minmal algebraic structure. Within each set, each operation – which, formally, is a endomorphism over $[\mathbf{2^{32}}]$ – has an inverse:

$$(\forall M \in S)(\forall x)(\exists y)M(x) = y \Rightarrow (\exists M' \in S)M'(y) = x$$

and an identity:

$$(\forall M \in S)(\exists x)M(x) = x$$

What this means is that over each set of mutation operators – and therefore over their union – the concatenation or succession their application forms a cyclic group.[4]

In practical terms, this is a generally beneficial property for genetic operators to possess: all else being equal, they should be designed with an eye towards neutrality with respect to an arbitrary choice of fitness functions. By ensuring that the mutation operators are involutive, or, more generally, that they form a cyclic concatenation

[4]The proof is left as an exercise for the author.

group, involution just being the smallest nontrivial form of such a structure, with a cycle of two, we (imperfectly) guard against a situation where they ratchet the population into a small corner of the genotypic (and, consequently, the phenotypic) landscape, *independent of the fitness function.* (Identity is less significant, in this context, and is introduced into the mutation operators only as a way of ensuring closure.) Ratcheting occurs when the genetic operators are not properly balanced. In the situation where the algebra defined by concatenation over the mutation operators does *not* form a cyclic group – when there is "no way back" from some mutation M by any succession of further mutations – ratcheting is inevitable. This problem is distinct from, but related to, the problem of genetic drift, which it exacerbates. Involutive pairs of operators, selected with equal probability, provide some safeguard against this. The *ideal*, in some sense, would be to select genetic operators that would engender an *ergodic* system under a null fitness function:[5] one whose behaviour is evenly distributed over the probability landscape it inhabits. In practice, even with fitness anulled, evolutionary systems rarely exibit such regularity, which has some very interesting effects on the paths that evolution pursues. We will study some of these consequences in Chapter 5.

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

[5]Thanks to Andrea Shepard for this insight.

each mating event to two. [6]

In single-point crossover between two genotypes, μ and φ, we randomly select a link index μ_i where $\mu_i < |\mu|$, and φ_i where $\varphi_i < |\varphi|$. We then form one child whose first μ_i genes are taken from the beginning of μ, and whose remaining genes are taken from the end of $\varphi_{\mu_i \ldots}$, 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 μ_i and φ_i, using a parameter we call "fragility", whose calculation I explain in Section 4.3.2.1. The details of the algorithm itself can be found in 6.

Crossover, so defined, has certain algebraic properties that allow it to interact harmoniously with the forms of mutation specified in Section 4.3.1. If we abstract away from the information loss enabled by the possibility of dropping of genes with low link fitness, a probabilistic parameter that can be tuned, in ROPER, then, under the assumption of a maximum genome length – an assumption that holds in ROPER, and which is imposed quite naturally by the practical limits of stack space in the target process – the crossover function, too, can be shown to form a cyclic group under concatenation. Let a and b be two chains selected for crossover, and $C(a, b)$ the set of *possible* offspring that may emerge from their mating. If we restrict the splice index of the crossover to be nonzero, then $a, b \notin C(a, b)$. But if we then take the set $\mathbb{C}(C(a, b))$ of all possible offspring resulting from an inbreeding of members of $C(a, b)$, then we *do* find that both a and b appear in this set. There is provable possibility for a chain, identified only by its packed composition and without considering its genealogical metadata, to be its own grandpa.

Crossover is therefore associative, and since the crossover operation is symmetrically defined – one of the two offspring that are *actually* produced by a mating of a and b will begin with clumps from a, the other with clumps from b, with the other parent providing the tail – we can show that the mutation operators defined in Section 4.3.1 commute with and distribute over crossover, if we consider them as functions that take probability distributions of offspring as their codomain.

This is still some distance from rigorously establishing that ROPER's populations

[6] 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: $(\vec{\mu}, \vec{\varphi})$: ($[\![$ **clump** $]\!]$, $[\![$**clump** $]\!]$), the parental genotypes

Require: *Fragility*: $[\![$**clump** $]\!] \times [\![$ **clump** $]\!] \times$ lineage $\to \mathbb{F}$

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

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

$\mathfrak{R} \leftarrow \mathit{seed}(\mathfrak{R}, s)$

splice-points $\leftarrow$ () {the indices at which the parental genes will be recombined}

for $\vec{\alpha} \in (\vec{\mu}, \vec{\varphi})$ **do**

  $t \leftarrow \sum_{\alpha \in \vec{\alpha}} 1.0 - \mathit{Fragilty}(\alpha)$

  $p, \mathfrak{R} \leftarrow \mathit{random\text{-}float}(\mathfrak{R}, t)$ {p is more likely to fall on a highly fragile link}

  $x \leftarrow 0$

  $i \leftarrow 0$

  while $x < p$ **do**

    $x \leftarrow x + \mathit{Fragility}(\alpha_i)$

    $i \leftarrow i + 1$

  end while

  push $(\vec{\alpha}, i)$ onto splice-points

end for

let $\mu^a, \mu^b =$ split μ at splice-points.lookup(μ)

let $\varphi^b, \varphi^a = \mathit{split\varphi at splice\text{-}points.lookup}(\varphi)$

let $\chi^a = \mu^a \cup \varphi^a$

let $\chi^b = \varphi^b \cup \mu^b$

return χ^a, χ^b {the offspring}

will, asymptotically, approximate an ergodic system – a system where any states that are reachable at the outset remain reachable, by an arbitrarily long but finite path of genetic operations, from any point in that system's evolution – but it does at least establish the plausibility of ergodicity. In practice, however, convergence and evolutionary deadends may remain commonplace, and it is only a very slight reassurance to know that such states of affairs are not irreversible or eternal, in principle.

4.3.2.1 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 μ_i and φ_i, instead of letting them be simply uniform. With each 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 μ who shared the same linked pair. Links that have a 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 μ, the link-fitness of each clump $f(\mu_i)$ (implicitly, between each clump and its successor) is calculated on the basis of the fitness of μ, $F(\mu)$:

$$f(\mu_i) = F(\mu)$$

if the prior link fitness $f'(\mu_i)$ of μ_i is `None`, and

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

otherwise. The prior link-fitness value $f'(\mu_i)$ is inherited from the parent from which the child receives the link in question. If the child μ receives its i^{th} clump from one parent and its $(i+1)^{th}$ clump from another, or if i is the final clump in the chain, then $f'(\mu_i)$ 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-liable 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.[7]

Selection, however, does not directly operate on genotypes but *phenotypes*. In the context of genetic programming, "phenotype" is the name given to the semantic

[7]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.

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 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 suprise: 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 at 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 precribe 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* resurfaces here in full force: the odds of a randomly generated chain of gadgets executing without crashing is extremely small – under 5%, on average, at the beginning of a run. 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 form 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. The 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. 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, he 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.[8] 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 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 [11], and *implicit*

[8]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."

fitness sharing, introduced in [29].

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 [23]. "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` vector, which begins empty. Every time an individual responds to the problem, its (its fitness assessment for that particular problem) score is pushed into its `predifficulty` vector.

3. after a one "season" of tournaments has elapsed, where the length N of a season is defined as
$$N \leftarrow \frac{\texttt{population\_size}}{\texttt{tournament\_size} * (1 - x)}$$
where 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 the mean of the predifficulty vector. More difficult problems, at this point, are associated with a higher difficulty score, which is always a float between 0.0 and 1.0.

4. once difficulty scores are available for each problem, the relative fitness of each creature responding to it can be assessed: it is just the base, or "absolute", fitness score, multiplied by the inverse of the difficulty. If, for instance, X receives 0.75 on a problem for which the average performance has been 0.1, then X's relative fitness is $0.75 * (1.0 - 0.1) = 0.75 * 0.90 = 0.675$. If, on the other hand, it receives 0.2 on a problem for which the average performance is a miserable 0.98, then its relative fitness comes to 0.004, reflecting the rarity of its talents.

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 $\vec{i}$ 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^{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.

4.5 Remarks on Implementation

The system described above has been implemented using the Rust programming language, and the Unicorn emulation engine [27] [26].

Rust was chosen for its speed, type-safety, and functional niceties, though this decision wasn't entirely unarbitrary – prototypes of the system in Lisp, Haskell, and

Algorithm 7 Headless Chicken Patch

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

Require: $\mathbf{G}, \mathbf{P}, \min, \max$: 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** $i <$ headless_chicken_rate **then**
3: $\quad \mathfrak{R}, \text{candidates} \leftarrow$ using $\mathfrak{R}$, pick $n - 1$ from Π
4: $\quad \mathfrak{R}, \text{candidates} \leftarrow \text{candidates} \cup \textit{spawn}(\mathfrak{R}, \mathbf{G}, \mathbf{P}, \texttt{min}, \texttt{max})$ {Using Algorithm 5}
5: **else**
6: $\quad \mathfrak{R}$, candidates $\leftarrow$ using $\mathfrak{R}$, pick n from Π {As before}
7: **end if**

OCaml are still strewn about my hard drives and Git repositories in various states of incompleteness. The decision to make use of the Unicorn framework remained somewhat more constant. Evaluating arbitrary ROP chains on bare metal turned out to be every bit as hazardous and messy as it sounds, and so the need to find a suitable virtualization framework became apparent very early in the project. Spinning up full-fledged Quick Emulator (QEMU) virtual machines (VMs) for each evaluation – or even for each evaluation that ended in a fatal system state – would bring with it a prohibitive amount of overhead. I needed something that would let me evaluate thousands upon thousands of individuals within a reasonable timeframe.

Unicorn, which its authors describe as, "a lightweight multi-platform, multi-architecture CPU emulator framework", exposes the CPU emulation logic of QEMU, while abstracting away from input/output (I/O) devices and any interface with the operating system, along with all their associated overhead. The machine state of the emulator remains transparent, and is easily instrumented by the user. This makes Unicorn ideal for performing a fine-grained semantic evaluation of ROP chains, under the assumption of a given CPU context. The evaluation, that is to say, is strictly "concrete" – it will tell us only how a given chain will behave, *assuming that the CPU context and memory space is in such and such a state*. This can be seen as an limitation of ROPER, as compared to procedurally deterministic but symbolically indeterministic ROP compilers like *Q* [28], which makes use of symbolic execution (via the Binary Analysis Platform (BAP)) to precisely determine the semantic valence of each

available gadget so as to *explicitly* fashion them into the instruction set architecture (ISA) targetted by Q's own compiler. What ROPER's procedurally stochastic and semantically concrete (i.e. "deterministic") approach loses in semantic precision and robustness, however, is made up for with a singular cunning when it comes to exploiting the *particular*, concrete state of its host process. Its task, after all, is not to craft a portable, reusable ROP payload that can be cut-and-pasted, off-the-shelf into arbitrary attack contexts, but to craft payloads that are as idiosyncratically adapted to the peculiarities of its chosen target as a moth to its orchid.

4.5.1 Initialization of the environment

In the discussion of the algorithmic specification of ROPER, above, we have, for the most part, abstracted away from the environment in which the evolutionary process occurs. Setting up this environment is the first task of the engine, and it proceeds as follows.

It begins by parsing and analysing the target binary – either a standalone executable, or a statically compiled library file. ROPER is currently only prepared to handle ELF binaries targetting 32-bit, little-endian ARM architectures, though there's no *essential* reason for any of these restrictions, and the system could be fairly easily adapted to handle other formats (such as Apple's Mach object file format (Mach-O) or Window's Portable Executable (PE) formats), or other architectures (some tenative work on adapting ROPER to is already underway, and it turns out to be fairly straightforward to take on other reduced instruction set computer (RISC) ISAs; complex instruction set computer (CISC) ISAs pose a few more challenges, but tend to make for extremely fertile gadget sets – the instruction set is so vast and intricate, for example, that it can be a challenge to find a string of bytes that *can't* be parsed as a series of machine instructions!).

ROPER reads the ELF program headers and loads the program data into the memory of a cluster of Unicorn emulator instances, at the appropriate addresses, and with the appropriate permissions, just as the Linux kernel would do when launching the executable on the metal. While doing this, ROPER (optionally) can ensure that $w \oplus x$ is enforced, even if not strictly required by the binary.

Since ROPER, like any genetic programming system, relies heavily on randomness,

a word or two about its is in order. The pseudo-random number generator (PRNG) used is supplied by Rust's default `std::rand::thread_rng` function, which, as of version 0.5 of the `rand` library, rests on an implementation of the cryptographically secure the HC-128 stream cipher (HC-128) algorithm [32], seeded on a per-thread basis by the operating system's entropy pool. In the current implementation, the seeds passed to this generator are not logged, and cannot be manually specified by the user, which makes the exact replication of a run impossible. I hope to address this shortcoming in a future overhaul of the codebase. The salient point about the PRNG, for now, is that it is of fairly high quality, and *should* not be vulnerable to being exploited by the populations.

Dr. Ian Malcolm: John, the kind of control you're attempting simply is... it's not possible. If there is one thing the history of evolution has taught us it's that life will not be contained. Life breaks free, it expands to new territories and crashes through barriers, painfully, maybe even dangerously, but, uh... well, there it is.
John Hammond: [sardonically] There it is.
Henry Wu: You're implying that a group composed entirely of female animals will... breed?
Dr. Ian Malcolm: No. I'm, I'm simply saying that life, uh... finds a way.

Michael Crichton, *Jurassic Park*

5

Experimental Studies

5.1 Overview

As of the time of writing, I have experimented with four distinct classes of fitness functions in ROPER, with a handful of variations within each class. Though ROPER has been tested with numerous executable ELF binaries, compiled for the 32-bit ARM architecture, for the sake of consistency, unless otherwise noted, all of the experiments discussed here make use of a web server binary blob, pulled from the `tomato-RT-N18U` router firmware image [1]. The distribution of gadgets harvested from this binary's `.text` section are plotted in figure 5.1, to which we will make frequent reference, in our heatmap overlay images.

In the following subsections (5.1.1, 5.1.2, 5.1.3, and 5.1.4), I will outline each task set that ROPER was given to perform, and specify the way in which each was incorporated into ROPER's fitness function.

In Section , I will walk through some of the more interesting results gleaned from each experiment, and in Section , I will lay out certain conjectures that address an

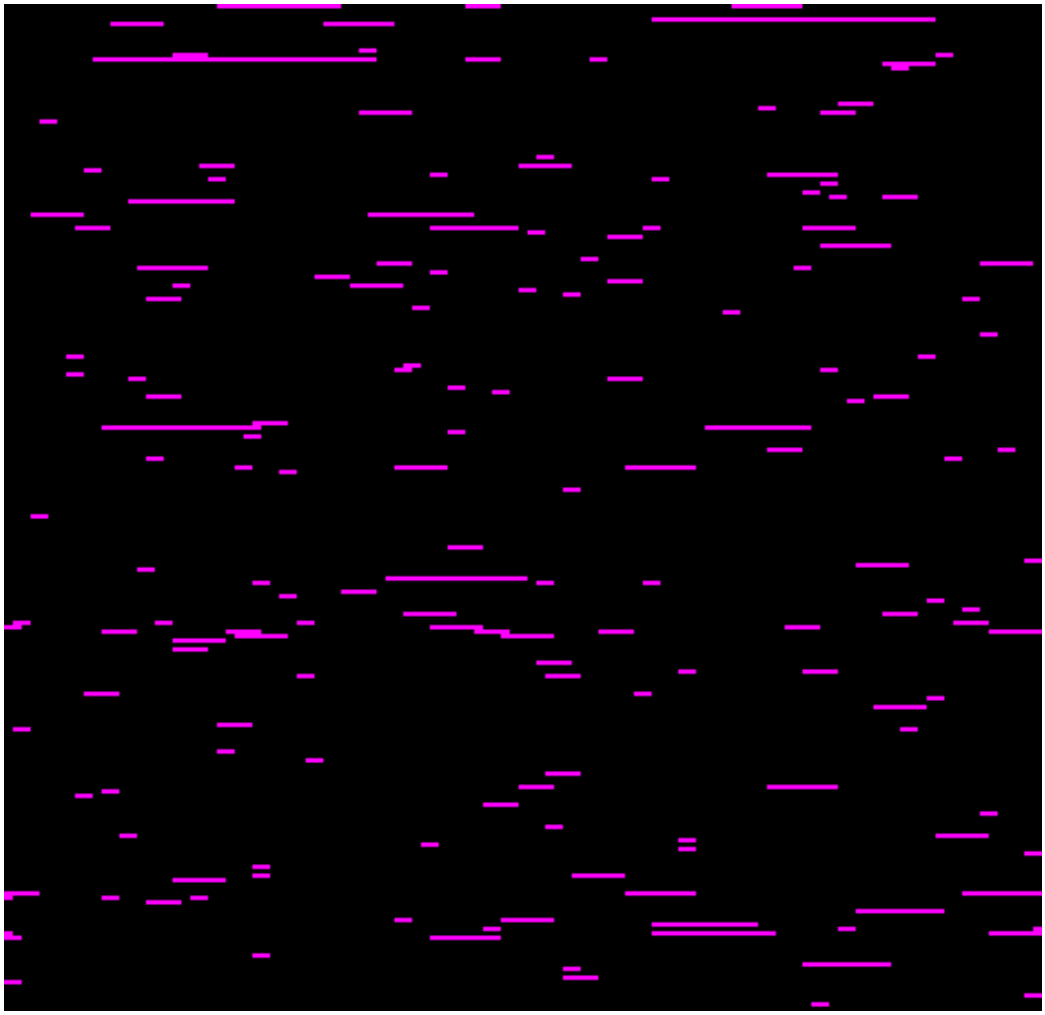

Figure 5.1: Bitmap representation of the gadget distribution in the `tomato-RT-N18U-httpd` ELF binary

enigma in the observed results, and set out another series of experiments to test it.

5.1.1 The null task

As discussed in Sections 4.3.1 and 4.3.2, when designing a genetic system, a question that naturally arises as to what constraints are brought into play by the shape of the system itself, in its genotypic and phenotypic topography, independent of any particular fitness function that could be applied to it. In the case of simple and unilateral genetic algorithm systems – where there is no distinction between phenotype and genotype (the genetic syntax of individuals and their operational semantics), it is possible for the system to be strongly *ergodic*. Under less harmonious conditions, however, there may exist various attractors in the genotypic and phenotypic landscapes, and in their interaction, that incline the system to converge in ways that are relatively independent of any specific fitness function.

One way to empirically study these dynamics is to define an effectively uninformative fitness function, one that metes out rewards arbitrarily and inscrutably, without any detectable pattern. A cryptographically secure random number generator is well suited to this particular sort of absurdity, and this is how ROPER implements its null task.[1]

And since, in each of the other experiments, I enforce some form of crash penalty as a component of the fitness function (see Section 4.4.3 for details), it would make sense to study what effects this pressure alone, abstracted from the others, has on the population dynamics. For this reason, I will introduce a second series of null task runs, in which the only non-random variable used in reproductive selection will be the crash penalty.

5.1.2 Preparing the parameters for a system call

This task is the most immediately practical of the set, and comes the closest to a practical, "real world" application of ROPER in the domain of application security. One of the most common use cases for a ROP chain is to prepare the CPU context for a particular system call, permitting the attacker to read or write to a file,

[1] In the current implementation, this task can be chosen by setting the problem type to `kafka`, in the configuration.

open a socket, execute a program, or any other task that requires the cooperation of the operating system. To do this in assembly or machine code, the programmer needs to set certain registers to contain and point to the relevant values. To call `execv("/bin/sh", ["/bin/sh"], NULL)`, for instance, and spawn a shell process, on the ARM architecture, she needs to set register `R0` to point to the null-terminated string `"/bin/sh\0"`, register `R1` to point to a pointer to that string, set `R2` to zero, and set `R7` to the code for this particular system call. Once this is done, dispatching the call is just a matter of executing the `SVC` instruction. To perform this operation with a ROP chain, the same requirements must be met, but in a more roundabout fashion, since the ROP hacker isn't able to execute any instructions directly, but must conduct the processor to execute a series of gadgets whose cumulative effect is to prepare just this machine state. The final gadget in the chain, however, is trivial: it is just the address of an `SVC` instruction, followed, perhaps, by whatever series of gadgets is necessary to clean up the process and restore the normal flow of execution, if stealth is desired. (Though it's entirely possible, in most cases, to simply let the host process crash after performing the desired call – it's just a bit sloppier, and can bring about other problems in the context of a real-world attack.)

The task we assign ROPER is to carry out the preparation stage of this operation, but it can easily be extended to complete the call – it is, after all, just a matter of appending a single, trivial gadget to the chain, which can almost always be found in binary, since the `SVC` call is perfectly generic and does not embed any of its arguments in the instruction itself. This step is omitted in our experiments only because Unicorn abstracts away from any interaction with the operating system, and doesn't handle system calls in a meaningful way. The cleanup stage is more context-sensitive and complex, but something I may experiment with in the future, in order to prepare fully deployable payloads with ROPER.

The fitness function defined for this task aims to be a gauge of the distance between the register and memory state resulting from an individual chain's execution, and the state required for the call. "Distance", here, however, is a tricky concept, and it isn't at all obvious how it should be defined in such as way as to both track material progress towards the goal in view, and remain efficiently calculable. Ideally, it would be defined as edit distance in the Markov chain that represents a genotype's trajectory through

probability space: how many generations, or applications of the genetic operators, are required to achieve the target state, and how probable are each of those genetic pathways? In practice, we make do with a very loose approximation of this ideal:

- For each target immediate value, we first check the execution result in the target register, and take the hamming distance between the two values (the number of 1s in target $\oplus$ result), and divide it by the maximum hamming distance (32, on this architecture), storing the quotient in the variable `nearest`;

- If `nearest` $\neq 0$, we iterate through the remaining result registers, and calculate the hamming quotient with the target for each value found there. We then apply a "wrong register" penalty to the quotient, and if the result is still less than `nearest`, we rebind `nearest` to the result.

- We then pass to our memory samples from the execution – one for each register that could be validly referenced in the machine state at the end of execution, each holding n bytes (currently, n is set to 512, but this may undergo tuning) from the engine's memory, starting with the address in the register. We scan the memory sample for the desired value, and, if found, we return the quotient of the value's offset by the length of the sample (e.g., if we find it at offset 256, then we return $\frac{256}{512} = 0.5$). If the index of the register dereferenced to that memory sample is the same as the target's, then we consider this quotient as a new candidate for `nearest`, after applying a "right value, but needs to be dereferenced" penalty. If there is a mismatch in the register index, we also apply the "wrong register" penalty, and do the same.

- After completing these iterations for a particular target register, we push the final setting of `nearest` into an error vector, to be considered later.

- `nearest` is then reset to the maximum value of 1.0, and we repeat the process for the next immediate target register.

- We then move on to the indirect targets in our target vector, and repeat more or less the same process, first scanning the memory samples returned from the individual's execution in the emulator, recording any findings, and then passing

on to consider the immediate values in the register, where we calculate the hamming quotients, applying the same penalties as above wherever there occur mismatches between target and resultant register index, or mismatches between desired indirection and resultant immediacy. At the end of each check, the final value of `nearest` is pushed to the error vector.

- Finally, we take the mean of the values in the error vector, and return it as a float between 0.0 and 1.0: this is the fitness value for that particular evaluation.

The function used for the "wrong register" and "needs to be dereferenced or indirected" penalties was, like most details in ROPER, arrived at through a great deal of trial and error, and at the time of writing has settled into $1 \sqcap \sqrt{x + 0.1}$, which seems to generate a reasonable amount of pressure while still maintaining a traversable gradient in the fitness space. Restricting the primary distance measure to hamming distance and forward linear scans feels like a fairly crude approximation, but seem serviceable enough for now.

5.1.3 Classification problems

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 series of experiments, 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, first, a toy data set, designed for simplicity, and, second, Ronald Fisher and Edgar Anderson's famous *Iris* data set.

The Iris data set is a well-worn benchmark for training elementary machine learning systems,[2] and to the machine learning specialist, there is nothing particularly interesting about yet another classifier churning out results for such a relatively unchallenging set. But, with these experiments, we enter essentially uncharted waters as far as return-oriented programming – or even, to the best of my knowledge, any form of low-level "weird machine" exploitation – is concerned. There is no real precedent for having anything like a ROP payload implement even a basic and rudimentary machine learning benchmark, and so this task is introduced here entirely as a proof of concept.[3] The interest, here, isn't in building a better mousetrap. It's in showing that one can be built – or bred – out of utterly alien materials.

The Iris data set comprises series of four measurements – sepal length, sepal width, petal length, and petal width – for 50 specimens belonging to three different species of iris flower, 150 specimens in all. The task of the classifier is to predict the species when given the measurements. "One class is linearly separable from the other 2," the documentation on the UCI Machine Learning Repository advises, "the latter are NOT linearly separable from each other." The fitness of each competing ROPER individual, in this context, will depend on its ability to correctly classify the sample of specimens presented to it. Since this is a balanced data set, detection rate alone is a sufficient proxy for accuracy. The four attributes of each specimen will be cast and normalized as integers, and, for each evaluation case, loaded into four of the Unicorn CPU's general purpose registers. The candidate chain is then executed, as described above, and at the end of execution we read the values contained in three distinct registers, designated as "output registers" for this purpose. Each register represents a "bid" for one of the three possible iris species. The values they contain are cast as signed integers, and the register containing the largest value is interpreted as a winning bid for its corresponding species.

As we will see in a moment, experimentation has shown that an accurate and, if

[2]"This is perhaps the best known database to be found in the pattern recognition literature," reads the data set information note at [12].

[3]If you're interested in developing an intelligent classifier, you're unlikely to consider doing so using the unweildy scraps of hijacked process's memory, and if you're interested in crafting a low-level attack payload, a reverse shell probably seems like a more sensible goal than a moderately clever flower sorter – unless, of course, what you're really after in either field are ways of making machines do strange, strange things.

not altogether efficient, at least timely[4] classification of the set can be achieved by ROPER, so long as the fitness sharing mechanism discussed in 4.4.4 is made available to it.

The base fitness function used in the classification experiments is based on register-bids: a subset of general purpose machine registers are designated as "output registers", and another, as "input registers". At the beginning of each evaluation case (each exemplar in the training set), the input registers are loaded with values from the attribute fields. At the end of execution, data is read from the output registers. Each output register is interpreted as a signed integer, and taken to represent a bid on a class. The register with the highest bid decides the individual's "guess" at which class the attributes passed to it represent. In the event of a tie or an incorrect guess, no points are awarded (the fitness value of this case is set to 1.0, which is the poorest rating). In the event of a correct guess, 0.0 points are awarded. That point value is then duplicated into a pair. The first remains untouched, and is factored into the detection rate or "absolute fitness" of the chain. The second is passed to subsequent modifications. If the chain threw an exception during execution, for instance, then the score is raised by a certain factor (which is tunable, and in some experiments, dynamically variable in response to population trends), but it is constrained to remain between 1.0 (worst fitness) and 0.0 (best fitness) at every point. If fitness sharing is in effect, then the modifiable component of the score has the inverse of the exemplar's difficulty added onto it. (If the chain scored a 0.0 on an exemplar for which the mean performance is 0.85, for instance, then it receives an additive boost of 0.15.)

5.1.4 "Would you like to play a game?"

Naturally, the first question that comes up when you find a way of eliciting rudimentary, intelligent behaviour from an exploited process is whether it can play any games. This brings us to our fourth fitness function: we want to train ROPER to evolve ROP chains that can play a game of Snake.[5] For this purpose, I wrote a simple, machine-friendly implementation of the classic arcade game in Lisp,[6] and set it up to communicate with ROPER over a TCP connection. ROPER's task, here, is

[4]Meaning: you will receive the results before you die.

[5]I'm reserving tic-tac-toe for emergencies.

[6]The code is freely available at `https://github.com/oblivia-simplex/snek`.

to evolve a ROP chain that, when run in a loop, can play a more or less competent game of Snake. The ROP chain is fed an array of first-person sensor readings, from the snake's perspective, which indicate the relative distances of objects in the playing field: *apples*, which increase the snake's length, *cacti*, which kill the snake on contact, segments of the snake's own body, which are similarly dangerous to collide with, and the walls of the field, which, again, result in death on contact. Points are awarded to the player relative to the number of unique grid coordinates visited, and the number of apples consumed, before dying.

5.2 A few notes on terminology

5.2.1 Iteration, generation, and season

ROPER employs a steady-state selection model in all of the experiments discussed in this thesis, and so it's important to distinguish what we mean by *generation* from what we mean by *iteration*. The notion of "iteration", here, is for the most part external to the actual evolutionary process – it's an artefact of implementation, in a sense, representing only a cycle through the "main loop" of the program, during which one or more tournements are thrown in parallel. (I say "artefact of implementation" with a very large grain of salt. If there are any fields of study that demand a healthy skepticism towards what counts as essential to the specification, and what counts as "mere implementation details", they're computer security and evolutionary computation!) An "iteration", in this context, becomes a meaningful evolutionary unit (modulo abstraction leaks) through a mediating notion that, in ROPER, I've called a "season": a season is defined as a sequence of iterations that is just long enough for it to become highly probable that more or less every member of the population has had their shot at a tournement (and, therefore, a shot at reproduction). In concrete terms, the length of a "season" is a function of tournement size and population size.

5.2.2 Naming scheme for populations

Mentally keeping track of numerous, varying artificial populations is a challenge at the best of times, and I've found that assigning short, pronounceable names to each batch is a useful mnemonic device. For this purpose, I've borrowed a trick from

the designers of *Urbit*, and used strings of six random letters, following the pattern "`consonant vowel consonant consonant vowel consonant`"[7]. Using these labels over the course of this chapter lets me avoid circumlocutions like "as seen in the second population discussed in Section $N", and has the further advantage of setting up a self-documenting correspondence between the analyses in this chapter and my system of log files.

5.3 Initial Findings

5.3.1 Surveying the landscape with the null task

Before looking at the behaviour of ROPER under the influence of and constraints of the semantically nontrivial fitness functions described above, let us first try to get a sense of how the system behaves *without* receiving any meaningful fitness information, and then move on to a constraint that is incorporated into each of the fitness function considers, and acts as their common factor: the one imposed by the crash penalty.

5.3.1.1 Arbitrary selection with no crash penalty

The first thing that we wish to learn about how our populations behave under randomized selection pressure is whether there are any discernible trends in the distribution of behaviours in those populations over time. One useful view on this distribution is the heatmap of addresses visited over the course of the run, sampled seasonally. Though not a precise or statistically significant representation of behaviour, it does, at least, serve to convey a general impression, at let the eye pick out the more obvious skews in distribution.

In figure 5.5, we see heatmaps of the the *xeqcyv* population's phenotypic distribution sampled at seasons 4 and 212. Qualitatively speaking, the two maps appear almost identical. There has been no collapse of phenotypic diversity, and the system even gives the impression of being ergodic. Acting freely, with no systematic selective constraints, the mutation operators appear to be successfully counteracting the pull of genetic drift.

[7] This superficially resembles the namespace that Yarvin and Wolff-Pauly allocate to "stars" in *Urbit*, but is a bit less constrained: their namespace has room for $2^{16} - 1$ names, ours accommodates $(20 * 6 * 20)^2 \approx 2^{11.2}$.

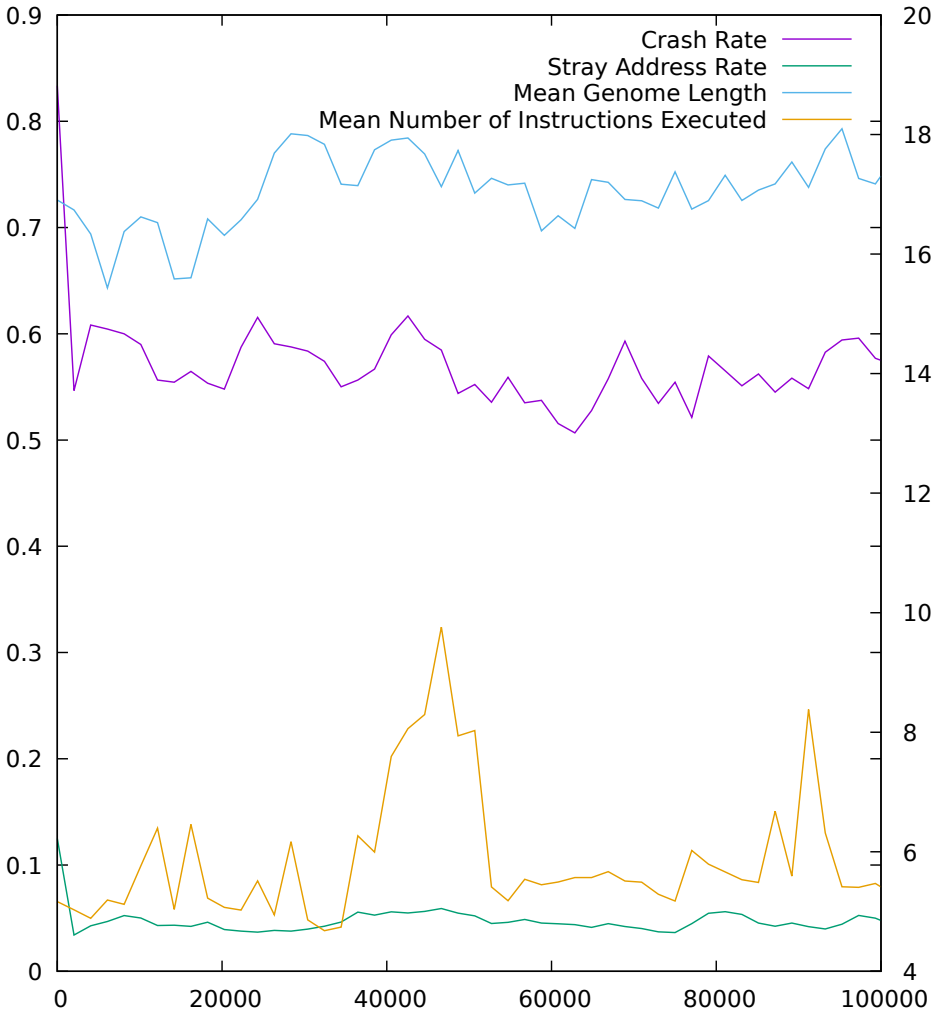

Figure 5.2: Performance metrics over *kurlig* population of 2048, evolving under the absurd fitness function. Crash rate and stray address rate map to left vertical axis, while mean genome length and mean instructions executed map to the right.

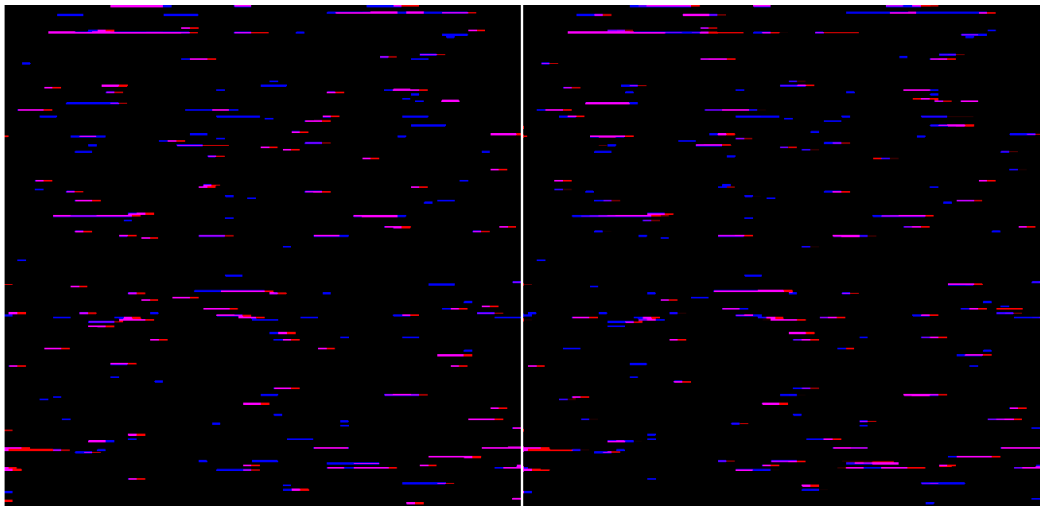

Figure 5.3: Address visitation heatmap over the `tomato-RT-N18U` process memory, by the *kurlig* population, evolving under the absurd fitness function with no crash penalty. Season 4 heatmap displayed on the left, season 212 on the right.

5.3.1.2 Arbitrary selection with crash penalty

Surprisingly, even with the crash penalty in place, the distribution of execution path frequency remained strikingly consistent, at least to a distant and qualitative inspection. The heatmap snapshots of the population's execution habits at Season 4, and again at Season 212, showed such little difference that I had to check carefully to make sure that I had not placed them in the wrong order, in figure 5.5.

The tendency of these populations to shrink from execution, as shown in figure 5.4, is not at all surprising. The longer they spend in execution, the greater their risk of crashing. Since crashing the the one *sure* way to draw selection's wrath, the population is highly disincentivized to spend any longer than they have to in execution. There is simply nothing for an individual under such selective pressures – and only those – to gain by running code on the CPU. There is no "task" to be completed in this setup, after all. Computation, for the creatures of *xeqcyv* and *kurlig*, it essentially vestigal, and rapidly atrophies. And so execution times contracted to as little as 4 instructions on average, with much less variation than seen in the *kurlig* population.

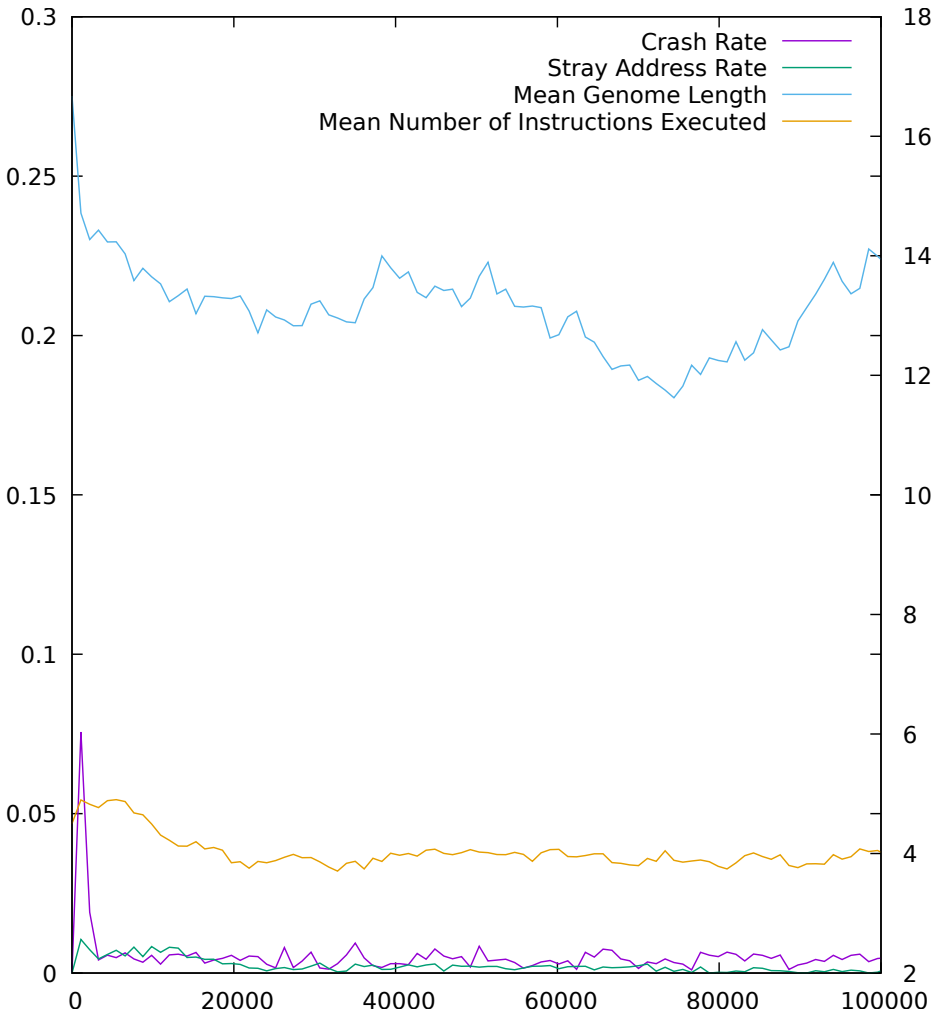

Figure 5.4: Performance metrics over *xeqcyv* population, evolving under absurd fitness function with crash penalty. Crash rate and stray address rate map to the left vertical axis, while mean genome length and mean instructions executed map to the right.

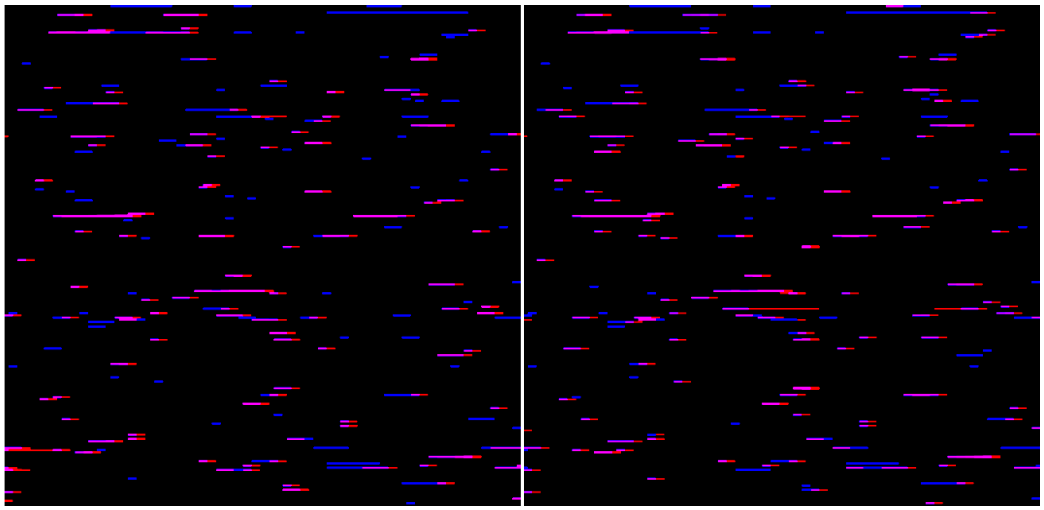

Figure 5.5: Address-visitation heatmap over the `tomato-RT-N18U-httpd` process memory, by the *xeqcyv* population, under the "Kafkaesque" fitness function: Season 4 on the left, Season 212 on the right. Blue swaths indicate where the harvested gadgets lie, red and magenta swaths indicate addresses visited by creatures in the population in execution.

5.3.2 Preparing an execv() system call in the Tomato web server

In the experiments discussed here, we targetted the Hypertext Transfer Protocol (HTTP) daemon extracted from the firmware binary of a popular, open-sourced ARM router, with the filename `tomato-RT-N18U-httpd`, and focussed on preparing a call to `execv()`. This binary has the fortunate (for us) property of not only containing a considerable number of gadgets, but several hardcoded path strings as well. `#!/bin/sh` sits comfortably at address `0x0001bc3e`, for example – though it is terminated by a newline (`0x0a`), and not by a null byte. Another interesting string that we can find there, residing at address `0x0001f62f`, is `"/tmp/flashXXXXX"` – just a few hundred bytes below an apparent error message, reading `"Unable to start flash program"`. It appears that the `XXXXXX` suffix is overwritten at runtime, at some point, yielding a valid path to an executable, which, presumably, reflashes the router and restores factory settings[8]. And, just as we'd expect, the pathname, along with `/bin/sh`, resides in *writeable* memory. The resources for exploitation are numerous here, and the situation is ripe for ROPER's exploration.

[8]This conjecture warrants a bit more reversing than I've carried out so far, but in the meantime this works as a plausible scenario, which is all we need for this PoC of ROPER.

To get our proof of concept underway, we'll set out CPU state pattern string, which ROPER will parse and use to parameterize its fitness function, to

```
0001f62f,&0001f62f,00000000,_,_,_,_,0000000b
```

Each comma-separated cell in the string represents a register. The commas indicate registers that we don't care about for the purpose of this task, and they will be ignored by the fitness function. The ampersand is the indirection operator, as in C, and it tells ROPER that `R1` should contain a *pointer* to the address of the string, "/tmp/flashXXXX", which, for its part, is known to reside at `0x1f62f`.

A satisfactory CPU pattern was soon produced by an individual in the *wiwzuh* population, seventeen genealogical steps from the initial population. This specimen isn't particularly complex, once you factor out a few of the detours it takes. The It essentially works by just popping the necessary values from the stack to the registers – all evolution needed to do here was to find the values that hadn't been provided in the initial integer pool (such as the pointer 0xfff8 to the value 0x1f62f) and to place them in the necessary order. All that's needed in order to actually launch this syscall is to place a single `SVC` instruction pointer at the end of the packed chain shown in figure 5.1. The Tomato binary has 859 to choose from, and the choice is more or less arbitrary.

Even with so simple a task, the fitness landscape traversed by ROPER displays a surprising degree of ruggedness, as we can see in figure 5.6.

```
Clumps:
[*] 0000b4ac   00000000   00000b17   0000000b   0000000b   00000000
[*] 0000d1a0   0001f62f   0001f62f   0001f62f
[*] 00016654   706d742f
[*] 0001706c   0001f62f   0000fff8   0001f62f

Packed:
ac b4 00 00  00 00 00 00  17 0b 00 00  0b 00 00 00
0b 00 00 00  00 00 00 00  a0 d1 00 00  2f f6 01 00
2f f6 01 00  2f f6 01 00  54 66 01 00  2f 74 6d 70
6c 70 01 00  2f f6 01 00  f8 ff 00 00  2f f6 01 00

Execution Trace:
0000b4ac        pop {r4, r5, r6, r7, r8, pc}
```

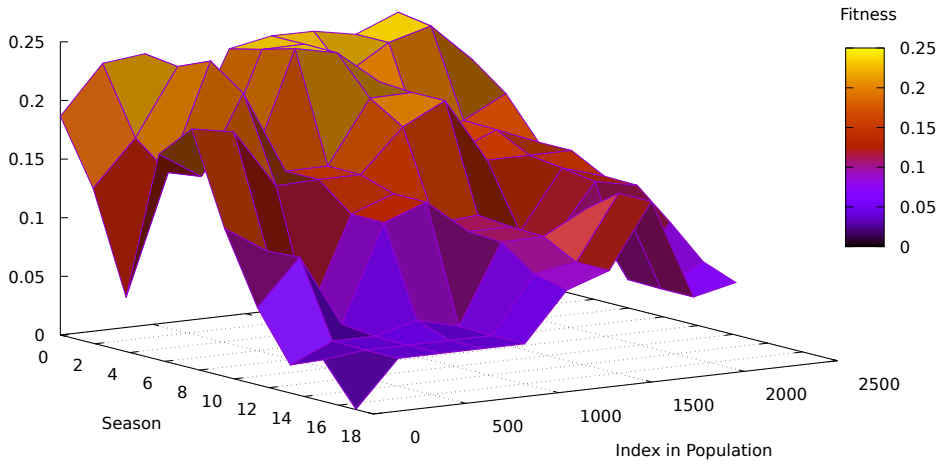

Figure 5.6: Gaussian approximation of the fitness landscape across the population and over evolutionary time, for the *wiwzuh* population, evolving `execv("/tmp/flashXXXXXX", ["/tmp/flashXXXXXX"], NULL)` syscall ROP chains in `tomato-RT-N18U-httpd`, yielding a specimen with perfect (0.0) fitness.

```
0000d1a0        cmp r0, #0
0000d1a4        popeq {r3, r4, r5, pc}

00016654        cmp r0, #0
00016658        ldr r3, [pc, #4]
0001665c        moveq r0, r3
00016660        pop {r3, pc}

0001706c        ldm sp, {r0, r1}
00017070        add sp, sp, #0x10
00017074        pop {r4, r5, r6, pc}

Registers:
R0:  1f62f->706d742f  R8:  0->68732e00
R1:  fff8->1f62f      R9:  0->68732e00
R2:  0->68732e00      R10: 0->68732e00
R3:  706d742f         R11: 0->68732e00
R4:  0->68732e00      FP:  0->68732e00
```

```
R5:  0->68732e00      SP:  2b053->0
R6:  0->68732e00      LR:  0->68732e00
R7:  b->746e692e      PC:  0->68732e00
```

Listing 5.1: A perfected execv() payload on `tomato-RT-N18U-httpd`, from the *wiwzuh* population, generation 17.

I must make a correction to a report stemming from this series of experiments that I had previously published and presented ([15], [14]). I had initially set the target pattern for these attacks to `02bc3e,02bc3e,0,_,_,_,_,0b`, where I had identified `02bc3e` as the address of the string `"/bin/sh"` in the httpd binary. This was inaccurate on three counts: first, the address was off by 0x10000 bytes, as a more careful inspection of the program headers – and subsequent improvement of ROPER's ELF loader – revealed. Second, that string is not null-terminated, and so we'd have trouble passing it as-is to `execv()` – though it is newline-terminated, sitting at the beginning of an embedded shell script, and residing in writeable memory, so it's still likely of some use, with a bit of manoeuvering. Most seriously, though, was a rookie mistake I had made in relaying `execv()`'s signature: the second parameter requires another degree of indirection, and so this particular payload would have immediately resulting in a segmentation fault, if deployed in a real scenario. These errors all stand corrected in the current document, and ROPER's handling of pointer indirection and dereference has been vastly improved, along with its ELF loader.

Some good nevertheless came from that oversight, as it called to my attention an extraordinarily interesting and counterintuitive pattern in the behaviour of many of ROPER's strongest specimens, which was thrown in sharp relief by the champion chain of that particular run, system call bugs notwithstanding. The specimen that caught my interest is reproduced here, in figure **??**.

5.3.3 Results of the classification problem

In what follows I describe some of the findings arrived at by experimenting with assigning classification tasks to ROPER. Two different datasets are considered: an artificially simple and linearly separable dataset consisting of two classes, and the well-known "iris dataset" by Fisher. But first, a few remarks on the character of the problems we're looking at seem worth making.

5.3.3.1 Particular challenges imposed by ROPER's phenotypic landscape

Studying ROPER's handling of simple classification problems, where the odds of success by any standard classification algorithm can be more or less anticipated, throws into relief certain challenges that are peculiar to ROPER, owing to the unusual terrain of the phenotypic landscape that it is forced to traverse. The routes that it tends to carve out between problem and solution are often extremely indirect and circuitous, due to pressures that are external to the shape of the problem space, as specified in the data set, for example, but internal to ROPER's operational semantics.

To begin with, of course, there is the constant threat of segfaulting, which is never a concern for genetic programming systems that have been designed, from the ground up, with their intended purpose in mind (assuming there are no bugs in the implementation). ROPER must not only find a solution to each problem in the problem set, but do so without throwing the CPU into an exceptional state. At the beginning of a run, *most* execution paths lead to a crash, as may be expected when executing randomly assembled ROP chains, on which very little prior sanitization has been performed. A great deal of evolutionary time appears to consist in winnowing useful components out of their unviable, crash-prone encasings.[9]

Furthermore, even without considering the probability skews caused by genetic drift, there is an unevenness in the space of possible outcomes – represented, for example, by register states – that is imposed by the materials that the host binary makes available. Consider, for instance, the distribution in the frequency of register usage shown in figure 5.7. If the output registers used for the bid-based classification function are, say, `R0`, `R1`, and `R2`, then, all else being equal, we can expect to see the class designated by `R0` receive more attention, until the population calibrates itself.[10]

At the present time, no explicit adjustment is made to the fitness functions to account for this uneveness, though the *difficulty* mechanism and its use in fitness sharing

[9]This is a conjecture based on indirect evidence, such as the frequency with which the "minimum fitness" measurement (the best performer, crashing aside) outpaces the "best fitness" measurement (the best, non-crashing performer), but more fine-grained collection of genealogical data, and further experimentation, is necessary for us to be sure that this is, indeed, a common tendency.

[10]At some point, this needs to ascertained through experimentation. For that, we'll need finer-grained information on the classification confusion matrices than the current version of ROPER provides (since the focus has been on balanced data sets, the detection rate data has remained fairly coarse grained), but this should be simple enough to implement.

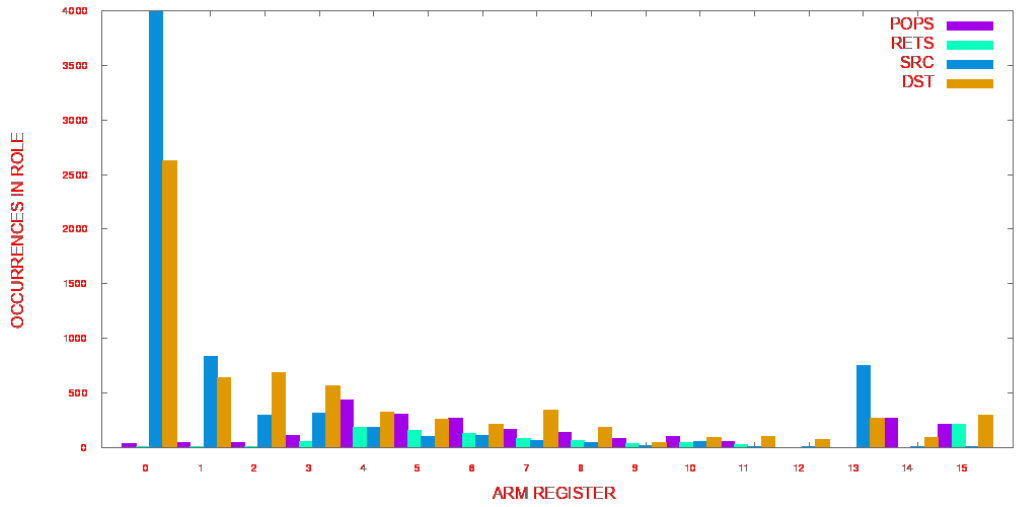

Figure 5.7: Register use histogram for the `tomato-RT-N18U-httpd` ARM ELF binary, used in many of the experiments documented here. The general shape of this distribution is representative of all of the compiled ARM ELFs I have looked at. The height of each bar represents the frequency with which the register, indexed on the X-axis, appears in pop-type instructions, in returns, and in the source and destination of data manipulation instructions.

(see Section 4.4.4) provides a means for the evolutionary process to, eventually, gain some traction on it.

These qualities should be borne in mind when assessing ROPER's performance on the following classification tasks. Its performance is in no way impressive, when compared to any standard classifier implementation, and these results are not intended to showcase ROPER's merits *as* a classifier. They're put forward as a proof of concept, demonstrating, by construction, that anything as strange and improbable as *training a population of ROP chains to recognize patterns in data* is possible.

5.3.3.2 Classification of a simple, linearly separable dataset

As a way of establishing the minimal feasibility of performing classification tasks with ROPER, I tested the system with an extremely simple, linearly separable dataset that I had generated artificially, with just two attributes and two classes (see figure 5.8).

ROPER had very little difficulty with this problem set, and was able to generate fairly good classifications of the data in the initial trials (figure 5.9). Its best specimen clearly relied on a single attribute in forming their classifications – basing them, it

appears, on the sign of the x parameter – but it seems likely that some fine-tuning of the (currently very naive) fitness function should be able to encourage a subtler approach. A fine-tuning of the fitness sharing parameters would perhaps be sufficient to draw selective pressure's attention to the two blue points misclassified as red.

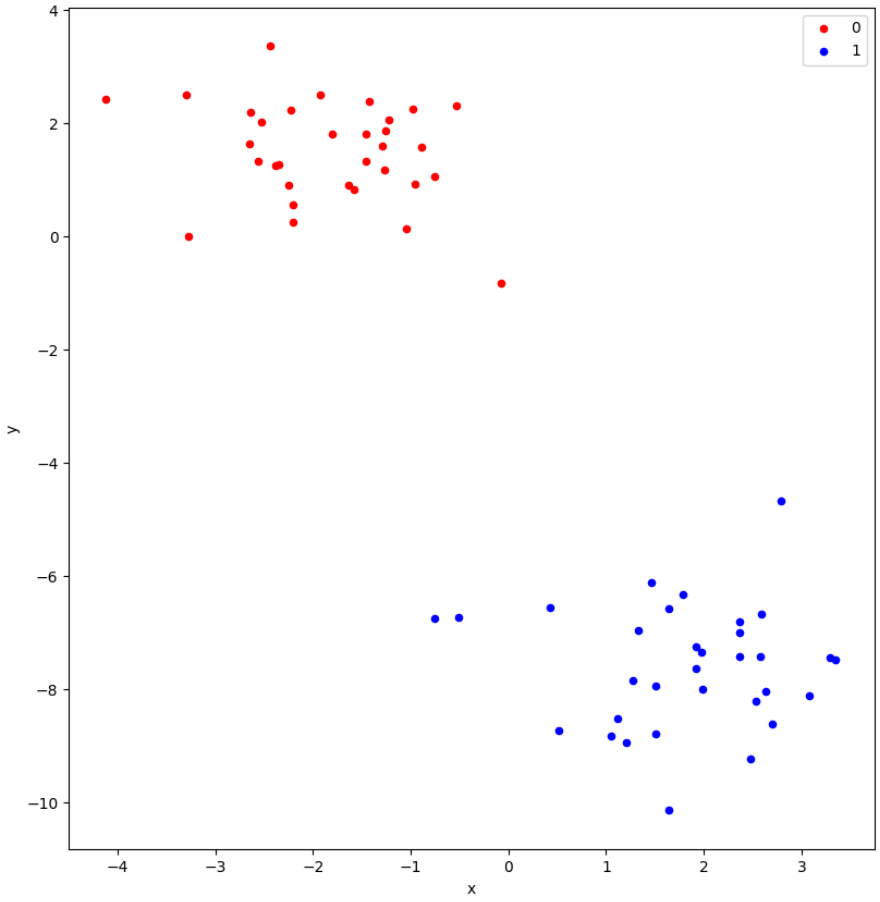

Figure 5.8: An extremely simple, artificial data set, with two linearly separable classes determined by two points.

5.3.3.2.1 A case study of malignancy in the *fizwej* population Meanwhile, in the *fizwej* population, a genetic strain emerged that had found a way to classify the simple dataset flawlessly – or nearly so. Though it had achieved an error rate of 0.0, it terminated its execution, every time, with an instruction that caused the CPU to

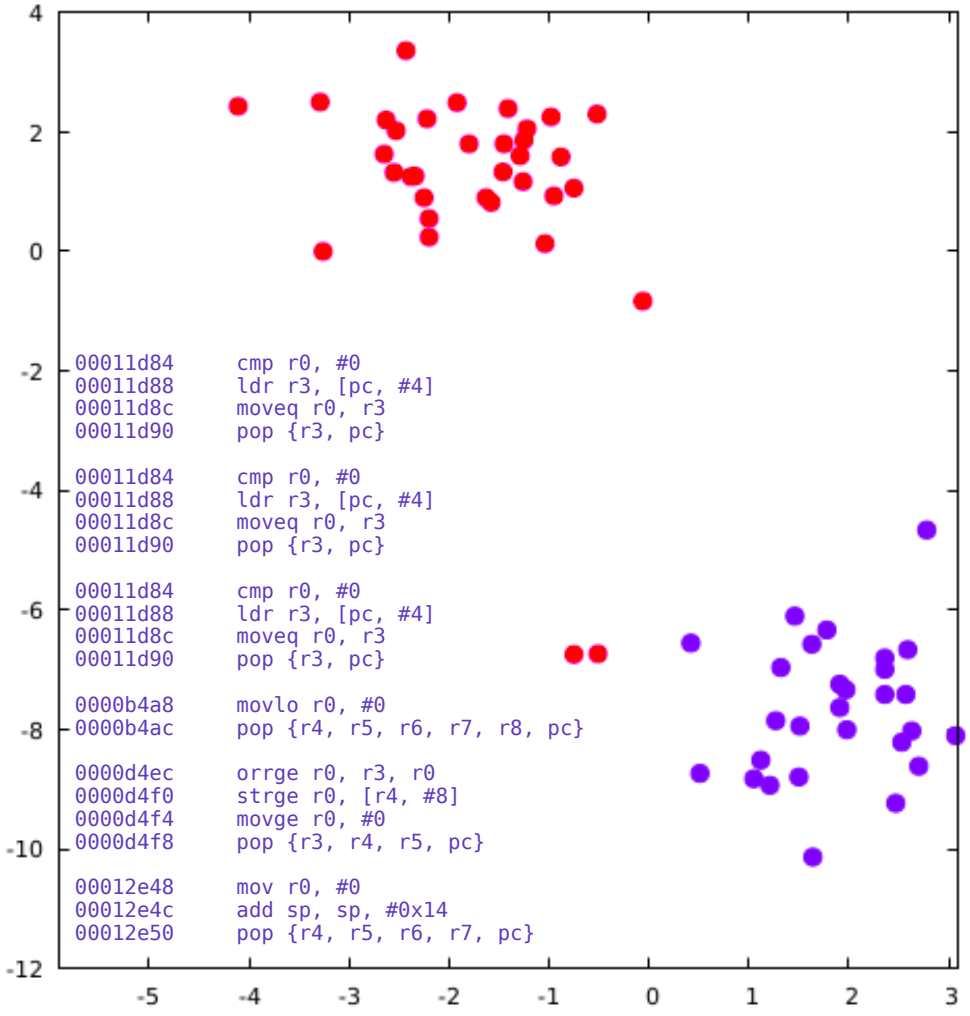

Figure 5.9: The classification on the data displayed in figure 5.8 performed by the best specimen in the *kathot* population, 35^{th} generation (execution trace inset).

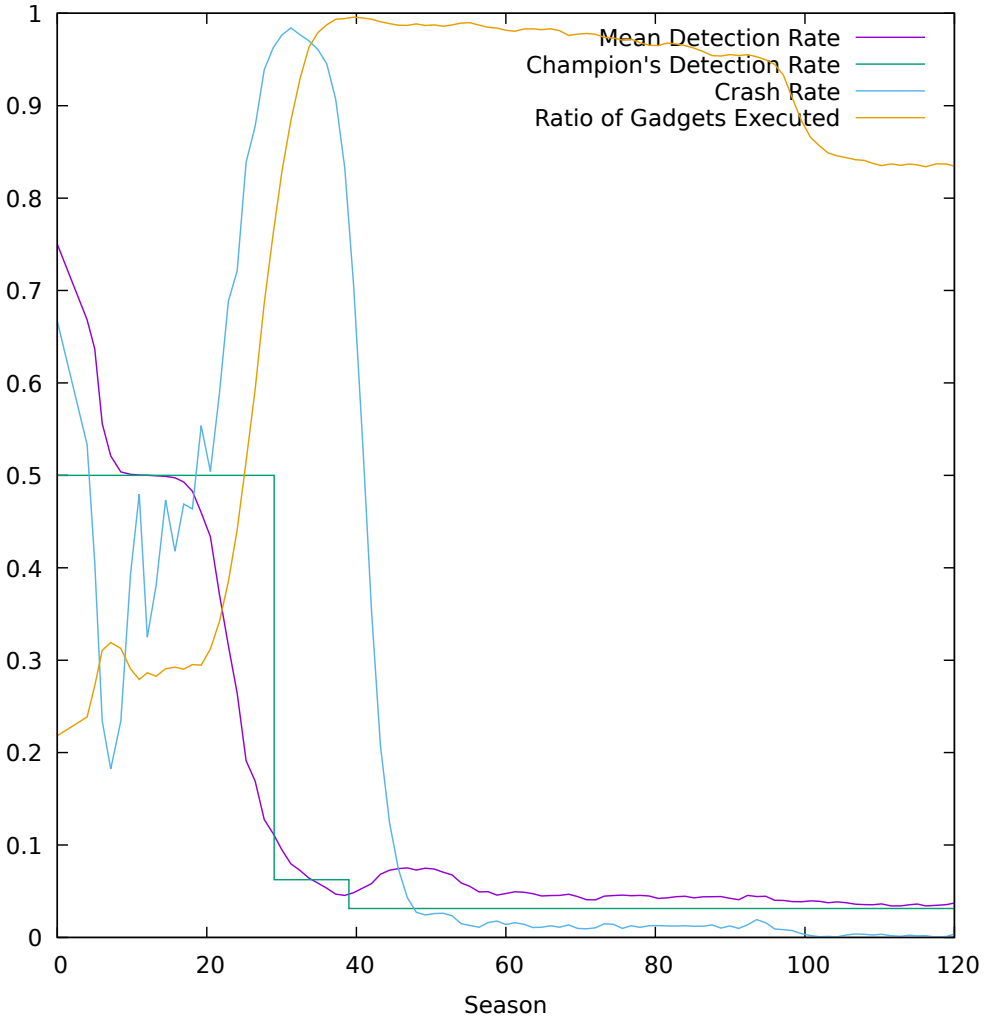

Figure 5.10: Performance profile of *kathot* population, with gadgets in `tomato-RT-N18U-httpd`, on the "two simple blobs" classification problem.

crash with a write access violation. This specimen illustrates a difficulty encountered often in ROPER populations: the offending instruction was part of the same gadget that did most of the work in solving the classification problem, which can be reduced to the following sequence of instructions, given some loose constraints on the constant popped into `R4`:

```
0000acf4    lsr r2, r4, #5            ;; a    := r4 << 5
...
0000acac    add r3, r3, r2, lsl #2    ;; Blue := Blue + (a << 2)
0000ad00    and r2, r4, #0x1f         ;; Red  := r4 & 0b11111
```

After solving the classification problem with these instructions, the phenotype has no possibility of avoiding the next instruction in the gadget:

```
0000ad04    stmib r1, {r4, sl}
```

This instruction increments the value in `R1`, dereferences it, and then attempts to write the contents of `R4` and `R10` (i.e., the stack-limit register, `SL`) to that address. In the memory space mapped for `tomato-RT-N18U-httpd`, however, the only legally writeably region of memory is the space allocated to dynamic memory (where the heap resides) and to the stack, from addresses 0x28000 to 0x28fff and from 0x29000 to 0x2cfff, respectively.[11] In the 7289 different specimens recorded from the *fizwej* population, by the time of its 64$^{\text{th}}$ season, that had this particular gene and that had activated it to achieve a perfect detection rate on "two simple blobs", not a single one had managed to reliably dereference `R1` to a writeable address at the time it dispatched that fatal instruction. To do so, indeed, they would have had to had changed their calculation strategy entirely – their entire *modus operandi* consisted in setting `R1` to a negative value when the exemplar belonged to the **red** class, and to a positive value when it belonged to the **blue**, leaving `R0` more or less constant. All of the writeable addresses in the process space, however, fall within the positive range mentioned. The crash rate of this population rapidly reached 100%, driving itself into an evolutionary dead end.

[11] A sneaky "gotcha" that crops up when converting permission flags between ELF binaries and Unicorn or QEMU images is that while ELF's permission bits have the same semantics as the Unix file permissions: `100` for read, `010` for write, and `001` for execute, Unicorn encodes its permissions the other way around: now, `100` is execute and `001` is read, while `010` is still write.

Start Address	End Address	Protections
0x00000000	0x00000fff	READ
0x00008000	0x00020fff	READ | EXEC
0x00028000	0x00028fff	READ | WRITE
0x00029000	0x0002cfff	READ | WRITE

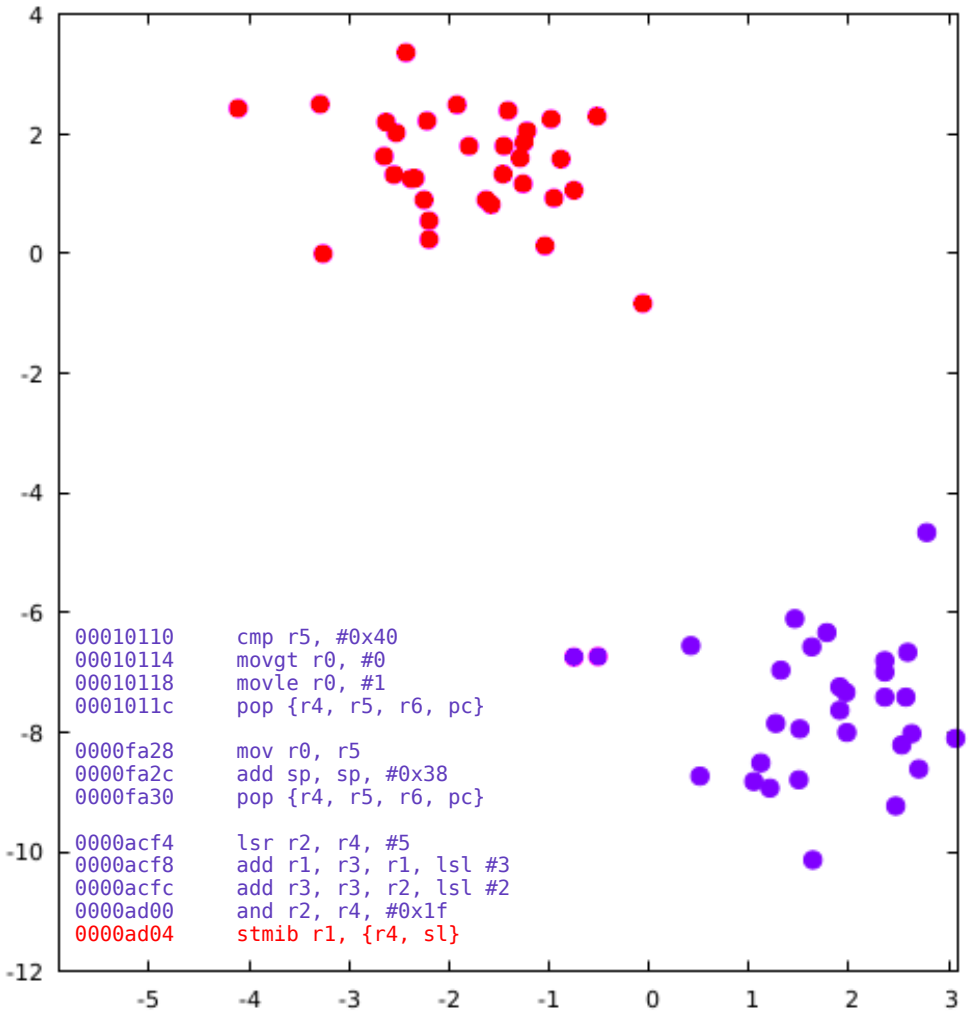

Figure 5.11: A genetic union of a harmful phenotypic trait with an advantageous one, in the *fizwej* population. As before, gadgets are separated by blank lines.

The obstacle that we're encountering here is that, owing to the complex structure of the virtual instruction set that ROPER is working with – an instruction set made up of gadgets rather than single machine instructions – it is possible, perhaps even common, for a dangerous phenotypic trait to secure its place in the population when it

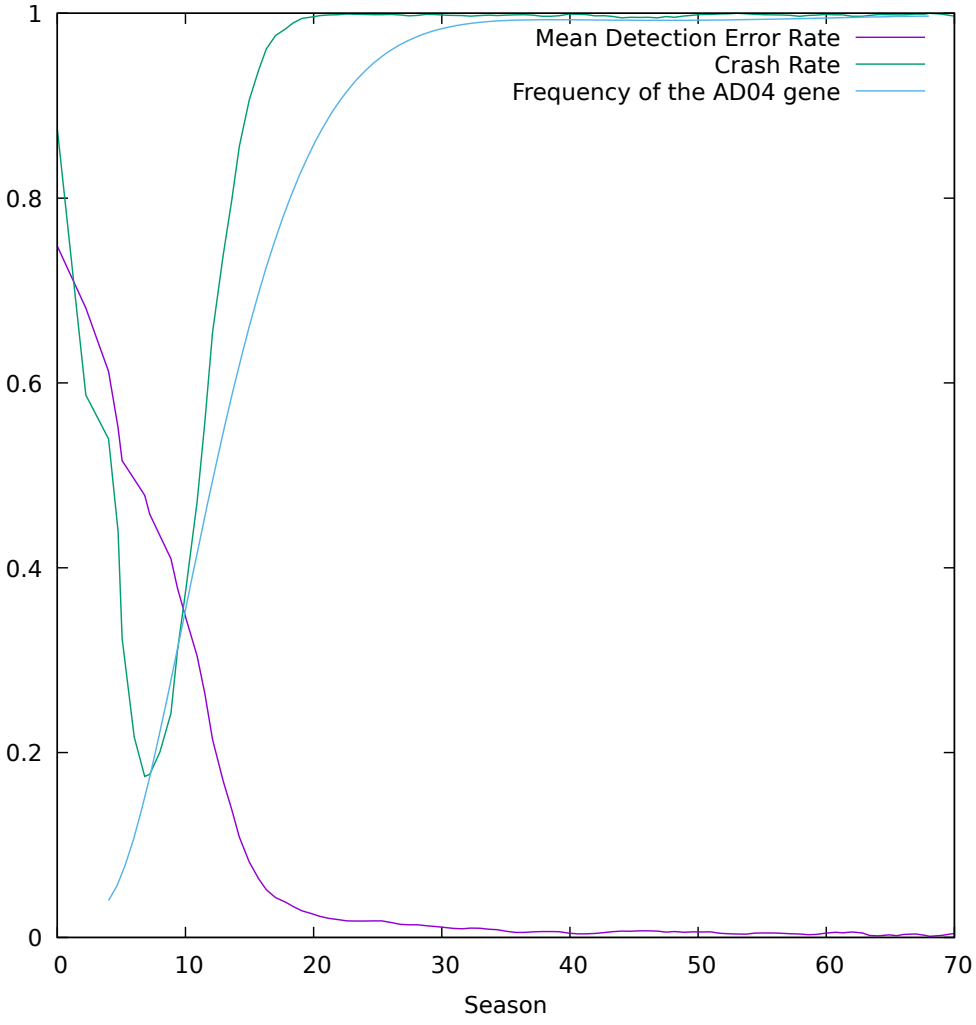

Figure 5.12: The evolutionary dead-end of the *fizwej* population, with the spread of the "`0000ad04 stmib r1, {r3, sl}`" gene.

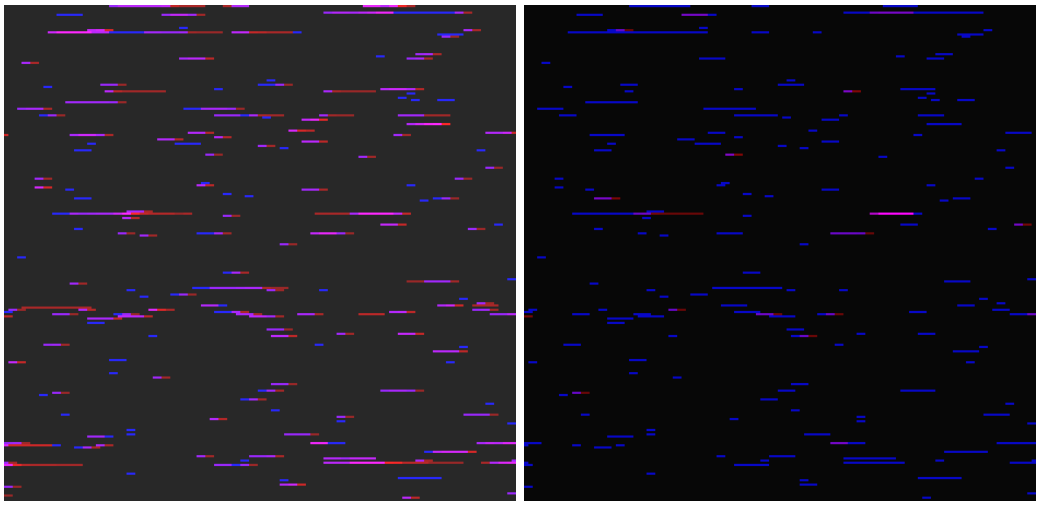

Figure 5.13: Address visitation heatmap of the *fizwej* population, at season 4 on the left, and season 124, on the right, showing a dramatic decline in phenotypic diversity. Compare with the *xeqcyv* and *kurlig* heatmaps in figures 5.5 and 5.3.

is genetically united with an advantageous trait. Of course, this isn't just a problem for ROPER, but for biological organisms as well – we see something similar, for example, in the relation between malaria resistance and sickle cell anemia [17].

The question of whether or not a particular gadget is malignant is highly context-sensitive – since crashes are most often brought about through a register dereference and a subsequent attempt to either read from, write to, or execute data at an address for which the requisite permission has not been granted – and the pressures borne by selection are sometimes too coarse-grained to answer it before a selective epidemic occurs. It may be interesting to attempt to incorporate a more sophisticated semantic analysis routine to inspect gadgets that trigger a crash, and to determine whether or not they should be removed from the gene pool – an idea we will take up further in the discussion of future directions for this project, in Section 6. Incorporating a gauge of genetic, or low-level phenotypic, diversity into the fitness function – scaling fitness by the relative frequency of either the individual's machine-word composition, or its execution path through process memory – may also help mitigate the threat of malignant genetic convergence.

5.3.3.2.2 An early encounter with a segfault plague, due to an vulnerability in the crash handling mechanism This wasn't the first time that I had seen

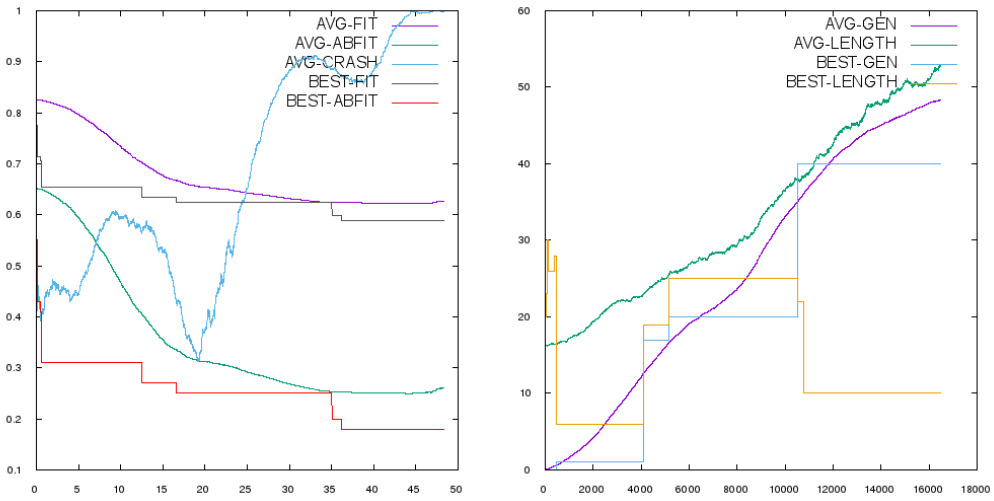

Figure 5.14: A plague of segfaults: the cyan line indicates the crash rate, and the lower left index shows the average genealogical generation, and not the number of iterations, as used in later graphs. The raw data for this experiment has unfortunately been lost, leaving only this artifact as a historical curiosity.

a segfaulting strain rise to dominance in one of ROPER's populations. In one, particularly fascinating and unusually pathological trial with the Iris classification problem (which we will look at more closely in Section 5.3.3.3) , which I stumbled across early in the experimentation, the crash rate skyrocketed, and the *entire population* fell victim to a congenital plague of segfaults before its 20,000$^{\text{th}}$ tournament iteration (figure 5.14). Due to the data that was collected at the time being insufficiently coarse-grained, I cannot say whether dynamics similar to those described in Section 5.3.3.2.1 played a role in this occurrence. At least one of the factors leading up to it was uncovered, however, by reviewing the mean "ratio run" metric of the population: the population had discovered a means of exploiting the sloppy implementation of the gadget-return counter, which was responsible for tracking how many gadgets had executed before a crash occurred, that was then in effect. New to the Unicorn library and the Rust language, and somewhat frustrated with the complications involved in having a callback to the Unicorn engine soundly pass data back to the Rust context from which the engine was dispatched, I'd taken a shortcut and built a counter in what I expected would be an unused region of the emulator's memory space, and then used a straightforward API to read the counter from memory after execution had terminated. At the time, I imagined that though there was some chance the counter

could *perhaps* get corrupted, I would be able to treat that corruption as inconsequential noise. This was a mistake, and the population wasted no time in exploiting it. A dominant genetic strain had evolved to hijack the return counter, setting it to an artificially high value before wantonly crashing. The apparent success of those chains in executing numerous gadgets before crashing meant had reduced the crash penalty to near zero, and since enough of this line had managed to perform fairly well on the classification task – achieving an 82% detection rate against Iris – in addition to exploiting the experimental framework, they soon wiped out every single lineage that they ran up against.[12]

This was by no means the norm, however. The bug was fixed, and a secure conduit for the return counter was implemented, avoiding any in-band communication that could be hijacked by the population it was meant to assess. As an additional safeguard, a second factor was incorporated into the crash-penalty gradation: the penalty would steepen in proportion to the global frequency of crashes in the population.

This bolstered the tendency of the crash rate to oscillate, of course, though the oscillations appear to occur to some degree with or without the global crash frequency penalty. The global crash rate would generally settle into a comfortable oscillation between 1 and 20 percent of the population crashing, at any given time.

A stricter penalty could easily reduce the crash rate to almost zero, but this appeared to negatively impact long-term performance. The fitness landscape inhabited by ROPER's populations, it seems, is extremely jagged, and too strict a penalty to crashing would prevent the population from crossing from one fitness peak to another. It made more sense to work with the tendency for crash rates to oscillate than against it, and allow exploration of more dangerous waters so long as it doesn't threaten to risk the long-term well-being of the population as a whole.

In view of the phenomena described in Section 5.3.3.2.1, however, this approach to regulating the crash rate may warrant revision.

5.3.3.2.3 A proposed method for regulating malignant genes with a TTL field

Another mechanism that we could use to mitigate this dynamic may be to add a TTL field to our `clump` datatype. With each crossover or cloning event, the

[12]This was a very nice example of what Lehman et. al have called "The Surprising Creativity of Digital Evolution" in [22].

TTL would be decremented, and when it reaches zero, the clump would be excised from the genome and replaced with a new, randomly generated one, perhaps with an identical SP_Δ and immediate component. In the event of a crash, we could examine the address visitation path tracked by the emulator to find the last clump entered before crashing. Once the guilty clump is found, its TTL could be halfed (or some order of magnitude greater than the linear decrement that occurs in crossover). We could then experiment with the results of different tunings to the TTL field and its methods, and see if this might be a better way of regulating the crash rate than the current approach, which simply factors a crash penalty into the same, scalar fitness value that determines reproductive odds.

5.3.3.3 *Fleurs du malware*: Classification of the Iris data set

The Iris dataset, though relatively simple by contemporary classifier standards, is considerably more complex than the "two simple blobs" dataset explored in Section 5.3.3.2. Its attribute-class mapping is plotted in figure 5.15

5.3.3.3.1 Without fitness sharing The fitness curve of our best specimens *without fitness-sharing* typically took the form of long, shallow plateaus, against the backdrop of a population swaying, it seemed, more in response to genetic drift than selective pressure.

An interesting phenomenon seemed to recur in several populations, however, following a prolonged fitness plateau: long after it had been effectively quelled by selective pressure, the rate at which individuals crash during execution would begin to rise again, climbing, in fits and starts, from near zero to up to 40% of the population. This is what we see happening in figure 5.16, for example, which documents one of the early experiments performed with ROPER.

One possible explanation for this strange behaviour is that a second-order selective pressure encourage intron formation, of which the crash rate may be a symptom, if the method for forming introns, favoured by this population, involved individuals altering their own call stacks so as to escape dependency on certain segments of their genomes – a strategy that might be viable for one or two generations, but may result in more fragile chains in in subsequent ones. I explore this possibility in more depth

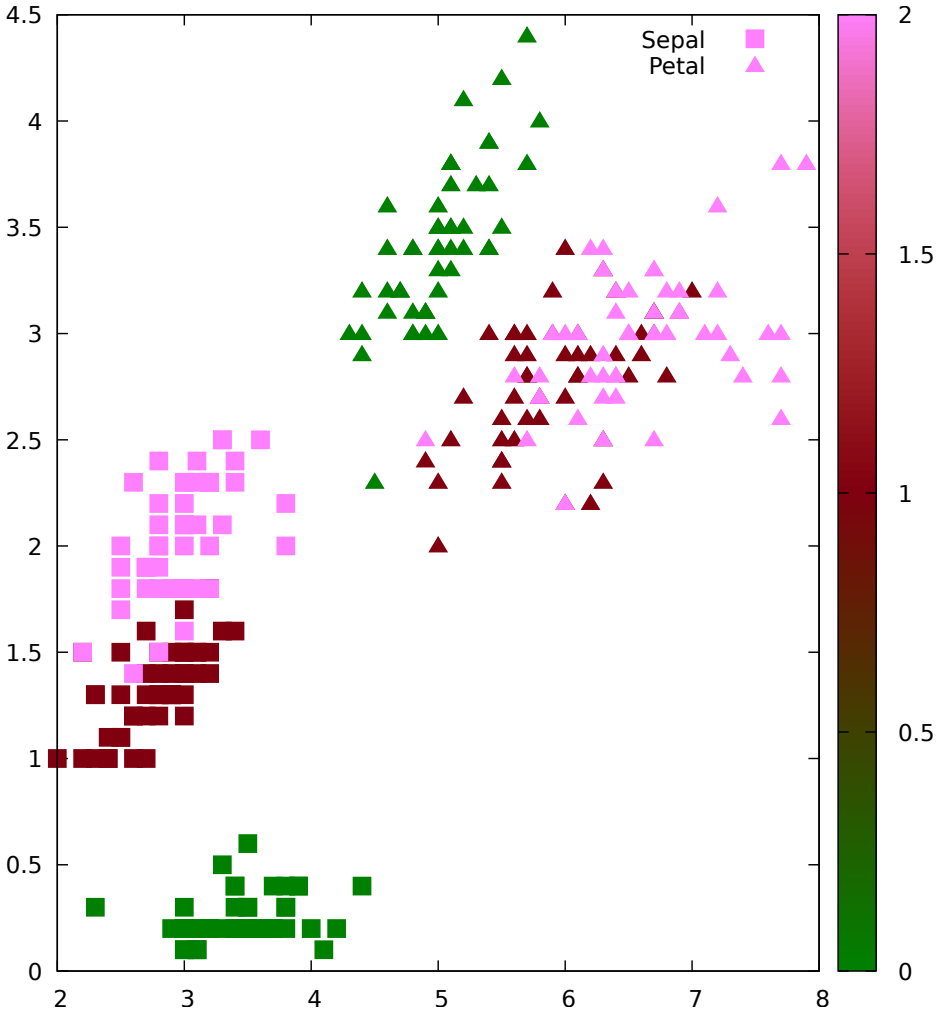

Figure 5.15: Map of the Iris dataset. Triangle points represent petal measurements, and square points represent sepal measurements, with length on the X-axis and width on the Y-axis. Colour maps to species: green for *setosa*, maroon for *versicolor*, and pink for *virginia*.

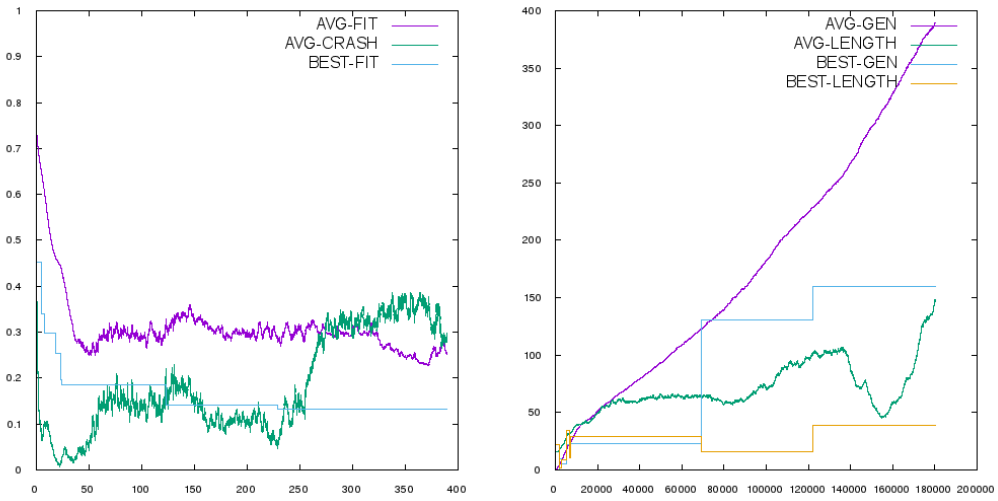

Figure 5.16: Resurgence of the crash rate during a fitness plateau, in an early run of the Iris classification task, before implementing fitness sharing. The X-axis in the left-hand pane tracks the average generation of individuals, while the X-axis on the right tracks iterations of the main loop. The relation between the two measures is linear, and so these panes can be more or less superimposed.

in section 5.4.2.

Curiously, the peak in crash rate coincides with a dip in the average length of the population, which appears contrary to the supposition that the population was undergoing a strong pressure for intron formation at this point – code bloat is almost certainly the *simplest* means of forming introns. This may be an artefact of the fragility mechanic detailed in Section 4.3.2.1, however: a sufficiently fragile gene has a strong likelihood of simply being dropped during crossover, and passed to neither offspring, and when a crash occurs, the fragility of the last gadget to have been executed is automatically maximised. We could, therefore, be seeing a *consequence* of the rising crash rate in the corresponding drop in genome length – this does, at least, appear to me to be a likely hypothesis.

Another, non-exclusive possibility is that the fitness plateau we can see, stretching across the second half of the history documented in figure 5.16 is a symptom of (premature) genetic convergence. Perhaps it converged on a fitness peak that was robust enough to survive a certain number of reproductive cycles – long enough for it to establish dominance in the population, and drive out competing phylogenic strains, but which was surrounded on all sides by steep ravines. By the time we start to reach

mean generation 220 or so, descendents of that strain may be starting to reach the edges of that fitness plateau, and *their* descendents begin dropping off the edge, en masse. This seems incongruous with the mean fitness line continuing to progress, but this is just the mean, and may be the effect of that strain completing its ill-fated domination of the gene pool.[13]

5.3.3.3.2 With fitness sharing What would consistently seem to occur in the classification runs performed using a static fitness function, keyed to detection rate, was that the strains in the population would emerge that could reliably distinguish the linearly separable species of iris from the others, but whose competence would end there. This is, of course, not a trivial accomplishment in itself for a randomly generated ROP chain, and so this strain would rapidly outperform its competitors in almost every tournament in which it was represented, soon dominating the gene pool, and bringing about a genetic convergence of the population, whose performance would then plateau. If there had been any resources in the population that *could* have made some headway on distinguishing the two less tractable, and more entangled, species, had they had time to be tuned by genetic operators, those resources would likely have been expunged by the gene pool in their carriers' fierce competition with the "bottom feeders" whose high absolute fitness scores represented only a facility for solving relatively *easy* problems.

For this reason, I introduced into ROPER the *fitness sharing* mechanism that I have outlined in Section 4.4.4. The result (after some persistence and plenty fine-tuning) was a superb run – achieving 96.6% detection rate (0.034 absolute fitness) on the Iris set in 27,724 tournaments, 216 seasons of difficulty rotation, and an average phylogenic generation of 91.3. Figure 5.18 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 informatively compared with the one illustrated in figure 5.17. Note the tight interbraiding of problem difficulties in figure 5.18, as compared to their gaping – but still, slowly, fluctuating – disparity in figure 5.17. The ballooning

[13]The raw data from this run has unfortunately been lost, and so we're restricted to speculating on the basis of its surviving artefacts.

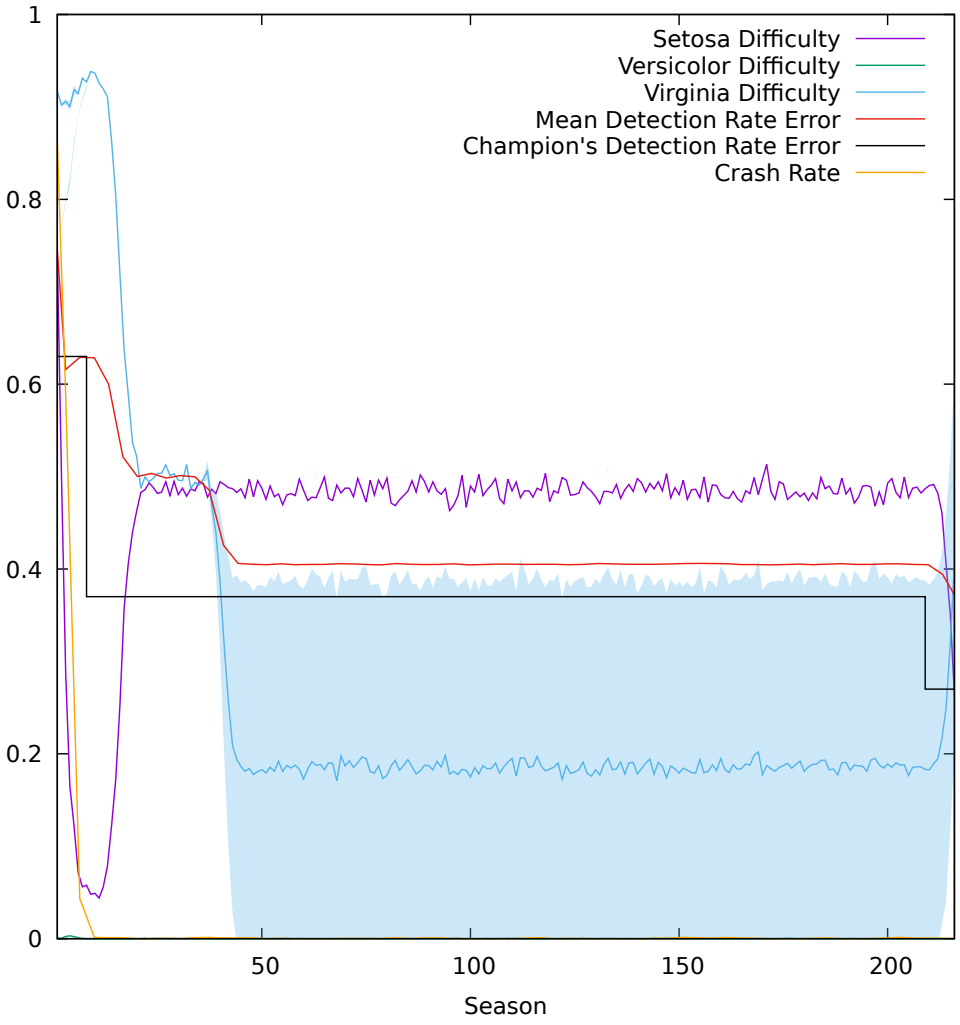

Figure 5.17: A trial similar to the one documented in figure 5.18, with per-class difficulty recorded, but with the fitness sharing mechanism suspended (*cazmud* population). The filled curve surrounding each mean difficulty class, here being of visible breadth only in the case of the *iris virginia*, represents the standard deviation of difficulty for each exemplar class.

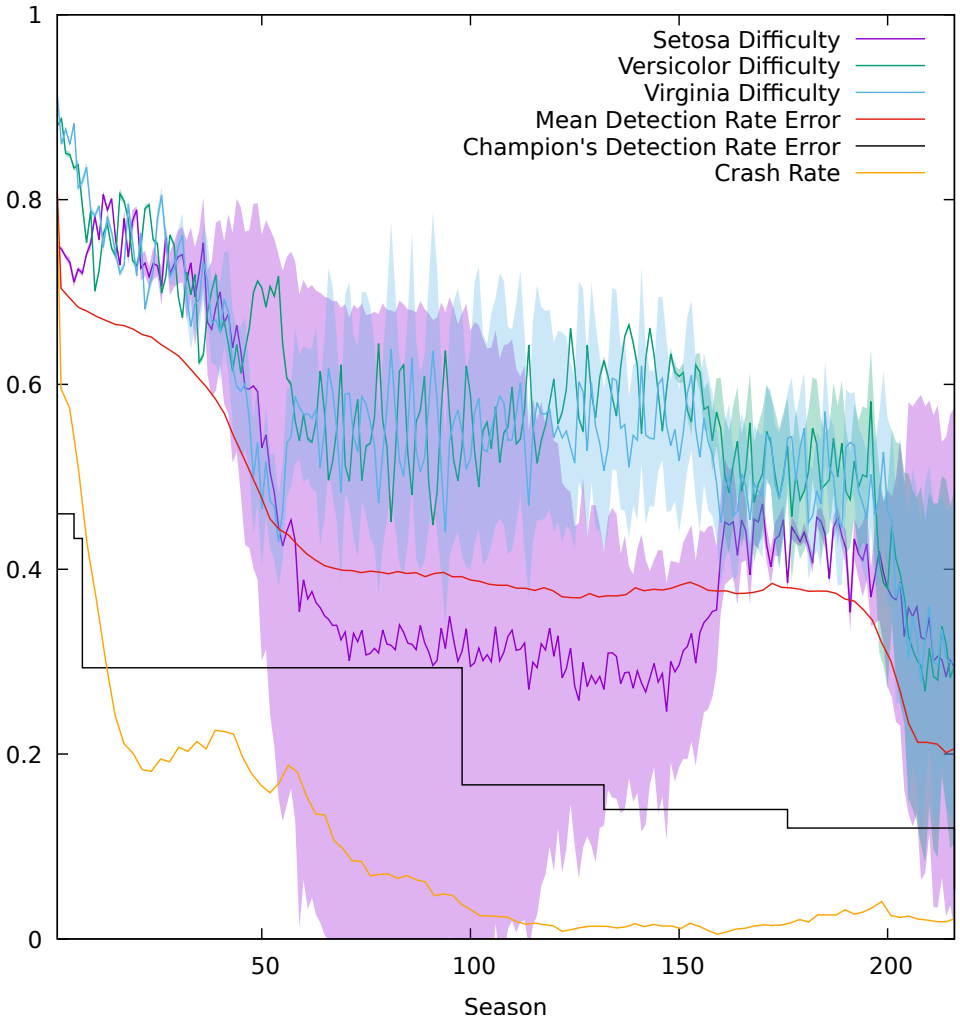

Figure 5.18: A very good run on the Iris classification task, employing the fitness sharing algorithm documented in Section 4.4.4 (*ragweb* population). The filled curve surrounding each mean difficulty line again represents the standard deviation of difficulty for that exemplar class.

standard deviation of difficulty by class in figure 5.18 also suggests a dramatic increase in behavioural diversity in the population, which is precisely what we aim for with the fitness sharing algorithm.

5.3.4 Preliminary results of the *Snake* experiments

In the final problem class put to ROPER, I wrote a simple *Snake* game [8] for its populations to play, to see how they may respond to a dynamically changing environment, with moderate to high degrees of randomness. The specifications of the game are described above, in Section 5.1.4. To control the degree of randomness, I set up the game control protocol so that the ROPER engine could pass a random seed to the game in order to initialize the placement of "apples" and "cacti" on the game board. Surprisingly, the system appeared to perform better when the seeds were randomly generated and highly unique, rather than drawn from small set of seeds that would remain fixed for the duration of the evolutionary run.

Results on this experiment remain largely preliminary and anecdotal in nature, but evidence of its basic feasibility can be found in the time-lapse recording of the final champion of the *misjax* population playing a semi-competent game of Snake in figure 5.19.

5.4 Intron Pressure, Self-Modifying Payloads, and Extended Gadgetry

In all of the experimental trials performed, a certain number of peculiar, interrelated phenomena appeared in the dynamic behaviour of the population as a whole.

5.4.1 Crash Rate Oscillations

The first has to do with the observed crash rates. In the beginning – as is to be expected, since our initial gadget harvest is deliberately roughly hewn and approximate, with no attempt to formally verify the individual reliability or usefulness of each gadget or combination thereof – the vast majority of our specimens (80 to 90 percent) would crash before completing execution. This would most often be the result of a segmentation fault, or memory access violation error, when the specimens would attempt to dereference and invalid pointer, or jump to an address outside of

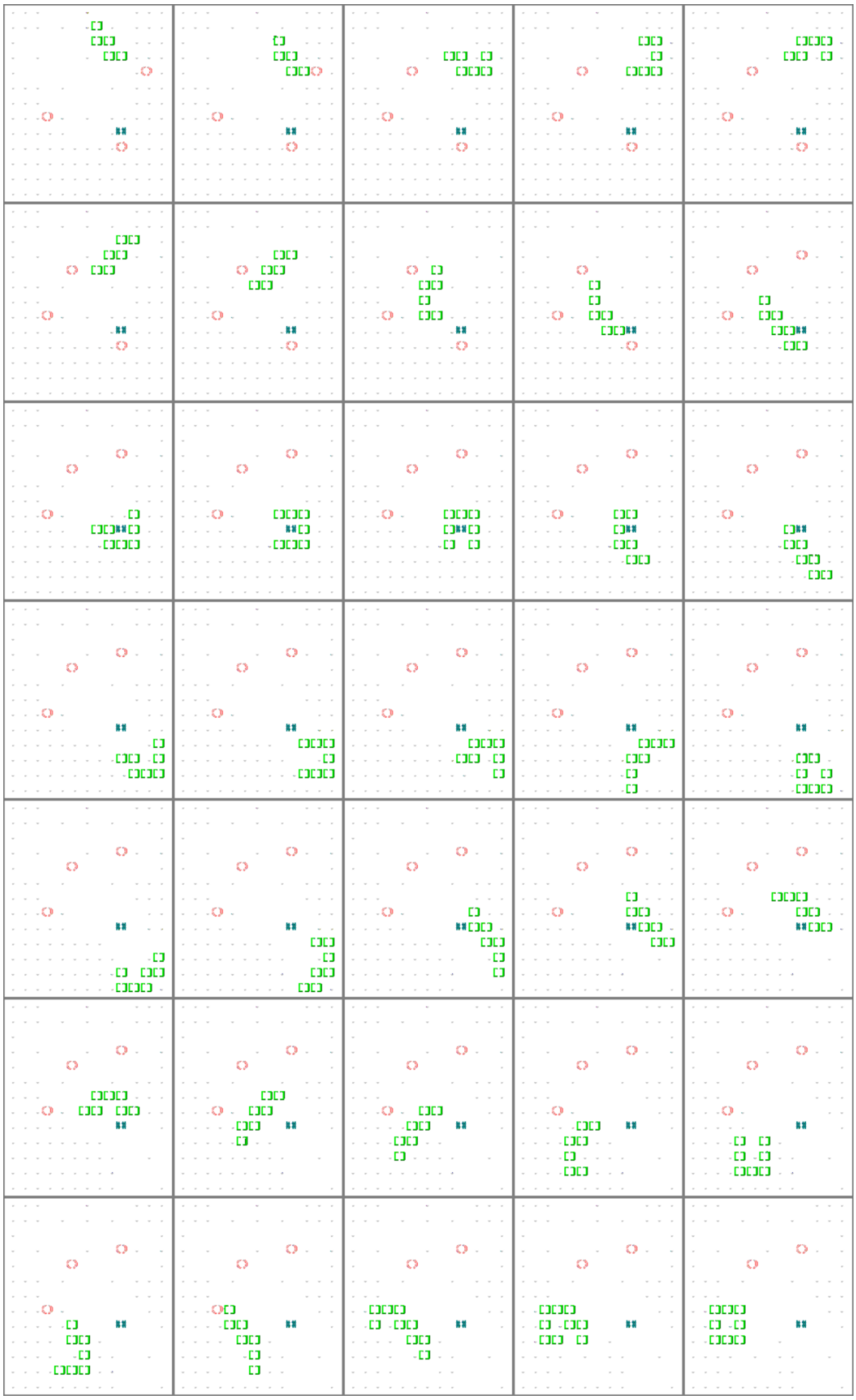

Figure 5.19: Time-lapse rendering of one of the fitter specimens encountered in the Snake trials (*misjax* population). Snake segments: `[]`, apples: `()`, cacti: `##`.

an executably-mapped segment of memory. Since such violations carry with them a fairly steep penalty to individual fitness, *and* increase the likelihood that the genes responsible for exposing the individual to such hazards are dropped from the gene pool, the rate of crashes would always drop fairly quickly within the first few thousand tournament iterations. There is nothing unusual or unexpected about this – reducing the likelihood of crashing is the simplest way for any of our individuals to ensure their survival and prospects of reproduction, and appears to be much easier for our populations to accomplish than the specific task-oriented components of the fitness function. And whereas the prospect of crashing "in the wild", before having a chance to reproduce, is completely and utterly fatal to evolving malware strains (see the discussion of this problem in Section 3.1.1), our "in vitro" populations have the advantage of a gently gradated crash penalty: the fitness penalty incurred by a crash is inversely proportionate to the number of gadgets viably executed before the crash.

What *is* surprising, however, is that the crash rate does not *continue* to decline or stabilize at the extremely low level that it tends to reach after its initial dropoff. The curve it traces relative to the number of iterations doesn't even *approximate* a monotonic reduction. Instead, it eventually – but consistently – begins to rise soon after the average fitness of the population reaches a plateau, and then starts to oscillate, until the plateau is broken and the fitness equilibrium is punctured.

5.4.2 The stray rate and extended phenotypes

Once a secure conduit had been implemented for passing detailed execution information from the emulator back to the evolutionary engine (see Section 5.3.3.2.2, another rich vein of information had opened up, which helped to provide evidence for a tendency that, until then, I had only been able to observe by manually dissecting the disassembly dumps of individual specimens post-mortem[14]: I had noticed a tendency for ROPER populations' best performers to frequently be those that take strange and enigmatic risks with their own control flow – manipulating the program 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 behaviours that seem, *prima facie*, destined for failure in most cases. Their apparent frequency was an

[14]As discussed in my 2017 report on this project [15], and my talk at AtlSecCon 2017 [14].

enigma.

Consider, for example, the specimen displayed in 5.2 which achieved a perfect fitness in trial of the CPU context matching experiment, where the task was to prepare the register vector for an

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

system call – the sort of task that manually crafted ROP chains are often designed for, so that the attacker can spawn a shell with the privileges of the exploited process.[15]

```
;; Gadget 0
000100fc         mov r0,r6
00010100         ldrb r4,[r6],#1
00010104         cmp r4,#0
00010108         bne #4294967224
0001010c         rsb r5,r5,r0
00010110         cmp r5,#0x40
00010114         movgt r0,#0
00010118         movle r0,#1
0001011c         pop {r4,r5,r6,pc}

;; Gadget 1
00012780         bne #0x18
00012798         mvn r7,#0
0001279c         mov r0,r7
000127a0         pop {r3,r4,r5,r6,r7,pc}

;; Gadget 2
00016884         beq #0x1c
00016888         ldr r0,[r4,#0x1c]
0001688c         bl #4294967280
0001687c         push r4,lr

00016880         subs r4,r0,#0
00016884         beq #0x1c
000168a0         mov r0,r1
000168a4         pop {r4,pc}
```

[15] See the semantic correction noted in Section [?], which spoils the real-world usability of this *particular* chain, but which does not logically impact the analysis and conjectures formed here, in relation to it.

```
;; Extended Gadget 0
00016890         str r0,[r4,#0x1c]
00016894         mov r0,r4

00016898         pop {r4,lr}
0001689c         b #4294966744
00016674         push {r4,lr}

00016678         mov r4,r0
0001667c         ldr r0,[r0,#0x18]
00016680         ldr r3,[r4,#0x1c]

00016684         cmp r0,#0
00016688         ldrne r1,[r0,#0x20]
0001668c         moveq r1,r0

00016690         cmp r3,#0
00016694         ldrne r2,[r3,#0x20]
00016698         moveq r2,r3

0001669c         rsb r2,r2,r1
000166a0         cmn r2,#1
000166a4         bge #0x48
000166ec         cmp r2,#1
000166f0         ble #0x44
00016734         mov r2,#0
00016738         cmp r0,r2
0001673c         str r2,[r4,#0x20]
00016740         beq #0x10
00016750         cmp r3,#0
00016754         beq #0x14

00016758         ldr r3,[r3,#0x20]
0001675c         ldr r2,[r4,#0x20]
00016760         cmp r3,r2

00016764         strgt r3,[r4,#0x20]
00016768         ldr r3,[r4,#0x20]
```

```
0001676c         mov r0,r4

00016770         add r3,r3,#1
00016774         str r3,[r4,#0x20]
00016778         pop {r4,pc}

;; Extended Gadget 1
00012780         bne #0x18
00012784         add r5,r5,r7
00012788         rsb r4,r7,r4
0001278c         cmp r4,#0
00012790         bgt #4294967240
00012794         b #8

0001279c         mov r0,r7
000127a0         pop {r3,r4,r5,r6,r7,pc}

;; Extended Gadget 2
000155ec         b #0x1c
00015608         add sp,sp,#0x58
0001560c
pop {r4,r5,r6,pc}

;; Extended Gadget 3
00016918         mov r1,r5
0001691c         mov r2,r6
00016920         bl #4294967176
000168a8         push {r4,r5,r6,r7,r8,lr}
000168ac         subs r4,r0,#0
000168b0         mov r5,r1
000168b4         mov r6,r2
000168b8         beq #0x7c

000168bc         mov r0,r1
000168c0         mov r1,r4
000168c4         blx r2
```

Listing 5.2: Execution trace of a chain that generates the register pattern required for a call to `execv("/bin/sh", ["/bin/sh"], NULL)` in `tomato-RT-N18U-httpd`, by

modifying its own call stack and executing numerous “stray” or “extended” gadgets, in the *poclux* population.

It’s an extaordinarily labyrnithine 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.[16]

By tracking every address visited by every chain in its movement through the host process, I was able to confirm my suspicion that this bizarre behaviour is not at all uncommon in ROPER’s populations – nor is it only to be found in pathological, crash-prone specimens.

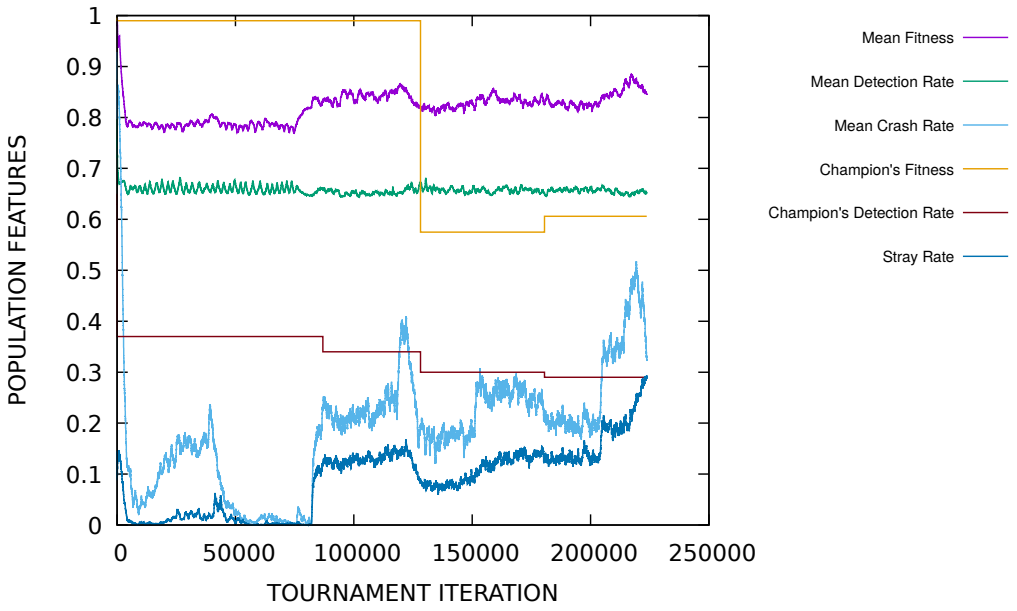

Figure 5.20: A run on the Iris classification task, with a high stray rate, by the *hepfap* population.

[16]I’ve given the name ’extended gadgets’ to these units of code, meant to suggest analogies with Dawkin’s notion of the extended phenotype [10], for somewhat speculative reasons that will be explained in a moment.

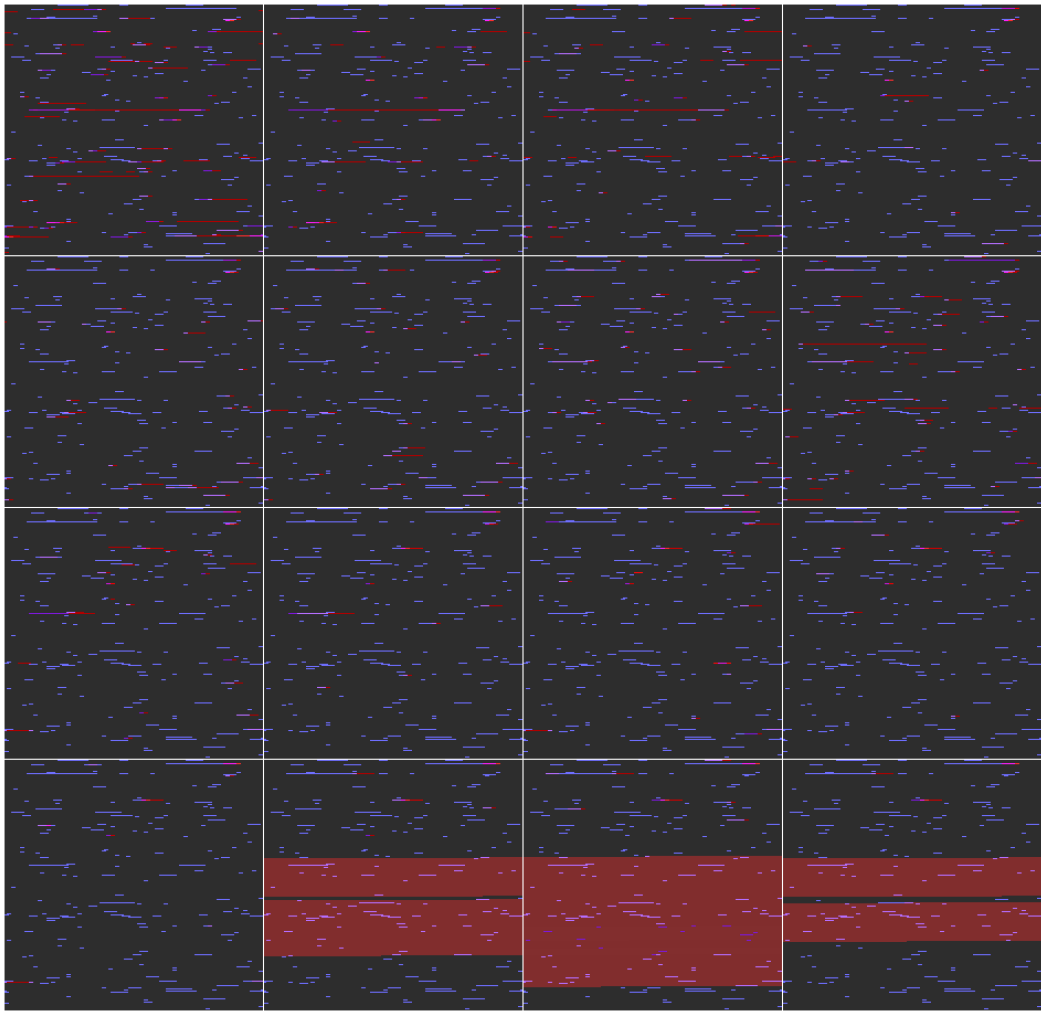

Figure 5.21: Heatmap montage over the *fimjek* population, showing range of addresses executed, in red tint, superimposed on a map of the explicit gadgets that were harvested to form that population's initial gene pool, in blue tint. The intensity of red tint indicates the frequency with which the corresponding address was visited. Magenta and purple cells indicate orthodox gadget traffic, while red swaths with no tint of blue indicate stray activity. From top-left to bottom-right, each cell is a snapshot of the evolving heatmap at intervals of 20 seasons. The underlying gadget map is taken from the `tomato-RT-N18U-httpd` binary that we used in this experiment.

5.4.3 A conjectural explanation of stray-rate fluctuations as a result of intron pressure

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 proliferate 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 [31], 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 is uncertain that ROPER would be able to discover these labyrinthine passageways through its host if the selection pressure against errors were more severe. The breaking of a fitness plateau, in most of the populations observed, was forecasted by a momentary peak in the crash rate. This often appears hand-in-hand with a periodic increase in genome length, which chimes with some of Banzhaf's findings regarding the relation between intron bloat and punctuated equilibrium in evolutionary processes [31]. During such periods, 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.[17]

We see this clearly enough in our champion ROP-chain displayed in Figure **??**, 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.

[17]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 [5]

5.4.4 Testing the Extended Gadgetry Conjecture with Explicitly Defined Introns

The explanation given above for the strange behaviour observed seems to me to be compelling enough on strictly theoretical grounds, but it still remains to be seen if it can withstand experimental testing. If, as I have conjectured, this behaviour appears because it represents a rich, even if 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 much safer supply of **explicit** introns into the gene pool. All we need to do is to define a 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. The simplest way to do this is just to attach an `enabled` flag with each clump, which can be set to either `true` or `false`. When `false`, the clump is ignored by the serialization procedure that prepares the payload, so that it's never sent to the emulation engine. We will also add a new mutation operator, which is able to toggle the `enabled` flag during reproduction. This lets the intron serve the additional, potentially useful purpose of acting as a repository for genetic information. If the selective pressure responsible for "extended" or "stray" gadgets is indeed derivative of the well-known pressure to generate introns, then these explicitly defined intronss (EDIs) should be able to undercut their market share.

Patience, and an adaptively disadvantaged experimental setup, was eventually rewarded with a compelling corroboration of the intron conjecture regarding the proliferation of stray gadgets.

Eight populations of ROP chains over the `tomato-RT-N18U-httpd` binary were initialized with identical parameters, with the sole exception of the EDI rate: four (*xufmoc*, *simtyn*, *surjes*, and *mycwil*) began with an EDI rate of zero, with no further possibility of acquiring EDIs through mutation. The other four (*megkek*, *huzqyp*, *rofted*, and *qatjaq*) were initialized with an approximate 10% EDI ratio, and a 5% per-clump EDI mutation rate – meaning that in the event of a mutation, which occurs in 50% of reproduction events, the other 50% being the result of single-point crossover, each individual clump has a 5% chance of being toggled. In the event of crossover, the `enabled` flag is simply inherited, along with its clump, unaltered. This is a fairly

aggressive mutation rate, giving us a probability of

$$\frac{\sum_{n-1} \frac{1}{20} * (1 - \frac{1}{20})^n}{2}$$

that at least one clump in a chain of length n will be toggled on or off, in each reproduction event.

The task component of the fitness function for these trials was to match a precisely specified CPU context, similar to the register-matching task discussed in Sections 5.1.2 and 5.3.2, but with one key difference, incorporating a recent update to the engine: the population would be responsible for matching not just a series of immediate register values, but correctly dereferencing pointers as well (but only up to one degree of indirection). The exact pattern in question, in ROPER's (updated) syntax, is:

```
0002cb3e,&0002cb3e,000000000,_,_,_,_,0000000b
```

Corresponding to the preparation of the CPU for the system call, `execv("/bin/sh", ["/bin/sh"], NULL)`, with `/bin/sh` at address 0x2cb3e. We will be disregarding the fitness results for these experiments, however. The populations were deprived of mutation operators, with the exception of an EDI toggling mutation in the cases of *xufmoc*, *simtyn*, *surjes*, and /mycwil. Their task evaluation function was also hobbled, so that it would feed them very little information regarding the proximity of the registers to a correct match – nothing but the bitwise hamming distance, for both immediate and indirect register values. They were designed to either fail indefinitely, or evolve for an extremely long period of time before converging, the better to study their long-term dynamics.

The results, presented in figures 5.22 and 5.23, were found to corroborate the intron hypothesis, though not as dramatically as anticipated. One of the four EDI populations still showed a visible history of stray address visitation, but only one, as compared to the four EDI free populations, all of which showed signs of straying. Though fine-grained genealogical data is not available for these populations, the relative continuity of the stray line in the *rofted* population, appearing in figure 5.22, suggest that this *may* be the result of a single bloodline, in contrast to the EDI free *surjes* and *simtyn* populations, where the incidence of strays appears in fits and

starts, suggesting multiple genealogical origins. It would be interesting to repeat this experiment with larger populations, and the collection of genealogical data, as time (and memory space for exponentially accumulating data) allow.

The trajectory of the genome length curve in each population group is consistent with what we know about introns and genetic bloat (cf. [31]), and it is to be expected that the populations lacking EDIs would supplement their absence with an accumulation of "spare" genes. But what's surprising is that those same populations appear to be executing the majority of their tremendously large genomes – and doing so with relatively infrequent crashes. The non-EDI populations have an anomalous member in this respect, too: the *xufmoc* population exhibits a sharp downwards trajectory in its genome length, following its 2000^{th} season.

A surprising result is the the non-EDI populations appear to have a higher ratio of executed gadgets – even though this ratio is calculated only with respect to *enabled* gadgets, excluding EDIs. The difference on this measure is rather subtle, however, and may be the effect of noise, or overrepresented and idiosyncratic bloodlines.

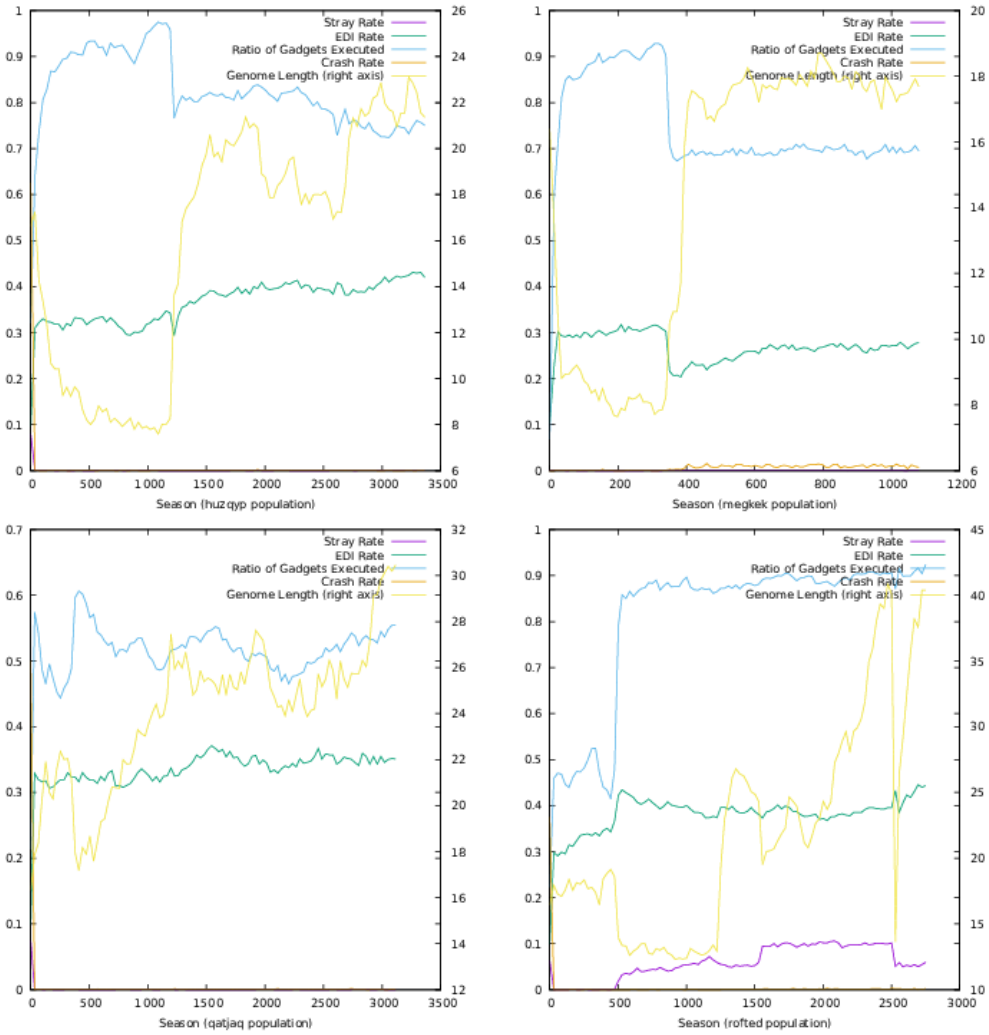

Figure 5.22: Plot of EDI frequency to stray rate, crash rate, and the ratio of gadgets run in the *hyzqyp*, *megkek*, *qatjaq* and *rofted* populations, with non-EDI mutation disabled.

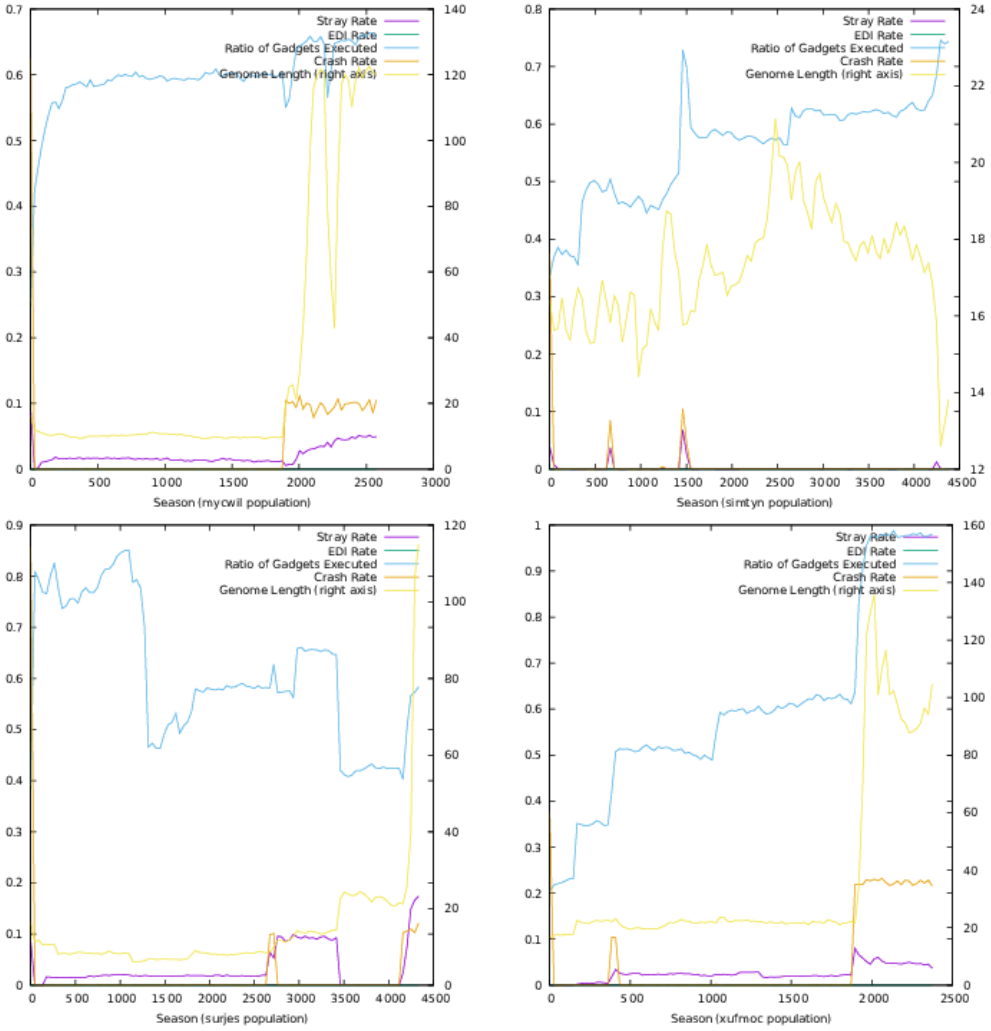

Figure 5.23: Plot of stray rate, crash rate, and the ratio of gadgets run in the *mycwil*, *simtyn*, *surjes* and *xufmoc* populations, with mutation and EDI toggling disabled, with a base EDI rate of zero.

6

Conclusions and Future Work

6.1 The devil in the details

ROPER has been explicitly designed as a "proof of concept", and not as a refinement on existing prior art. The experiments documented here, that is to say, represent the first glimpse we have had of the possibility of ROP chain evolution. The application of an existing body of knowledge and techniques – here, genetic programming – to an otherwise familiar application domain, admittedly, seems to be something somewhat narrower than a *conceptual* innovation.

But, on the one hand, the devil's in the details, and this is why I chose to spend the majority of Chapter 5 labouring over peculiarities in individual cases and idiosyncracies in population trends, rather than focus on accumulating quantitative performance metrics. What we find there, I think, are challenges *specific* to the domain of evolving populations of programs on such "weird machines" as the ROVM. It is one thing to design a system that is optimised and streamlined, from the ground up, for use in machine learning or evolutionary computation, but it is something quite

different to look for ways to foster evolution in uneven assemblages of found materials. These are assemblages that supervene on designed and documented machines, but which, considered at the level of abstraction on which they operate, *no one* has designed, or documented, strictly speaking. No one *builds* a ROVM; they emerge accidentally through leaks in the procedural machine model around which most software is designed and with an eye to which it is implemented. They are systems that are *discovered*, more than invented, even as they lie at the heart of artificial and meticulously crafted systems. They form a sort of junkyard wilderness at the heart of computational civilization. Their exploration and study is the art of hackers, but this project began with the intuition that something more might be learned of this strange terrain if we approach it from the side of the wilderness, instead of from the side of the explorer, and see how far we can get in coaxing unexpected structures and behaviours from it by applying just a bit of selective pressure, and letting the system tell us about itself through the constraints it imposes on the resulting evolutionary process.

And ROP is far from unique or special in this regard. The entire cybernetic universe of processes and protocols is replete with accidental and supervenient "weird machines", unintentionally Turing-complete byproducts of their intentionally designed substrates. From programmable ELF headers to javascript engine heap allocators, the technological environment bristles with computational potential that no one *intended* to extend to its various, vulnerable interfaces. I believe that the type of research conducted here, on the evolution of ROP chains, can be fruitfully extended to these other strange domains, and many more.[1] Very little is necessary, after all, to create an evolutionary system, and with a bit of care and craftsmanship, I believe that we can learn a great deal about what strange forms of computation can be bred in the crevices of our familiar, algorithmic abstractions.

Exploring some of this terrain using the techniques employed above, and extending them, is the first article of on my agenda of "future work", after this project. Even within the framework of ROPER, however, there are a number of interesting open problems and refinements to be explored, and I will use the remainder of this chapter to document a few of the most pressing.

[1] For an ongoing compendium of "weird machines" and their exploitation by the hackers who love them, I direct the reader to the pages of *PoC||GTFO* [2].

6.2 Futher experiments and modifications suggested by above results

The reader will have noticed that Chapter 5 raised at least as many questions as it answered, and that only a fraction of them were dealt with directly in that chapter.

6.2.1 Adding a TTL field to clumps to contain malignancy and promote novelty

This was touched on in Section 5.3.3.2.3, in response to an interesting case of a malignant gene being aggressively promoted by selection, in virtue of the beneficial trait with which its malignancy was indissociably bound. It's the sort of problem to which evolution systems whose components are semantically complex and multivalent are prone. Attempting to regulate it by tinkering with the weight of a single, scalar fitness value involved tradeoffs whose merit was difficult to anticipate in advance[2], and so another approach, orthogonal to the fitness mechanism, might be worth exploring. It's implementation seems fairly straightforward, and it will be one of the first experiments I perform with the system when time allows.

6.2.2 Collecting comprehensive genealogical data on the population during runtime

Frequently, in my analysis of various peculiar behaviours in ROPER's populations, in Chapter 5, my enquiry was cut short by a simple lack of data. It would be interesting to know, for example, *just how often* a fit chain descends from a crashing ancestor – information that could be used in calibrating an optimal crashing penalty or TTL, for instance, and which would be very interesting in its own right. There's nothing difficult in principle in collecting such data, but doing so efficiently will be an interesting optimization challenge.

6.2.3 Refactoring and optimization

I have a tendency to take new and ambitious projects as an opportunity to learn a new language, and in the case of ROPER, this was Rust. Unfortunately, this means that

[2]It's possible that using a vectorial fitness value, and applying an algorithm like Pareto optimization, may be another way of sidestepping this problem, but that too remains to be explored in the context of ROPER.

the entire codebase is *rife* with rookie cruft, and drowning in technical debt. The system needs a complete rewrite, and one made with polymorphism, extensibility, and optimization in mind. This will let future experiments be performed faster, and future modifications easier to make, as new ideas present themselves. The current state of the system has left the code difficult to reason about, and cumbersome to modify. It is very much a first draft, and a beginner's project. It was a joy to write, but I think it has long since outgrown its current form, and very much deserves an overhaul – one that includes unit tests and documentation, to boot.

6.2.4 The Snake game, and other interactive problem spaces

The experiment touched on in 5.3.4 remained largely tangential, and it cries out for completion and more careful study. This, too, is on the agenda.

6.2.5 Testing ROPER's payloads on fully realized systems

In the experiments discussed above, we never went beyond studying the behaviour of ROPER's populations *in vitro*, trapped in their reasonably complete, but still somewhat simplified and abstract emulated environment. It would be interesting to see whether ROPER can evolve specimens that can be used in a "real world" scenario – testing them, at the very least, against a full-fledged QEMU instance of a router, for example, with an actual RCE vulnerability as their point of entry. This would mean *much* slower evolutionary cycles, of course, but perhaps a pipeline could be set up between the Unicorn "nursery" where the chains evolve until they reach a certain threshold of fitness, and a more fully realized VM where their *in vivo* viability is put to the test.

6.3 Broader strokes

On the back burner, at present, sits a side project where I had begun to rewrite ROPER from scratch, before realizing that this was perhaps a bit too ambitious, given the time constraints of the thesis. This overhaul was initiated with an eye to what I took to be a handful of serious limitations in ROPER's design (some of which are, at the same time, items of interest peculiar to ROPER):

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.

6.3.1 "ROPUSH" or "ROPER II"

A design idea for the second iteration of ROPER was spurred by an interesting suggestion made by Lee Spector, in a discussion of some of my preliminary results and challenges at GECCO 2017 (where I presented [15]). Spector suggested that the opacity and brittleness that I was grappling with in "ROPER I" 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 – 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.

At present, an initial implementation of a PUSH-style virtual machine, designed for building ROP chains, that would then run on the Unicorn emulator – for which I wrote a basic library of Common Lisp bindings – sits, like I said, on the back burner of the current ROPER git repository. It would be interesting to see this through to completion, when time allows.

6.3.2 A return to Q, through the Binary Analysis Platform

I've also become increasingly interested, lately, with the domain of semantic binary analysis, and have been giving some though to switching from Unicorn to BAP as my ROP chain evaluation framework. Here, I would be following in the footsteps of *Q* [**?**], one of the projects that had first inspired me to work in this problem space. I'm interested in seeing where a marriage between *Q*'s semantic-analysis driven compiler of programs for the ROVM, and ROPER's evolutionary approach could take things. Having a richer source of precise, semantic information to nudge the evolutionary process through local hill-climbing searches, and a meaningful type system for gadgets that could be used in fostering homologous crossovers and saner mutations, could be an extraordinarily rich vein to mine. If ROPER II ever sees the light of day, it will probably try to move in this direction as well, using BAP to extract type information and mediate evolutionary synthesis with intelligent semantic analysis, with a highly flexible and evolvable, type-aware ROPUSH abstraction layer coordinating those activities.

Until then, I will tend to my ROPs.

Bibliography

[1] tomato-rt-n18u binary firmware blob. Version 3.5-140.

[2] International journal of proof-of-concept or get the fuck out (poc||gtfo), 2013 – 2017.

[3] Sperl Thomas (aka Second Part to Hell). Artificial evolution in native x86 systems. Retrieved from the author's website., 2010.

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

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

[6] Sergey Bratus. What are weird machines?

[7] Sergey Bratus. Offense and defense: Notes on the shape of the beast. presented at BSides Knoxville, June 2016.

[8] Luke Burns. Faq: The "snake fight" portion of your thesis defense.

[9] Fred Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1985.

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

[11] 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.

[12] R.A. Fisher. Iris plants database.

[13] 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.

[14] Olivia Lucca Fraser. Return oriented programme evolution with roper. Atlantic Security Conference (AtlSecCon), 2017.

[15] Olivia Lucca Fraser, Nur Zincir-Heywood, Malcolm Heywood, and John T. Jacobs. Return-oriented programme evolution with roper: A proof of concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1447–1454, New York, NY, USA, 2017. ACM.

[16] William Gibson and Tom Maddox. Kill switch, Feb 1998.

[17] J.B.S. Haldane. Disease and evolution. *La ricerca scientifica Suppl.*, 19(68).

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

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

[20] D. Iliopoulos, C. Adami, and P. Szor. Darwin inside the machines: Malware evolution and the consequences for computer security. 2011.

[21] Tim Kornau. PhD thesis, 2009.

[22] Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Julie Beaulieu, Peter J. Bentley, Samuel Bernard, Guillaume Belson, David M. Bryson, Nick Cheney, Antoine Cully, Stephane Donciuex, Fred C. Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Frénoy, Christian Gagneé, Leni Le Goff, Laura M. Grabowski, Babak Hodjat, Laurent Keller, Carole Knibbe, Peter Krcah, Richard E. Lenski, Hod Lipson, Robert MacCurdy, Carlos Maestre, Risto Miikkulainen, Sara Mitri, David E. Moriarty, Jean-Baptiste Mouret, Anh Nguyen, Charles Ofria, Marc Parizeau, David Parsons, Robert T. Pennock, William F. Punch, Thomas S. Ray, Marc Schoenauer, Eric Shulte, Karl Sims, Kenneth O. Stanley, François Taddei, Danesh Tarapore, Simon Thibault, Westley Weimer, Richard Watson, and Jason Yosinksi. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *CoRR*, 2018.

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

[24] Sergey Bratus Meredith Patterson and Len Sassaman. A patch for postel's robustness principle. *Secure Systems*, (March/April), 2012.

[25] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[26] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn: Next generation cpu emulator framework. http//www.unicorn-engine.org/BHUSA2015-unicorn.pdf, 2015.

[27] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn cpu emulator framework. http//www.unicorn-engine.org/, 2015–2018.

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[29] R.E Smith, S. Forrest, and A.S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1:127–149, 1992.

[30] Second Part to Hell. Chomsky hierarchy and the word problem in code mutation. Retrieved from the author's website., 2008.

[31] Robert E. Keller W. Banzhaf, Peter Nordin and Frank D. Francone. *Genetic Programming: An Introduction.* Morgan Kaufmann, 1998.

[32] Hongjun Wu. *The Stream Cipher HC-128.* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008.