Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2022 | LIU-IDA/LITH-EX-A--22/085--SE

ROP-chain generation using Genetic Programming

– GENROP

ROP-kedjegenerering genom Genetisk Programmering

Jonatan Branting

Supervisor : Ulf Kargén
Examiner : Nahid Shahmehri

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Return Oriented Programming (ROP) is the de-facto technique used to exploit most of today's native-code vulnerabilities hiding in old and newly developed software alike. By reusing bits and pieces of already existing code (gadgets), ROP can be used to bypass the ever-present Write ⊕ eXecute (W⊕X) security feature, which enforces memory to only be marked as *either* executable *or* writable; never both at the same time. Even with its widespread use, crafting more advanced ROP-chains is mostly left as a manual task. This paper attempts to explore the viability of automating ROP-chain generation by leveraging *genetic programming* (GP), and describes the implementation and design of the ROP-compiler GENROP in this endeavour. We introduce a novel approach to adapt GP to work within the environment of ROP, which attempts to guide the algorithm and preemptively remove pathways which are known ahead of time to be unable to generate a solution. GENROP is tested by attempting to generate a working payload against a number of binaries, and is then evaluated based on success rate and payload size when compared to *angrop* (another ROP-compiler). The results show that the algorithm is able to generate functioning payloads in most of the tested cases, although it does perform worse than angrop. This can partly be explained by the fact that GENROP uses gadget definitions generated by angrop, which reduces the potential viability of the ROP-compiler, as more unwieldy but potentially usable gadgets are not available. Additionally, it was found that extensively guiding the algorithm has negative consequences in terms of solution diversity. Relying on faster execution times and more iterations might produce better results. Further work is required to assess whether or not generating ROP-chains using genetic programming is a viable approach.

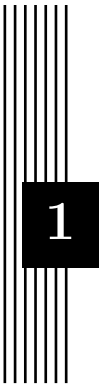
Contents

List of Figures

[1] This exploit only works if the memory is zeroed when not used, as otherwise `/bin/sh` will not be null-terminated.

Part I

Introduction



1 Introduction

Return Oriented Programming (ROP) is the de-facto technique used to exploit most of today's native-code vulnerabilities hiding in old and newly developed software alike. To be more precise, ROP is often used to bypass the ever-present Write ⊕ eXecute (W⊕X) security feature, which enforces memory to be marked as either executable *or* writable; never both at the same time. This feature hinders the more classic buffer overflow code injection attacks as they rely on being able to write machine code to memory, after which it is executed. To attack a system protected by W⊕X, you would either have to use an exploit which does not rely on injecting code, or you would have to disable this protection somehow. In some cases both options would be part of a multi-stage attack. ROP can be used in both of these stages, as it does not rely on injecting code, while potentially being a tool which might be used to disable W⊕X.

Even with its widespread use, crafting more advanced ROP-chains is mostly left as a manual task, regardless of the many attempts at developing automatic solutions. This paper attempts to explore the viability of automating ROP-chain generation by leveraging *genetic programming* (GP), and introduces the ROP-compiler GENROP in this endeavour.

1.1 Background

This section details background information intended to introduce some of the concepts most important to the research questions posed in this thesis.

1.1.1 Function Calls

Function calls are implemented on the machine-code level by issuing a special call-instruction that pushes the return address (i.e., the address of the instruction directly following the call instruction) to a region of memory known as the stack. When the called function returns, the saved return address is popped from the stack into the instruction-pointer register, resuming the execution of the calling function.

1.1.2 Return Oriented Programming

Return oriented programming is in essence a control flow hijacking attack. The ROP environment depends heavily upon the state of the CPU and memory at the time of the hijack,

and can be described as an emergent domain: it is not designed for the task, but emerges out of another process. It is a perfect example of a "weird machine", a term coined by Bratus [5] to describe code execution happening outside the original program specification as a result of computational artifacts following compilation.

The instructions available in the ROP environment consists of bits and pieces of the executable region of the targeted process' memory, and are called *gadgets*.

Crafting a non-trivial ROP-chain is however both challenging and time consuming, and the instruction set available does not lend itself to human programmers very well; there is a very good reason modern programming languages are as high-level as they are. Writing ROP-chains is akin to writing assembly with an obfuscated and randomly limited instruction set.

1.1.3 Previous Approaches

Due to the difficulties present in developing ROP-chains by hand, automating the process is an attractive idea. Multiple ideas and implementations have sprung up over the past years, with varying degree of success. [10, 25, 35, 31, 11, 36, 12, 33]. These tools all utilize the semantic information about each gadget, i.e. their effects on the state, coupled with constraint solvers or predetermined "recipes" to generate ROP-programs, an approach which works quite well in most cases and can reliably generate functioning *payloads*[1] for a given binary file.

An example of a ROP-compiler leveraging semantic analysis is Q [10], which achieves an 80% success rate for payload generation against binaries in `/usr/bin/`[2] programs larger than 20KB.

All of these ROP-compilers are however limited by the fact that these tools can only utilize gadgets and pathways that the compiler's authors explicitly implemented. An example of this is Q's lack of support for using the stack as an intermediate storage space, [35], and angrop's lack of support for working around the otherwise problematic `leave`[3]instruction. [35] While this is not inherently a flaw (you cannot expect the machine to do something it has not been instructed to do), it does raise an interesting question: how explicitly do you actually have to express yourself in this domain to get satisfying results?

In an effort to shed some light on this topic, Fraser set out to design a system called ROPER [13] which implemented *genetic programming* to evolve ROP-chains with various purposes for the ARM architecture.

1.1.4 Genetic Programming

Genetic programming (GP) is a metaheuristic evolutionary algorithm, with which computer programs are encoded with a set of genes, and then evolved using genetic algorithms (GA). The idea is that given the right selective pressures and mutations, a program which is "good enough" for its purposes — it solves the given problem, although not necessarily optimally — can be generated through this method. It is a method well suited for problems which are hard to reason about analytically, where methods that produce optimal solutions, such as constraint solvers, may not be able to find a solution within a reasonable amount of time. It might also be used as a way of more efficiently finding an answer to a given problem, if an approximation is all you need.

The reasons why GP is attractive as a method for generating ROP-chains are multiple:

1. It moves the problem from one domain to another: instead of having to deal with developing support for each possible behaviour and route a ROP-chain could take, we can

[1] A "payload" is a term commonly used to refer to malicious code.

[2] /usr/bin is a standard directory in most Unix-like operating systems such as Linux, which contains most of the executable files on the system.

[3] The `leave` instruction frees part of the stack which was allocated for a given function call, which has to be taken into consideration when crafting the ROP-chain, as part of that stack that is freed is not usable.

take inspiration from nature. This might allow us to reduce the solution to a specification of the problem and let selective pressure take its natural course towards a conclusion.

2. As the method is not analytical in the same sense as conventional methods of generating ROP-chains, GP might produce chains that does not look like ordinary ROP-chains, which could allow a ROP-chain generated by GP to dodge multiple defensive measures designed to pattern-match ROP-chains based on their behaviour.

3. It might produce ROP-chains which a conventional method could not find, allowing the method to produce ROP-chains for problems which otherwise would require a human analyst.

To underline reason (1): a quick look at the source code example of the angrop ROP-compiler [8] reveals multiple comments indicating lacking support for possible paths to working chains, such as the previously mentioned missing handling of the `leave` operation. By utilizing an evolutionary algorithm, we can leave this task to the computer, and instead focus on describing the problem as best we can. We are banking our luck on the chance that the sought after behaviour will emerge "naturally", without us having to explicitly implement support for it. In simple terms, this allows us to tell the computer that we want a problem solved, without having to tell it exactly how.

According to Koza [19], genetic programming very rarely produces *the* correct solution to a problem, but will in most cases supply an approximation not far from that. In the environment of ROP, there is rarely one single solution to a problem. There are however multiple *specific* solutions, and getting close to one of these solutions is simply not good enough, one specific solution will *have* to be found. Whether or not this provides enough leeway for a genetic programming system to properly work is likely highly dependant on the complexity of a given problem. Whether or not genetic programming is a viable approach for solving these kind of problems is what we will be attempting to answer in this work.

1.2 Aim

The aim of this thesis is to explore the viability of genetic programming in the domain of ROP-program compilation, by designing and implementing a ROP-compiler which heavily leverages genetic programming.

The result will not be a state-of-the-art ROP-compiler, but rather a proof-of-concept exploring the viability of using genetic programming to generate ROP-programs, blazing the trail first scouted by Fraser's ROPER.

1.3 Research Questions

The questions this thesis will attempt to answer are listed point-by-point below:

1. What are the challenges inherent in generating *ROP-chains* using genetic programming?

To help answer this question, we will be implementing a rudimentary ROP-compiler based upon GP with basic capabilities, in addition to conducting a literature study. This requires answering an additional question:

2. How do we adapt the genetic programming algorithm to work in a ROP environment?

In essence, this divides the thesis into two parts, first we will identify the challenges we are facing in attempting to implement a ROP-compiler using genetic programming, and subsequently we will be proposing solutions to these challenges or problems.

1.4 Delimitations

The research in this thesis is limited by the following:

1. We will only be targeting and generating ROP-chains for the x86 platform.

Part II

Theory

2 Theory: Return Oriented Programming

Theory pertaining to the intricacies of binary file analysis and ROP.

2.1 Binary Exploitation In General

Most of today's binary exploitation is a multi-step process consisting of getting past multiple security measures put in place over the years, which include using:

(1) Information leakage to bypass ASLR[1], for example using *Format String* exploits

(2) ROP to bypass W⊕X by calling e.g. `mprotect`

(3) *Shellcode* injection, to execute the actual payload

In this sequence of exploits, (2) is often the most time consuming and troublesome step. The focus of this paper is to generate ROP payloads, and as such we will not go into much detail regarding step (1) and (3). As such, we will not go in depth on topics which are not required to understand ROP and its underlying concepts.

2.1.1 Buffer Overflow and Shellcode Injection

A buffer overflow is a very common vulnerability where, while writing data to a buffer, the buffer is overrun and memory adjacent to this buffer is written to as well. The cause is often improper memory management or usage of unsafe methods.

An example of a vulnerable C program (or more specifically a single function):

[1] Address Space Layout Randomization (ASLR) is a defensive mechanism employed by most modern operating systems and is enforced by hardware, which randomizes process key memory locations such as the stack, heap and base of the executable and its imported libraries. This makes it harder to successfully inject a working payload, barring any form of *information leakage*, which could reveal where the attack has to perform the injection of the malicious payload.

```c
#include <stdio.h>
void vuln(){
  char buf[16];
  return gets(buf);
}
```

The `stdio` function `gets` does not check for length, and will write whatever input it receives to the buffer provided, even if this will cause an overflow. This vulnerability will then allow an attacker to potentially take complete control over the computer, via a shellcode injection or a more complex exploit if necessary.

A classic exploit which makes use of a buffer overflow is a *shellcode injection*. In such an exploit, the attacker attempts to inject a payload, i.e. a piece of machine code[2], somewhere in the process' memory, and then redirect the control flow of the program to execute the injected code.

Given the program defined earlier: if we input a string longer than 16 bytes, we will start to overflow the buffer, writing to memory adjacent to that of the buffer. Local variables — such as the buffer we are overflowing — are stored on the stack, where the return address also lies, allowing it to be quite easily overwritten.

To take advantage of this, we can in this case input a string with the format. `A*40 + <address of start of shellcode>`, which will cause the return address to be overwritten by the address of the start of the shellcode. When the function attempts to dereference the return pointer to resume execution at the calling function, the control flow of the program will be redirected to the injected shellcode, and the attacker's payload will have been executed.

2.2 Defensive Mechanisms

In this section we list typical defensive measures and mechanisms employed by modern processors and operating systems. The active defensive mechanisms of a system dictate which type of exploits will be required to take advantage of a given vulnerability. For example, if W⊕X is enabled for a certain binary file, chances are we must make use of ROP (or another similar method) to get past that security measure.

2.2.1 W⊕X and DEP

Write XOR eXecute W⊕X is a security mechanism employed by all modern CPU architectures, which enforces each page in memory to be either marked as executable *or* writable; never both at the same time.

The protection is called W⊕X in Linux and Data Execution Prevention (DEP) in Windows.

This feature hinders the more classic buffer overflow code injection attacks as they rely on being able to write the actual machine code to memory, after which it is executed. To get around W⊕X, an attacker would have to either employ some other exploit to disable such protections, or bypass it entirely using *code reuse* attacks — such as ROP —, as code reuse attacks will completely side-step the W⊕X, since no machine code needs to be injected.

Disabling W⊕X through ROP entails executing the system call `mprotect`.[3] This allow programmers to change the permissions of a page in memory, and are mostly used to allow for generating code at runtime, for e.g. interpreted languages. [11]

[2] That typically spawns a shell, giving the attacker access to the victim's machine. This is where the name *shellcode* stems from.

[3] In Linux systems, specifically.

```
""" How a set of gadgets suited for
executing an execv or mprotect attack
could look like.  """

pop_eax = pop eax; ret;
pop_ebx = pop ebx; ret;
pop_ecx = pop ecx; ret;
pop_edx = pop edx; ret;
write_what_where = mov eax, [edx]; ret;
zero_edx = xor edx, edx; ret;
syscall = int 0x80; ret;
```

Figure 2.1: A typical list of gadgets used to implement an `execv` exploit.

2.3 Binary Analysis

This section details binary analysis, which analyses the raw binary of an application to assess vulnerabilities and potential threats.

2.3.1 Dynamic vs Static Analysis

During static analysis, a program is analysed without executing it, and a model of the program is developed. In the case of statically analysing binary programs (i.e. compiled code), this can be a challenging task. In the effort to build an accurate model of the program and its control flow graph each branching instruction might take arguments from both memory addresses and registers, each respectively being affected by other parts of the program and its inputs. Due to this and other factors, completely accurate models are mostly not achieved by available tools. [17]

Dynamic analysis, on the other hand, attempts to paint a picture of how the program will function by analysing the program *as* it runs, which can allow the analyser to record execution paths and behaviour of the program.

2.4 Return Oriented Programming — In Depth

This section goes details ROP, and how it works, in depth.

2.4.1 Basic ROP-chain Execution Example

As mentioned previously, in Chapter 1, the instructions available to a ROP programmer consists of artifacts in the executable region of the targeted process' memory, following compilation. These instructions are called gadgets, and are conventionally limited by:

1. Being mapped to the executable part of memory for the targeted process.
2. Being terminated by a return operation or something semantically equivalent.

As ROP operates upon the underlying mechanics on which the original program ran, which relies on storing the return address on the stack to maintain control flow during function calls, we can control the flow of the program by controlling the values written to the stack.

Keeping limitation #2 in mind, a ROP-loop is executed as follows (and is showcased in Figure 2.2.):

1. A ROP-chain, i.e. a sequence of gadget addresses, is written to the stack. The first address of the chain overwrites the stored return address.

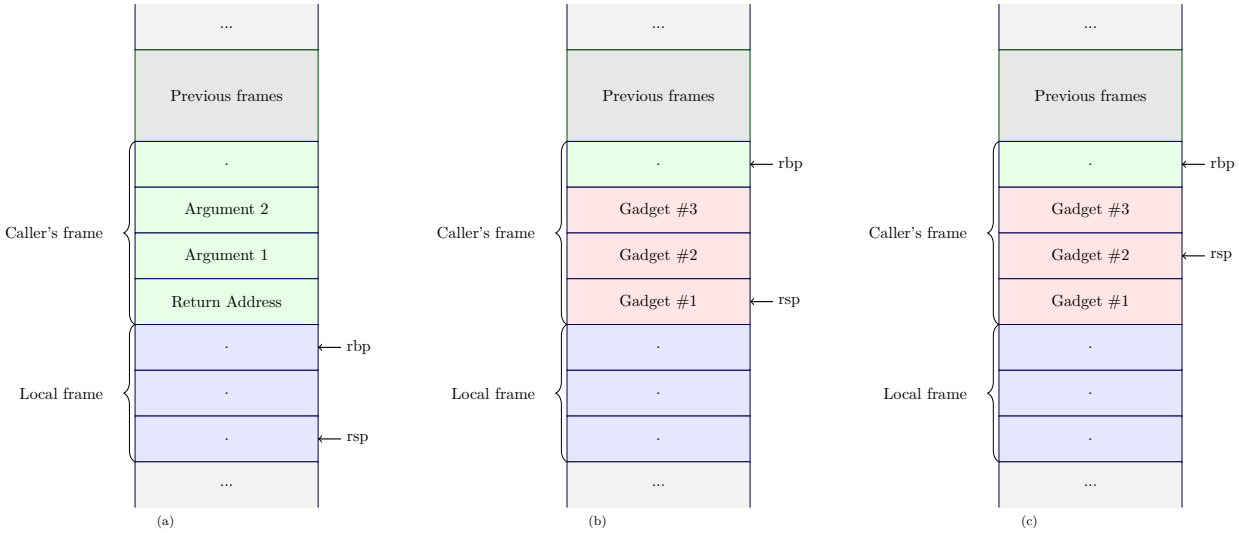


Figure 2.2: Example of a ROP-chain with three gadgets executing on x86.

2. The currently running function completes and a return instruction is issued. As a result, the return address (which now contains the address to the gadget that is next in the ROP-chain) is popped into the instruction pointer register`rip`.

3. The gadget is subsequently executed.

4. The gadget finishes execution, in turn executing a return instruction, which will transfer execution to the next gadget in the chain. Return to step 2.

2.4.2 Example `execv` payload

ROP is probably the most known *code reuse* attack, which, as its name might allude to, reuses pieces of a binaries which already exists, to execute something other than what the original author intended.

A very typical and standard ROP-exploit, given that enough gadgets are available, is an *execv* system call attack. The `execv(const char * filename, char const to * argv[])` system call is a function which executes the program specified by the *filename* parameter. As such, it can be used to execute a number of programs, most commonly the `/bin/sh` program, granting full control of the machine.[4]

```
eax = 0xb
ebx = <address in memory of the string "/bin/sh">
ecx = <address of a pointer pointing to the string "/bin/sh">
edx = null
```

Listing 1: CPU-state required for reaching shell-access using an `execv`-attack.

The goal of this attack is to set the CPU in the state described in Listing 1. Here, the value contained by `eax`, i.e. `0xb` corresponds to the `execv` system calls on 32-bit Linux platforms,

[4]More specifically, this grants as much control of the machine as the user had which ran the service which was exploited.

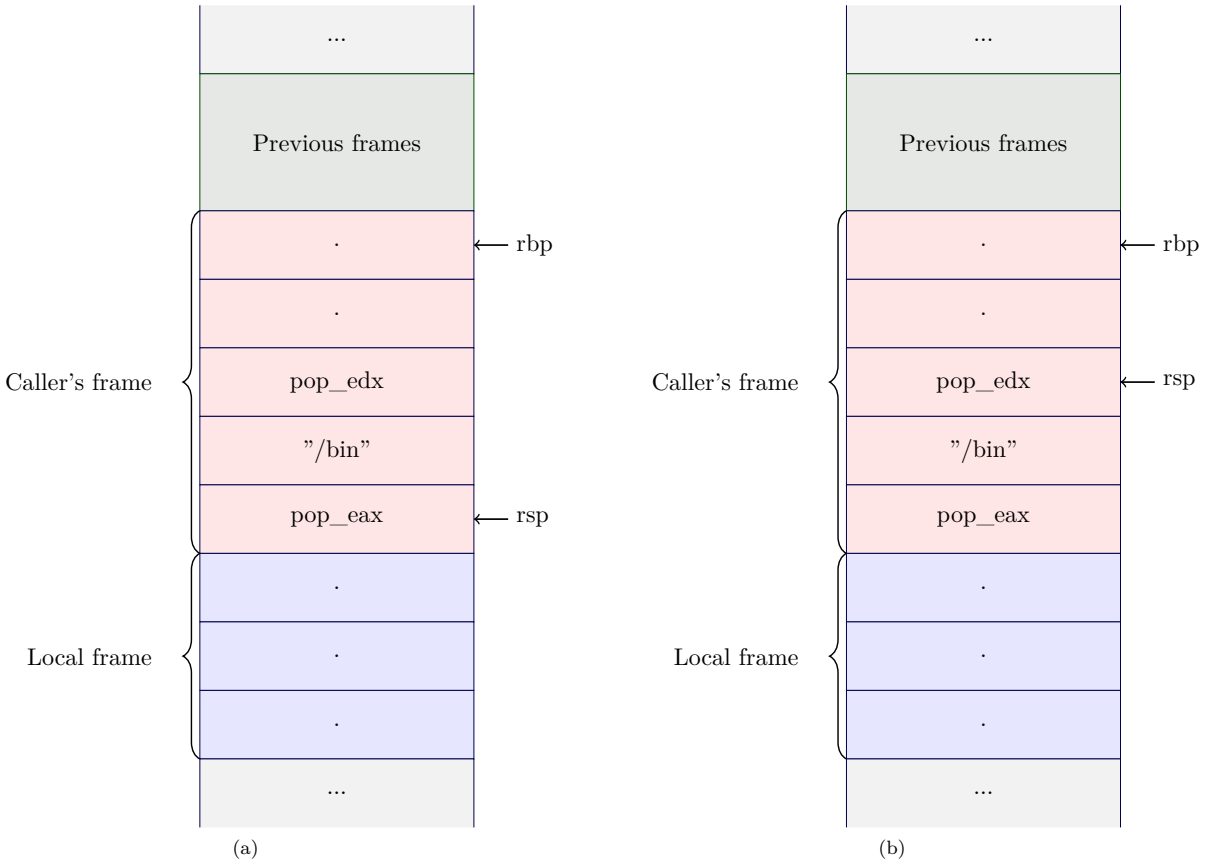


Figure 2.3: Example of the first gadget execution of the `execv`-payload.

and differs from architecture to architecture and kernel to kernel. An example gadget execution is visualized in 2.3.[5]

Given that the relevant gadgets are available, such as the gadgets displayed in Figure 2.1, the ROP-chain could be implemented as in Figure 2.4.

As can be observed from this example, even with the best of gadgets available, the payload is still quite complex. Although, it should be noted that, if the string `"/bin/sh"` is already defined in memory, this attack becomes much easier to execute, as a lot of the complexity comes from actually writing the string to memory in the first place.

[5]It might be interesting to note some of the registers available in x86 and their purposes. To this purpose a non-exhaustive list of registers follow:

1. accumulator register `ax`, is used in artithmetic operations.
2. counter register `cx`, used in loops and shift/rotate instructions.
3. data register `dx`, used during I/O arithmethic operations.
4. base register `bx`, used as a pointer to data.
5. stack pointer register `sp`, points to the top of the stack.
6. stack base pointer register `bp`, points to the bottom of the stack.
7. instruction pointer `ip`, points to the next instruction to be executed.

Note that the register names are prefix with either an *e* in 32-bit mode or an *r* in 64-bit mode.

```
"""
In this example we are generating a string payload, where each gadget is
followed by its arguments.

When this is properly injected and executed, the CPU would pop each
gadget address into the instruction pointer, which would start the
executing of that gadget. This would in turn pop their corresponding
arguments off the stack.

Once a gadget has finished execution, it would `return` to the next
gadget and the chain would continue.

"""

# `write_mem` points to a place in memory where we want to write "/bin/sh"

payload = ""
payload += padding

# Write "/bin" using the `write_what_where` gadget
payload += pop_eax
payload += "/bin"
payload += pop_edx
payload += p32(write_mem)
payload += write_what_where

# Write "/sh" using the `write_what_where` gadget
payload += pop_eax
payload += "//sh" # 4 bytes, "/" is ignored by system-calls
payload += pop_edx
payload += p32(write_mem + 4)
payload += write_what_where

# Write address of `write_mem` just after the "/bin/sh"-string itself
payload += pop_eax
payload += write_mem
payload += pop_edx
payload += p32(write_mem + 12) # We need to include null-termination of "/bin/sh"
payload += write_what_where

# Set registers to properly indicate what we want `syscall` to do
payload += pop_eax # Indicates that we want to execute `execv`
payload += p32(0xB)
payload += pop_ebx # Where does the string "/bin/sh" start in memory
payload += p32(write_mem)
payload += pop_ecx
payload += p32(write_mem + 12) # Where does the the address of the string "/bin/sh" start
↪  in memory?
payload += zero_edx

# Issue syscall, to actually execute whether the CPU-state implies
payload += syscall

```

Figure 2.4: A typical `execv` ROP payload implemented in Python using gadgets defined in Figure 2.1, executed on a x86 32-bit platform. [6]

2.4.3 How a Gadget Is Born

As ROP-programs function on the bits and pieces of code meant to run the underlying program, the instruction sets (which differ from binary file to binary file) can become extremely unwieldy. The original compiler had access to each and every instruction by themselves, without side effects, it could generate very effective programs using exactly the instructions and

their corresponding arguments required to execute what the author of the program had in mind.

The subsequent ROP-compiler, on the other hand, has access only to the instructions which the original compiler generated, and only in that specific order. As the function of the ROP-program is in most cases completely different from that of the original program, this can causes a few troubling issues.

As the developer of a ROP-chain, you will be presented with a list of "instructions", with a wildly varying number of effects, and side effects alike, depending what you want to accomplish with said gadgets.

```
pop rbx
and al, 8
mov eax, dword ptr [rax]
mov dword ptr [rsp + 0x10], eax
ret
```

Listing 2: A basic block, which can easily be analyzed.

An example is the so-called *basic block* visualized in Listing 2; a block of code with no jumps or indirections, allowing it to be easily analyzed.

This block can be used in four different ways: as we can freely choose where to jump, we can point the CPU towards any line in the block. This means that we can from this block extract the gadget `mov dword ptr [rsp + 0x10], eax; ret;`, which would execute that single instruction without side effects. If we really want the `pop rbx` instruction however, we need to go through all instruction until the `ret` instruction is issued, leaving us with a number of side effects, such as overwriting `rax`.

Effectively using gadgets which do *not* make up basic blocks (e.g. gadgets which conditionally jumps to different locations in memory), is an interesting field of study, as it would allow the usage of many more gadgets than what is typically focused on today when automatically generating ROP-chains.

2.4.4 Register Clobbering

Register clobbering is the term used to describe the events in the example above. For example, if we have the following gadget: `pop rax; mov qword ptr [rsp + 0x60], rbx; add rsp, 0x48; ret;`, and we want to utilize it for its `pop rax` instruction, we cannot avoid modifying `rsp` and `rbx` as well. In this case, we have therefore *clobbered* `rsp` and `rbx`.

2.4.5 Missing Functions/Instructions

In many cases, compilers designed to generate ROP-chains will be left without many rather crucial functions. For example, if you want to set `rax` to a specific value, it is very convenient to have access to a `pop rax` gadget. This is not always the case, which means that you have to take another route to achieve your goal, by e.g. utilizing a `mov <from> <to>` gadget, by first setting `<from>` to the target value, and then executing the `mov` gadget.

2.4.6 Illegal Memory Accesses

Due to the unwieldy instruction set available, almost all practical applications of ROP will results in the programmer or tool having to handle a lot of side effects, either through a very explicit execution order, or by more advanced means, in the more difficult cases.

A common side effect is reading memory defined by an address either in memory or in a register. If that address is set something that is *not* accessible by the process, executing the corresponding gadget will cause a so-called segmentation fault, and the program will crash.

It is possible to work around this by taking note of this memory dereference, and controlling this address intentionally, so that it points to something the process *does* have access to.

2.4.7 Payload Size Limitations

One of the reasons the ROPC tool is unusable for practical applications is the fact that it generates very large payloads. This is a problem for two reasons:

1. The entry point to the attack that was used might disallow payloads larger than a certain size.

2. As you effectively overwrite memory that is being used, a too large payload could mean that you are overwriting crucial parts of the program, disallowing the program to cleanly shut down after the exploit has run its course, or it could mean that you are starting to write to unavailable memory, which would result in a segmentation fault.

2.4.8 Other Hoops to Jump Through

There are a number of other problems which can arise due to the nature of the environment in which the ROP-chain is executed in, as well as the instruction set it is built of. To effectively generate exploits for any binary, a potentially very large number of edge cases have to be considered.

A practical example of this is correctly handling the `leave` instruction [7], which is definitely possible, but would require the programmer or tool to modify the payload explicitly to handle this instruction. For reference, this would require overwriting the new stack frame as well, to continue executing the ROP-chain after the `leave` instruction had executed in the chain.

Another not entirely obvious pathway, could be to utilize the stack as a scratchpad to store intermediate values which are being juggled, which is something angrop [33], amongst others, utilizes.

Many tools quickly give up trying to find a solution if not enough ”obvious” gadgets are available, as the pathway to a solution becomes to complex and far fetched.

[7] The `leave` instruction is equivalent to `mov ebp, esp; pop ebp`

3 Theory: Genetic Programming

Theory detailing the workings of genetic programming.

3.1 Genetic Algorithms Core Principles and Ideas

Genetic algorithms (GA) are a family of domain-independent algorithms which are inspired by natural evolution and utilize selective pressure in an attempt generate solutions to a multitude of problems such as search and optimization problems.

Genetic programming is a subset of GAs, with the goal of evolving *programs* themselves, which are "good enough" at solving a specific task.

As stated previously, in 1, these algorithms are not good at finding *the* correct solution, but rather they are good at approximating solutions, and excel where analytical solutions fail or are not applicable.

Typical applications for genetic programming are: [2]

1. Functions approximation given sample points.
2. Classification tasks.
3. Planning, or other scheduling problems.
4. Evolution of game-playing strategies.

Genetic programming is not the optimal way of solving these problems in most, or any, of these cases, instead, what makes genetic programming interesting is the fact that the same algorithm (although slightly adapted to the specific problem) is capable of solving *all* of these problems; theoretically, if you can properly model the problem, genetic algorithms can in many cases approximate a rather precise solution, without requiring much manual effort.

In essence, the goal of a genetic programming algorithm is to generate a program which meets certain criteria (is highly fit), from a *population* of less fit programs. Population is in this case is used to refer to the pool of programs currently being evaluated, and programs in this context are often referred to as *individuals*.

3.2 How A Genetic Programming Algorithm is Constructed

For genetic programming in particular, Koza [29] defines five preparatory steps — or variables to be defined — which needs to be executed before the genetic programming loop can take over, these include: [1]

1. The set of terminals for each branch of the to-be-evolved program.
2. The set of primitive functions for each branch of the to-be-evolved program.
3. The fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population).
4. Certain parameters for controlling the run.
5. The termination criterion and method for designating the result of the run.

Here, step 1–2 corresponds to the building blocks of the target program, namely the available functions and terminal values (e.g. concrete values and zero-argument functions). Step 3 corresponds to the fitness function measuring how well a given program performs.

Step 4 corresponds to the available genetic operators, namely: *selection, mutation and crossover*, and their respective variables, which controls how often the operators are applied to the population. Another crucial variable to correctly specify is that of the population size p, as this can greatly affect the performance of the algorithm. A rule of thumb is that more complex problems require larger populations and vice versa.

3.3 Genetic Operators

A genetic operator is the typical name of the functions applied on the individuals of a population in a GA. This include operators such as the selection operators, which corresponds to which scheme is used to select pairs for mating (in the sense of passing on their genes, i.e. reproduction.), the crossover operator, which corresponds to *how* parents mate (i.e. how are their genes combined to generate their offspring), and the mutation operator, which corresponds to how individuals change slightly by themselves throughout the generations, irrespective of reproduction.

3.3.1 Selection Operators

When discussing selection strategies for GAs, it is referring to how mating pairs are selected, i.e. which individuals will be passing their genes on through the crossover operator. As the selection operator itself does not differ much between GP variants, a small set of the selection rules that have been used and studied in literature will be directly listed here.

Tournament Selection

A very commonly used selection scheme is *tournament selection*, where n individuals are chosen to participate in a tournament. Each participant is randomly chosen weighted by their fitness score. This is repeated k times, where k is the number of individuals chosen for crossover.

Tournament selection is probably the most used selection operator, due to its ease of implementation and the perk that it allows for easy adjustments to the selection pressure. Additionally, it has been proven to be independent of the scaling of the fitness function, which is a very attractive characteristic when exploring potential functions to use for a problem where the fitness function might not be obvious. [16, 39]

[1] Taken directly from *A Genetic Programming Tutorial* by Koza et al. [29]

Elitism and Deterministic Selection

Elitism is a term for always keeping the fittest individuals alive to partake in the next generation. This has been shown to increase the performance of a GA significantly in some cases, [40, 30] and has also been proven to reduce *bloat* [28], which is a topic which that will be detailed further down. In "A universal eclectic genetic algorithm for constrained optimization" [23, 20], elitism coupled with a deterministic selection scheme was implemented, where high performing individuals always breed with a low performing individual to maintain diversity.

If elitism is implemented without any form of interbreeding with worse performing individuals, or another form of tactic for maintaining diversity, it can in many cases lead to premature convergence. This happens because too many individuals similar to the few strong individuals that survive each season are generated. In other words, the population becomes too homogeneous to effectively continue evolving.

3.3.2 The Crossover Operator

During the crossover step in a GA, two or more individuals are chosen from the mating pool to "give birth" to a new individual or in some cases, individuals in plural. The idea is that if we recombine parts of fit individuals, we can similarly to natural selection generate better solutions as time go by. How crossover is performed is dependent upon the implementation of the system as well as how the individuals are *encoded*, i.e. how the problem is represented. This is further detailed in Section 3.4.

In essence, crossover strategies should be selected with the *Building Block Hypothesis* (BBH) in mind, which is detailed in Section 3.6.1.

The percentage of the population which is affected by crossover is controlled by the crossover rate variable c_r, with typical values ranging between 0.6 and 0.9. [19]

3.3.3 The Mutation Operator

The mutation operator is used mainly to introduce "new blood" into the population, and such, its function is mainly to stave of *premature convergence* (which is detailed in Section 3.6.4), as a completely random mutation rarely generates something fully functional directly. A mutation can be viewed as a source of new genes and gene-links, which can be very useful for the population down the line.

The mutation operator differs heavily from implementation to implementation, as it has to be implemented so that no non-functional individuals are generated, as well as make sense in the domain of the problem. A typical example of a mutation can be:

1. In the case of linear GP, a mutation can be a flipping one or more bits.
2. In the case of a tree-based GP, a mutation can be replacing a sub-tree with a new, randomly generated one.

Changing how these operators behave can have huge impacts on the performance of the genetic algorithm, as it is an important source of diversity, and correctly deciding upon how they modify the population is of utmost importance.

The percentage of the population which undergoes mutation is affected by mutation rate variable cm_r. Typical values for most problem is in the range of 0.01–0.05 according to Koza. [19]

3.4 Encoding/Variants of Genetic Programming

We will define a variant of GP as that of variations of how an individual is represented, i.e. how are instructions and potential arguments encoded, which affects how these can be modified when affected by genetic operators.

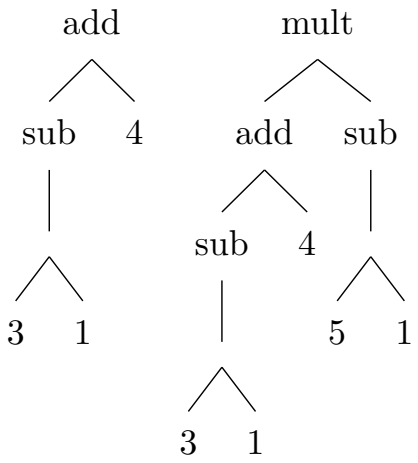


Figure 3.1: Simple example of the tree-based structure used for tree-based GP, and Lisp

```
input/   # gets an input from user and saves it to register F
0/       # sets register I = 0
save/    # saves content of F into data vector D[I] (i.e. D[0] := F)
input/   # gets another input, saves to F
add/     # adds to F current data pointed to by I (i.e. D[0] := F)
output/. # outputs result from F
```

Listing 3: Example of a program expressed in Slash/A syntax.

3.4.1 Tree-based GP

The most commonly used form of encoding individuals is tree-based GP, where each individual, and thus program, is encoded as a tree, as the name alludes to. [29, 19, 2, 3] Tree-based GP generates programs with a form similar to those expressed as *S-expressions*, commonly used in Lisp programming languages.

An example program encoded in this was is visualized in Figure 3.1.

Mutation and Crossover

In tree-based GP systems, crossover is often done by picking a crossover point in each parent tree, and combining these to form the offspring. The offspring will in this case be a merge between the two sub-trees extracted from its parents.

Mutation is often done by replacing a sub-tree with another, randomly generated, sub-tree. Which sub-tree is replaced is in many cases completely random, relying on purely natural selection, but can in other cases be weighted in some way to subtly guide the evolution.

3.4.2 Linear GP

As most computer programs are fundamentally linear, it makes sense intuitive sense to encode GP-programs the same way. The concept of encoding programs in a linear fashion is easy to grasp, as each instruction will be executed in order from start to finish. An example program implemented in the programming language Slash/A is displayed in Listing 3.

An important note is that in Slash/A, every instruction is atomic and thus argument-less, which makes it very robust and suitable for usage with genetic programming, as every random sequence of instructions is a semantically correct program. [1]

Mutation and Crossover

For linear GP, crossover is often implemented similarly to how it is implemented in tree-based GP systems: as the individuals of the population are encoded as vectors, the offspring is the result of each bit of the first individual up until the *crossover point*, and each bit of the second individual from that point on. How the crossover point is chosen can differ, and in some cases this is a weighted point, chosen based on the fitness score of each individual, while in other cases this is statically set, always resulting in a half-half split.

Alternatives to 1-point crossover includes annular crossover, which defines as a start- and endpoint, and creates a cycle (i.e. after the last bit of a vector representing an individual, the following bit is the first of the same vector) of the genetic bits from each individual parent partaking in the mating. For example, the start- and endpoints could be defined as $[-1, 3]$, which would result in an offspring with all genetic bits from $[-1, 3]$ from the first parent and the rest from the second parent.

Mutations can vary wildly depending on problem domain, but typical mutation operators include modifications such as:

1. Permute the order of operations.
2. Add random operations.
3. Delete random operations.
4. Other problem specific mutations.

3.4.3 Stack-based GP

Stack-based GP is a super-set of tree-based GP, as the individuals (and thus the programs) can take the form of any possible digraph. It was originally proposed by Perkis [26], with the idea that evolving this form of programs would increase the flexibility of GP, and allow for programs which might not otherwise be explorable.

These GP systems function as a stack machine: the instructions and their arguments reside on the stack, and after a successful execution, the results of the operation is in turn pushed onto the stack. The individuals themselves are represented as a vector of instructions and corresponding arguments. [26]

Mutation and Crossover

For stack-based GP, the crossover and mutation operators are very similar to the operators typically used in linear-GP systems. The authors of "Stack-Based Genetic Programming," implemented a form of annular crossover for their version of stack-based GP, which is described in Section 3.4.2.

3.4.4 Graph GP

As trees are a type of graph, the representation of individuals in a graph GP system is similar to that of a tree-based GP-system, with the exception that the hierarchies between nodes can be intertwined. As such, the operators of tree-based GP and graph GP are similar, albeit with a few important differences.

Mutation and Crossover

A common crossover operator for graph GP is the Subgraph Active-Active Node (SAAN) crossover, as defined by its authors, and functions in the following manner: [27]

1. A random node (crossover point) is selected for each parent.

2. For parent 1, a sub-graph containing all nodes required to calculate the results of the selected node are extracted.

3. The crossover point, or node, in parent 2 is replaced by the nodes extracted in step 2.

There are numerous other variants of this, which is one one of the reasons graph-based GP is an interesting field of study: as the encoding is so powerful, its operators can be selected to fit each specific problem in a much broader sense when compared to other variants such as tree-based GP.

Mutations meanwhile include operations such as:

1. Generating a random sub-graph and inserting it randomly in an individual.

2. Replacing or redirecting links between nodes.

3.4.5 Linear-graph GP

A merge between linear and graph-based GP systems, linear-graph GP attempts to mimic the appearance, or form, of ordinary programs, as developed by humans. A normal procedural program (implemented in e.g. Python) is typically made up of functions containing a linear set of instructions. These functions are then used and reused in conjunction with higher order operators to create the wanted program. This can be thought of as a set of linear instructions connected throughout a graph, which is exactly the idea of linear-graph GP.

$$t = \{x,\ c \mid x \in \mathbb{N},\ c \in [a, b]\}$$

This approach outperformed linear GP variants by a wide margins in a small set of test cases, and performed well even with very small initial populations. [18].

Mutation and Crossover

For linear-graph GP, we are left with an interesting choice when it comes to what to do during the crossover operation: do we recombine the linear programs themselves, or do we simply merge sub-graphs, as in graph-based GP. The method used by Kantschik et al. was to do both, with different probabilities.

During linear crossover, one program from each parent is picked and recombined. This results in two children with the same modified linear program, but with differing node structures (taken directly from each parent).

During graph crossover, the method used in graph GP, detailed in in Section 3.4.4 is used, but instead of only generating one child for each crossover operation, two children are generated, with differing base structures.

3.5 Basic Example Implementation of GP

Genetic programming is often used to evolve a program well-suited to solve a certain task. This is often done using a domain-specific language, or a set of functions and terminals specifically designed for the given problem. To illustrate this, we will first pose an example problem, which we then will solve using genetic programming.

Problem description

A good example problem to solve is a simple symbolic regression problem. We want to find a program which outputs values equal to a given polynomial r. In this case, we want to find a program which outputs result of $r = x^2 + 3x^2 + x$, given an input variable x.

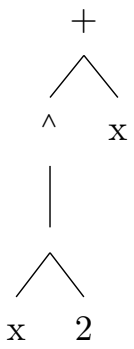


Figure 3.2: Example of how an equation can be expressed in a tree-based structure.

Terminals

We will be using a terminal vocabulary of the following variables:

$$t = \{x,\ c \mid x \in \mathbb{N},\ c \in [a, b]\}$$

where $[a, b]$ is a specific range of values speculated to be part of the solution. x is the variable x of the equation, which will be integrated during the fitness function.

Fitness function

We will define the fitness function as $f(a)$ for individual a:

$$f(a) = \left| \int_{-1}^{1}(a) - \int_{-1}^{1}(r) \right|$$

where r is the reference polynomial, i.e. $x^2 + 3x^2 + x$ in this case.

Termination criteria

The termination criteria can simply be defined as fulfilled if an individual of the population has achieved a fitness score below a certain threshold, i.e. $f(a) < \epsilon$ for any individual a in the population. For this problem we could most likely set $\epsilon = 0.01$ without problems.

Encoding

We will be encoding the population using a tree-based structure. Meaning every individual holds a tree of nodes, with functions and terminals as their arguments. An example individual could in this case look like this:

```
r = (node + [(node ^ [(node x []) (node 2 [])]) (node x [])])
```

This is in essence a very verbose way of writing the following equation, but expressing it like a tree of nodes does hold the benefit of allowing the structure to be easily manipulated.

$$r = x^2 + x$$

This is also visualized in Figure 3.2.

Genetic Operators

This defines the different operators used by the genetic algorithm to modify its population.

Selection Simply using tournament selection, without any tweaks, should prove to be good enough, as we simply want to weigh the fitness of the individuals when considering which individuals should be allowed to mate. As we are happy with a close enough approximation of the problem, and the problem is as simple as it is, premature convergence is not an immediate threat.

Crossover We will simply use the conventional method of crossover for tree-based GP systems:

1. Pick a random node from each parent, these will be the crossover points.
2. For each parent, extract the sub-tree following the crossover point.
3. For each parent, replace the sub-tree following the crossover point with the sub-tree extract from the other parent. This will be the offspring.

This method will output two children for each application of the crossover operator.

Mutation We will define our mutation operator as an application of one of the following operations, with equal probability:

1. Replace a sub-tree with a randomly generated one of roughly the same size.
2. Move a sub-tree from one node to another.

3.6 Genetic Programming Concepts

While designing a fitness function has been described as an artform, so has creating the actual representation, or encoding, of an individual. If a program cannot be properly modified by the available genetic operators, then a well performing fitness function will not help us much. [19, 26] For this reason, many different encodings have been researched, which is further detailed in Section 3.4.

3.6.1 Building Block Hypothesis

A hypothesis regarding GAs, which attempts to describe why they achieve the results they do is the Building Block Hypothesis (BBH), which consists of:

1. A description of a heuristic that performs adaptation by identifying and recombining "building blocks", i.e. low order, low defining-length schemata with above average fitness.
2. A hypothesis that a genetic algorithm performs adaptation by implicitly and efficiently implementing this heuristic.

Paraphrased, Goldberg [15] describes this hypothesis as: *by identifying blocks of high fitness, these can be recombined to create blocks of increasing fitness, which reduces search space compared to trying every possible combination, this can in turn reduce the complexity of the specified problem.*

As code and programs are in essence made up of "blocks of high fitness", given a fitness function which captures this, then we can understand why GAs and GP would be suitable to evolve programs effectively, if the building block hypothesis holds.

3.6.2 Ratcheting

An important aspect to keep in mind while designing genetic operators is *ratcheting*. Ratcheting occurs when the genetic operators are not properly balanced, and there is *no way back* from a mutation or crossover. If the population has started tipping one way, it then cannot tip back, even though the average fitness has started to plummet. To make sure this does not happen, the genetic operators which is used by a GP algorithm should form a cyclic group in some way, meaning that given any combination of operators executed upon an individual, some other combination can bring it back to its default state. [13]

3.6.3 Bloat

Bloat occurs during a GP loop when the population reaches a local optimum, and almost no modification to the well-performing individuals yields a performance increase anymore. The only modifications that will not immediately be discarded by the algorithm will then be additions of instructions which do not affect the result, also called *introns* in literature. As such, the algorithm will output individuals containing more and of these introns, and will thus be longer (they have more genes, or instructions encoded within them), effectively bloating the population with non-functional genes. [2, 4]

Why this happens is still somewhat of a mystery, and the cause most likely differs from problem to problem and implementation to implementation. It is not entirely agreed upon whether or not this actually *is* a problem however, and advocates for bloat argue that it mirrors reality, as 90% of our DNA does not affect us directly, and could be described as filler DNA. [2]

Many attempts have been made to stave off bloat, most notably that of enforcing hard limits of this size of the individuals, or employing some form of anti-bloat selection. There are numerous versions and implementations of this, for example weighing the probabilities an individual will be selected for crossover on the basis of length as well as other factors (such as fitness). [2]

3.6.4 Premature Convergence

Populations which converge too early (i.e. they converge to a local optimum before they can reach an acceptable solution) are said to have been subject to premature convergence. More specifically, premature convergence is often referred to as the vent of the population becoming too homogeneous, disallowing further progress baring lucky mutations. [24]

Premature convergence is a very common problem, as avoiding getting stuck in a *local optimum* is exceedingly difficult.

While this is one of the strengths of genetic programming (as opposed to e.g. hill-climbing algorithms, which a *very* prone to getting stuck in local optimum), it is still a problematic area, as the more difficult or the larger the problem, the higher the risk the algorithm will run head-first into this issue. This is the reason genetic programing and its ilk is recommended as way to *approximate* a solution.

To combat premature convergence, researchers have proposed numerous possible methods, such as *fitness sharing*, and the *island model*, which will be discussed more thoroughly later on in the thesis.

3.6.5 The Multiple Domains Problem

A typical problem for programs evolved using genetic programming is the fact that the programs have to excel in two very different domains: first, they must thrive in the environment they were evolved in, meaning they are ”easy” to manipulate using the various genetic operators, or rather: prone to survive, meaning their fitness-altering genes remain intact. Secondly, they must effectively solve the preset problem they were tasked to solve. [26]

1. The evolutionary domain: The individual has to excel at passing down its genes through generations. As such, genes which are not directly affecting the fitness can quickly be removed from the gene pool, even though it might be hugely beneficial in the longer run.
2. The program domain: The individual has to perform as a program solving the problem it was evolved to solve.

This is one of the contributing factors of bloat, as the individuals try to excel in the evolutionary domain while still fulfilling the requirements of the program domain, they gather a lot of introns.

3.7 Niching Methods

A niching method is a method used to promote diversity in a population, or in other words, to promote individuals to have a certain *niche*. This increases heterogeneity, which should help combat premature convergence.

This can be done in a number of ways, from imposing penalties, to almost completely isolating parts of the population from one another.

The most common niching methods found in literature is listed in this section.

3.7.1 Fitness Sharing

To maintain diversity, and thus combat premature convergence, fitness sharing is an often used *niching* method. Fitness sharing modifies the search landscape (i.e. the fitness function), by reducing the payoff for performing well in common tasks. The penalty given is:

$$f_i' = \frac{f_i}{m_i}$$

where $f_i^{'}$ is the shared fitness score for an individual i with fitness score f_i, and m_i is the niche count, corresponding to the number of individuals sharing the fitness f_i. In other terms, an individuals fitness score is scaled proportionally to the number of individuals sharing the same niche.

The niche count m_i is in turn defined by:

$$m_i = \sum j = 1 N sh(d_{ij})$$

where N is equal to the population size, d_{ij} is the distance between individual i and j, and $sh(\cdots)$ denotes the similarity-, or sharing-, function.

With this representation in mind, the most commonly used sharing function is defined as:

$$sh(d_{ij}) = \begin{cases} 1 & -/(d_{ij}/\sigma_s)^{\alpha}, if d < \sigma_s \\ 0, & otherwise \end{cases}$$

where σ_s is defined as the niche radius, i.e. how similar we allow individuals to be.

The distance metric d_{ij} is defined as either some genotypic or phenotypic similarity, where the genotypic similarity is often the Hamming distance between the bit-string of the two individuals, while the phenotypic similarity relates to how they affected the in which it executed.

This niching method works very well for easy problems, but begins to show its limitations and flaws as the complexity of the problem grows. To begin with, fitness sharing requires priori knowledge about the distance between the local optimum, or the peaks of the fitness landscape. Without this information, we cannot properly set the value σ_s. For most problems, this information simply is not available. The constant modifications to the fitness values of the population that this method causes also help to aggravate this issue, as it makes it harder and harder differentiate between different optima. [32]

3.7.2 Island Model

Another approach for maintaining diversity is the so-called island model [38, 37]. The idea is to divide the population into multiple so-called islands, which are isolated except for rare points of contact. In theory, this could allow each island to develop their own niche, effectively exploring the search space multiple local optimum at a time, while sharing information through *migration*.

During migration, n individuals are sent to replace the n worst performing individuals on the island k steps away from their origin island. Here, k is a rotating number, in the range of $1 \leq n \leq j$, where j is the number of islands. After j migrations every island has sent individuals to every other island.

This has proven to be quite an effective way of staving off premature convergence. Another very attractive feature — if it is appropriate for the given problem — is that it allows the programmer to parallelize the algorithm, with each island running in its own thread, and thus potentially greatly increasing the performance of the algorithm. [38, 37]

3.7.3 Clearing

Clearing works similarly to fitness sharing, with a few key differences. Instead of directly sharing the fitness between each individual in a certain niche, each niche only accepts a set number of individuals, allowing the fitness of these individuals to be preserved.

According to Sareni et al. [32], this was at the time the most promising niching method, greatly outperforming the common fitness sharing in terms of maintaining multiple niches.

Due to its similarities, it does share the same limitations as fitness sharing, such as requiring priori knowledge of distance between local optimum, making it difficult to use for many problems.

3.7.4 Crowding

In standard crowding, the selection part of a GA is modified to randomly select a small percentage of the population for crossover. The offspring then replaces the most similar individuals of the population, instead of the least fit ones, where the similarity is often of the genotypic variant. [9]

This technique underperforms for multimodal problems, i.e. problems where all or most of the available solutions are to be returned. More generally this can be viewed as the method having problems keeping multiple niches alive. [32]

Modifications to this technique include variants which introduce additional tournaments between parents and children, and only replaces the parents if the children has a higher fitness score, which has proven to be an improvement to the method in almost all cases. [22, 32]

Very similar is that of the so-called restricted tournament selection, where individuals are selected as in normal tournament selection, but instead of replacing the individuals with the lowest fitness in the population regardless of niche, restricted tournament selection has the offspring compete in a tournament with the most similar individual of the population, where the winner is inserted in the population while the loser dies.

3.8 Other Modifications

3.8.1 Fitness Scaling

Scaling is usually combined with a niching algorithm to increase the difference between optima. A common way of scaling is the so-called power-scaling technique:

$$f_i' = \frac{f_i^\beta}{m_i}$$

where β can be modified to control the scaling. It can be difficult to correctly set this parameter, as a too high scaling can cause premature convergence, where a few strong individuals might completely take over and force the population to get stuck in a local optimum.

Setting the scaling too low, on the other hand, might not do much to help alleviate the problem that scaling was attempting to solve in the first place.

3.8.2 Adaptive Genetic Algorithms

An adaptive genetic algorithm (AGA) refers to a GA which exposes one or more of its parameters (as defined during the preparatory steps by Koza et al. [2]) to evolution in addition to evolving the individuals in the population.

This has been successfully implemented by Angel et al. [23], where the following parameters were adapted during evolution:

1. The number of offspring.
2. The crossover probability.
3. The mutation probability.

AGA:s can be hugely effective if finding the proper values for these parameters cannot effectively be done (this can happen if there are too many uncertainties for a given domain or problem). It is however not free to use, as it does almost paradoxically also increase the search space, as we are now attempting to solve for three (or however many parameters you are trying to adapt) additional parameters.

4 Related Work: Pre-existing tools and frameworks

Relevant tools, other papers, and work done which either directly or indirectly affected the writing of this paper.

4.1 BAP: Binary Analysis Platform

Binary Analysis Platform (BAP) is a framework or infrastructure for performing program analysis on binary code. Like other platforms of its kind, BAP works by first disassembling the binary code into assembly instructions, subsequently lifts these to an intermediate language (IL), and then analyses the binary at the IL level. This allows BAP to function on a variety of architectures, as any architecture which can be lifted to the IL is also supported by BAP. [6]

Notably for this paper, BAP exposes the semantics of assembly, i.e. the effect of instructions on the CPU-state, including all implicit side effects packed into many often used assembly instructions.

4.2 angr: another binary analysis framework

angr is a binary analysis framework similar to BAP which combines both static and dynamic analysis into a framework with a huge set of features, developed primarily by Shoshitaishvili et al. [33].

It performs semantic analysis by first lifting the code of the target binary file to VEX, an intermediary representation (IR) of assembly code of various architectures. This allows angr to work with any format and architecture supported by VEX itself. In addition, VEX also exposes side effects of instructions which are otherwise non-obvious, such as conditional flags. [33]

This approach allows a ROP programmer a much clearer view of what a gadget actually does when executed.

A ROP-compiler built upon this framework is called *angrop*, and supports functionality such as:

1. Setting register to arbitrary values.
2. Writing arbitrary values to memory.

3. Calling functions.

4.3 PSHAPE: Practical Support for Half-Automated Program Exploitation

PSHAPE (Practical Support for Half-Automated Program Exploitation) is a ROP compiler in lieu of Q and similar compilers. The core loop used by PSHAPE is defined as: [12]

1. Gadget extraction & analysis: In this step gadgets are extracted from the given binary and sub-sequentially analysed, and summaries are generated for each gadget, exposing side effects in a readable manner.

2. Gadget filtering: The gadgets are filtered to only keep those which affect registers used to pass parameters to functions, which include `rdi, rsi,rdx, rcx, r8, r9` on the x64 platform.

3. Gadget grouping: The gadgets are grouped into two categories, `load` and `mod`, which corresponds to gadgets which overwrite a certain register — for example `pop rcx` — and gadgets which modifies a register, respectively.

4. Gadget chaining: The permutations of all possible gadget combinations are then generated, and for each permutation, the pre- and postconditions are computed (i.e. the state before the chain is executed, and the state after the chain is executed, respectively). When a viable permutation is found (i.e. the postcondition fulfills the requirements), the algorithm terminates.

5. Pre-condition solving: In the event of a chain which dereferences a register, a pre-chain will have to be generated setting the register to a viable value.

Follner et al. [12] also touched upon the interesting topic of the future of ROP, and the increasing need to make use of *all* gadgets available, and noted that a branching gadget (i.e. a gadget containing a conditional jump) could be separated into two separate gadgets given a controllable precondition.

4.4 Q

Q was one of the first fully automatic ROP-compilers, and was also capable of ”hardening” binaries against protections such ASLR and DEP, by which it took an already existing payload and modified it so that it bypassed such protections. As mentioned earlier in Chapter 1, it achieves an 80% success rate for payload generation against binaries in `/usr/bin` with a size above 20KB.

Q splits the compilation of the ROP-chain into multiple steps:

1. Program expression: the target program is expressed in a higher-level language, which maps functions or operators to gadget types depending on their semantic operations, e.g. gadgets which writes to memory, performs arithmetic functions, etc.

2. Gadget discovery: compiling a list of potentially useful gadget addresses.

3. Gadget arrangement: arranging gadget types in differing ways expressing the target program.

4. Gadget assignment: assigns the discovered gadgets in a manner matching the arrangements found during the earlier steps.

This approach allows the user to express their wanted program in a simple high-level programming language called QooL. This language was designed to allow for easy interaction with the environment in which it is executed, meaning it mostly deals with memory and register writes and reads, and is thus not Turing Complete.

The language of Q, QooL, is defined as:

<stmt> ::=

| StoreMem <exp> <exp> <type>

| Assign <var> <exp>

| CallExternal <func> <exp list>

| Syscall

<exp> ::=

| LoadMem <exp> <type>

| BinOp <binop type> <exp> <exp>

| Const <int> <type>

Since the inception of Q, there have been many compilers which have gone past the capabilities of Q. The developers of angr and angrop touches upon the absence of utilizing the stack as a scratchpad for moving values in Q. This functionality is present in angrop, and should in theory allow angrop to find many chains Q were not able to, as this allows the angrop compiler to find chains through an extra layer of indirection. [33]

4.4.1 ROPC

ROPC was written as an attempt by the author to write their own version of Q, which is not publicly available. Q and ROPC therefore share many similarities in design; apart from the compilation step, the approach is identical (gadget discovery, classification, and verification).

They however differ greatly during the compilation step. ROPC programs are defined in a higher, domain specific language called ROPL, and allows expressing Turing Complete programs including recursion and loops, which is something Q does not support. [25]

Many features of ROPC requires specific gadget types to function correctly. To implement conditional jumps, `lahf` or `add esp, <reg>` gadgets are used. This limits the number of binaries ROPC is able to generate payloads for.

Due to the complexity required allowing expressing Turing complete programs, ROPC injects a preamble, or self-contained ”stack”, in every payload which is used as a scratchpad in further calculations. Due to this — and other design decisions such as allowing loops and other complex constructs — payloads generated by ROPC are very large, which makes using this ROP compiler in practice almost impossible.

4.5 ROPER: ROP-chain compilation using GP

ROPER is a ROP-chain compiler designed to generate chains for the ARM architecture, and utilizes genetic programming to achieve this. The idea was that this approach might allow the program to generate ROP-chains a human analyst might not even be able to think of. ROPER evolved ROP-chains for the ARM processor architecture, which allowed it to sidestep problems and complexities you would otherwise encounter while developing a similar compiler for e.g. the x86 architecture. Notably, the length of instructions in an ARM-architectures is static, which makes building chains which actually execute almost trivial. ROPER was able

to successfully "compile" multiple quite interesting results, among them a successful `execv` attack.

An interesting observation by Fraser was the fact that in a lot of the best performing, or fittest, individuals, they explored so-called *extended* gadgets, or gadgets which branched unexpectedly, and went off-script. When this occurred, the instruction set was not inherited directly by its ancestors, and was instead an example of so-called *emergent behaviour*.

ROPER was additionally subject to a huge amount of bloat, as observed in the `execv` example: only 2 out of its 32 gadgets executed as expected, and was thus the only two gadgets which contributed to its fitness. The job of the rest of the gadgets was more or less to make sure that the individuals fit genes had a higher chance to be passed down through the generations. This is a prime example of the Multiple Domains problem programs evolved using genetic programming often suffer from.[1]

4.5.1 Encoding or Genotype Representation

An individual in ROPER is encoded as a vector of *clumps*, which contains a gadget address plus its corresponding arguments, if the gadget contains instructions which modify the stack pointer (e.g. `pop` instructions).

In addition to this, each individual contains information about gene linkage, or *viscosity*. This is relevant to the Building Block Hypothesis, as it is used to encourage formation of building blocks, described as clumps which tend to improve fitness when occurring together. This viscosity metric affects the start- and endpoints, used during crossover, and is derived from:

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

if the prior link fitness is `None`, and

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

otherwise. $f(A[i])$ corresponds to the fitness of the i-th clump of individual A. The prior-link fitness $f'(A[i])$ is passed down from its parents. This is in simple terms the average fitness score of ancestors in which this gene occurs.

4.5.2 Fitness Function

The fitness function, or measure, used by ROPER is dependent upon the specified task. The most relevant function is the pattern matching fitness function, which attempts to set the CPU-context to a specific state. The value of an individuals fitness is the average of the following measures:

All fitness functions are first subject to a behavioural profile of sorts, containing:

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.

These measures are subsequently passed to a task specific fitness function. In the case of a pattern matching function, such as setting registers to certain values, Fraser used the following: for each target, the best of the following is returned as its fitness score, and stored in a vector. The individual's fitness is then the average value of this vector.

[1] The multiple domains problem is further detailed in Section 3.6.5

1. The hamming distance for each target register and their value compared to each target value.
2. The hamming distance for each non-target register and their value compared to each target value plus a ”wrong-register” penalty.
3. A memory sweep of the dereference of the value of each register. This value is calculated by the offset from the entry point: if we find the sought after value e.g. 50 bytes after the entry point, $50/n$ is returned, where n is the size of the dereference.

4.5.3 Fitness Sharing

Additionally, ROPER also makes use of fitness-sharing to stave off what Fraser deems the most serious problem for ROPER when dealing with more complex problem spaces: the threat of depleting population diversity. Especially when the problem is divided into multiple parts, targeting low-hanging fruit becomes a strategy that without mitigation often completely takes over, causing a catastrophic decrease in diversity.

In ROPER, fitness-sharing is implemented as:

1. Each problem is initialized with a base `difficulty` score, specified by the user.
2. Each season, each problem's `difficulty` score is updated to the mean of the populations fitness score for that specific problem.
3. Once the `difficulty` score is available for each problem, the fitness score of each individual can be weighted depending on the `difficulty` of the problem or problems it managed to solve.

This allows individuals capable at solving rare tasks who would otherwise not survive to pass down its capabilities to the new generations, hindering premature convergence.

Part III

Methodology and Implementation

5 Pairing ROP and GP

This section discusses the pairing of ROP and GP, and how it differs from areas in which GP historically has been used.

5.1 Strengths of ROP & GP

Since every single binary exposes a different underlying set of functions for the unfortunate ROP programmer to make use of, there is an argument to be made that perhaps a computer could learn to understand this instruction set and how to utilize it better than a human could.

Previous implementations of automatic ROP-generators outperform humans in most *basic* cases, if the instruction set allows for it. If the most obvious gadgets and building blocks exists, and no complex hurdles are in the way, such as e.g. a too small stack, an automatic approach will always produce a ROP-chain must faster than a human ever could.

The problem, however, lies in the aforementioned complexity. Given a problematic enough binary, automatic approaches of generating ROP-chains have proven to be insufficient, as they fail to find more complex paths towards potential solutions. [11] The reason for this was, according to Follner, complexities regarding indirect jumps and branching, which required the ROP-chain to initially set certain memory addresses and registers to make sure certain jumps were either made or not.

This is a problem which is certainly solvable, given sufficiently advanced automatic compilers, which take these kinds of problems into account. This is not an easy feat though, which is most likely why no such tools exists yet as of writing.

The idea is that if we could replace the constraint solver employed by most automatic ROP-compiler with a GA, we could transform the problem, and thus perhaps reduce the complexity. If we find a fitness function which accounts for problems like we described above, we could in theory find paths to solutions which would otherwise require a *very* complex implementation. A concrete example of this actually occurring is the pathways Fraser's ROPER took during the evolution of a ROP-chain supposed to execute an `execv` exploit, which overwrote its own code stack and injected ROP-chain to explore executable code not contained in the pool of gadgets initially harvested from the binary.

In other words: if we can restrict the search space enough for the GA to function well, we can give vague guidelines as to what we want the algorithm to achieve, in the form of a fitness

function, which could in theory allow us to disregard complex implementations which might otherwise be very difficult to properly implement in the general case.

5.2 Problems For GP & ROP

The pairing of GP and ROP is not a without problems, and domain specific problems pertaining to genetic algorithms and more specifically genetic programming when paired with ROP include:

1. Very large number of GP-operators (gadgets), causing state-space explosion.

Given large enough binary files, we run into problems due to the sheer number of instructions (gadgets) available to us. A problem normally solvable by GP uses a set language of a few functions and/or operators. In the case of ROP, we may have access to tens of thousands of gadgets, causing an exceedingly large state space-explosion. The number of possibilities to evaluate then becomes so large that getting anywhere close to a solution becomes almost impossible.

A good analogy would be this: in typical GP systems, we evolve a program using a static set of functions and operators towards a goal system state. The goal is then to approximate a goal state, on top of a well defined base. Applying this directly on top of the ROP-domain, by evolving gadget addresses directly towards a very specific goal state, we have reversed the typical operating function of GP. Instead of using a specific set of instructions to approximate a goal state, we are using a non-specific set of instructions to evolve a very specific goal state.

2. Non-robust underlying system, which disallows stable mutations and crossovers.

It is important for a genetic algorithm to function properly that any mutation or crossover on a functioning program results in another functioning program. In the case of ROP, many programs will simply crash, due to attempts to access memory which is not readable, trying to write to memory which is not writable, or by simply modifying parts of the memory so that the control flow breaks.

This often results in the unfortunate scenario in which two relatively successful parents cross over and produce something that instantly crashes, and can cause issues such as premature convergence (as individuals which does anything at all are better than individuals that crash).

3. Very specific target programs, requiring very specific program flows: getting stuck in local optimum is a very real problem.

If we look historically at GP and GA, these techniques are apt at approximation (as with most solutions generated by artificial intelligence, the perfect solution is rarely found). The environment upon which ROP operates, however, requires utmost precision in its execution, leaving no room for error. There is often a very specific pathway that needs to be taken, and getting close to that solution is not good enough.

A concrete example of this given a ROP-program would be the case of a chain containing an individual which in turn contains a gadget which does a necessary action, while at the same time dereferencing a register causing a segmentation fault. As we cannot possibly know what the gadget would have done had it actually executed correctly, this individual will quickly be discarded along with this perhaps crucial gadget, causing our evolution to never being able to find an answer. As such, we have a very fragile environment, causing a very fragile evolution. Had a human analyst found this gadget, this chain might have been easy to build.

Another problem, which is very hard to overcome without employing some form of analysis on the gadgets before we attempt to evolve chains on top of them, is the inconsistent instruction size and stack-offset. Without knowledge about *how* gadgets will affect the CPU-state, we

cannot intelligently build chains, as one gadget might completely multiple instructions or simply cause a crash due to irregular offsets (non-multiplicative of architecture specific instruction size).

6 Designing GENROP

Discussing the design of a ROP compiler using GP as a main component, using what we have learned so far.

6.1 The Art of Designing a Fitness Function

Designing a fitness function has been described as an artform [7], and as such, there is much trial and error involved. This is especially true in the case of designing a GA to generate ROP-chains, as finding a fitness function which appropriately shapes a fitness-landscape lending itself towards consistently traversing it is an open problem, yet to be concretely answered in any form of general case.

The difficulty in this endeavour becomes even greater when the genetic algorithm is designed to operate within a ROP-environment.

Even a quite basic problem, such as how to give points to an intermediate becomes challenging. If we, for example, want to set `rcx` to `0x5`, but the only relevant gadgets available to us that are the following:

1. `pop rax; ret`
2. `mov rcx, rax; ret`

As there are no gadgets available to us that directly set `rcx` to a given value, we need to first set `rax` to this value, through gadget (1). After gadget (1) has been executed we can set `rcx` to `0x5` by using gadget (2), moving the value from one register to another. In this scenario, both gadgets are critical to the task, but simply setting the fitness function to the *hamming distance* — which measures the bitwise distance between two byte arrays — for the target register to the target value would not yield a fitness landscape with a gradient pointing towards the solution. Such a fitness function would in this case be functionally identical to one which simply answers true or false whether or not a solution has been found or not.

A possible solution to this problem could be to make it beneficial for the individuals to set other registers to the wanted value as well. [13]

In the perfect scenario, a fitness function would possess the following four characteristics:

1. A continuous fitness landscape, allowing a GA to follow the fitness landscape towards a global optimum (or in other words, towards a solution). [1]

Maintaining a continuous fitness landscape in this sense is in most cases not a very difficult problem when applying GAs on a domain where the use of GAs is well-established. For the problem of function approximation, simply using the difference between the sample and the expected value, given a few sample points, is often enough to create a "good enough" fitness landscape for a genetic algorithm to function properly. In the case of a genetic algorithm designed for generating ROP-chains, this is a problem which might not even be solvable. Fraser showed that the fitness landscape created was very rugged and hard to traverse for multiple different problems and solutions, with no real discernible pattern or trend visible. [13]

2. Appropriately discouraging unnecessary side effects while not discarding gadgets which contain them completely.

To discourage side effects, we need to design the fitness function to properly weigh the advantages and disadvantages of a gadget. In most cases, gadgets affect the CPU-state in broader strokes than would be preferable. This has the effect of giving most of the gadgets our algorithm will be making use of many additional unwanted side effects. A gadget might for example include one desirable instruction, while also harbouring two additional instructions which also affect the CPU-state. If these undesirable instructions are so harmful that an individual with this gadget will never be able to reach perfect fitness, then it is said to carry a *malignant* gene. These might be very hard to properly identify however, as paired with another gadget (which might not do anything by itself), an apparently malignant gadget could begin to function very well.

3. Maintaining diversity, and thus reducing the risk of premature convergence.

One of the most difficult problems to solve when designing a fitness function to be used in a genetic algorithm is maintaining the diversity of the population. When a GA fails to find a solution, it is often attributed to *premature convergence*, which is to say that the population have converged towards a local optimum. [34]

Smith et al. developed methods for GAs to be effectively used for more complex tasks which requires what they call "cooperative structures which join together" [34]. Since a more complex end goal requires more complex sub-components, converging the population towards a global optimum from the get go would result in landing in a local optimum for any non-trivial task. This is opposed to optimization problems, which requires a less complex evolutionary algorithm. For a more "block-based" approach, which would be required for a GA system to effectively generate ROP-chains, the algorithm needs to maintain a diverse and cooperative population. Blocks of gadgets need to be identified, and these blocks need to be able to join together.

4. Encouraging non-direct paths, allowing the algorithm to find more complex solutions which are not immediately obvious.

An important feature needed by the genetic algorithm implemented in this paper is the feature of encouraging non-direct paths. ROP-programs often have to take problematic and complex paths to find a solution to a problem which might seem easy or obvious on the surface level. If not enough gadgets are available, the path towards setting the CPU context in a certain state might require setting multiple flags or addresses a certain way, to allow another gadget to actually function correctly.

[1] With *continuous* in this sense meaning that it is *continuously* followable; there exists a clear trend or gradient to follow.

An example of this is the task of setting a specific register to a certain value. This might seem quite easy at first glance — and it indeed often is — but if there are no gadgets which immediately sets exactly *that* register to any value, we might need to move that value from another location to that register first, which would imply the requirement of setting the value at that location to the desired value beforehand. The more indirect the solution is, the more difficult it is to find.

6.1.1 Feasability of Expressing Such a Powerful Fitness Function

Achieving the tasks listed above through the design of a fitness function is no easy feat, and it is uncertain if it is even possible. For a problem more fit to being solved by genetic programming this *can* be achieved, but when the environment in which we will be evolving our programs is as hostile as the environment in which ROP-chains live, this is a dream we will most likely not achieve, at least not in this paper.

6.2 The Art of Encoding The Problem

Kantschik [18] proposed that the encoding of the problem, i.e. how the population is structured, is of equal importance to that of other parts of a GP algorithm, and maintained that the encoding was something that was often overlooked.

Looking at the building block hypothesis, GAs should preferably during crossover exchange ”building blocks” of high fitness to generate offspring with even higher fitness. As such, we want to create favourable conditions for:

1. Allowing ”building blocks” of functional code to be identified.
2. These building blocks to thrive, or in other words, survive.

This is in many cases the same point, as creating conditions for the blocks to be identified in the first place almost by definition allows them to survive, as identifying them means finding a pattern in the population. Promoting a known pattern is trivial compared to actually finding it in the first place.

Fraser [13] attempted to achieve both of these points by using a linear vector encoding for her population, coupled with a ”viscosity metric” — which measured the success of the adjacency of gadgets — which was used to weigh the crossover point during crossover. The encoding itself, she argued, was not better suited for ROP than any other option.

Preferably, we want the encoding itself to aid with both condition (1) and (2), as this would very naturally guide the population towards finding different functional blocks, making a successful convergence much more likely to happen. Most importantly however, the encoding used has to make sense with regards to the available functions, or the instruction set the population has access to. For e.g. symbolic regression problem, this instruction set often contains the basic math operators: addition, subtraction, multiplication and division. This instruction set lends itself very well to a binary tree type of encoding, as each operator naturally accepts two inputs.

Additionally, we want to allow these blocks to be reused: code programmed by humans are often interconnected pieces of linear code, which is also the case for the assembly code generated by the compiler, and for many ROP-chains. In the example `execv` payload shown in Section 2.4, setting the `rax` register to a specific value was used multiple times. It would be preferable then to be able to naturally allow parts such as that to be reused during the chain.

Another point will therefore be added to the list:

3. Allow code to be effectively reused during the ROP-chain execution.

6.3 Implementation Decisions

With the points listed in Sections 6.2 and 6.1, we have some important decisions to make:

1. How do we encode our individuals?
2. How do we handle premature convergence, i.e. which niching methods do we use?
3. How do we properly decide upon the GA parameters (mutation rate, crossover rate, migration rate, etc.)?
4. How do we define our fitness function?
5. How will we define our genetic operators, i.e. crossover, mutation and selection?

6.3.1 Encoding

There are a few options available to us with regards to how we should properly encode our individuals, with advantages and disadvantages for each one. As such, we will go through them here.

We will first and foremost define *what* we will encode, as simply using a memory address itself, and nothing else, will not allow us to generate functional individuals (the search space will be too large, with the absolute majority of them being non-functional). We will in this case be using the same method as is used in ROPER: by utilizing so-called *clumps*, which is a shell containing a gadget coupled with its arguments. [13]

We also want to take into account the list defined earlier in Section 6.2. An encoding which seems like a natural fit at first glance is the encoding used in stack-based GP algorithms: ROP-programs also operate on a stack, after all. This design is sadly rather orthogonal to the requirements of a GP algorithm designed to generate ROP-chains. This is because the stack based genetic programming system primarily functions well because of the underlying architecture or environment, which is designed with the express purpose of functioning well for genetic programming. In the domain of ROP, this is not something we can control in any meaningful way.

Moving on, we have to account for an unfortunate fact about our population of clumps: they do not operate like pure functions. If anything, the operate like the exact opposite. To clarify, a pure function is a function, which: [21]

1. Always produces the same output for a given input.
2. Does not have side effects (running this function will not indirectly affect another function).

In contrast, our clumps *only* affect the global state, and often in unpredictable ways, while not producing a directly usable input.

Due to this, most encodings discussed in Section 3.4 disappear as options, as they either detail a very specific environment in which the evolution occurs, or they operators are functions which take arguments and output values themselves, allowing very logical structures to be built.

Connecting clumps as though they were part of a graph implies that somehow the output of one clump is the input of another. As we have established, this is simply not possible. Therefore, we cannot realistically encode our individuals with either *graph* or *linear-graph* encoding.

Using a *stack* encoding is not applicable either, as this encoding is very reliant on the environment in which it operates, which is not something that we can control.

This leaves us with linear encoding, which is used by the Slash/A [1] programming language, and ROPER [13].

This encoding does not share the same problems with incompatibility as the other encodings, but it is not a perfect fit either. A linear encoding is very simple: it only describes which instructions are to be run, and in which order. It does *not* consider how the gadgets will be modified by the genetic operators, or how ROP-chains are structured (which might not be as simple as its execution order). This results in linear encoding not inherently maintaining building blocks.

The solution that was chosen to combat the this problem is detailed in Section 6.3.3.

6.3.2 Fitness function

A perfect fitness function would perfectly map the fitness landscape and its intricacies, which would allow even a gradient descent type algorithm[2] the ability to find a solution. This is often not possible, and especially not in the domain of ROP, due to its unreliable and fragile environment.

Creating a fitness function in the ROP-environment which portrays the fitness landscape as a gradient, is sadly not a very tractable problem. Due to this, we have decided to let our fitness function be as simple as possible, while delegating much as possible to the other parts of the GP loop.

The fitness function is supposed to indicate how far from the solution a specific individual actually is, which can also be expressed as: how likely is this individual to evolve into a candidate solution in the coming generations. Many ideas were tested, but later discarded due to not resulting in increased performance. Some of these are listed here:

1. Scaling the fitness score by the number of gadgets executed divided by the number of gadgets in the chain, i.e. is this chain crashing early, if so we penalize the chain.

2. Scaling the fitness score by the number of gadgets in the chain divided by the number of gadgets executed, i.e. the inverse of the metric listed above. If the chain contains gadgets which are still to be executed, then it might contain a solution if the gadget which caused the crash is somehow removed. Another way of thinking about this is: if the entire chain executed, and we did not find a solution, then this chain is unlikely to evolve into a solution down the line.

The fitness function is borrowing ideas from ROPER, and is being kept quite simple. In GENROP, however, we do know which operation we are currently trying to perform (which goal), and as such, we can design one fitness function for each supported action[3].

Fitness function for changing registers:

$$f(r) = h(t_r, a_r)/n_r$$

where h is the hamming distance function, t_r is the expected content of register r, a_r is the actual content of the register and n_r is the size of the register.

The fitness function for writing to memory is slightly more complex:

$$f(m) = \begin{cases} h(t_m, a_m)/n_m, & \text{if } m \text{ has modified} \\ \infty, & \text{if } n \text{ has not been modified} \end{cases}$$

where t_m is the target content of of the target location in memory, a_r is the actual content, and n_r is the size of the target content.

It was decided after trial and error not to implement the conditional part of the memory fitness function for the register counterpart, as it did not seem to improve the results much, while significantly degrading performance.

[2] A gradient descent algorithm is an optimization algorithm for finding a local minimum in a function.

[3] GENROP supports writing to memory and settings registers. Actually issuing `syscall` is not handled by the genetic algorithm, as it's very straight forward to do given knowledge about the gadgets available.

6.3.3 Niching method

As we need to make up for the lack of sophistication in the encoding, we have to introduce a way for building blocks to be identified in other ways. This gives us two goals for the niching method:

1. Reduce state space.
2. Allow building blocks to be identified.

To achieve this, the algorithm designed in this paper attempts to implement the following features:

1. A pre-processing step, which groups the gadgets into islands with a specific sub-goal for each island. A visualization of these islands and how they fit together is shown in Figure 6.1. This is not the same type of islands as described in the Island Model, but regrettably shares the same name.
2. Each island only contains gadgets that are able to affect the state of the island's goal. If the target goal is to e.g. modify register `rax` then only gadgets which can modify `rax` will be included on this island. These gadgets can in turn have dependencies on other gadgets.[4]
3. Generates sub-goals based on the dependencies of each gadget on parent islands. Given a gadget which modifies `rax` but depends on `rbx`, a sub-island will be generated with register target `rbx`.
4. When generating an initial population, or when a mutation occur which generates a new clump, the gadgets are inversely weighed based on how many times they have occurred in other individuals on the same island.
5. When a solution has been found for a sub-island, the champion is migrated to all of its parent islands.
6. The champion of a sub-island is in this case first optimized (i.e. all clumps which aren't necessary to produce the results are removed from the chain), and then transformed into a single, immutable clump, allowing it to be treated as a single function or gadget on the parent island. Allowing variable length solutions without introducing a source of bloat.
7. Traditional fitness sharing, which divides the individuals into groups depending on fitness scores according to each evaluator on an island.
8. A *remigration* system, which entails reactivating an island's sub-islands if a champion has not been found in n seasons. The top k sub-islands are activated, sorted by the fitness-sharing vector (i.e. the sub-island(s) with the individuals with the most sought after traits are activated).

This system allows us to divide and conquer: separate the problem into small, easily (in many cases) solvable subproblems, which can then, according to the BBH, recombine to solve the larger, higher level problem (i.e. higher fitness individuals). This separation makes solving problems using GP more tractable for the human analyst, and it is in part a solution to the problem of finding a solution in very large search spaces.

Traditional fitness sharing was implemented, as it was identified that most computation time took place on the final island, where all previously identified clumps and gadgets attempted to assemble themselves into a functional payload. This environment, while being

[4] An example of this is a gadget which moves the content of `rbx` to `rax`. we have a dependency on `rbx`.

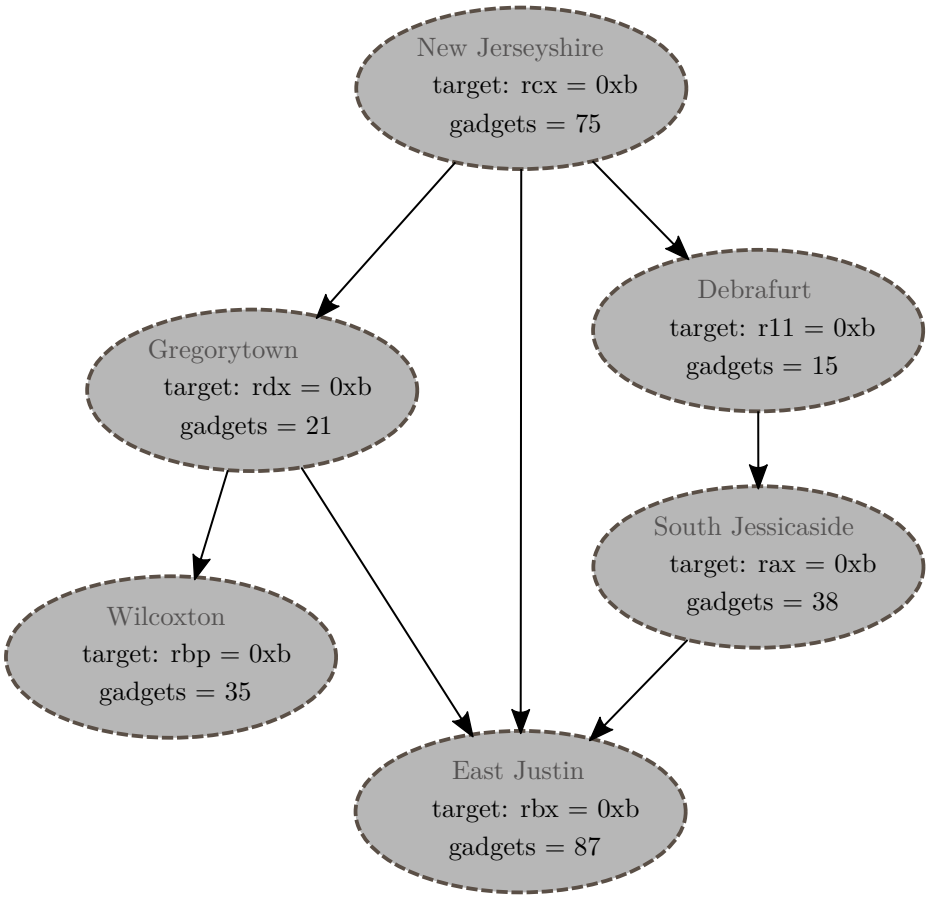


Figure 6.1: Example preliminary island diagram.

filled with a higher-than-average quality of clumps, was rather prone to finding and staying in local optimum, and thus fitness sharing was deemed an appropriate solution to this problem.

The remigration system introduced allows the algorithm to constantly introduce new blood into the populations, targeting the most pressing problem, thus increasing the odds of a successful convergence.

6.3.4 Selection

Trying to implement a non-destructive crossover operator suitable for evolving ROP-chains is quite difficult, as running the gadgets in the wrong order can not only introduce individuals with bad performance, but also individuals which crash during execution. Therefore, in the effort of maintaining a healthy population, we propose a new selection scheme dubbed *bin selection*, which is tightly coupled to the concept of island used in GENROP, and relies on islands having one or more goals. It functions like this:

1. Generate k bins which a dedicated goal picked from the island, where $k = \min(n, 3)$, where n is the number goals on the island.[5]

2. Each bin will contain individuals from the population, sorted by their fitness according to the bin's goal.

[5]The number 3 is arbitrarily set by trial and error.

3. Pick j individuals from the generated bins, by first randomly picking a bin, and then subsequently randomly pick an individual from that bin, weighted by their order in the bin. Where $j = p * c$, p is the population size and c is the crossover rate.

This enables us to effectively use the fact that the population is forced to specialize in different areas, to ensure that the population does not become homogeneous on parent islands, once the sub-islands have migrated upwards.

To negate unnecessary bloat, individuals will always be of a size in the range of its parents sizes, an example would be e.g. parent 1 is of length 3, parent 2 is of length 5, then the offspring can be of length 3–5.

6.3.5 Crossover

For the crossover operator itself we implemented a simple one point crossover scheme, with a randomly chosen crossover point. This does not have to be weighted, as finding building blocks is handled on a higher level as described in Section 6.3.3.

The crossover point is random, as there is no clear benefit on selecting it with any form of weight. This is partly due to the fact that we are encoding the problem as clumps, which can be variable in length, and encapsulated functional building blocks by itself. As such, there is no need to attempt to impart this information[6]during the crossover.

6.3.6 Mutation

The available mutations which are used in the algorithm borrows heavily from Fraser's ROPER [13], as the encoding is similar, and the mutations that can be used in this case cannot differ too much. The available mutation operations which modify an individual made up of clumps are:

1. Replace a random clump with another random clump.
2. Permute a clumps argument.
3. Rotate arguments a random number of steps.
4. Shift clumps left or right.
5. Replacing a value with its pointer dereference, if possible.
6. Replacing a dereferenced value with its origin address.

This makes up a balanced mutation set, as every mutation can be undone by a combination of other mutations. This prevents ratcheting, which is detailed in Section 3.6.2 .

6.3.7 Parameters

The parameters and rates used in the algorithm are listed below. It should be noted that these are not necessarily important to the algorithm design itself (as this differs greatly from a more traditional GP implementation), and are simply a result from trial and error.

- Crossover rate: 0.8, even though the crossover operator is quite destructive, we have allowed a high rate, as insertion into the population requires the individual to be fitter than some other individual in the population. At minimum 20% of the population will therefore always be made up of individuals from the previous generation.

[6]Information about functional building blocks.

- Mutation rate: 0.4, as opposed to normal GP implementations, where the mutation operator is used as a rare event, to introduce some new blood into the population, it is in this case highly required, as variations of each individual has to be explored in a much more minute sense, as following a gradient is more difficult than in other applications of GP. As such, the mutation rate is much higher than it usually is. It can however be argued that a mutation rate this high transforms the algorithm from a genetic algorithm into a form of random search instead.
- Termination condition: Currently this is set to a static number of iterations if no solution has been found, or until a solution has been discovered. Measuring convergence is difficult, as the fitness landscape is often very rough in the ROP domain.
- Population size: The population size for each island is decided by:

$$\max\left(45, \min\left(120, \frac{n_g}{l_c + 10}\right)\right)$$

where n_g is the number of gadgets in the islands population's gadget pool, and l_c is the average chain length of the individuals in the island's population.

6.3.8 Putting it all together

The result of this is summed up by the following list:

Pre-processing

1. Generate gadget definitions using angrop.[7]
2. Filter non-unique gadgets.
3. Group gadgets into islands with target sub-goals making up a directed graph with max depth n.

GP loop

4. Activate all islands without any sub-islands themselves.
5. Do crossover on the top $c = 0.8$, utilizing bin-selection to select mating pairs.
6. Mutate $r = 0.4$ of the population randomly.
7. If a champion has been found, and parent islands exist:
   a) Optimize the champion chain:
      i. Remove each gadget in the chain, one by one, to find gadgets which only contributed to increased bloat.
      ii. It is deemed optimal once the version with the fewest number of gadgets has been found.
   b) Encapsulate the champion as a single gadget.
   c) Mutate the parent island population $r = 0.4$ with the encapsulated champion as the sole gadget in the gadget pool.
   d) Deactivate the current island and activate parent islands.

[7] Using gadget definitions from angrop is a very convenient way of getting access to this information. It does however limit the algorithm's potential, as we will not have access to potentially interesting, powerful but difficult to use gadgets (e.g gadgets which are not made up of basic blocks). These gadgets would be troublesome to analytically develop support for (especially in the general sense), but allowing them to be used as an option in a GA would open up many options.

8. If no solution has been found for t generations, reactivate child islands if they exist.
9. If a champion has been found and no parent islands exist, we have a solution: terminate loop.

Post-processing

10. Output the solution as a python-script generating a string with the addresses to be supplied through a buffer-overflow or other exploit.

Through all of this, we have hopefully split the problem into sufficiently small parts for the GP loop to solve a decently complex problem, both in the general problem domain (i.e. in which the programmer thinks), as well as in the domain of ROP (making it possible for the GP loop to solve the problem).

Part IV

Evaluation and Conclusion

7 Evaluation

How we will go about answering the questions posed in this thesis.

7.1 Methodology of Evaluating GENROP

To evaluate the feasibility of generating ROP-chains through GP, we will be using our ROP-chain compiler, `GENROP`, to generate a payload which can execute the `execv` command, with `/bin/sh` as target binary file. This will be run on a Linux machine on top of the x86_64 architecture, with ASLR disabled.

The actual task of executing the `execv`-command entails:

- Write `"/bin//sh"` to memory, at address `<addr>`.
- Setting register `rax = 0x3b`.
- Setting register `rdi = <addr>`.
- Setting register `rdx = null`.
- Setting register `rsi = null`.
- …and then calling `syscall`.

where `<addr>` can be any writable part of the process' memory. The payload will then be tested on the real binary, by injecting the payload using GDB into the running process, which will hopefully grant us shell access, signalling a success.

Some binaries simply do not contain a `syscall` instruction, which is required to execute `execv`. For these binaries (a few binaries we are testing will have this limitation), we will still test whether or not we are able to successfully compile a ROP-chain (up until the point where the `syscall` is required). We will not test the payload on the real binaries in these cases, as we cannot achieve shell access.

This task requires the ability to set four different registers to different values simultaneously, while also writing a `/bin//bash` to a specific address in memory. Due to the nature of ROP-gadgets, and the prevalence of side effects when these instructions are executed, this is quite a challenge for a GP-loop to accomplish.

The difficulty of this task differs wildly from binary to binary, as the available gadget pool can allow for completely different tasks or execution pathways. For example, if you want to set registers `rax`, `rbx`, `rcx`, and `rdx` to distinct values, and you have a gadget which looks like this: `pop rax; pop rbx; pop rcx; pop rdx; ret;`, finding a solutions is really quite trivial. On the other hand, if you do not have a gadget which perfectly fits, it will be more difficult, especially if certain `pop <reg>` gadgets are missing entirely.

Due to this difference in complexity requirements, we will be testing the algorithm against multiple different binaries, which includes:

1. `/usr/bin/rsync`
2. `/usr/bin/ls`
3. `/usr/bin/git`
4. `/usr/bin/cat`
5. `level0`[1]

On a more high level note, we will test the following:

1. If the compilers can generate a payload for each given binary.
2. If the generated payload works as expected on a real binary. (i.e. does the payload execute, and can we actually get shell access?)
3. How large the payloads are that are generated. (The larger they are, the less likely it is it will work in a ”real” scenario, as the size of the payload you can inject is often very limited)

Additionally, for each test case, we will also be generating solutions using `angrop`. These will be used as reference solutions regarding solution complexity, and whether or not the binary exposed enough relevant gadgets to allow a solution to exist. [2]

[1] from VulnHub's ROP Primer 0.2[14]

[2] A solution could of course exist even though angrop could not find it, but comparing our ROP-compiler to another well performing conventional ROP-compiler will allow us to analyze the results in an effective way.

7.2 Results from testing GENROP

This section will go through what we found when testing GENROP, as defined in Section 7.1.

7.2.1 Summary of results of runs on all binaries

Figure 7.1 represents the summary of the various tests we did to put GENROP through its paces.

GENROP	level0	ls	rsync	git	cat
# total runs	10	10	10	10	10
# failed runs	0	0	0	0	10
syscall instruction exists	-	-	-	-	-
payload execution successful	yes	yes	yes	yes	no

GENROP	level0	ls	rsync	git	cat
avg. payload size	176	400	400	456	-
avg. # gadgets used	8.7	6.8	8.1	8.4	-
avg. # seasons elapsed	30.3	36.2	47.6	10.8	-

angrop	level0	ls	rsync	git	cat
payload size	88	120	152	104	256
gadgets used	6	8	5	6	6

Figure 7.1: Result summary of all binaries part of the evaluation.

The algorithm seems to be very consistent in either finding a solution or not finding one at all, with every single run succeeding for all binaries but `cat`, for which not a single run succeeded.

Worth nothing is that `angrop` was able to generate working payloads for all the binaries which we tested.

There also exists quite a large discrepancy in the payloads generated by the different ROP-compilers. Most notably, the payload size is much larger for payloads generated by `GENROP` than they are for payloads generated by `angrop`. This is very relevant, as if the bytesize of a payload is too large, it can be quite difficult, if not impossible, to actually use in any real scenario.

To generate the payloads, both tools were instructed to solve the problem in the same way, using the same supplied address to write to, using the same method.[3]

It is also interesting to note that `GENROP` was not able to accomplish anything `angrop` was not able to achieve. Which might not be entirely unexpected as `GENROP` is using the same gadget definitions as `angrop`. This might have reduced the capabilities of `GENROP`, which in theory could have utilized gadgets which were not supported by `angrop`.

We were, however, able to generate payloads which *actually worked* in a ”real” scenario, where we injected the payloads into a running process using `gdb`. That is a small victory in itself.

[3] As this problem would otherwise have different solutions, and comparing them would in that case be very difficult. A method refers to the higher-level concept of executing an `execve`-attack with `/bin/sh` as target binary, which can be done in multiple ways.

7.2.2 The failure of generating a solution for /usr/bin/cat

Binary	Size	Gadgets available	Citizens	Succesful
cat	39.1 kb	184	46	no
ls	141.9 kb	4208	120	yes
git	3.14 mb	33738	122	yes
level0	829.6 kb	10127	122	yes
rsync	442.3 kb	610	121	yes

Figure 7.2: Number of gadgets available on the island on which the payload generation for `/usr/bin/cat` consistently failed.

What exactly was it that stumped GENROP in the case of `cat`? What made the tested binary files different from one another?

Looking deeper in the logs, as well as simply analyzing the binaries, reveals the information presented in Figure 7.2. As with most automatic ROP-compilers, the binary size plays a crucial role in the success rate of payload generation.

In this case, at the part where GENROP could no longer proceed in any of the attempts, there was a huge discrepancy in the number of *relevant* gadgets available to choose from in that step. This number is directly affected by the size of the binary, as the larger the binary, the more gadgets or instructions exists.

This *could* have been all there was to it, had it not been for the rather interesting island graph in Figure 7.3. Here, the island Heatherport have four children islands, each which solves Heatherport's problem by themselves, in different ways. Due to this, as soon as one of its children found a solution, the problem on the Heatherport island *should* have been solved as well.

After some research, the reason this did not happen is quite apparent, and highlights the problematic nature of genetic programming, and the difficulties in finding and tuning the correct parameters for a given problem.

To reiterate what is mentioned in 6.3.7, the population size of each island is decided by:

$$\max\left(45, \min\left(120, \frac{n_g}{l_c + 10}\right)\right)$$

where n_g is the number of gadgets in the populations gadget pool, and l_c is the average chain length of the individuals in the population.

The effects of this design decision can be seen in Table 7.2. While this is explained more in depth in Chapter 6, it is in short done to allow as many gadgets as possible to be represented among the citizens. The more gadgets that are available, the more citizens are required (with a population cap in place, to keep computation times low)

When an island finds a solution and migrates upwards, the champion of that island will be encapsulated as a clump, which is then subsequently treated as a single gadget on the target island. The algorithm will then mutate the population on the target island in accordance to the mutation rate, with the before-mentioned clump as the sole gadget in the gadget pool.

In this case, any solution taken from one of Lake Matthewtown's sub-islands *should* immediately have solved Lake Matthewtown's problem (as the island basically acts as a branch: any of these solutions are viable). This did not happen, as the algorithm effectively worked against itself:

When one of Lake Matthewtown's sub-islands found a solution, the champion was encapsulated as a single gadget, and a migration-triggered mutation event took place. In this instance, we only want the champion clump to be paired with other (relative to that clump) no-op gad-

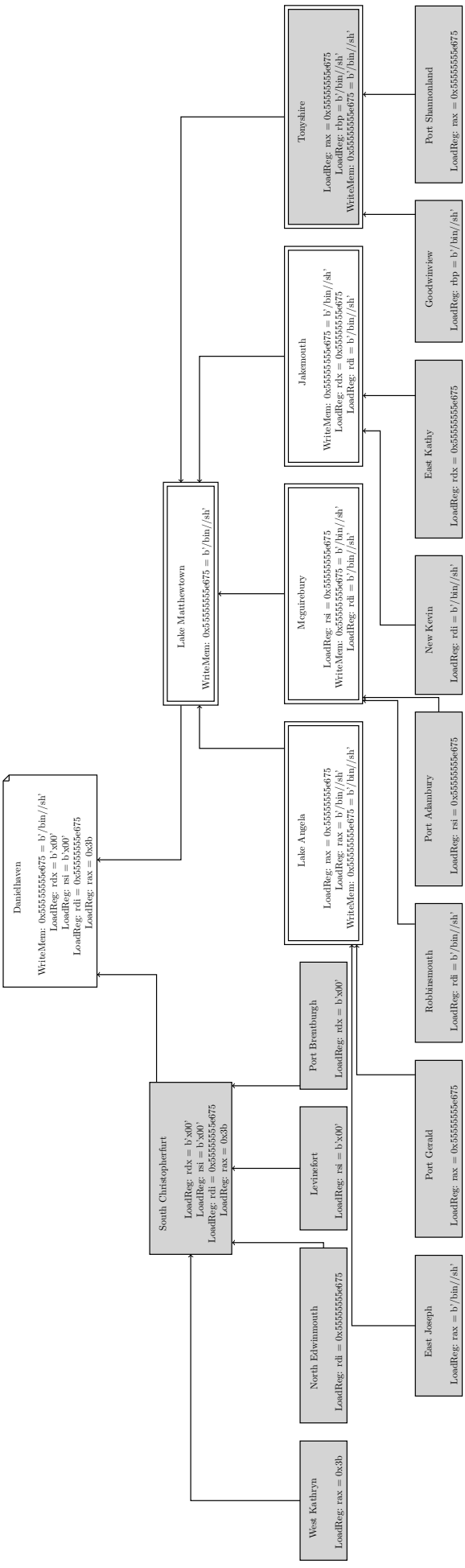


Figure 7.3: Island diagram of one of the failed attempts of generating a payload for the `cat` binary using GENROP. Double-edged boxes indicate active islands, and single-edged boxes indicate inactive islands. Filled boxes indicate islands on which a solution has been found and migrated upwards.

gets. Because the algorithm is designed to only put citizens on the island with gadgets which *could* affect the state of the island's problem, this becomes rather unlikely, to begin with.[4]

To make it even more difficult, due to there begin so few gadgets available in the binary to begin with, it is likely that the gadgets available on Lake Matthewtown had multiple overlapping side effects. It could even be the case that the sub-islands shared multiple multi-pop gadgets. This makes it even more unlikely find a combination where the results of the champion clump was not partially overwritten by some other gadget in the same chain.

This exposes a flaw with the algorithm: we rely too heavily on this migration-triggered event, and have difficulties actually converging towards a solution (and not just randomly stumbling into one).

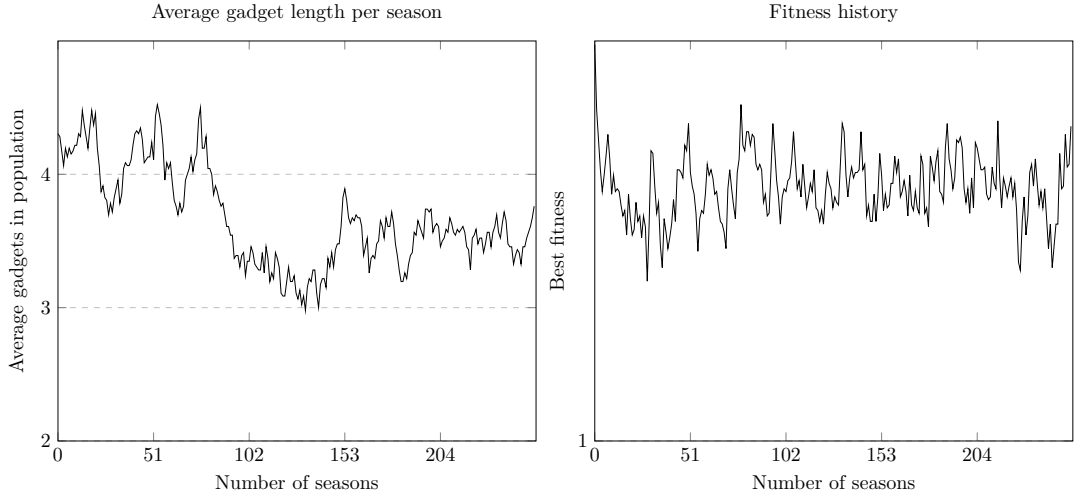


Figure 7.4: Average gadget length and fitness per season, for the island where the payload generation for `cat` stalled.

Looking at the data of the failed payload generation for the `cat` is interesting, because we know that to proceed from the point at which the algorithm failed, it would have to find a combinations of gadgets such that the migrating clump was either:

1. The sole clump in the chain.
2. Accompanied with no-op instructions (relative to the clump), which is highly unlikely, due to the nature of the algorithm.
3. Accompanied with instructions which the clump could ignore (for example through overwriting the results of the other instruction anyway).

In any of these cases, the fewer other gadgets present, the better. We therefore know the probable best strategy for the algorithm to take here: reduce the gadget length of the population; which is indeed what happened, although it did seem to be trending upwards again at the end.

The average fitness history of the population across the seasons does seem to be very rough and jagged, however, which is likely due to the very high mutation rate that was used (40%). This is a difficult but important metric set properly, as with a mutation rate that is set too low, the algorithm too easily fall into local minimum (causing the algorithm to be unable to actually find a solution), while a high mutation rate results in a very volatile population without much direction.

[4]It should be noted that while this is a problem with the implementation of GENROP, it is not a problem with the fundamental design, and is something that could be worked around.

7.2.3 Statistics from the successful payload generations

ls				
Number of runs	Successful		Failed	
	10		0	
Average ...	mean	std.dev	min	max
... # gadgets	6.8	1.4	6	11
... # islands used	6.4	0.47	6	7
... # modified registers	11.0	0.0	11	11
... # memory changes	0.4	0.47	0	1
... # complex clumps	5.4	0.47	5	6
... # mutations	3.4	2.14	1	9
... # season elapsed	36.2	14.73	17	71
... block length	89.5	14.23	67	113
... stack change	484.0	55.66	400	600

Figure 7.5: Result summary of /usr/bin/ls.

rsync				
Number of runs	Successful		Failed	
	10		0	
Average ...	mean	std.dev	min	max
... # gadgets	8.1	0.79	7	10
... # islands used	5.1	0.67	4	6
... # modified registers	10.0	0.0	10	10
... # memory changes	0.0	0.0	0	0
... # complex clumps	4.1	0.67	3	5
... # mutations	8.0	15.6	1	57
... # season elapsed	47.6	33.07	13	125
... block length	64.2	9.14	47	85
... stack change	306.6	38.56	225	393

Figure 7.6: Result summary of /usr/bin/rsync.

level0				
Number of runs	Successful		Failed	
	10		0	
Average ...	mean	std.dev	min	max
... # gadgets	8.7	0.74	8	10
... # islands used	7.6	0.87	6	9
... # modified registers	9.3	2.09	6	12
... # memory changes	0.0	0.0	0	0
... # complex clumps	6.6	0.87	5	8
... # mutations	2.6	0.87	1	4
... # season elapsed	30.3	12.55	12	51
... block length	41.1	11.37	26	70
... stack change	179.4	34.46	121	257

Figure 7.7: Result summary of /usr/bin/level0.

git				
Number of runs	Successful		Failed	
	10		0	
Average ...	mean	std.dev	min	max
... # gadgets	8.4	0.63	7	9
... # islands used	6.6	0.97	5	8
... # modified registers	9.6	1.22	8	12
... # memory changes	0.2	0.38	0	1
... # complex clumps	5.6	0.97	4	7
... # mutations	2.3	1.05	1	5
... # season elapsed	10.8	1.9	8	14
... block length	60.9	12.38	40	77
... stack change	333.0	88.05	209	489

Figure 7.8: Result summary of /usr/bin/git.

Going through the detailed statistics for successful payload generations sheds light on the algorithm from the other point of view, i.e. what went right.

We can see, by looking at the results tables for all the binaries for which we generated a payload successfully, that generally, the algorithm performed as we would expect it to.

A *complex clump* is a clump which encapsulates one or more other clumps (which in turn could be complex clumps themselves). This metric is quite high for each binary, and is consistently so, without much deviance between the runs. This tells us the divide and conquer approach of our algorithm is heavily leveraged in the generation of the payloads. This is also further emphasized by the "# island used" metric, which tells us how many islands were used as part of the final payload (i.e. some genes from here, some genes from there, etc.).

The "# mutations" that were used is also an interesting metric. It varied *wildly* from run to run, and from binary to binary. The metric explains how many times the final payload was part of a mutation during any point in time in its history. The variance can mostly be explained by the fact that the longer the algorithm runs, the more chance there are for mutations to occur. This can also be seen in Figure 7.9. With a Pearson correlation coefficient of $r \approx 0.53$, this indicates a decently strong correlation between the number of seasons elapsed and number of mutations.

Variance in run time is normal for a GP-loop, as being lucky and picking the correct gradient to follow would lead to a very fast solution, whereas in other cases it can take much longer as the algorithm chases multiple local minima.

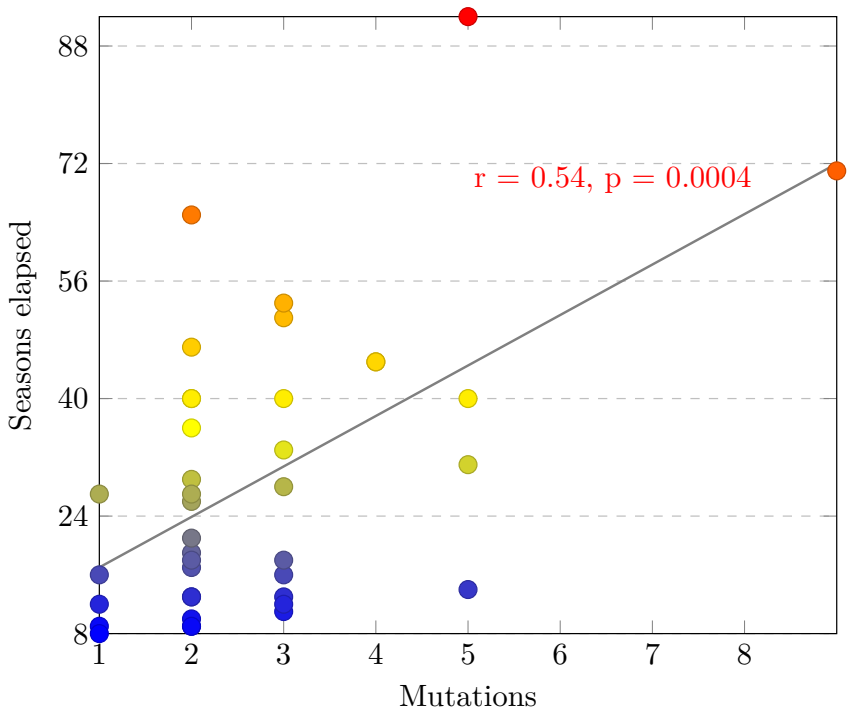


Figure 7.9: Correlating number of mutations and seasons elapsed.

Other metrics, such as the average ”stack change” and ”block length” are, on the surface, less interesting, as the algorithm works on the basis that it either worked or it did not work, and these metrics were simply side effects of the chosen solution. I was however curious to whether or not these metrics had anything to do with solution difficulty, as this could then be useful when designing the fitness function. A large stack change might not only result in a less useful final payload, but might also correlate with a lower chance of even generating a payload in the first place.

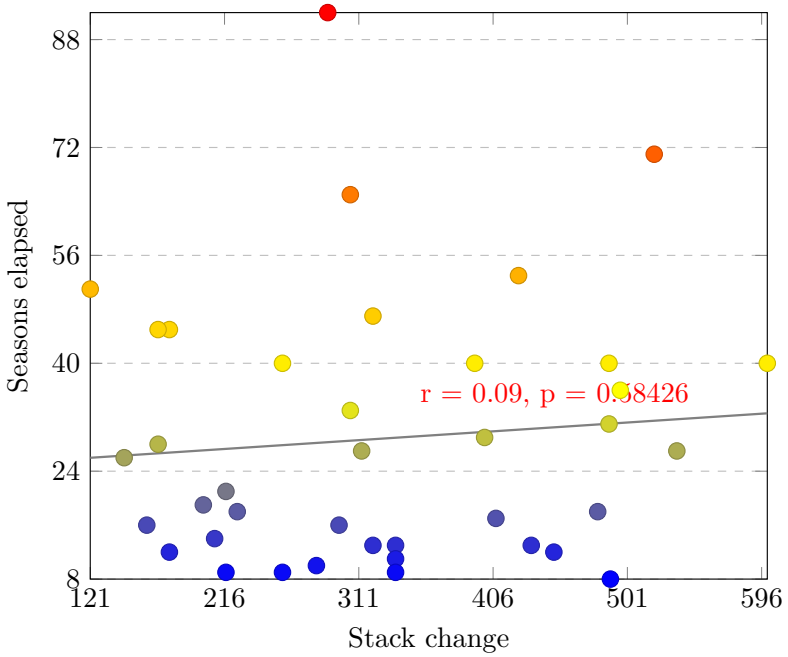


Figure 7.10: Correlating stack change and seasons elapsed.

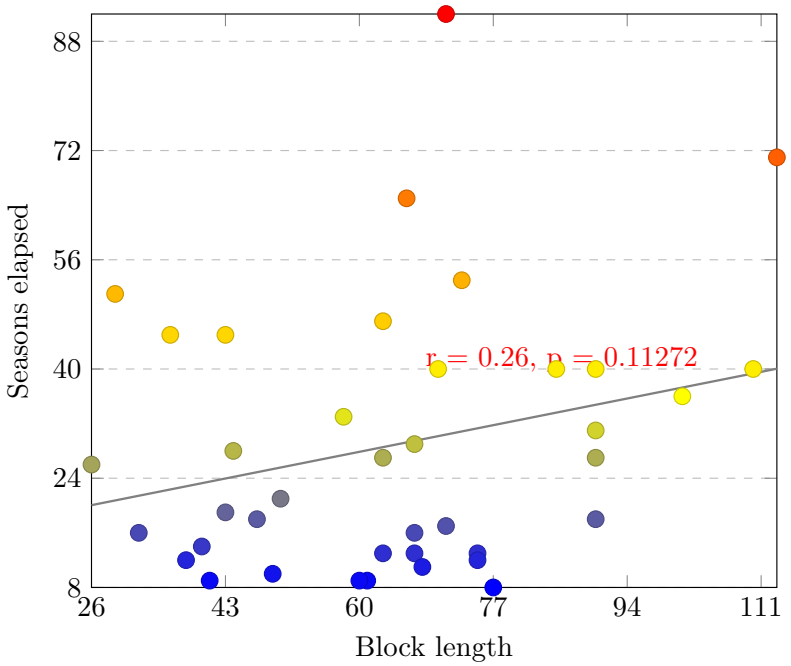


Figure 7.11: Correlating block length of the final payload and seasons elapsed.

The plots in Figure 7.10 attempt to visualize if there is a correlation. Here we are using seasons elapsed as a proxy for difficulty in finding a solution. The parameter p is the Pearson correlation of the sample, and r is the probability that an uncorrelated population would produce a sample with Pearson correlation equal to p.

These plots showcase that there was absolutely no evidence of this correlation between the number of "stack changes" and seasons elapsed. This is not entirely surprising, however, as this does not directly affect how the gadgets function, and is more or less only metadata.

There does however seem to be, according to Figure 7.11 some hints of correlation between the average "block length" of the gadgets used and the difficulty of generating a solution. This correlation was not very strong, however, with a Pearson correlation coefficient of $r \approx 0.26$. Additionally, with a p-value of $p \approx 0.12$, we cannot be very confident that this correlation was not just the result of random chance.

7.2.4 Comparing payloads from `angrop` and `GENROP`

When directly comparing payloads generated from `angrop` and `GENROP` it sadly looks rather bleak, as the payload quality is night and day between the two ROP-compilers.

```
base_addr = 0x0
chain = ""
chain += p64(0x778c4 + base_addr) # pop rax; ret
chain += p64(0x3b)
chain += p64(0xaf004 + base_addr) # pop rdi; ret
chain += p64(0x555555847f65)
chain += p64(0x4cb99 + base_addr) # pop rdx; ret
chain += p64(0x0)
chain += p64(0x1c6001 + base_addr) # pop rsi; ret
chain += p64(0x0)
chain += p64(0x188cf3 + base_addr) # pop rax; pop rbx; ret
chain += p64(0x68732f2f6e69622f)
chain += p64(0x555555847f65)
chain += p64(0xfd332 + base_addr) # mov qword ptr [rbx], rax; pop rbx; ret
chain += p64(0x0)

```

Figure 7.12: angrop payload generated for /usr/bin/git

Figure 4 is a great example of the bad payload quality generated by `GENROP`. Among other problems, the payload overwrites `rax` *six* times before finally setting it to `0x3b`. There are two completely unnecessary gadgets as part of the payload: number three and four. While the remaining gadgets cannot obviously be merged, these are probably not the most optimal gadgets either. Looking at the additional padding required by the gadgets also shows that they are extremely space inefficient. Additionally, there are side effects all over the place, and many registers which did not have to be modified at all have been touched.

In comparison, `angrop` very efficiently uses small gadgets, that only affect as little as possible, while maintaining a small memory footprint. These payloads are indeed worlds apart.

The reason for the bad quality of payloads generated by `GENROP` lies in the difficulties inherent in designing a decent fitness function. If we want to set `rax` to a specific value, we gather all gadgets which can affect `rax` in *any* relevant form. We then attempt to create a heuristic which can decide whether or not `rax` got any closer to the target value *after* a chain of gadgets have been executed.

Let us say that initially, the value in `rax` was `0x77710000efefef`, and after a chain has successfully executed, it becomes `0x400ef`. Numerically, `0x400ef` is indeed closer to `0x3b` than the original value was. The chain therefore gets a relatively good fitness score, never mind the fact that `0x400ef` is (in the case of ROP, and for our intended purposes), not any closer to `0x3b` than `0x77710000efefef` is.

You, could, on the other hand, use a binary fitness heuristic (for this metric only), and say that either `rax` was modified, and that is a good thing, or it was not modified, and that is a bad thing[5] In the end it has the same result: the population is filled with citizens that most likely modify a certain register multiple times, as simply touching a register is positive, through the eyes of this heuristic.

How to improve this fitness-function dilemma is not entirely clear, though, as it is *very* difficult to tell how far from a solution a certain chain is, simply by looking at how it affected the CPU-state.

Another issue is the fact that while a single gadget could potentially set all relevant registers in the same instruction (a multi-`pop` call), this will likely never happen with the design of this algorithm: We split the task of finding a solution into multiple islands — each island with their own goal — and then subsequently wrap these solutions, or clumps, as single gadgets.

[5]If a register was modified, it means that potentially that chain could be mutated so that the correct value is added as an argument to a pop instruction for the register.

```
p = b""
p += p64(0x555555579908) # pop rax; pop rbx; pop rbp; ret ;
p += p64(0x68732f2f6e69622f)
p += p64(0x555555847f65)
p += p64(0x68732f2f6e69622f)
p += p64(0x555555658a9d) # mov qword ptr [rbx], rbp; add rsp, 8; pop rbx; pop rbp; ret ;
p += p64(0x68732f2f6e69622f)
p += p64(0x68732f2f6e69622f)
p += b"A"*8 # padding
p += p64(0x5555555f2d70) # pop rsi; add rsp, 0x30; xor eax, eax; pop rbx; pop rbp; pop r12; ret ;
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += b"A"*48 # padding
p += p64(0x5555555a2585) # pop rax; pop rbx; pop rbp; pop r12; pop r13; pop r14; pop r15; ret ;
p += p64(0x68732f2f6e69622f)
p += p64(0x68732f2f6e69622f)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x68732f2f6e69622f)
p += p64(0x5555555f2d70) # pop rsi; add rsp, 0x30; xor eax, eax; pop rbx; pop rbp; pop r12; ret ;
p += p64(0x0)
p += p64(0x0)
p += p64(0x0)
p += p64(0x0)
p += b"A"*48 # padding
p += p64(0x5555557633e2) # pop rdx; xor eax, eax; add rsp, 8; ret ;
p += p64(0x0)
p += b"A"*8 # padding
p += p64(0x555555579908) # pop rax; pop rbx; pop rbp; ret ;
p += p64(0x3b)
p += p64(0x3b)
p += p64(0x3b)
p += p64(0x5555555f1c3f) # pop rdi; add rsp, 0x28; pop rbx; pop rbp; pop r12; pop r13; ret ;
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += p64(0x555555847f65)
p += b"A"*40 # padding
p += p64(0x5555555eaa91) # syscall

```

Listing 4: genrop payload generated for /usr/bin/git

It is then highly likely that both of these clumps will be part of the final solution separately, even though they could theoretically have been merged.

Because of this design choice, GENROP will always generate rather large payloads.

8 Conclusion

Arriving at a conclusion based upon the implementation and its results.

8.1 Differences between design and implementation

While most of the ideas from the design of GENROP worked out as expected, some aspects did not entirely function as originally intended. As these differences might be important to understand when evaluating the success of the ROP-compiler, they are listed here:

1. The encapsulation and migration of the solution to a subproblem was not general enough to be usable by more than a single parent island. This results in a more ordinary tree-based structure, where a child can only have a single parent. This counters part of the building block hypothesis, and subsequently reduces the effectiveness of the algorithm, as we cannot reuse blocks in the way originally envisioned.
2. As the machine used for running the algorithm had limited computational resources, the algorithm is heavily skewed for converging quickly. This will have affected both the design of the algorithm, the parameters used, and the results themselves.

8.2 Answering the research questions

We will start by first explicitly answering the research questions posed during the introduction.

8.2.1 What are the challenges inherent in generating *ROP-chains* using genetic programming?

The challenges inherent in generating ROP-chains using genetic programming are multiple:

1. Exceedingly large number of potential GP-operators (gadgets), which cause state space explosions. In other cases, where genetic programming has been successfully used, the number of GP-operators are very limited.[1]

[1] Examples include the different problem solutions posed in Chapter 3, where a multitude of GP structures were proposed.

2. The ROP-domain, or the environment in which our ROP-chains operate, is not robust, which results in a high percentage of ROP-chains that crash early. This makes the fitness of an individual difficult to measure.

3. Due to the discrete nature of the underlying architecture and any typical problem description (such as an `execv` attack), very specific solutions are required, and approximations are mostly not good enough.[2]

4. The ROP-domain inherently nets the GP-loop a *very* uneven fitness landscape, which makes it difficult to identify any form of gradient to follow. A ROP-chain that might solve the entire problem description might also be a single argument value change from doing absolutely nothing to the CPU-state. This degenerates the GP-loop to something more akin to a random search.

All in all, the ROP-environment is *not* suited for working properly with classic implementations of genetic programming.

8.2.2 How do we adapt the genetic programming algorithm to work in a ROP environment?

To effectively use genetic programming to generate ROP-chains, we introduced GENROP, of which its most important characteristics can be summarized as following:

1. Counteract the problem with too many GP-operators. The solution presented in this paper was to lean on a new niching method wherein the population was split into multiple islands with different target goals, and with different gadget pools (only relevant gadgets were included in each gadget pool).

2. Create a selection scheme that can utilize the unique nature of the island graph structure. In this paper we introduced *bin selection*, which effectively keeps the population from becoming homogeneous, by selecting the individuals from multiple niches.

3. Use traditional fitness sharing, to further emphasize the importance of niches.

4. Allow solutions for a given sub goal to be encapsulated and reused as gadgets for higher level goals. This allows for variable length solutions, while not introducing additional bloat.

This system allows us to separate the problem into small subproblems, which are much more easily solvable in isolation than trying to solve all the problems simultaneously. These solutions are then recombined to generate a solution to all the subproblems. In other words, divide and conquer.

8.3 Discussion regarding the viability of pairing ROP and GP

All in all, we were able to adapt GP to work as a semi-viable strategy for generating ROP-chains. There are a few problems with the design and implementation, which were highlighted in Chapter 7, all of which have a number of solutions not too terribly difficult to implement. The question of whether or not this is something to pursue further, however, remains.

By adapting GP to work for the ROP-domain, we have lost much of what makes GP good in the first place. There are two extremes here:

[2] It is however the case that one solution might be better than another, in terms of side effects and size of the payload. A worse solution could be seen as an approximation to a better solution. In this paper we focused on achieving a solution at all as the only measure of success. If a solution was worse than that, it did not solve the problem, and was therefore not good enough.

1. Either you guide the GP-loop in very strict ways, putting up railing along the path, refusing the allow it to explore paths that you know (or think) are not worth it. This implementation *does* generate working payloads, but they still resemble payloads generated by other, more traditional tools. In most ways this implementation approximates or imitates a more analytical approach.

2. You loosen the grip, and allow the GP-loop to explore more freely, with less hand holding. This approach has trouble generating any solutions at all, due to the nature of the ROP-domain. Solutions are bound to more novel, however, if they exist.

Finding the middle ground was the goal, but it seems exceedingly difficult to find, if it even exists.

It does seem like, in the case of `GENROP`, that we edged too close to extreme (1). When a chain is easy to build, we are consistently capable of generating a solution. On the contrary, when a chain is difficult to build, we are consistently unable to generate a solution. While the exact solutions and their generation times might differ, the binary nature of success and failure still holds true.

The inherent challenges of matching an algorithm which implicitly works best with imprecise domains where only an approximation of a solution is required, and where the fitness landscape is continuous, is too orthogonal to the ROP-domain; which is jagged, precise and discrete. While there often exists multiple viable solutions to a problem in any given binary, an approximation of one such solution is not good enough. The fitness landscape is such that a candidate might be a single mutation away from the solution, while its effect on the CPU-state might indicate something else entirely. This is one property among others that disallows an effective coupling of GP and ROP.

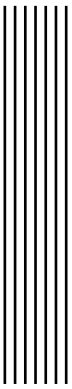
While it does sounds like a neat idea in theory: let the machine solve the problem for you, while you specify the problem and its parameters; in practice, this approach is much too weak and fragile, and still needs just as much hand holding as a more traditional approach to the problem. The domain simply is too specific, too precise, and too demanding to be properly solvable by a GP-loop. When attempting to make the solution fit the problem, we have lost much of what made the proposed solution worth pursuing in the first place.

8.3.1 Future Work

Possible future work on a ROP-compiler which leverages GP should attempt to explore the opposite direction of the approach put forward in this thesis; instead of attempting to find ways of guiding the algorithm as much as possible, we should attempt to design an environment in which we can evaluate ROP-chains as fast as possible. This would allow for less guided GP-loops, and would properly enable the genetic part of the algorithm.

Another interesting topic to explore would be that of gadget extraction. In this thesis we used gadget definitions generated by `angrop`, which limited the algorithm by only ever attempting to use gadgets which do not branch or alter the control flow in difficult to control ways. By also extracting branching and other more complex gadgets, we could theoretically unlock the ability to generate otherwise impossible solutions, which could be applicable for binaries for which the current compiler cannot find solutions for.

All in all, although the results in this thesis are somewhat negative, there are still avenues to explore in this domain.



Bibliography

[1] Artur Adib. *Slash/A, Programming language and library for C++*. https://github.com/arturadib/slash-a. [Online; Accessed 26 August 2019].

[2] William B. Langdon, Riccardo Poli, Nicholas Mcphee, and John Koza. "Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications." In: vol. 115. Jan. 2008, pp. 927–1028. DOI: 10.1007/978-3-540-78293-3_22.

[3] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*. Vol. 1. Morgan Kaufmann San Francisco, 1998.

[4] Markus Brameier and Wolfgang Banzhaf. "Neutral variations cause bloat in linear GP." In: *European Conference on Genetic Programming*. Springer. 2003, pp. 286–296.

[5] Sergey Bratus. *What Hacker Research Taught Me*. https://www.cs.dartmouth.edu/~sergey/hc/rss-hacker-research.pdf. [Online; Accessed 8 March 2019]. 2009.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. "BAP: A binary analysis platform." In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469.

[7] Catherine Collin and Stephen Eglen. "An Introduction to Natural Computation." In: *Trends in cognitive sciences* 2 (June 1998), pp. 235–6. DOI: 10.1016/S1364-6613(98)01182-6.

[8] angrop community. *angrop source code*. https://github.com/angr/angrop/blob/571e28ac07d4cec208a9e8e0327bdcea5f9e76bb/angrop/chain_builder/__init__.py. [Online; Accessed 06 April 2021].

[9] Kenneth Alan De Jong. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." AAI7609381. PhD thesis. USA, 1975.

[10] David Brumley Edward J. Schwartz Thanassis Avgerinos. *Q: Exploit Hardening Made Easy*. https://www.usenix.org/event/sec11/tech/slides/schwartz.pdf. [Online; Accessed 8 March 2019].

[11] Andreas Follner. "On Generating Gadget Chains for Return-Oriented Programming." In: (2017).

[12] Andreas Follner, Alexandre Bartel, Hui Peng, Yu-Chen Chang, Kyriakos Ispoglou, Mathias Payer, and Eric Bodden. "PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution." In: *Security and Trust Management*. Springer International Publishing, 2016, pp. 212–228. DOI: 10.1007/978-3-319-46598-2_15. URL: https://doi.org/10.1007%2F978-3-319-46598-2_15.

[13] Olivia Fraser. *Urschleim in Silicon: Return Oriented Programming Evolution With ROPER*. Tech. rep. 2018.

[14] g0tmi1k. *VulnHub*. https://www.vulnhub.com/. [Online; Accessed 07 21 2021]. 2021.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[16] David E. Goldberg and Kalyanmoy Deb. "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms." In: *FOGA*. 1990.

[17] Steve Hanov. "Static Analysis of Binary Executables." In: 2007.

[18] Wolfgang Kantschik and Wolfgang Banzhaf. "Linear-graph GP-a new GP structure." In: *European Conference on Genetic Programming*. Springer. 2002, pp. 83–92.

[19] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5.

[20] Angel Fernando Kuri-Morales, Edwin Aldana-Bobadilla, and Ignacio López-Peña. "The best genetic algorithm II." In: *Mexican International Conference on Artificial Intelligence*. Springer. 2013, pp. 16–29.

[21] Brian Lonsdorf. *Professor Frisby's Mostly Adequate Guide to Functional Programming*. https://github.com/MostlyAdequate/mostly-adequate-guide/blob/master/ch03.md. 2015.

[22] Samir W Mahfoud. "Crowding and preselection revisited." In: *PPSN*. Vol. 2. 1992, pp. 27–36.

[23] Angel Kuri Morales and Carlos Villegas Quezada. "A universal eclectic genetic algorithm for constrained optimization." In: *Proceedings of the 6th European congress on intelligent techniques and soft computing*. Vol. 1. 1998, pp. 518–522.

[24] Elena Simona Nicoară. "Mechanisms to Avoid the Premature Convergence of Genetic Algorithms." In: *Petroleum-Gas University of Ploiesti Bulletin, Mathematics-Informatics-Physics Series* 61.1 (2009).

[25] pakt. *Turing complete ROP compiler*. https://github.com/pakt/ROPC. [Online; Accessed 8 March 2019].

[26] Timothy Perkis. "Stack-Based Genetic Programming." In: *ICEC* (1994).

[27] Riccardo Poli et al. "Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming." In: *ICGA*. Citeseer. 1997, pp. 346–353.

[28] Riccardo Poli, Nicholas Freitag McPhee, and Leonardo Vanneschi. "Elitism reduces bloat in genetic programming." In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. Citeseer. 2008, pp. 1343–1344.

[29] John R. Koza I and Riccardo Poli. *A Genetic Programming Tutorial*. Tech. rep. Standford University, University of Essex, June 2003.

[30] G Rudolph. "Evolutionary search under partially ordered sets." In: *Dept. Comput. Sci./LS11, Univ. Dortmund, Dortmund, Germany, Tech. Rep. CI-67/99* (1999).

[31] Jonathan Salwan. *angrop is a rop gadget finder and chain builder*. https://github.com/JonathanSalwan/ROPGadget. [Online; Accessed 8 March 2019].

[32] Bruno Sareni and Laurent Krahenbuhl. “Fitness sharing and niching methods revisited.” In: *IEEE transactions on Evolutionary Computation* 2.3 (1998), pp. 97–106.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis.” In: (2016).

[34] Robert E Smith, Stephanie Forrest, and Alan S Perelson. “Searching for diverse, cooperative populations with genetic algorithms.” In: *Evolutionary computation* 1.2 (1993), pp. 127–149.

[35] Nick Stephens. *angrop is a rop gadget finder and chain builder*. `https://github.com/salls/angrop`. [Online; Accessed 8 March 2019].

[36] Jeff Stewart and Veer Dedhia. “ROP Compiler.” In: (2015).

[37] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. “Island model genetic algorithms and linearly separable problems.” In: *AISB International Workshop on Evolutionary Computing*. Springer. 1997, pp. 109–125.

[38] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. “The island model genetic algorithm: On separability, population size and convergence.” In: *Journal of computing and information technology* 7.1 (1999), pp. 33–47.

[39] Tian-Li Yu, David E Goldberg, Ali Yassine, and Ying-Ping Chen. “Genetic algorithm design inspired by organizational theory: Pilot study of a dependency structure matrix driven genetic algorithm.” In: *Genetic and Evolutionary Computation Conference*. Springer. 2003, pp. 1620–1621.

[40] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” In: *Evolutionary computation* 8.2 (2000), pp. 173–195.