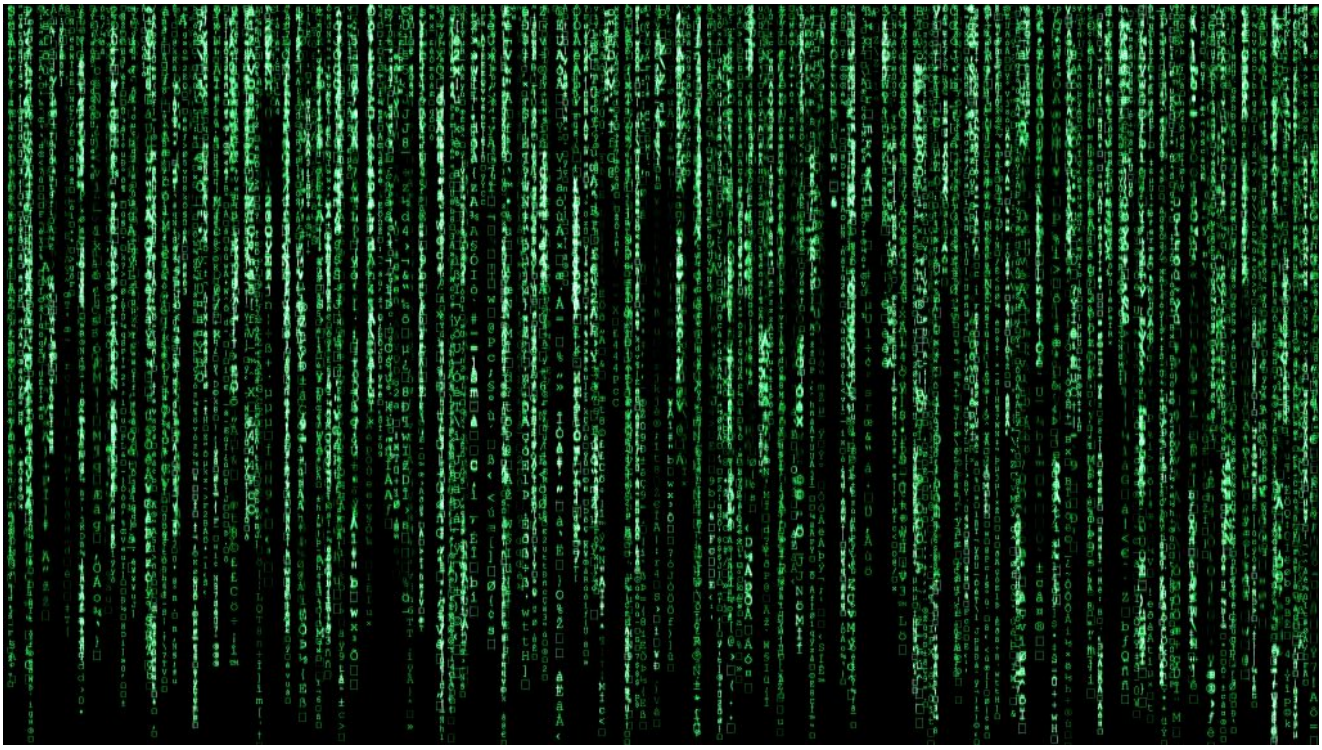
Yara: In Search Of Regular Expressions

engineering.avast.io/yara-in-search-of-regular-expressions/



This blog post is based on my paper Pattern Matching in YARA: Improved Aho-Corasick Algorithm and a pull request that I opened on the upstream version of Yara. My main goal is to describe the changes from a more practical point of view and also provide some updates and changes that were made from the initial design proposed in the paper, as practical usage in Avast showed me some additional ways how to improve my code.

Yara is a pattern-matching tool used mainly to classify and identify malware. It is also an open-source project with an active community that is working hard to improve this tool every day. In this post, I will not go into details about what Yara is and how it can be used. For that, you can visit my personal blog or watch my talk about Yara at Botconf 2022.

Motivation

We use the Yara tool every day at Avast, and it is a critical help for our malware analysts. We are also trying to give back and bring as many of our improvements and additional features into the upstream version as possible.

The changes I am proposing address a problem faced by our analysts: the "*slowing down*" warning, where rules can generate the following:

```
$ ./yara yara_rule.yar -r /directory
warning: rule “can_be_slow” in ../yara_rule.yar(4): string “$slow_string” may slow
down scanning
```
Code language: Bash (bash)

Besides the fact that we should be aware of every warning, not just straight errors, this negatively influences the usability of rules. For example, the VT Hunting platform does not accept rules with warnings.

Most notably, one of our analysts was trying to create a rule that would detect Bitcoin addresses, as in the rule below, and had to combat this warning constantly.

```
// btc.yar rule
rule contains_btc_address
{
   strings:
       $btc = /[13][a-km-zA-HJ-NP-Z1-9]{25,34}/ fullword ascii wide
   condition:
       $btc
}
```
Code language: PHP (php)

However, we would like to use some of these rules, so what is the solution? And what exactly is the reason that Yara is warning about low performance? Can this be changed or improved?

Problem analysis

After analyzing the problem, I found the potential issue that decreased the number of primarily regular expressions that could be used in Yara rules without warnings.

To understand that, I will briefly explain how Yara works internally. For more details, you can visit an excellent summary in Performance Guidelines, where I also added a few of my notes, or look directly at the documentation and the code itself.

Yara firstly searches for potential matches with the help of the Aho-Corasick machine. In this step, it finds all possible places in input, where the whole strings can be found, and additionally, where rules can hit. Just after that, the bytecode engine is run to verify the matches, and then the conditions of the rules are also checked.

For this first step, short substrings, up to 4 bytes, are selected from strings. A series of heuristics are used to choose the unique sequences.

This is wanted as we do not want to search for the `00 00 00 00` sequence as it would hit too many times.

The main problem is that the heuristics strongly prefer text strings and cannot work with regular expressions very effectively.

In our case, the Bitcoin addresses let to zero-length atom. What does that mean? The bytecode engine must check every byte from the input because every position could be a potential match. And this is a truly time-consuming task, thus the warnings.

But what if we could help Yara better understand the regular expressions and help it narrow down the number of potential matches? Would the speed of scanning improve?

After creating the proof of concept, I can say yes, we can do it.

Proposed Changes

You can read about all the details and formal theory behind the changes in my paper, but I will try to describe the changes in a more practical way here.

First, I extended the definition of atoms from text strings to regular expressions. That means that they are able to represent the subclass of these languages. I kept the length of 4 bytes or rather a sequence of four classes. The class is a set of characters. It can be one character, that would be a literal, a collection of characters, like in `[a-d]` representing the set `{a, b, c, d}`. The last case is noted as a dot in regular expression as any character (sometimes without the new line, depending on the implementation).

In this version, the Bitcoin address would be represented as `[13][a-km-zA-HJ-NP-Z1-9]{2}`, noting the repetition of the second class. It also could be `[13][a-km-zA-HJ-NP-Z1-9]{3}`. Still, experiments showed that the additional class does not improve the speed as significantly as the repetition does not provide a substantial decrease in the number of potential matches.

The heuristics for selections of atoms were broadened and changed to prefer still simple text literals where they can be found and include more complex ones, as in our case at the same time.

This part underwent several changes, and in the end, I tried to keep the changes as minimal as possible. My goal is not only to improve the scanning of the regular expressions but also scanning of other cases the same in the worst scenario. Even a tiny change in heuristics can dramatically affect the number of potential matches and thus the scanning speed, so this had to be handled carefully.

The class itself is represented as a bitmap, an array of integers where individual characters are coded with a bit value set to 1 if the character is set, 0 otherwise. For example, for character `a`, the bit on position `bitmap[12][1]` is used, as its ASCII code (97) is divided by 8, and the resulting number is 12. The presence of the number is indicated by a bit with the value one on the position given by operation modulo 8.

The selected atoms are then inserted into the Aho-Corasick automaton, a prefix tree machine. It is an extended version of a general finite automaton, where we are adding strings from the root state and detecting the common prefixes as we create states only when necessary. In my proposed version, it can also accept the classes instead of just simple text strings. In this step, the same classes are detected in the common prefix in a similar fashion as before.

This leads to a problem because the Aho-Corasick contains so-called failure links that allow several matches (and even interleaving ones) to be detected at one pass from the input. This requires that the machine is deterministic even with failure links that return to previous states. My change, however, broke this determinism. Additional changes had to be made, and the machine was transformed into a deterministic one even for failure links.

With these changes, we detected a subset of potential matches again, and the bytecode engine is run less often, even with the use of regular expressions. Also, the Bitcoin address can be detected without warnings about the slowing down scanning with my proposed changes.

But what about the actual speed of the scanning? Was it improved?

Tests

For tests, I run both versions on the server with the Debian operating system in release 4.9.168 on a 64-bit architecture with a processor Intel Xeon 5–2697 v4 with a clock rate of 2.30GHz. The memory clock rate was 2.40GHz. As a compiler, the GCC in version 8.3.0 was used.

I selected two rules, one with the Bitcoin address and one from unit tests, to provide additional information on the overall effects of the proposed changes. More tests can be found in my paper.

To measure speed, I scanned two larger folders with various subfolders and files with 15 GB and 760 GB of data in total.

BTC Address Rule

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

Code language: PHP (php)

The rule btc.yar	Set 1 (15 GB)	Set 2 (760 GB)
The upstream version	1m 20.134s	8h 55m 21.727s

The classes version	6.683s	40m 22.879s

The results for btc.yar YARA rules

Hex Value Rule

```
// hex.yar rule
rule hex_value_test
{
   strings:
       $hex = {61 6? 63 [1-5] 65 66}
   condition:
       $hex
}
```

Code language: PHP (php)

The rule hex.yar	Set 1 (15 GB)	Set 2 (760 GB)
The upstream version	6.399s	39m 7.329s
The classes version	6.285s	37m 43.513s

The results for hex.yar YARA rules

Conclusion

Yara is a fascinating tool for pattern matching, and I am happy that there is an online community that is very active and helpful.

At Avast, we want to give back as much as possible, and for that reason, I also published the changes that we have been using in the company for a while now.

Note that there will be a discussion with the authors of Yara about this commit, and there can be some additional updates and changes, so the updated version is the most suitable for general usage.