A Preliminary Whitepaper on ROPER: A Genetic Tool for the Evolution of ROP-chain Payloads

Olivia Lucca Fraser
Dalhousie University
Halifax, NS
Email: lucca.fraser@gmail.com
Github Repo: github.com/oblivia-simplex/roper.git

***Abstract*—We describe ROPER, an exploit development tool, which remains a work in progress at the time of writing. ROPER analyses a target ELF file, extracts its executable section (.text), and decomposes it into a collection of 'gadgets' – sequences of instructions ending in a return. It then spawns a population of gadget chains ('ROP-chains'), and uses a genetic algorithm to find an optimal combination of gadgets for the task at hand. 'Tasks', for now, are considered as system calls, and a chain of gadgets is said to be fit for such a task when it succeeds in placing the necessary values in the required registers. (More complex tasks can no doubt be envisioned and implemented as well, within this framework.) The aim is to make ROPER as flexible as possible with respect to target architecture, but for now we're focussing on the 32bit ARM architectures.**

***Index Terms*—ROP-chain attacks, binary exploits, genetic algorithms, Unicorn Engine, security**

I. Introduction

There has been surprisingly little work done, as far as I can see, on uses of genetic programming (or even machine learning, more generally) in offensive security, or even in malware development and blackhat applications. Defensive applications of machine learning and evolutionary algorithms abound,[1] but are forced to rely on human – and often malicious – attackers for their training data. An ideal situation would be to allow defensive AIs benefit from an evolutionary arms race with equally capable offensive AI.

One noteable exception is a fuzzing tool recently developed by Michal Zalewski (AKA lcamtuf), called American Fuzzy Lop, which employs a genetic algorithm to seek out uncharted control flows through a target programme (targetting either source code or compiled binary), in search of bugs and vulnerabilities.

Another, closer to my own work, is a research project undertaken by Gunes Kayacick, Nur Zincir-Heywood and Malcolm Heywood in the NIMS Lab, circa 2005-2008. Kayacik et al., developed a framework that uses a genetic algorithm to evolve traditional, "Smashing the Stack"-style shellcode attacks, but which are optimised for IDS undetectability, and which are able to adapt to syscall frequencies that appear "normal" on the target host, in the eyes of various intrusion detection systems.

[1] At the recent AtlSecCon security conference, for example, it seemed that every other vendor was marketting ML-driven firewalls and IDSes of variety or another. I do not currently have the figures to back up this anecdotal impression.

II. Obsolescence of Stack-Smashing Attacks & Rise of Return-Oriented Programming

The attacks evolved by Kayacik's tool, however, attacks are only viable so long as stack memory, or, more generally, writeable memory, is mapped as executable. An attempt to redirect the instruction pointer to nonexecuteable memory will result in a relatively harmless segfault (which may still be of concern as a potential DoS vector, but this is nowhere near as serious as the threat of arbitrary code execution). Non-executable stacks, however, are today the rule rather than the exception, thanks to security features baked into most compilers (both GCC and CLANG use this feature, for example), and so there are relatively few programmes in the wild, nowadays, that are vulnerable to the genus of attacks evolved through Kayacik's framework.

The hacking community discovered ways of circumnavigating the obstacle of non-executable stacks as early as 1997, when Solar Designer posted his now-famous return-to-libc technique to the Bugtraq mailing list. Unlike traditional shellcode attacks, where the attacker places executable code onto the stack (either in an input buffer or an environment variable, for instance) and then redirects the instruction pointer to that code, Solar Designer's attack simply uses code that is *already* mapped to executable memory. Since libc is almost always going to be resident in the executable memory of a Unix process, it makes for a convenient target – all that we need to do is to redirect the instruction pointer to, say, the `system()` function, with the desired parameters, such as a pointer to the string "`/bin/sh`", are placed on the stack. This works when we are dealing with an ABI that expects to find parameters there – the matter is somewhat more complicated when dealing with more modern ABIs, such as ARM and x86_64, where the first few function arguments are passed in registers.

Return-oriented-programming (ROP) is a generalization of this technique. Instead of redirecting execution to already-resident functions, ROP targets a far more general class of sequences, which it assembles into complex chains, cannibalizing the target process and cobbling together new, arbitrary programmes from the scraps of its executable memory.

III. Gadget Extraction and ROP-chain Construction

The sequences that make up a ROP-chain are called "gadgets". What they have in common is just that they culminate in a "return" instruction, or an instruction that restores the programme counter (RIP on x86_64, R15 on ARM, or PC when we're speaking generically) to a predetermined state. Typically, and by convention, this "predetermined state" is the address of the instruction immediately following a function call. The RET instruction on x86 and x86_64 processors, for example, does this by pushing the return address onto the stack (followed by the function's parameters) before jumping to the first instruction in the function. The ARM processor has at least two basic ways of restoring the instruction pointer: when the function is a "leaf" in the control flow, it can quickly restore the return address from the *link register*, where the BL instruction stores it before branching. But if two or more functions are called in a row, or if recursion is used, the return address is stored on the stack, as it is with x86 systems. To simplify matters, I restricted my attention to stack-based returns when dealing with the ARM.

The beauty of stack-based ROP-gadgets is that it is easy to control the address that they "return" to, and nothing but convention compels this address to be the address from which the function was called. If a different address is placed on the stack, then the PC will take it instead (just as it does in traditional shellcode attacks), and execute whatever it finds when that pointer is dereferenced.

A. Specific Challenges Posed by the ARM Architecture

The ARM architecture, in particular, makes for a particularly interesting target of ROP attacks. On the one hand, it is the most widely deployed architecture in the world, at present, residing on innumerable smartphones and embedded devices, and underlying countless insecure and under-scrutinized apps and firmware binaries. The attack surface it represents is tremendous. On the other hand, ARM's Harvard bus architecture, which physically segregates instruction (executable) and data (read/writeable) caches, making it invulnerable to traditional shellcode attacks.

ARM is a RISC architecture, meaning that the instructions for the ARM processor are all of a fixed width. When the processor is operating in arm mode, this width is 32-bits,[2], but the processor can also switch to thumb mode on a dime, whereupon it begins reading and executing 16-bit instructions from the instruction cache (with a handful of 32 bit instructions thrown in, but we can ignore this complication for now). The fixed-width nature of ARM instructions makes them easier to search and scan for returns than CISC architectures, where instructions are of variable sizes and parsing them is a nontrivial operation.

The uneven lengths of CISC instruction sets, however, also affords a greater variety of gadgets. In a CISC instruction set, the most frequently-used instructions tend to be shortest in length, and since RET is extremely frequent, it is usually assembled to a single byte. This means that a crafy misalignment of an instruction sequence can trick the processor into treating *any* occurrence of the return byte (`0xC3` on x86/x86_64) as a return, with the surrounding instructions reinterpreted according to the resulting offset. ARM doesn't have this feature (or bug?), but approximates it to a lesser degree with its dual execution modes: so long as we find a way to set the mode bit, any sequence of n arm instructions can be reinterpreted as a sequence of $2n$ thumb instructions, though the probability of those instructions being meaningless or malformed is greater, since the set of all valid instructions is considerably smaller (though individually more expressive) on the ARM (it is a *restricted* instruction set).

B. Suitability of the Problem for Genetic Approaches

In his 2009 treatise, *Return Oriented Programming for the ARM Architecture*, Tim Kornau advances the following thesis:

> Return-oriented programming on the ARM architecture is possible. If the binary code of libraries for a given operating system can be analysed, there exists an algorithm which can determine whether the given code can construct the necessary gadgets for return-oriented programming. If the necessary gadgets for return-oriented programming exist, there exists an algorithm which can extract the pre- and post-conditions necessary to craft an arbitrary programme with the given gadgets.[3]

Kornau's work seems to place its emphasis on analysis, the proof of a given attack's possibility, and the provision of heuristics for finding and assembling chains of gadgets. The actual work of ROP-chain assembly, however, remains manual (ibid., p. 6).[4]

It seems that genetic approaches may be well-suited to this task, and outperform human chain-composition in efficiency and variety. The process of assembling ROP-chains makes for messy and noise-ridden work, and gadgets selected for a particular function typically carry with them unintentional side-effects, which complicate matters substantially. Automating this process is one of the goals of ROPER.

IV. The Design of ROPER

ROPER divides its operations between a "phylogenetic" client, which manages population dynamics and applies the various evolutionary operators, and an "ontogenetic" server, which maps genotypes to phenotypes and returns the result to the client for purposes of selection.

[2]I'm restricting the discussion to 32 bit ARM processors for the moment, but 64 bit ARM devices are already on the market, and will be examined at a later time

[3]Tim Kornau, *Return Oriented Programming for the ARM Architecture*, thesis submitted to the Ruhr-Universität Bochum, December 22, 2009, pp. 5-6

[4]Disclaimer: I have not yet had the opportunity to study Kornau's work in detail. Doing so wil be a priority for this project, however.

A. Population Initialization and ROP Gadget Extraction

For the time being, I have restricted my attention to arm-mode, stack-based returns in ARM binaries – returns that restore the PC by popping the stack, rather than transferring the value from the link register. Finding stack-based ROP-gadgets in ARM machine code is simple: search for instructions that POP into PC (i.e. STDMI instructions that take the stack pointer (R14) as their address pointer and R15 as their destination register. The ARM instruction set is about as human-readable as machine code gets, and finding a pop that targets R15 is as simple as finding a word whose first two bytes are E8 BD and whose remaining halfword H is such that $H\&(1 << 15)! = 0$. (ARM lets us pop into any combination of registers in parallel: the second halfword of the instruction is simply a bitmap of register space.)

B. Client-side Phylogeny

The ROPER client is written entirely in Common Lisp, and handles the "phylogenic" side of the evolutionary process – the manipulating the population as a whole, applying selection and variation operators, and essentially everything aside from the mapping of genotype to phenotype, which is delegated to an "ontogenetic" server. In this way, it is able to maintain a relatively light memory footprint, as genetic programming goes, while the heavy lifting can be left to a remote machine. Writing the client in Lisp affords the user a rich and interactive environment for experimentation, as well, so long as she is comfortable working with the REPL, which is the only user interface that ROPER has as of yet.

In what follows, I will review three main aspects of the ROPER client: how it represents the ROP-chain genotype, what selection operations it is currently using, and what variation operators are available. All of this is subject to change in future iterations.

1) Genotype Representation: Once harvested from the target binary, the gadgets are entered into a hashtable, indexed by initial address. These will be the basic "genes" in our gene-pool.

Individuals are variable-length sequences of these gadgets (or rather, to save space, of their addresses/hash-keys). The population is initialized by first taking the set of all singleton chains (chains containing just one gadget) and then taking an equal number of chains consisting of between two and five gadgets. This is, of course, easily adjusted (and one of my reasons for coding the client in Lisp is that it makes for quick and easy experimentation and reparameterization in an interactive environment), and there is no reason to expect this setup to be the best. The guiding intuition, though, is that since the pool of gadgets already consists of complex operations, it is reasonably probable that a single gadget already solves the problem at hand, and this might be missed if we neglected any singleton chains. I leave it to the variation operations to generate longer and more complex chains from these basic "building blocks".

2) Selection Operators: I have experimented with two different selection operators so far, though neither has yet resulted in much success. Both were chosen with an eye to achieving a common objective: the evolution of ROP-chains that would bring a subset of the machine's registers into the desired state. This state is represented by a fairly basic schema language, whose expressions are vectors of fixed-width unsigned integers and wildcard tokens. The vector

```
#(1 _ _ 8 _ 4 _ 1024)
```

, for example, matches any configuration of registers that has 1 in R0, 8 in R3, 4 in R5, and 1024 in R7 (or similarly for x86_64).

a) Tournement selection with a euclidean fitness function.: The schemas described above can be seen as a hypersurface in eucldean space, given by the nonwildcard coordinates. This suggests one obvious candidate for a fitness function: the euclidean distance between the hyperplane described by salient register-values set by the ROP-chain, and the hyperplane described by the target schema. A few minor tweaks should be kept in mind, allowing for integer overflow, and so on (on a 32-bit register, 0xFFFFFFFF should be seen as just 1 unit away from 0x00000000, etc.), but this seemed like a good place to start.

A moderate degree of convergence was seen in some early trials, but nothing altogether encouraging,[5] given the expected simplicity of the task (by which I mean, the optimism of the programmer). I considered that quasi-continuous approximation might not be the way to go, given the exactitude of the goal – if we need to have the value 17 in R7 for a syscall, 18 will never be "good enough". (There are, of course, going to be cases where approximation is acceptable – if I want my chain to call mmap and allocate about two hundred bytes of executable memory, for example, I wouldn't turn up my nose at 256, or 512, for that matter, but 0xFFFFFFFF might be a bit excessive, and 3 would be insufficient.)

b) Lexicase selection, or something similar.: Though we are not dealing with a "classification problem" in any meaningful sense, it occurred to me that it might be fruitful to exploit something similar to the Lexicase technique introduced by

The basic idea would be to first calculate the register states resulting from each individual in the population, when executed on the same initial register context (simply zeros for

[5]UPDATE: While I was writing this paragraph, ROPER was humming along in the background, and actually *did* achieve convergence using the tournement/euclidean-distance method, even while I was explaining away its failures. It may have been the case that I had given the engine a problem that it simply could not solve – I have, after all, not ascertained the Turing completeness of the gadgets extracted from the test binary (a static compilation of ldconfig.real, pilfered from a Raspbian installation). The winning ROP-chain, in this case, was a thirty-two-gadget-long chain (and therefore one that was several generations away from the origin, since the maximum starting length was just four gadgets), and the target pattern was

```
#(1_4___5_)
```

now, but eventually this will be set to the register state observed at the moment(s) in the target proces at which control-flow hijacking becomes possible), shuffle the target schema (which is stored as an key-value list of indices and register states), and then begin filtering the entire population (or some large subset thereof) with respect to *absolute* correctness on each target register. When the entire population is eliminated, or the entire sequence of key-value pairs has been exhausted, a winner is chosen from last survivors. The process is repeated using the same key-value list in reverse, and a second winner is selected. This is how the parents of each successive generation are chosen. The intuition is that by reversing the list, we increase the odds of combining different "specialist" parents – a parent that is "good at setting R0", for example, with one that is "good at setting R7".

This method has not met with any noticeable success.

3) Variation Operators: Both crossover and mutation algorithms have been implemented. The population is initialized as a set of very short chains, which is counterbalanced by variation operators that favour growth. For sexual reproduction, one-point crossover is used. For asexual production, one of a handful of mutation functions is chosen. These include: push a new, random gadget onto the chain; pop a gadget off the chain; shuffle the chain; and shrink a randomly selected gadget in the chain, in the direction of the return instruction.

C. Ontogenesis as a Service

ROPER is built according to a server-client architecture. The client takes care of the strictly "phylogenic" operations – managing the population, reproduction, variation, and so on – while the server (or servers) handle "ontogeny", translating genotype to phenotype by executing the code that makes up each ROP-chain and returning the resulting register state.

Common Lisp was chosen as the development language for the phylogenic/client components, due to its flexibility in handling high-level list-manipulation operations, and the ease at which different evolutionary methods can be prototyped and experimented with.

As nimble as lisp is for high-level operations, however, it can be a relatively frustrating tool for working on the level of machine code. There is no implementation-independent way, for example, to simply poke machine code into memory and excute it, as there is in C or BASIC, for example. (The foreign-function-interface library CFFI can be used, to some extent, but the situation turned out to be far from stable, especially when executing arbitrary sequences of bytes in rapid succession! And it meant depending on C, in any case.)

I decided to split the workload of the programme, and code a server entirely in C, where I could take advantage of the language's precision and transparency with respect to the underlying architecture. Though this meant incurrent some additional overhead in runtime – data can't be packetized and sent across the wire for free – it had the added potential benefit of delegating the demanding work of code excution to other, more powerful computers than my own workstation, and would leave the phylogenic engine unharmed when the ontogenic engines miscarry. (The server eventually turned out to be quite stable, especially once the decision was made to virtualize the architecture with Unicorn, but instability seemed like a reasonable expectation when running arbitrary bytevectors on bare metal.)

1) Protocol and Packet Header: Communication between the server and the client makes use of a simple protocol, running on top of TCP. ROP-chains are sent to the server as a sequence of gadget-packets. Each packet begins with a seven-byte header, which is laid out as follows:

- bit 0 is the BAREMETAL field, and indicates whether the code is to be run on bare metal, or in the emulator;
- bit 1 is the RESET field, and tells the server whether or not to reset the registers to their initial state before executing the code, or to keep them in the state resulting from the most recent packet;
- bit 2 is the RESPOND field, and tells whether or not to transmit the register state back in response to the packet, in the form of a symbolic expression readable by Lisp – typically requested when the last packet in a chain is sent. When RESPOND is set to 0, the server just responds to each packet with "OK".
- bit 3 is the ACTIVITY_TEST field, and instructs whether or not to run a simple 'activity test', which tries to detect semantic introns – gadgets that have no phenotypic effects, at least in isolation – by seeding the registers with a test pattern (0x02020202 in each register), and checking to see if the pattern is unchanged on return (bit 3);
- bits 4-7 the ARCH field, which specifies the target architecture, when using the emulator (otherwise the native architecture is used, of course). Only two architectures have been implemented so far, so 1 gets you 32-bit ARM, and 0 gets you x86_64.
- bytes 1-2 (note the shift to bytes) is the EXPECT field, and gives the length of the gadget in bytes;
- bytes 3-7 is the STARTAT field, and specifies the memory address at which the emulator should begin execution. This is also, conveniently, the hash-table key for the gadget back home in the client.

Once the server receives the gadget, it passes it to the appropriate execution function. If BAREMETAL is selected, a pointer to the gadget (which is just an array of bytes representing machine code) will be passed to hatch_code, which casts it as a function pointer, forks, and calls the function in the child process, while tracking its behaviour in the parent using ptrace. When the code terminates, it returns a pointer to its register state, which is either stored and used as context for the next gadget in the chain, or converted to a symbolic expression and sent back over the TCP connection.

If the emulation is selected (i.e. if BAREMETAL(buffer) == 0), then a pointer to the code is passed to an instance of the Unicorn Emulation Engine, which takes over its execution and tracing, and again returns the register state as a byte array.

2) Undocumented Shortcomings of the Ptrace Library on ARM: Originally, I had planned to execute all the code

natively, or perhaps within a full-fledged virtual machine (using QEMU or VirtualBox). It was while attempting to use the above methodology on the Raspberry Pi, to test ARM ROP-chains, that I hit a dead end with native execution. It turned out that the ptrace library was broken in some virtually undocumented respects when it came to the ARM architecture. A bit of experimentation and digging around revealed that the PTRACE_SINGLESTEP function was not operational on ARM, and the alternative – peeking ahead in the code, poking breakpoints, replacing the instruction, and so on – was arduous and error-prone. A search for alternative methods brought me to the Unicorn Emulation Engine.

3) Unicorn Emulation Engine: Unicorn is a remarkable emulation library, which was just introduced at BlackHat USA in 2015. It repurposes elements from the QEMU library to provide standalone, context-free CPU emulation, making it the ideal tool for testing arbitrary sequences of machine code, and observing their effects on CPU context. My own innformal tests have shown it to be only twice as slow as running the code on bare metal (in a forked process), but with far greater fault tolerance and memory safety. It also allows for the emulation of a variety of architectures, independent of the architecture of its host. These features make it an extremely important element of the ROPER system.

V. Findings

ROPER is still very much under development, and not quite ready for a series of robust experimental trials. However, some of the preliminary findings have been encoruaging, and even with somewhat erratic and fault-prone phenotype expression to contend with, it has proven capable of discovering chains that succeed in bringing about specified register states. Much work remains to be done before this can be used as a practical tool for ROP-chain evolution, but the way forward is sunny.

VI. Conclusion

A. TODO

The project is still very much a work in progress, and comes with a lengthy todo list. These range from the immediate and crucial, to more ambitious, but further-afield features.

1) As it stands, the execution ("ontogenesis") server only handles register manipulations. It should be extended so as to at least manipulate stack memory, which is usually crucial for a working ROP-chain. The return stack, at present, is entirely simulated by the client, which feeds the server its gadgets one at a time, taking for granted that they can be executed sequentially. But this is not always the case, and a gadget can break the chain if it corrupts the stack.
2) It might also be feasible to pre-load the server with an image of the expected heap memory of that target process, so that it can be leveraged by the ROP-chains as well.
3) Once the system works more or less flawlessly, it could be seen as a strange sort of "compiler", which takes a set of specifications on the one hand, and a target binary on the other, and compiles a ROP-chain that cannibalizes said binary to meet those specifications. In this case, why not write a simple declarative scripting language that targets the ROPER compiler? A high-level language like Lisp or Prolog would make this straightforward enough, and would greatly increase the usefulness of the tool.
4) I've been impressed by Gunes Kayacik's work on coevolving buffer overflow attacks with intrusion detection systems, and think that something similar could be done here. Kayacik's work, as groundbreaking as it was, is primarily focussed with a class of attacks that are now, for the most part, obsolete – compilers nowadays protect binaries against traditional shellcode attacks by default, by marking writeable memory as non-executable, as a rule. (This is why the ROP-chaina attack was invented, after all.) The principles exploited in Kayacik's research are perfectly generalizable, however, and there's no reason why they can't be put to work here, such that ROPER treats *undetectability* as a secondary objective, once the primary objective – of properly executing a given task – is satisfied.
5) Tim Kornau's work on return oriented programming on the ARM deserves much closer scrutiny, and I expect it to be very useful in minimizing the work that needs to be left to ROPER's genetic algorithms. I will be studying it closely over the next few months, and seeking to incorporate its insightsi into ROPER's design.
6) ROPER stands in need of proper backronymization. I may go with GNU tradition on this, and go with "Return Oriented Programming Evolution with ROPER".

References

[1] Tim Kornau, *Return Oriented Programming on the ARM Architecture*, thesis submitted to the Ruhr-Universität Bochum, December 22, 2009.

[2] J. Drake, P. Oliva Fora, Z. Lanier, C. Mulliner, S.A. Ridley, G.Wicherski, *Android Hacker's Handbook*, Indianapolis, Wiley, 2014. See Chapter 9, in particular.

[3] G. Kayacik, N. Zincir-Heywood, M. Heywood, "Evolving Buffer Overflow Attacks with Detector Feedback", EvoCOMNET 2007, LNCS 4448.