An initial evaluation of ROP-based JIT-compilation

Pablo Bravo[1], Francisco Ortin[2]
[1]*Malwarebytes Corp., Anti-Exploit Technologies, EEUU*
[2]*University of Oviedo, Computer Science Department, Spain*

Abstract

Return-oriented programming (ROP) is a security exploit technique that allows an attacker to execute code in the presence of security defences. By modifying the contents of the runtime stack, the program control flow can be changed to execute specific machine sequences called *gadgets*. This new way of thinking about program flow may be useful for improving the runtime performance of specific language features such as structural reflection, dynamic code evaluation, and function composition. This article presents an initial evaluation of ROP as a JIT-compilation technique. We compare runtime performance, memory consumption and compilation time of four different back-ends, including ROP, of a simple stack-based virtual machine.

Keywords: Return-oriented programming, JIT compilation, runtime performance, memory consumption, stack-based virtual machine

1 Introduction

Return Oriented Programming (ROP) is a security exploit technique that allows the attacker to execute code in presence of Data Execution Prevention (DEP) [1]. Essentially, ROP is based on the idea that all the code a program needs to run could be found inside any other process. The organizational code unit in ROP is a *gadget*, a sequence of instructions ending in a `ret` instruction. With the control of the runtime stack, addresses of gadgets can be pushed and the execution path will flow through the gadgets code [2]. A successful attack can be composed based on code that already lives inside the attacked process.

Although ROP has mostly been used for software attacks, it represents a new way of thinking about program flow. Any program can be codified with a minimum fixed-length set of gadgets [10]. Then, programs are executed as a sequence of invocations to these gadgets. These invocations are performed by pushing the correct memory addresses of the gadgets to be called. Therefore, programs are codified as data, and pushed onto the stack.

The idea of identifying programs as modifiable data is widely used in dynamic languages such as Python, Ruby and JavaScript [6]. Meta-programming services such as structural reflection and dynamic code generation make use of this idea, providing a high level of runtime adaptability. Therefore, the implementation of a ROP-based JIT-compiler for these language features may involve better runtime performance and simplicity, since the high- and low-level languages follow the same pattern: program representation as modifiable data.

The main contribution of this paper is an initial evaluation of the pros and cons of ROP-based JIT compilation. For that purpose, we implement a simple stack-based language with 4 different back-ends including a ROP JIT compiler. Runtime performance, memory consumption and compilation time are evaluated.

2 Return-oriented programming

ROP is based on the fact that the computer stack has not actually to store the real return addresses and arguments to chain function calls in an imperative language model. It can store any address in the process space. The addresses of common computation elements (gadgets) can be stored in the stack. Since each gadget ends with a ret instruction, the values in the stack control the execution flow of the program.

The idea of ROP may be applied for program translation. A Turing-complete set of gadgets can be elected to run any program as a concatenation of gadgets. The set of gadgets would behave effectively like an interpreter or virtual machine, as can be seen in Figure 1. Execution flow jumps between the different gadgets due to two facts: gadgets end in a `ret` instruction, and programs are represented as addresses in the stack. Gadgets are executed following the program flow, obtaining the expected program behavior.

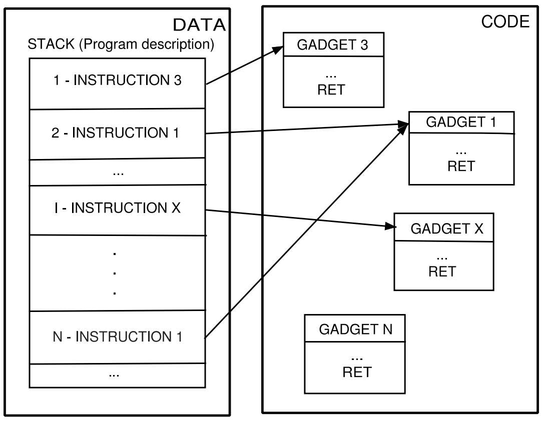


Figure 1: Program representation using gadgets.

There is a similar approach to ROP denominated Jump-Oriented Programming (JOP). This variant is a replacement in architectures which lack a stack, and demonstrates that ROP is neither tied to a stack nor a `ret` instruction. Jump-Oriented Programming (JOP) is based on gadgets ending in a jump instruction, whose address is also read from a linear structure [9]. This way, the common stack, if any, is free to be used in any other way required except to write down function calls to control program flow.

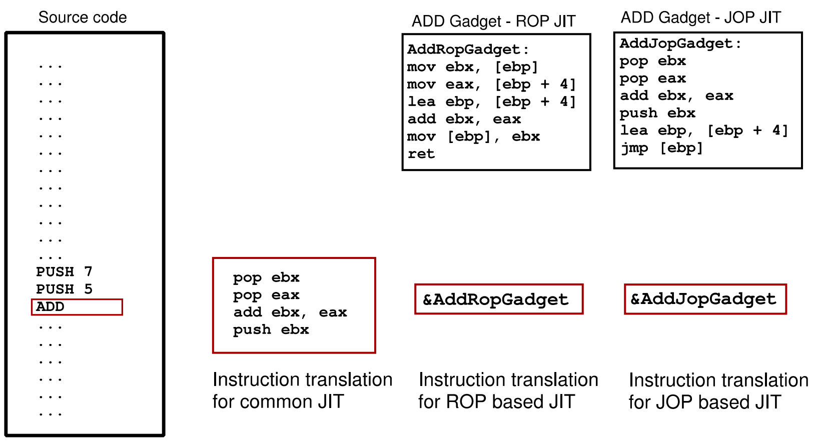
One thing to notice is the high locality of the executed code. Program description lies sequentially in a stack in the form of gadget addresses, but the real code executed by the computer is a handful set of gadgets. Those gadgets can probably be stored in few pages of memory, thus improving code locality.

3 Evaluation

3.1 Methodology

We implemented a simple language of a stack-based virtual machine taken from a compiler construction course [8]. It contains the following instructions:

- Basic arithmetic operations performed by popping the operands off the stack, computing the operation, and pushing result back onto the stack.
- Push constant values and variable addresses.
- Load a variable value: the variable address is popped off the top of the stack, and its value is pushed.
- Store: given a variable address and a value on the top of the stack, both are popped and the value is assigned to the variable.
- Output: pops the value off the stack and shows it in console.


Source code
ADD Gadget - ROP JIT
ADD Gadget - JOP JIT
Instruction translation for common JIT
Instruction translation for ROP based JIT
Instruction translation for JOP based JIT

```
...
...
...
...
...
...
...
...
...
...
...
...
PUSH 7
PUSH 5
ADD
...
...
...
...
...
...
```

```
AddRopGadget:
mov ebx, [ebp]
mov eax, [ebp + 4]
lea ebp, [ebp + 4]
add ebx, eax
mov [ebp], ebx
ret
```

```
AddJopGadget:
pop ebx
pop eax
add ebx, eax
push ebx
lea ebp, [ebp + 4]
jmp [ebp]
```

```
pop ebx
pop eax
add ebx, eax
push ebx
```

```
&AddRopGadget
```

```
&AddJopGadget
```

Figure 2: Source code translations for different back-ends.

The implementation consists in a basic interpreter with four back-ends: as an optimized interpreter, with typical JIT-compilation [4], and ROP and JOP JIT-compilation. The three JIT-compiler approaches provide no optimization.

The implementation of the ROP/JOP back-end comprises a set of gadgets that faithfully resembles the JIT implementation, as Figure 2 shows. In particular, the output routine and its invocation stay exactly the same among the back-ends.

We measure execution time, memory consumption and compilation time of some synthetic source programs containing different number of output instructions (0%, 10%, 16%, 20%, 25%, 33% and 50%). As we saw in Section 3, the output instruction has a strong influence on the evaluation. The executed programs have increasing code sizes. The binary sizes of the JIT-compiled programs vary in powers of two from 4KB to 1024KB.

3.2 Execution time

Figure 3 shows the execution times of the 4 different back-ends. We evaluate the influence of program size and output instructions on the runtime performance. The first chart in Figure 3 presents the little influence of the program size, when no output instructions are used. We can see how the classical JIT compilation technique provides the best performance results for every program size. The execution time of the JIT approach is between 6.9% and 8.5% the execution time of the interpreter version. The JOP JIT-compiler provides the second better performance, requiring on average 44% more execution time than the classical JIT-compiler approach.

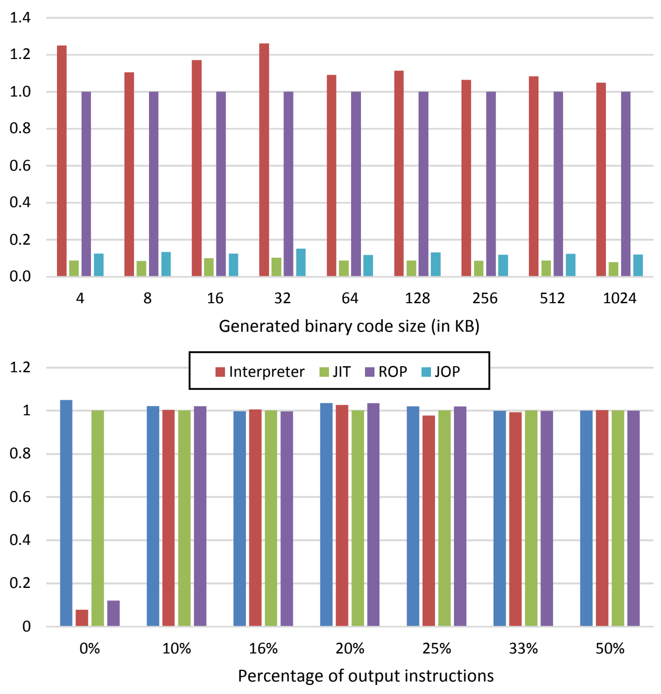


Figure 3: Execution time relative to ROP.

The ROP back-end is significantly worse than the JIT and JOP approaches, but it is still 12.9% faster than the optimized interpreter. If we compare the two new ROP and JOP back-ends, we can see how JOP is surprisingly 787% faster. This difference is due to the different semantics of `ret` and `jmp` instructions. Return instructions are designed as companions of `call` instructions, performing a quite complex operation in x86 architectures compared to jump. On the other hand, modern architectures have quite effective branch prediction hardware to minimize the effect of `jmp` instructions in the pipeline of the processor.

The second chart in Figure 3 shows the important influence of output instructions. For programs with at least 10% output instructions, execution times of the 4 approaches are not statistically significant. The time consumed by output instructions diminishes the differences among the approaches, as the `out` instruction require much more execution time that the rest of instructions.

3.3 Memory consumption

We also measured the memory consumed at runtime in the execution of each program. The size of the program had no influence on the relative values, but the kind of compiled instructions did. Figure 4 shows the average memory consumption relative to the interpreter approach. As expected, the 3 JIT-compiler approaches consume more memory than the interpreter [3]. JIT-compilers provide better runtime performance, but also require additional memory resources.

The ROP and JOP approaches always consume the same memory resources: twice the memory used by the interpreter version. However, the memory consumption of the classical JIT-compiler depends on the program compiled. In fact, if the number of output instructions is 50%, it consumes 46% more memory resources than the ROP and JOP approaches (almost 3 times the memory of the interpreter).

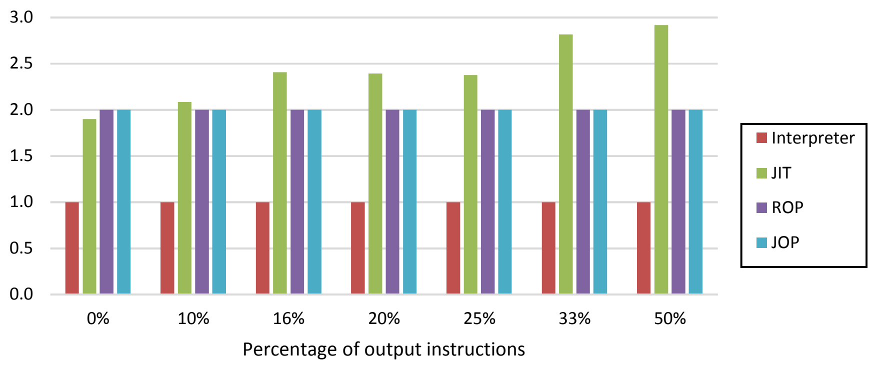


Figure 4: Memory consumption relative to the interpreter.

This difference between the two JIT-compilation techniques is caused by the way both approaches generate code. ROP and JOP include a fixed collection of gadgets in the generated code, and most instructions simply push a 4-byte value onto the stack (Table 1). These values are memory addresses of the corresponding gadget. However, the traditional JIT-compilation is based on the generation of a collection of binary instructions per source instruction. That is, instead of calling a gadget, they write their body for each instruction. As Table 1 shows, the classical JIT approach commonly requires more memory than the ROP/JOP alternative, even for our low-level stack-based language. We think this difference would be greater when compiling high-level languages.

Table 1: Instruction sizes in bytes.

Instruction	JIT	ROP/JOP
`ADD`	5	4
`SUB`	5	4
`MUL`	5	4
`DIV`	7	4
`PUSH`	5	8
`PUSHA`	7	8
`LOAD`	4	4
`STORE`	4	4
`OUT`	18	4

3.4 Compilation time

JIT compilation includes the binary code generation in the application execution. Therefore, the compilation time may increase the global performance of short-running applications, and it must be measured [5].

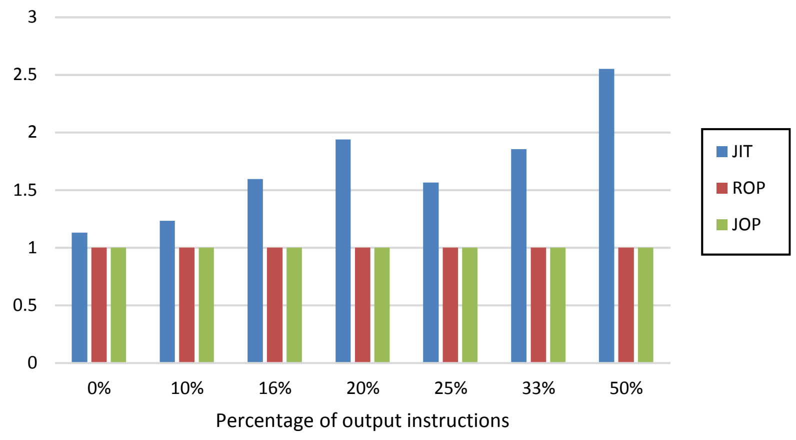


Figure 5: Compilation time relative to ROP.

Figure 5 shows compilation time relative to the ROP JIT-compiler. As expected, there is no significant difference between the ROP and JOP approaches, since code generation is analogous. However, these techniques generate binary code faster than the traditional approach. Besides, compilation time of the classical JIT compilation grows with the percentage of output instructions. The rationale is the same as for memory consumption. Since the JIT-compiler must generate more binary instructions for the output instruction (Table 1), the compilation process takes longer.

4 Conclusions and future work

An initial evaluation seems to imply that the ROP-based JOP technique can be used as an alternative mechanism for implementing JIT-compilers. JOP provides important performance benefits compared to interpretation, and 44% more execution time than the classical JIT-compiler approach. However, its memory consumption grows linearly with the number of instruction, and it is generally lower than current JIT-compilation techniques. Finally, it requires less compilation time than the traditional JIT-compilation.

We think that applying JOP compilation to high-level programming languages may increase the identified benefits. In these languages, the difference between the sizes of binary code generated for high-level instructions is higher than in a simple low-level stack-based machine. This higher difference will imply lower memory consumption and faster compilation of the JOP approach.

We are currently working on the JOP/ROP compilation of a high-level language to improve our evaluation. We think that this new compilation mechanism will improve the runtime performance of meta-programming features such as structural reflection and dynamic code evaluation [6]. These language features are based on the identification of code as data that can be modified, which is the idea behind ROP. Concatenative programming is also based on the idea of composing functions to create programs [7]. A common implementation of concatenative languages is with a stack machine, so ROP/JOP JIT-compilation may be applicable.

We also believe that execution time of JOP programs may be increased when compiling high-level languages. Our stack language does not have any instruction related with control flow, so its pipeline in the processor is ideal. Even in this worst-case scenario, the JOP-based JIT performed well. Thus, we think the effect of locality may decrement the impact in performance of breaking the pipeline.

Acknowledgements

This work was partially funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978. We have also received funds from the Principality of Asturias to support the Computational Reflection research group, grant GRUPIN14-100.

References

[1] Shacham, H., The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). *Proc. 14th ACM Conference in Computer and Communications Security (CCS 07), ACM*, pp. 552–561, 2007.

[2] Prandini, M. & Ramilli, M., Return-Oriented Programming. *IEEE Security & Privacy*, pp. 84-87, 2012.

[3] Ortin, F., Labrador, M. A. & Redondo, J. M., A hybrid class- and prototype-based object model to support language-neutral structural intercession. *Information and Software Technology*, Volume 56, Issue 2, pp. 199-219, 2014.

[4] Aycock, J., A brief history of just-in-time compilation. *ACM Computing Surveys (CSUR) Surveys*, Volume 35 Issue 2, pp. 97-113, 2003.

[5] Georges, A., Buytaert, D., & Eeckhout, L., Statistically rigorous Java performance evaluation, *OOPSLA '07*, ACM, New York, NY, USA, pp. 57–76, 2007.

[6] Redondo, J. M., Ortin, F., A Comprehensive Evaluation of Widespread Python Implementations. *IEEE Software*, doi: 10.1109/MS.2014.104 (to be published).

[7] Diggins, C., What is a concatenative language, 2008 http://www.drdobbs.com/architecture-and-design/what-is-a-concatenative-language/228701299

[8] Ortin, F., Zapico, D., & Cueva, J. M., Design Patterns for Teaching Type Checking in a Compiler Construction Course. *IEEE Transactions on Education*, Volume 50, Issue 3, pp. 273-283. August 2007.

[9] Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z., Jump-oriented programming: a new class of code-reuse attack. *ASIACCS '11 Proceedings of the 6th ACM Symposium on Information*, Computer and Communications Security, pp. 30-40, 2011.

[10] Homescu, A., Stewart, M., Larsen, P., & Brunthaler, S. Microgadgets: size does matter in Turing-complete Return-oriented programming. *6th USENIX Workshop on Offensive Technologies*, Bellevue, WA, 2012.