Yara Studies: A Deep Dive into Scanning Performance

Dominika Regéciová
Gen

This paper was presented at Botconf 2023, Strasbourg, 11-14 April 2023, www.botconf.eu
It is published in the Journal on Cybercrime & Digital Investigations by CECyF, https://journal.cecyf.fr/ojs

Abstract

You probably know this scenario — you spent a while analyzing new samples, which was not easy, but you're finally done. You also created a neat Yara rule to match the samples, and you're ready to send it off and move on to your next task (or lunch). But oopsie — the Yara rule is warning of slowed scanning. Or your colleague comments they do not like a particular part and wants to be sure the rule is effective.

While working with Yara, I consulted with many analysts about this problem. They knew what they wanted to detect, but Yara was not always helping them write the rules more effectively. Based on my experience with algorithms used in Yara, we worked together to find a solution to improve scanning speed and limit potential hurdles for future usage.

This paper presents five studies with descriptions of the five problems, an explanation of why Yara does not like the first solution, and tips on what can be improved. Note that no sensitive information is disclosed in this paper. All studies were anonymized, so the general problem is the same, but there is no direct link to a specific malware family mentioned, nor can it be tracked.

Keywords: Malware detection, pattern matching, performance, regular expressions, Yara.

1 Introduction

Yara is a well-known tool used by many analysts and security specialists around the world. It is a very practical tool that can detect patterns based on static and dynamic information, and it is simple to use — the syntax of the rules is straightforward. Even a quick look at the official documentation [1] or examples of publicly available rules [2] can give you an idea of how the matching works.

That is not the hard part of the process, however. If you want to describe a specific malware family (the most common use case for Yara), the situation is much more complex than just simply putting together a list of strings that need to be matched. Creating a good rule that correctly detects a described family and does not cause false positives requires skill and experience. Even after all the necessary static and behavioral information is extracted and described in the rule, another problem can arise — the performance of the scanning. Yara has several mechanisms to detect potential slow scanning and generates warnings, such as in Figure 1.

```
# Running Yara with a rule over the input
directory (recursively)
$ ./yara rule.yar -r input_directory
warning: rule "family_XYZ" in rules.yar(x):
string "$re" may slow down scanning
```

Figure 1: Example of Yara rules warning

These warnings are based on heuristics that evaluate the quality of the rules. They inform us that the rule could be written more effectively. They often lack information on what change is needed to improve performance. What makes the issue worse, the rules with warnings can not be used in some systems like VirusTotal Hunting[1].

This paper aims to present several studies where the performance of Yara rules was in question. Based on the practical examples, the critical aspects that can influence the scanning speed will be described as well

[1] https://www.virustotal.com/gui/hunting-overview

as a guide on how to approach them. The goal is to make the lives of analysts easier. The goal is also to create simple guidelines that are easy to follow and use, so writing Yara rules can become more manageable.

Note that this paper is a follow-up to the previous work — *Yara: Down the Rabbit Hole Without Slowing Down* talk for Botconf 2022 [3] and *Pattern Matching in Yara: Improved Aho-Corasick Algorithm* [4].

2 Introduction to Yara

This section briefly provides an overview of what Yara is and how it works. If you are familiar with Yara, you can skip this section. Or just go through it to ensure you are familiar with all syntax constructs used in this paper. More information can be found in the official documentation [1].

Yara is a language and tool for pattern matching. The patterns and additional context are described in syntax units called rules. An example of one rule is shown in Figure 2.

Every rule has three parts — meta information, strings, and condition. The only mandatory part is the condition that contains Boolean expressions. The rule matches the file only if the condition is True. The meta section usually includes the author's name and other information about the malware the rule describes. Strings can be text strings, regular expressions, or hexadecimal strings.

```
rule example_rule
{
  meta:
    author = "Dominika Regeciova"
  strings:
    $str = "Hello World!" fullword nocase
    $re = /abcd[x-z]/
    $hex = { 63 62 61}
  condition:
    $hex at 0 or
    $re or
    $str
}
```

Figure 2: Example of Yara rule

In the rule named *example_rule* we have three expressions where at least one must be true to match a sample. Either a text string that is delimited by non-alphanumeric characters is found in the sample (while ignoring the case of the characters), or regular expressions are found or a hexadecimal string in starting position 0 (at the beginning of the file). If a sample meets the condition, Yara will match it and report it as a result.

To better understand the following studies, let us summarize the algorithms used in the process, from matching the strings to evaluating the condition on specific samples, so you later understand how each step can influence the evaluation of the rules. The entire process can be split into four steps: atoms selection from strings, creation of the Aho-Corasick automaton, bytecode engine run, and evaluation of conditions.

2.1 Atoms Selection From Strings

From all the strings from all rules, so-called atoms are selected. The atoms are substrings with lengths from zero to four bytes. Yara has several heuristics for choosing the most unique and, thus, most effective atoms. In our example, from a regular expression */abcd[x-z]/*, Yara will select the part *abcd*.

The heuristics, however, have their limits, and for strings that are too general, such as */\w*/*, even zero-length atoms can be chosen. This is problematic because a later, much slower state does all the matching, and the input is searched byte by byte. This will lead to a warning about slowed scanning and limiting the rule's usability in systems like VirusTotal Hunting.

2.2 Aho-Corasick Automaton

All selected atoms are used to build a prefix tree called the Aho-Corasick automaton. This automaton works as a sifter — it quickly scans through every input and finds all potential matches for the atoms. This is a crucial step that influences how fast or how slow the scanning is. If we have too many general atoms, many — even each very byte, is selected as a potential match and must be inspected further. We want to limit the set of potential matches, so the rest of the evaluation is much faster.

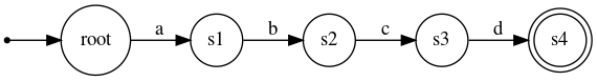

Figure 3: An example of the Aho-Corasick automaton for atom *abcd*

The basic idea behind using Aho-Corasick automaton is that it effectively matches every occurrence of multiple strings simultaneously, even when the matches are interleaving. Starting in the root state, the automaton reads one symbol from the input at a time and changes states accordingly. The atom is found in the input if the final state is visited.

We can find multiple matches in the same position because we have so-called failure functions. The failure function is used when there is no transition based on a combination of the current state and input symbol. The default action is to go to the root state and try again. However, if the prefix of the other string was partially matched, we can go there instead. All failure functions point to the root in our example because there are different characters, but we can choose another group of strings to demonstrate this situation.

Let us say we are looking for words *she* and *his* in text *shis*. We started matching *sh* in the keyword she, but now, we need *e*, but we have *i* instead. We could go

to the root again or try to match *his* instead because we know we already read *h*. And yes, we can find *his* in the rest of the text.

2.3 Bytecode engine

The bytecode engine gets the list of potential matches and verifies which are true positives based on the full definition of strings (including modifiers like wide, nocase, and others). This process takes time, so as we discussed before, making a list as accurate as possible is a clever idea. Yara also reports the position in the input if a match is found. In the example in Figure 4, there is a match in position *0x0*.

```
0x0: abcdx => match
0x7: abcdf => not a match
```

Figure 4: example_rule found one match in the input

2.4 Condition Evaluation

Evaluating the condition for the whole rule is the last step. It is important to remember that conditions are evaluated after matching the strings. If you want to limit the file size that you want to match with the filesize function, it will not prevent the previous steps from being executed when your rules match static strings. The good news is there is short-circuit evaluation, so changing the order of conditions can positively impact performance.

In our example, we found the match on position *0x0*, and the condition is evaluated as True.

3 Experiments

All studies will be explained based on algorithms used in Yara, but often, the best arguments that positively influence performance are numbers. For this reason, all studies were tested, and the scanning speed was measured. For tests, the upstream version of Yara in its latest release (v4.2.3) was used [5].

Experiments were run on a server with the CentOS Linux 7 operating system on a 64-bit architecture with an AMD EPYC 7502 32-Core Processor with a clock rate of 2.50GHz. The memory speed was 3200 MT/s.

Publicly available datasets were used for scanning. The first two were published in 2018 by Eduardo De O. Andrade and contain 31,220 samples of cleanware, and malware [6], consisting of 8.2 GB of binaries. Please, be careful with handling the samples, mainly the malicious samples. The second dataset is a large 14GB binary blob [7]. The third addition to the dataset is a text file of *The Entire Project Gutenberg Works of Mark Twain* by Mark Twain, available as a testing file for the Regex Performance testing tool [8].

Together we scanned around 22 GB of data. The most straightforward Yara rule was created to measure the minimal time in which Yara can go through all the samples (Figure 5). The evaluation is very simple as all samples are evaluated as not matching right away based on the condition. It is the fastest rule possible, taking around 36 seconds to scan all the samples in our dataset.

```
rule test_00
{
  condition:
    false
}

# False rule
$ time ./yara 00.yar -r dataset/
real    0m36.297s
user    0m57.502s
sys     0m18.932s
```

Figure 5: Experiments: the simplest "false" rule

Also, note that the following studies are just a selection of common performance issues in Yara. As additional resources, the *Yara Performance Guidelines* [9] is an excellent page with many examples and tips on how to use Yara more effectively.

4 Study I: Strings vs. Condition

The first example is a case where a few first bytes were important, and the analysts wanted to match them. Their rules looked like this:

```
rule test_01
{
  strings:
    $h00 = { 42 ?? ?? 00 00 61 62 }
  condition:
    filesize < 1KB and
    $h00 at 0
}
```

Figure 6: Study I: the original rule

Note that the values of the bytes were changed, but the idea remained the same. The reviewer, in this case, did not particularly like the general nature of the strings. Two arbitrary bytes with two zeroes bytes are not particularly specific and unique; that is true.

The position specification can look like a good solution, mainly with the limitation of the maximal size of the input. However, we need to keep in mind how Yara works. In the first phase, the statics strings are searched, and the condition is applied after. So, in this case, each input is scanned as a whole, and all strings matches are returned. After that, we evaluate the filesize function and search, which starts at position 0. This is a relatively minor example, but this can principally grow exponentially and slow down scanning dramatically.

So, what is a better solution? If we have a specific position in mind, the best solution is to use *intXY* or *uintXY* functions. You can match *uint8(0)* or

unit32(3), where the number in the function name represents how many bits you want to match starting with the offset or virtual address described as an argument. Be aware that both 16 and 32-bit integers are little-endian. For big-endian, you can use functions *intXYbe()* or *uinXYbe()*. The rewritten rule then looks like the following:

```
rule test_02
{
  condition:
    filesize < 1KB and
    uint8(0) == 0x42 and
    uint32(3) == 0x62610000
}
```

Figure 7: Study I: the updated rule

There are two advantages of this change — we are not scanning whole files but just a specified position and if, and only if, the limit for the input size is met. The difference in speed can vary due to the nature of input files. During my initial investigation and on the data set, the difference was around 9-10%. This may appear insignificant, but the difference can be significant when scanning large datasets.

```
$ time ./yara 01.yar -r dataset/
real    0m39.934s
user    1m7.489s
sys     0m19.024s

$ time ./yara 02.yar -r dataset/
real    0m36.565s
user    0m56.980s
sys     0m19.227s
```

Figure 8: Study I: comparison

5 Study II: Potentially There

How do you want to match something that could be there, but may not necessarily be there? This was the case in our second study. The analyst was working on samples that were creating a set of strings with a combination of special characters that could be included between two characters but sometimes were missing. If the string were *powershell* and special character ^, generated strings would look like this: *powershell*, *p^owershell*, *p^o^wershell*, and so on. The first version of the rule had a problem — warning about slowing down scanning.

```
rule test_03
{
  strings:
   $re=/p\^?o\^?w\^?e\^?r\^?s\^?h\^?e\^?l\^?l/
  condition:
    $re
}
```

Figure 9: Study II: the original rule

This was a tricky case because the changing nature of strings is problematic for all pattern-matching tools. To maximally help Yara limit the potential set of matches, we split the string definition into two — one with a caret between the first two characters and one without. This change will help Yara select substrings *p^o* and *po* as atoms, and this is generally better for the algorithms rather than using only one character *p* as in the first version.

```
rule test_04
{
  strings:
   $re1=/p\^o\^?w\^?e\^?r\^?s\^?h\^?e\^?l\^?l/
   $re2=/po\^?w\^?e\^?r\^?s\^?h\^?e\^?l\^?l/
  condition:
    $re1 or $re2
}
```

Figure 10: Study II: the updated rule

```
$ time ./yara 03.yar -r dataset/
warning: rule "test_03" in 03.yar(3):
string "$re" may slow down scanning
real    0m43.650s
user    1m7.416s
sys     0m21.658s

$ time ./yara 04.yar -r dataset/
real    0m37.430s
user    0m59.172s
sys     0m20.318s
```

Figure 11: Study II: comparison

The second version is about 14% faster, just based on the nature of pattern-matching algorithms used in Yara. The meaning is the same, they match the same samples, but the second version allows Yara to be more optimized during the matching process and makes it faster. Again, the difference in numbers is not astronomical, but even 10-15% can save you a good amount of time when dealing with extensive datasets.

6 Study III: Problematic Alternations

There are some instances where strings you want to match are short, for example, just three bytes. In these cases, be careful how you write them down because it can impact Yara negatively. Let us say we want to detect two hexadecimal strings — *44 03 33* and *44 2E 33*. It can be tempting to write them into one string as in Figure 12.

However, this format is far from ideal for Yara. Not that it cannot work with alternation during the atom selections process. For instance, in a case such as */(hey|bye)\.\w*/*, it would prefer two atoms *hey* and *bye* over searching just for a dot (*\w cannot be part of the atom*). However, Yara does not concatenate the strings together, even if it means the atoms would be

longer. In this case, the alternation does not provide enough information, and the problem is even bigger because even values outside the alternations are just one character in length. We want to match four or more bytes for effective matching, but sometimes it is not very easy. In this case, we want or need to work with these strings. So, what can we do?

```
rule test_05 {
  strings:
    $hex = { 44 (03|2E) 33 }
  condition:
    $hex
}
```

Figure 12: Study III: the original rule

```
rule test_06 {
  strings:
    $hex0 = { 44 03 33 }
    $hex1 = { 44 2E 33 }
  condition:
    $hex0 or $hex1
}
```

Figure 13: Study III: the updated rule

```
$ time ./yara 05.yar -r dataset/
warning: rule "test_05" in 05.yar(3):
string "$hex" may slow down scanning
real    0m46.142s
user    1m12.487s
sys     0m21.040s

$ time ./yara 06.yar -r dataset/
real    0m37.241s
user    0m59.962s
sys     0m17.781s
```

Figure 14: Study III: comparison

The solution here is simple yet effective. We just need to write down both strings separately, even though they are similar. The updated version of the rule can look too descriptive, but it helps Yara understand how it can work with these strings. With more information, the selected atoms are longer (in this case, they are identical to the strings themselves), and the scanning process can be much more effective. The updated version was about 19% faster on our dataset.

7 Study IV: Too General

Warnings about too many matches or about *.**, *.+* or *.x,* indicate that one of the strings in the rule is too general and needs to be narrowed down. This happens when the prefix part is defined very loosely, like in Figure 15. Regardless of how good the rest of the rule is, even one general string can slow down a whole rule or even a ruleset if you are using multiple rules at once. For this reason, try to avoid using general regular expressions for more context. It can backfire very quickly.

```
rule test_07
{
  strings:
    $re = /.*\.exe/ nocase ascii wide
  condition:
    $re
}
```

Figure 15: Study IV: the original rule

Using this rule, Yara will warn us about what is causing scanning speed problems. The first one is caused by the star symbol, which indicates that we want to match alphanumeric characters and _ zero or more times. In this specific warning, Yara provides us with advice on solving their issue, as seen in Figure 17. We need to limit the range to small values, that describe the strings we need to match. Because the part before the dot is not mandatory, we can skip it entirely.

The problem of too many matches can be triggered depending on the input, and it is also connected to the first warning. Because the beginning of the string is not set to a fixed position, Yara searches for every variant, such as *.exe*, *a.exe*, *aaa.exe*, and so on. In these cases, Yara returns an incorrect number of matches in the second case. We want to avoid both warnings.

```
rule test_08
{
  strings:
    $re = ".exe" nocase ascii wide
  condition:
    $re
}
```

Figure 16: Study IV: the updated rule

```
$ time ./yara 07.yar -r dataset/
warning: rule "test_07" in 07.yar(3):
$re contains .*, .+ or .{x,} consider using
.{,N}, .{1,N} or {x,N} with a reasonable value
for N
real    1m2.497s
user    13m41.575s
sys     1m9.662s

$ time ./yara 08.yar -r dataset/
real    0m37.388s
user    0m59.060s
sys     0m19.649s
```

Figure 17: Study IV: comparison

Yara can try to match more than you want or need if you don't setup string length ranges that you want to match or if strings don't have a fixed starting position. The original version of the rule might look more general, but it will most likely cause trouble in the future. It is better to keep it more specific, as shown on the updated rule, which is about 40% faster (Figure 16).

8 Study V: Secret Agent 6, IPv6

How can you effectively search for IPv6 addresses? When an analyst asks this kind of question, the solution often lies in a detail. What kind of addresses do we need to match? The rule's first version matches all kinds of IPv6 addresses and more. This is time-consuming but also generates a lot of false positives.

```
rule test_09
{
  strings:
    $ipv6 = /([a-f0-9:]+:+)+[a-f0-9]+/
        fullword nocase ascii
  condition:
    $ipv6
}
```

Figure 18: Study V: the original rule

After discussion, we figured out we must only match global unicast addresses starting with the prefix 2001. This is good news for Yara because it can use this prefix for faster scanning. We also limit the range of hexadecimal symbols to match the strings.

```
rule test_10
{
  strings:
    $ipv6 = /2001:([a-f0-9]{0,4}:){1,6}[a-f0
        -9]{0,4}/ fullword nocase ascii
  condition:
    $ipv6
}
```

Figure 19: Study V: the updated rule

```
$ time ./yara 09.yar -r dataset/
warning: rule "test_09" in 09.yar(3):
string "$ipv6" may slow down scanning
real    1m20.158s
user    2m26.389s
sys     0m24.285s

$ time ./yara 10.yar -r dataset/
real    0m39.236s
user    1m0.937s
sys     0m21.562s
```

Figure 20: Study V: comparison

The difference between the two versions of this rule is incredible. The second variant was about 50% faster and had zero matches, while the first version of the rule had 20,625 matches, all false positives. Both aspects caused issues, and solving them at the same time was a win-win situation.

9 Conclusion

Yara is an amazing tool but not always user-friendly. When you need to create detection rules quickly, there is often not enough time to think about potential performance issues. As the selected studies show, even simple tasks, such as matching strings in a certain position, could be written differently with vastly different resulting performances.

This paper describes five studies to provide a form of guidelines that have the potential to improve the scanning speed with minimal effort so analysts can focus on their work and not worry about how to overcome barriers in the form of Yara syntax construct issues.

Acknowledgment: The author would like to thank Jakub Křoustek, Marek Milkovič, and Threat Labs team at Gen for their guidance and support.

Author details

Dominika Regéciová

Gen
Brno, Czech Republic
Dominika.Regeciova@gendigital.com
ORCID iD: 0000-0001-8729-6999

References

[1] "The Official Yara Documentation." https://yara.readthedocs.io/en/v4.2.3/.

[2] "Awesome YARA." https://github.com/InQuest/awesome-yara.

[3] D. Regéciová, "Yara: Down the Rabbit Hole Without Slowing Down," *The Journal on Cybercrime & Digital Investigations*, vol. 7, 2022.

[4] D. Regéciová, D. Kolář, and M. Milkovič, "Pattern Matching in Yara: Improved Aho-Corasick Algorithm," *IEEE Access*, vol. 9, pp. 62857–62866, 2021.

[5] "Yara GitHub." https://virustotal.github.io/yara/.

[6] E. de O. Andrade, "MC-dataset-binary and MC-dataset-multiclass." https://figshare.com/authors/Eduardo_de_O_Andrade/4923649.

[7] B. Bosansky, D. Kouba, O. Manhal, T. Sick, V. Lisy, J. Kroustek, and P. Somol, "Avast-CTU Public CAPE Dataset," 2022.

[8] "Regex Performance." https://github.com/rust-leipzig/regex-performance.

[9] "Yara Performance Guidelines." https://github.com/Neo23x0/Yara-Performance-Guidelines.