Return-Oriented Programme Evolution with ROPER

Olivia Lucca Fraser
NIMS Laboratory, Dalhousie University
6050 University Ave
Halifax, NS B3H 1W5
lucca.fraser@gmail.com

Nur Zincir-Heywood
NIMS Laboratory, Dalhousie University
6050 University Ave
Halifax, NS B3H 1W5
webmaster@marysville-ohio.com

Malcolm Heywood
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

John from Raytheon
Brookhaven Laboratories
P.O. Box 5000
lleipuner@researchlabs.org

ABSTRACT

This paper provides a sample of a LaTeX document which conforms, somewhat loosely, to the formatting guidelines for ACM SIG Proceedings[1].

CCS CONCEPTS

•**Computing methodologies** → **Genetic programming;** •**Security and privacy** → *Malware and its mitigation;* •**Software and its engineering** → Assembly languages;

KEYWORDS

Genetic programming, Exploit development, ROP attacks, ARM architecture

ACM Reference format:
Olivia Lucca Fraser, Nur Zincir-Heywood, Malcolm Heywood, and John from Raytheon. 2017. Return-Oriented Programme Evolution with ROPER. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 5 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Roper is a genetic compiler that evolves payloads for return-oriented programming (rop) attacks. These are attacks that manipulate their host's control flow in subtle and fine-grained ways, and, unlike traditional shellcode attacks, they do this without at any point introducing foreign code, or writing to executable memory. Since it is becoming increasingly rare for processes to map *any* segment of memory as both writeable and executable – due to a defensive measure called 'Data Execution Prevention' (dep) when implemented on Windows, or 'Write xor Execute' ($W \oplus X$), when implemented in a Unix environment – return-oriented programming (or rop) has become the industry standard approach to payloads in binary exploit development.

ROP works by sifting through the host process's executable memory – its `.text` segment, if we're dealing with elf binaries – and finding chunks of code that can be rearranged in such a way that they carry out the attacker's wishes, rather than their intended design. For these chunks to be usable in an attack, however, it must be possible to jump from one to the other in a predetermined sequence. This is where the 'return-oriented' nature of the attack comes in: most architectures implement subroutine or function calls by first pushing the address of the instruction *after* the call to the stack, and then jumping to the first instruction of a subroutine that, itself, ends by popping the bookmarked 'return address' from the stack. In a rop attack, we exploit this manner of implementing returns. We set things up so that the 'return address' popped from the stack at the end of each 'gadget' is just a pointer to the next gadget we wish to execute. This lets us chain together indefinitely many rop gadgets in sequence, and it is, in principle, possible to implement complex attacks in this fashion, without ever needing to use any executable code that isn't already there, waiting for us in the process's executable memory segment.[2]

These chains are the kind of entities that our engine evolves. The genetic material consists of the set of gadgets extracted from a target executable binary – we focus for now on elf binaries compiled for 32-bit arm processors. The individual genotypes are rop-chains, formed from this material. The phenotype, on which selection pressures are brought to bear, is the behaviour these genotypes exhibit when executed in a virtual (but realistic) cpu. The entire set up resembles a variation on linear genetic programming, but with a few key differences, required by the nature of the problem at hand. The goal is to not simply automate the somewhat tricky and time-consuming human task of assembling rop-chain payloads – though roper does that quite well – but to explore an entirely novel class of payloads: rop-chains that

[1] This is an abstract footnote

[2] **Cite that research paper that describes an entire rootkit written in rop chains**

GECCO '17, Berlin, Germany

DOI: 10.475/123_4

exhibit the sort of subtle and adaptive behaviour for which we normally turn to machine learning.

As a proof of concept, here, I will show how we succeeded in evolving ROP-chain payloads that cannibalize arbitrary binaries into mosaics capable of solving a few traditional benchmark classification problems with high degrees of accuracy, beginning with the famous Iris dataset. Without injecting a single foreign instruction, we will coax system and backend binaries into tasks that resemble nothing they were designed to do, and nothing that has previously been attempted in low-level binary exploitation: we'll have them sort flowers.

2 BACKGROUND

2.1 Return-Oriented Programming

2.2 Prior Research and Development

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

We're aware of two such projects at the moment. The first, named *Q*, is described in [CITE].

The second, ROPC, grew out of its authors' attempts to reverse engineer *Q* from its desrciption in [CITE], and extend its capabilities to the point where it could compile ROP-chains for scripts written in a Turing-complete programming language.

The project has since inspired a fork that aims to use ROPC's own intermediate language as an LLVM backend, which, if successful, would let programmes written in any language that compiles to LLVM's intermediate language, compile to ROPC-generated ROP-chains as well.

Another, particularly interesting contribution in the field of automated ROP-chain generation is *Braille*, by Andrea Bittau *et al.*[3] which automates an attack that its developers term "Blind Return-Oriented Programming", or BROP. BROP solves the problem of developing ROP-chain attacks against processes where not only the source code but the binary itself in unknown. *Braille* first uses a stack-reading technique to probe a vulnerable process (one that is subject to a buffer overflow and which automatically restarts after crashing), to find enough gadgets, through trial and error, for a simple ROP chain whose purpose will be to write the process's executable memory segment to a socket, sending that segment's data back to the attacker – data that is then used, in conjunction with address information obtained through stack-reading, to construct a more elaborate ROP-chain the old-fashioned way. It is an extremely interesting and clever technique, which could, perhaps, be fruitfully combined with the genetic techniques we will outline here.

2.2.2 Evolutionary Computation in Offensive Security.

Prior research at NIMS Laboratory. This is not the first exploration that our lab has made in applying evolutionary computation to the domain of offensive cybersecurity. Gunes Kayacik ...

Patrick LaRoche [port scans]...

Other applications. Other applications of evolutionary methods in offensive security: lcamtuf AFL

3 METHODOLOGY

3.1 Genotype Representation

3.1.1 Gadgets, Clumps, and Chains. Individuals, here, are essentially vectors of 32-bit words, which may be either pointers into executable memory addresses, intended to be popped into the program counter, or other values, intended to be popped into the CPU's other registers.

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

To do this, we calculate the distance the stack pointer will shift when each gadget executes, $\Delta_{SP}(g)$, and then clump together each gadget pointer g with a vector of $\Delta_{SP}(g) - 1$ non-gadget values.[4] These values will populate the CPU's registers when the final, multipop instruction of the gadget is executed. The program counter (`PC`) is always the final register populated through a multipop, and so the address of the next gadget g' should be found exactly $\Delta_{SP}(g)$ slots up from g.[5]

3.1.2 Variation Operators.

Mutation. Structuring the basic units of our genotypes in this way also lets us apply variation operators more intelligently. The genotype is much more tolerant of mutations to the non-gadget values in each clump than to the gadget address itself. The gadget address *may* be safe to increment or decrement by a word or two, but negating, multiplying, or masking it would almost certainly result in a crash. The

[3] CITE THIS

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

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

rest of the words in the clump can be mutated much more freely, either arithmetically, or by indirection/dereference (we can replace a value with a pointer to that value, if one is available, or if a value can already be read as a valid pointer, we can replace it with its referent).

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

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

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

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

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

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

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

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

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

3.2 Phenotype Evaluation

The phenotype, here, is the CPU context resulting from the execution of the genotype (the ROP-chain) in a virtual machine, passed through one of a handful of 'fitness functions'. Let's look at these factors one at a time.

3.2.1 Execution Environment. The transformation of the genotype into its corresponding phenotype – its 'ontogenesis' – takes place in one of a cluster of virtual machines set up for this purpose, using Nguyen Anh Quynh's superb Unicorn Engine emulation library. A cluster of emulator instances is initialized at the beginning of each run, and the binary that we wish to exploit is loaded into its memory. We enforce non-writeability for the process's entire memory, with the sole exception of the stack, where we will be writing our ROP-chains. There are two reasons for this: first, since the task is to evolve pure ROP-chain payloads, we might as well enforce $W \oplus X$ as rigorously as possible – the very defensive measure that ROP was invented to subvert. Second, it makes things far more reliable and efficient if we don't have to worry about any of our chains corrupting their shared execution environment by, say, overwriting instructions in executable memory. This lets us treat each chain as strictly functional: the environment being stable, the output of a chain is uniquely determined by its composition and its inputs.[6]

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

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

The reason a ROP-chain controls the execution path, remember, is that each of the snippets of code ('gadgets') that its pointers refer to ends with a return instruction, which pops an address into the program counter from the stack. In ordinary, non-pathological cases, this address points to the

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

instruction in the code that comes immediately after a function call – it's a bookmark that lets the CPU pick up where it left off, after returning from a function. The cases we're interested in – and engineering – of course, *are* pathological: here, the address that the return instruction pops from the stack doesn't point to the place the function was called from, but to the next gadget that we want the machine to execute. This gadget, in turn, will end by popping the stack into the program counter, and so on, until the stack is exhausted, and a zero is popped into PC. So long as a specimen controls the stack, it's able to maintain control of the program counter.

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

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

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

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

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

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

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

3.3 Selection scheme

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

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

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

There is also a certain probability, in each tournement, that only `t_size` $- 1$ contestants will be chosen, and that instead of being the second-best performer in the tournement, the second parent will be a new chain, randomly generated from scratch. This provides a constant trickle of fresh blood into the gene pool, and helps stave off stagnation.

4 EMPIRICAL STUDY

5 CONCLUSIONS

ACKNOWLEDGMENTS

The authors would like to thank Dr. Yuhua Li for providing the matlab code of the *BEPS* method.

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. The work is supported by the National Natural Science Foundation of China under Grant No.: 61273304 and Young Scientsts' Support Program (http://www.nnsf.cn/youngscientsts).

REFERENCES