Yara: Down the Rabbit Hole Without Slowing Down

Dominika Regéciová
Avast Software

Abstract

Terry and John are two malware analysts working for an unnamed antivirus company. Terry has worked there for many years, and he is helping John, who started recently, to learn more about their work. John is starting to use Yara – an excellent tool for the description and detection of malware families. With Terry, they are analyzing potentially malicious samples, and they are creating so-called Yara rules. This is not a simple task to do – Yara may be easy to use, but it is difficult to master. How to write the best rule possible? The rule that is good in detection, precise, but also fast? Luckily, they have help - a researcher Caitlin, who is not scared to get really deep into Yara. Today, all three of them will go deeper into Yara than ever before – the journey to the rabbit hole can begin.

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

1 Introduction

This presentation aims to provide practical information for analysts who create Yara rules and explain this topic to wider audiences without previous experiences with Yara.

Firstly, the basics will be explained – what is Yara? What can it do, and how is it being used? Why should we care about details, how Yara works internally? This section will provide enough information for those who are not yet familiar with Yara.

In the second part, we will explain how Yara works. We will show examples, and we also point out the problematic places that are doing the analyst's job harder than it could be.

At Avast, we are working on these areas, and we are trying to fix them. Some changes will be presented. Last but not least, the performance will be discussed. How to optimize rules? What to do if errors or warnings pop out?

To avoid getting too technical and theoretical, we will work with three imaginary colleagues - Terry[1], John[2], and Caitlin[3]. They will help us understand real-world applications of the provided information and represent a different perspective based on profession and level of experience.

Note that this presentation is slightly inspired by a light talk given at Botconf 2019 with the same name.

2 Why Yara and Why We Should Want to Know More?

Yara was created by Victor Manuel Alvarez from the company VirusTotal, which Google Inc. acquired in September 2012. The primary motivation for Yara was to create a tool for the classification and identification of malware and make the whole process of working with malicious files more effective. Before that, these processes relied more on the individual knowledge of analysts, which was highly unproductive and information was almost impossible to share.

With Yara, companies can create a set of rules that can be later used for threats detections. Each rule identifies key aspects of some malicious software or

[1]Terry Pratchett (1948-2015)
[2]John Irving (1942)
[3]Caitlin Doughty (1984)

behavior. Yara can scan files searching for statical information as strings or use reports from sandboxes and emulators to found suspicious behavioral information.

Today, many companies that are not exclusively antivirus companies, namely Avast, Kaspersky Lab, and ESET, use Yara every day for their cybersecurity. The list of these companies can be found on the project's Github page (see [1]). We can expect the popularity of this tool will only grow, mainly because of the active community that is working toward improving Yara and bringing important information to the public.

At one of the companies, there is currently an ongoing meeting. Malware analysts Terry and John decided to meet with a researcher from their team, Caitlin, to discuss how Yara works in more details.

John, still learning about all tools necessary for his job, hesitates. Why do they need to know so much about Yara? He saw some rules already, he even wrote a few of them, so what is the fuzz about it?

Caitlin, on the other hand, has a different point of view. She has been working with the Yara project directly for some time. She knows the algorithms used in the code, their strengths, and their limitations. But she needs to understand which information is actually helpful to the analysts and create more confusion.

To bring these two perspectives together, Terry will be moderating the meeting. He has been working as an analyst for a few years now. He can write pretty good rules, but there are cases where he is not sure how Yara works under the hood. There is not much time to search how Yara evaluates different versions of rules in detail in his job. The fact that Yara is still evolving and things are changing does not help either. Luckily, this is a part of the job for Caitlin. She is providing them with more detailed information and helping them with more complicated rules.

3 How Does Yara Work?

```
rule rule_number_one
{
 meta:
  author = "John"
 strings:
  $re = /abcd[x-z]/
 condition:
  $re
}
```

Figure 1: Our first Yara rule

How does Yara work then? To start, let us have a simple rule as shown in Figure 1.

Every rule has three parts - meta information, strings, and condition. Only condition part is mandatory, and it contains Boolean expressions. If the condition as a whole is True, the rule is matching the file.

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

The meta section usually includes the name of the author and other information about the malware the rule describes. Strings can be text strings, regular expressions, or hexadecimal strings.

In rule named *rule_number_one* we are searching one of three strings *abcdx*, *abcdy* or *abcdz*. If any of these three will be found in a file, Yara will match it.

But before the result is known, four steps have to be completed: atoms selection from strings, creation of Aho-Corasick automaton, bytecode engine run, and evaluation of conditions. Let us talk about them a little bit, so we understand later how each step can be optimized and used in practice by our analysts.

3.1 Atoms Selection From Strings

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

Caitlin explains that Yara has limits when selecting atoms and when strings are too general, such as */\w*/*, even zero-length atoms can be chosen. This will lead to a warning about slowing down scanning. We want to avoid that when possible because rules with errors or warnings can not be used in some systems like VirusTotal Hunting[4].

3.2 Aho-Corasick Automaton

After all atoms are selected, the prefix tree, called Aho-Corasick automaton, is created.

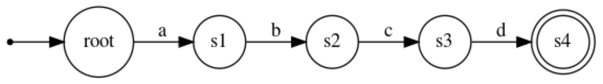

Figure 2: An example of the Aho-Corasick automaton for atom abcd

Aho-Corasick automaton is an effective way to match 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 a time and changes states accordingly. If the final state is visited, the atom was found in the input.

When there is no transition based on a combination of the current state and input symbol, the failure function is used. 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 try to go there instead. All failure functions are pointing to the root in our cases because there are all different characters, but we can choose a different case.

Let us say we are looking for words *she* and *his* in text *shis*. We starting matching *sh* in keyword she, but now, we need *e*, but we have *i* instead. We could go to the root not 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.

After the automaton is created, the first matching process is run. This will find all places inside of the file where actual matches could be found.

3.3 Bytecode engine

To found all actual matches, a bytecode engine is run with the list of potential matches. This phase can take a long time, so it is important the list of the potential matches is the most accurate as possible. If a match is found, Yara also reports the position in the input. In an example in Figure 3, there is a match on position *0x0*.

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

Figure 3: rule_number_one found one match in the input

3.4 Condition Evaluation

The last step is the condition evaluation. The condition has to be True to match the file. In our case, we only need one string present at least one time, but the condition can be more complex, as we will show later.

In our case, we found the match on position *0x0*, so the rule matched the file.

4 Our Changes in Yara

This part of the presentation is about some changes and improvements we have made in Avast. The core principles and theory behind them were described in paper *Pattern Matching in Yara: Improved Aho-Corasick Algorithm* [2]. Here we will present mainly the practical aspects of these changes and their impact on Yara. The code is also available online on the Github repository[5].

These changes were motivated by problems that Terry and John, and other analysts are facing. Write a good Yara rule is a challenging task itself, but often cryptical errors and warnings are popping out from the tool itself. *This rule is slowing down scanning; there are too many matches*, and so on.

As Caitlin and other researchers found out, one source of these problems is how regular expressions in Yara are being handled. Yara is accepting a pretty extensive set of regular expressions based on PCRE notation. However, the inside of the code, regular expressions are being ignored by the first two steps of the scanning process (atom selection and matching process with Aho-Corasick automaton), resulting in overloading bytecode engine. Thus the warnings and errors, because the Yara cannot handle the use of regular expressions effectively.

At Avast (and in the unnamed company where our team works), however, we want to include at least a subclass of the regular expressions into this process to search for them more efficiently and effectively.

[5]https://github.com/regeciovad/yara/tree/classesv2

4.1 Bitcoin addresses and more

To demonstrate the changes, let us look at an example in which John wants to look up the string that could be Bitcoin addresses. Based on the samples he works on, he is considering the `P2PKH` and `P2SH` types, omitting the `Bech32` type (see [3]). The rule detecting such addresses could look like shown in Figure 4.

```
rule rule_number_two
{
 strings:
  $re = /[13][a-km-zA-HJ-NP-Z1-9]{25,34}/
      fullword ascii wide
 condition:
  $re
}
```

Figure 4: The Yara rule for the detection of Bitcoin addresses

When trying this example, he gets a warning about slowing down scanning. Why? Because in the upstream version, the atom of the length of zero is being selected. This means that all bytes in the input must be checked by the bytecode engine if the match is there or not. This is an extremely time-consuming task, so the warning is correctly generated.

In our version, the atoms are not only substrings, they are a subclass of regular expressions. For that reason, we can work with atoms such as *1[a-km-zA-HJ-NP-Z1-9][a-km-zA-HJ-NP-Z1-9][a-km-zA-HJ-NP-Z1-9]*. In this specific case of the Bitcoin addresses, our version can be almost ten times faster than the upstream version. This change will lead to a much faster scanning process and enables Terry and John to use a broader spectrum of regular expressions without errors and warnings.

4.2 Nocase

Text strings in Yara are case-sensitive by default. However, we can turn a string into case-insensitive mode by appending the modifier *nocase* at the end of the string definition, in the same line [4].

However, this modifier can slow down Yara, mainly in the upstream version. For example, for string *cmd.exe*, the atom *cmd.* will be selected. If the *nocase* modifier is used, Yara will create eight atoms presenting all variants of lower and uppercases. This could not look as much, but when this modifier is used extensively, it can rapidly increase the number of atoms.

More atoms mean more potential matches, and that will lead to overall slowing scanning.

In Avast, we are evaluating the *nocase* differently. Instead of creating more atoms, we generate with consistent of classes such as */[Cc][Mm][Dd]./*. Again, this change can speed up the scanning process. More specifically, we were able to speed the scanning by about 27% in specific cases.

5 Yara Performance

Performance is important when we have many new samples every day that have to be scanned for potential threats. Yara, in general, can be very fast, but there are limitations. Luckily, we can improve Yara's performance if we keep some basic principles in mind during writing rules. These section are based on research of Inquest [5], [6], Florian Roth [7], and the author of this proposal.

5.1 Atoms Selection From Strings

As we explained before, the selection of atoms can have a significant impact on the overall performance. The goal is not to force to create strings just for the sake of the atoms but to be more mindful about mostly regular expressions and hexadecimal strings. Are those strings really what we need to match? Are they bringing the real meaning to the detection process of the malware, or are they just matching some random stuff?

```
/abc.{1,20}def/ => abc
/(one|two)three/ => thre
{ 00 00 [1-4] 01 02 03 04 } => {01 02 03 04}
/a(c|d)/ => /(c|d)/
/\w.*\d/ => "" (0-length atom)
```

Figure 5: Examples of how Yara is choosing atoms from strings

In Figure 5 we can see, Yara is trying to find the unique atoms for a given string. However, too general strings like */\w.*\d/* do not give Yara enough to work with. Omit strings like this. They will not bring additional value to your rules, and they will slow down Yara significantly.

5.2 Strings

Beyond the atoms, strings can additionally attribute to the scanning speed, mainly in the verification phase.

When working with modifications as ascii, wide, nocase, fullword, be careful and think twice about what you need. In general, any additional information will mean more steps that Yara has to do.

```
Ascii - 1 byte per character {42}
Wide - 2 bytes per characters {00 42}

$s1 = "cmd.exe" (ascii only)
$s2 = "cmd.exe" ascii (ascii only as $s1)
$s3 = "cmd.exe" wide (UTF-16 only)
$s4 = "cmd.exe" ascii wide (ascii and UTF-16)
```

Figure 6: Strings modifiers in Yara

If we need to use *nocase* option for case-insensitive mode, we recommend to use a regular expression instead and locate the possible variants like *$re = /[Pp]assword/*.

When working with regular expressions, be aware that most of them will impact the scanning speed negatively. Avoid greedy quantifiers *.** and *.+* or even *.*?*. Also, do not forget the upper bound (e.g. *.{2,}*). Think how many strings can be matched by a regular expression if you want to avoid too many matches or slowing down scanning warnings.

```
// $re1 will find Tom, xTom, xxTom in "xxTom"
$re1 = /.{0,2}Tom/

// $re2 will find Tomxx in "Tomxx"
$re2 = /Tom.{0,2}/
```

Figure 7: How Yara evaluate infixed prefix and suffix differently

Based on Figure 7, we can see that Yara evaluates the infixed prefix and suffix differently. It is caused partially by the Aho-Corasick automaton, which will report two potential matches in case of *$re1*, but only on in case of *$re2*. For that reason, it is better to use strings that have the most likely fixed location of the prefix.

5.3 Too Many Matches

Before Yara version 4.1.0, too many matches was error during scanning, where a string was found in input too many times. This means that the scanning process was stopped, and the results were invalid. From version 4.1.0, it is no longer an error but rather a warning, the scanning is finished, but the results could still be invalid.

The problem can be triggered during the evaluation of potential matches, where the number of verified matches overcomes a specific value (currently set up to 1,000,000).

These can be caused by a few reasons. Strings could be too general, as a sequence of zeroes or common sequences in binaries. As we also showed before, if the string does not have a fixed beginning of the match, Yara will try to find them all. For string */aaa/* with text *aaaaaaaa* (eight times *a*), Yara will find six matches.The problem can grow even more when no upper bound is used (such as */aaa*/*).

There is no one simple solution for these cases. Because the problem is triggered during the validation of potential matches, the changes of the conditions will not solve the problem.

In some cases, the following steps can fix the problem:

- Check for the quantifiers .* and .+, .*?
- Check for quantifiers without upper bound such as x{14,}
- Check for too large range (e. g. x{1,300000})
- Check for big jumps in the hexadecimal strings
- Check for wild-cards characters - can they be specified more preciously, or could be string split into two, omitting the wild-cards character?
- Check for alternations: can it be split into two or more strings?
- Try to add specification for words matching (fullword, \b, . . .)

5.4 Conditions

What about conditions, then? Can they improve the matching speed? Yes, and no. They have some potential, but there are two problems. Matching of strings comes first. Condition *filesize < 100 and $expensive_regex* will not help because Yara will match with a regular expression first and then check the file's size.

Also, if the statements are more or less equally expensive, the order does not really matter, as in the case *$string1 and $string2 and uint16(0) == 0x5A4D*.

However, there are cases where the order of the parts of the condition can make a difference. The first of them is based on short-circuit evaluation. If we have conditions like *False and b*, the b is not evaluated because a whole expression is False. The same logic applies to the *True or b*, where no matter what b is, the expression is True. For that reason, it is a good idea to write condition elements that are the most likely to be "False" first.

```
// EXPENSIVE and CHEAP
math.entropy(0, filesize) > 7.0
and
uint16(0) == 0xFFFF

// CHEAP and EXPENSIVE
uint16(0) == 0xFFFF
and
math.entropy(0, filesize) > 7.0
```

Figure 8: Examples of conditions where the order matters

From version 3.10, the integer range loops were also optimized:

```
for all i in (0..100): (false)
for any i in (0..100): (true)
```

Figure 9: For loops in Yara are also optimized.

Both of these loops will stop iterating after the first time through.

6 Conclusion

In this presentation, we discussed the unique tool for malware analysis, Yara. We presented how does it work, how does it perform, and how we can get the best out of its potential. We also presented changes that are also available online.

The future is difficult to predict, but Yara is most likely to be used in the following years more and more due to still-growing numbers of malware. For that reason, a deep understanding of Yara could be a key advantage in moving forward.

Acknowledgment: The author would like to thank Jakub Křoustek, Marek Milkovič and Threat Intelligence team in Avast Software for their guidance and support.

Author details

Dominika Regéciová

Avast Software
Brno, Czech Republic
dominika.regeciova@avast.cz
ORCID iD: 0000-0001-8729-6999

References

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

[2] D. Regéciová, D. s. Kolář, and M. Milkovič, "Pattern matching in yara: Improved aho-corasick algorithm," *IEEE Access*, vol. 9, pp. 62857–62866, 2021.

[3] "Bitcoin wiki: Address." https://en.bitcoin.it/wiki/Address.

[4] "The official yara documentation." https://yara.readthedocs.io/en/v3.10.0/.

[5] "Short-circuiting boolean operators in yara." https://inquest.net/blog/2018/12/18/yara-short-circuiting.

[6] "Stringless yara rules." https://inquest.net/blog/2018/09/30/yara-performance.

[7] "Yara performance guidelines." https://github.com/Neo23x0/Yara-Performance-Guidelines.