IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received April 8, 2021, accepted April 17, 2021, date of publication April 21, 2021, date of current version April 30, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3074801

Pattern Matching in YARA: Improved Aho-Corasick Algorithm

DOMINIKA REGÉCIOVÁ[1], DUŠAN KOLÁŘ[1], AND MAREK MILKOVIČ[2]
[1]Faculty of Information Technology, Brno University of Technology, 612 66 Brno, Czech Republic
[2]Avast Software s.r.o., 602 00 Brno, Czech Republic

Corresponding author: Dominika Regéciová (iregeciova@fit.vut.cz)

This work was supported by the Brno University of Technology project "Application of AI methods to cyber security and control systems" under Grant FIT-S-20-6293.

ABSTRACT YARA is a tool for pattern matching used by malware analysts all over the world. YARA can scan files, as well as process memory. It allows us to define sequences of symbols as text strings, hexadecimal strings and regular expressions. However, the use of regular expressions is limited because of the concern that it can slow down the scanning process. In this paper, we analyze the true nature of regular expressions in YARA and their implementation. We have, in fact, discovered several reasons why regular expressions can slow down scanning based on the nature of the used algorithm, Aho-Corasick. We have proposed a new version of this algorithm and have implemented it in the original version of this tool. The experiments are presented, proving that the speed of pattern matching with regular expressions can indeed be improved. In selected cases, the proposed version was about 27% faster than the original version. And in instances where strings were optimized for the original version, their speed was found to be comparable.

INDEX TERMS Aho-Corasick algorithm, pattern matching, regular expressions, YARA.

I. INTRODUCTION

In this paper, the pattern matching problem is understood by finding all occurrences of patterns in the input. Our goal is to target exact matches and we typically search for them in files or process memory. We may want to search strings that likely indicate malware, and suspicious activity in the network, or specific DNA sequences. The pattern matching problem's formalization is based on string-matching from the book *Introduction to Algorithms* [1]. We modified and extended the definition for the more general form and we match several patterns instead of one. The use of regular expressions is allowed as well:

We assume that the text is an array $T[1 \ldots n]$ of length n, where the characters are drawn from a finite alphabet Σ. We define a finite set of patterns P. The pattern is a string or regular expression. Each pattern p represents one or more literal string arrays $p'[1 \ldots m]$ of length m over the Σ alphabet.

We say that pattern p' occurs with shift s in text T if $0 \leq s \leq n - m$ and $T[s + 1 \ldots s + m] = p'[1 \ldots m]$. If p' occurs with shift s in T, then we call s a valid shift; otherwise, we call s an invalid shift. The pattern matching problem is the problem of finding all valid shifts with which all given patterns p' occur in a given text T.

Generally, the pattern patching problem is divided into two parts. In the first so-called preprocessing phase, the model for matching is created. The models are finite automata, prefix trees, and other models. The second phase is the matching process itself. It should be noted that even algorithms with high time complexity of the preprocessing stage are still applicable in practice. The reason is we can generate the model once, store it, for example, in the bytecode, and then repeatedly run over data.

YARA is a defacto industrial standard for pattern matching in computer security and other fields. For that reason, there is a large community that is publicly providing tools and patterns for YARA.

There are, for example, many projects for an automatic generation of patterns (see [2], [3] and [4]). From its nature, malicious software is constantly evolving, and automatic tools help analysts maintain so-called YARA rules using these patterns for detection.

Some tools are expanding the usability of YARA. There are YARA bindings for Python,[1] Java,[2] or Go[3] languages. There

[1]https://github.com/VirusTotal/yara-python
[2]https://github.com/p8a/yara-java
[3]https://github.com/hillu/go-yara

The associate editor coordinating the review of this manuscript and approving it for publication was Yilun Shang.

```
Input:  The cute cat can sleep at a tea party.
Matches:    cute cat           at   tea
                 at
```

FIGURE 1. An example of the Aho-Corasick algorithm: matches.

are also projects such as support distributed read mapping for extensive databases (e.g.,> 10 GB) [5] used for DNA sequencing. Some additional tools, services, and rules repositories are listed on Github page.[4]

With that being said, companies, such as Avast, are using YARA on massive scales and are starting to notice the problems with scanning speed and effectiveness. For example, the limitations for using YARA as intrusion detection systems in the network were described in a paper [6], where the authors used two-dimensional stages and pipelines based on an FPGA hardware accelerator.

The [6] paper is the closest research found with the aim of improving YARA capabilities to matching patterns. Nevertheless, in this paper, we do not limit improvements to HW changes, instead, we propose changes to improve general CPU-based implementation.

II. AHO-CORASICK ALGORITHM

Introduced first in the paper *Efficient String Matching: An Aid to Bibliographic Search* from 1975 [7], the algorithm got its name from its authors, Alfred Vaino Aho and Margaret John Corasick [8]. Their goal was to create a simple, yet effective, way to search multiple patterns in a text quickly. The general idea combines the Knuth-Morris-Pratt algorithm [1] with finite state machines.

From that day, the Aho-Corasick algorithm is still a popular solution, and is optimized for several purposes. In this paper, we describe the changes that were made to fit the malware analysis needs better. Several other proposals can be found in the Snort[5] for intrusion detection systems (IDSs) and optimizations for hardware implementations [9]–[11].

The algorithm itself will be presented in a practical example. Four strings were selected for matching: *at*, *cat*, *cute*, and *tea*. The input string will be a sentence *The cute cat can sleep at a tea party*. Several matches should be found, as shown in Figure 1. The pattern *at* is, for instance, found twice — on positions starting with the 10th and 23rd characters in the input (counting from 0).

As we mentioned before, the first step is to create a model from a set of strings. In the case of the Aho-Corasick algorithm, the prefix tree machine is built (Figure 2). The construction of the pattern machine takes time proportional to the sum of the patterns' lengths. Every pattern is being added from the root state, so that the common prefixes are detected, and we do not create additional and redundant states. The edges between these states are labeled by letters that must be read from input to make a transition to the next state, defined as the *goto* function. In the final states, the so-called *output* function (Figure 3) will provide information about matched patterns and their input text positions.

[4]https://github.com/InQuest/awesome-yara

[5]Snort is a free open source network intrusion detection system and intrusion prevention system: https://www.snort.org/

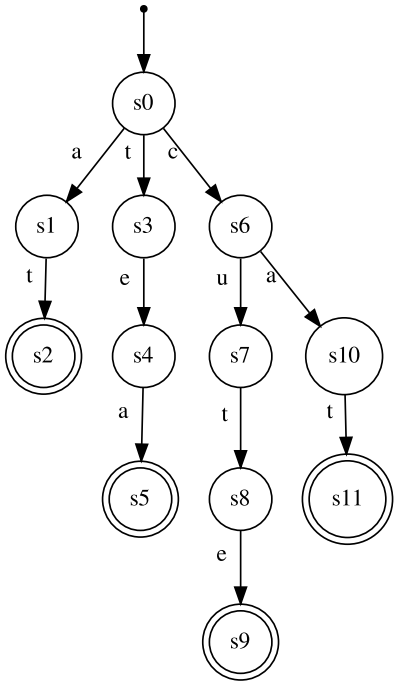


FIGURE 2. An example of the Aho-Corasick algorithm: machine and goto function.

TABLE 1. An example of the Aho-Corasick algorithm: failure function.

i	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11
f(i)	s0	s3	s0	s0	s1	s0	s0	s3	s4	s1	s2

```
i       output(i)
-----------------------------------------
s2      at
s5      tea
s9      cute
s11     cat, at
-----------------------------------------
```

FIGURE 3. An example of the Aho-Corasick algorithm: output function.

Additionally, the Aho-Corasick machine has a *failure* function. If there are no transitions, the algorithm can do during matching, the failure function (Table 1) is used. Instead of returning to the root state, the machine returns to a state where the match is likely based on patterns. For instance, in the state `s9`, the next state will be `s4`, because the end of pattern `cute` can be the beginning of the pattern `tea`. Thanks to this function, the algorithm can match even interleaving patterns as `[cu[tea]]`.

In the second phase, the algorithm starts at the input text's beginning and in the root state during searching for patterns. With every character from an input, a new state is decided with the transition function and the next character is read. If the state is final, the output function is used as well, thus reporting the match. In case there is no transition for the current state and character from the input, the failure function

is used, and the state is changed and the current character is used again as an input for the transition function.

The algorithm processes the input string in a single pass, and all occurrences of keywords are found, even if they overlap each other. For the input of the length n, the machine makes less than $2n$ state transitions [7]. The machine makes at most n goto transitions. The number of failure transitions is connected to the depth of the state (the length of the shortest path from the root state), and is always at least one less than the number of goto transitions (that equals the depth of the state).

III. YARA

YARA is a popular tool in the world of malware analysis and beyond. This open-source project defines itself as the pattern matching Swiss knife, describing malware families and more with textual or binary patterns [12]. YARA is a tool containing the custom languages that enable users to search inside the files and process memory. The one analogy uses another popular tool for matching purposes: YARA is to data what Snort is to network traffic [13].

The automation and re-usability are the substantial advantages of YARA. Experts or malware analysts' knowledge can be shared; the information is not limited to their memory or the time they are working for the team. With YARA, the needed data for malware detection can be shared all around the world. The tool itself does not contain any rules, and syntax units that define logic and structure of patterns, but there are several ways to share them through online services like Malpedia [14].

The company, VirusTotal, which created YARA and is owned by Google, provides the VirusTotal Hunting system. It is a service where the users can import their rules, matching them against the files submitted to VirusTotal. For many companies using YARA, this is a critical part of the infrastructure, monitoring their YARA rules' matches. VirusTotal also provides massive infrastructure with high-speed scanning. On the official website, VirusTotal claims that their data set includes more than 2.4 billion files, and the system is scanning around 1.8 million new files a day [15].

As explained in Section IV, YARA changes' initial motivation came from limitations to the infrastructure itself. Not only changes in YARA matters, since maybe an even bigger goal is to get these changes into VirusTotal's official YARA implementation and then into VirusTotal Hunting.

A. YARA USAGE EXAMPLE

The basic logical unit in YARA is called a rule. The rule is a collection of strings section where we define patterns and the condition part where we specify logic and use of the patterns. We can define not only patterns, but also additional conditions, such as the number of matches or a position of matched pattern. On top of that, YARA's syntax (influenced by C language) is more intuitive and more powerful than in *grep* or any similar searching tool.

In order to illustrate YARA's elegance, let us say we want to list all files in a directory with the *PNG* format. Based on the official specification (see [16]), the first eight bytes of a PNG file always contain the hexadecimal values shown in Figure 4:

```
89 50 4E 47 0D 0A 1A 0A
```

FIGURE 4. The first eight bytes of a PNG file format.

```
rule png_file
{
  strings:
    $hex = {89 50 4E 47 0D 0A 1A 0A}
  condition:
    $hex at 0
}
```

FIGURE 5. The YARA rule for the detection of PNG files.

```
$ ls cats/
cat_jpg.jpg  cat_png.png
```

FIGURE 6. The directory `cats`.

```
$ ./yara png_file.yar -r cats/ -s
png_file cats//cat_png.png
0x0:$hex: 89 50 4E 47 0D 0A 1A 0A
```

FIGURE 7. The results of the use of YARA rule.

To detect the PNG format, we need to specify a hexadecimal string describing the first eight bytes. Additionally, we want to specify that the pattern should be at the beginning of the file since, otherwise, it would not be a PNG format. As we can see in Figure 5, the resulting rule has a quite simple form. The hexadecimal string is in the braces defined as a variable `$hex` (variables in YARA always start with a dollar sign). The condition about a position is written as `$hex at 0`. YARA reports only files starting with a string `$hex`.

With this rule, saved like `png_file.yar`, we can detect the `.png` files. Consider a directory `cats` with two files inside (Figure 6):

YARA is a command-line tool, so we need the terminal with `yara` binary code available to scan the directory.

We can see the results of the scanning in Figure 7. The first argument is our rule for scanning; the second one is the name of the directory. YARA can scan files and processes as well. The `-r` option is for a recursive search in the directory, and the `-s` option prints matched strings.

The result says that we found one matching file. The *cat_png.png* has the searched hexadecimal values starting on position 0×0. The other file, *cat_jpg.jpg*, does not contain these values, and YARA did not match it. This is only an elementary example of YARA rules. For more information, we recommend reading official documentation (see [17]).

B. AHO-CORASICK ALGORITHM IN YARA

YARA uses the Aho-Corasick algorithm for matching, and also brings additional techniques and heuristics in its implementation to provide faster scanning. First of all, Yara uses so-called *atoms* to search for potential matches. The atoms are short sequences of characters up to four bytes. Only these substrings are inserted into the Aho-Corasick machine.

The output function stores the atoms found in file or memory to the list of positions where the potential match can be located. This list is later checked by a complex algorithm that verifies if the match is true or not for the whole string. The main idea is to select a unique set of atoms to provide the complete match list (not missing any potential matches) and not overload the checking mechanism with false-positive cases. Just this decision alone makes a massive difference in the scanning speed, as we will see in Chapter V. For this reason, we must be careful on which atoms are used and which are not.

Secondly, in the original article by Aho and Corasick [7], the machine accepted only text strings. In YARA, there is more broad implementation supporting hexadecimal strings and also a subclass of regular expressions. Based on the official documentation [17], this subclass is based on Perl-like syntax, and is quite large, even though it does not contain backreferencing and similar features. However, in Chapter IV, we will show how strict the subclass is in reality. Even if we can express a large number of regular expressions, YARA ignores them most of the time.

As in the original version, every state (except for the root state) has exactly one character as an input symbol for which the transition function returns this state. For this reason, when we want to search for case-insensitive patterns, we cannot combine lower and upper case letters in a transition function but instead we create all possible permutations in several atoms. For string *abc*, we create additional atoms: *Abc*, *aBc*, *abC*, etc. Based on the maximal length of atoms, 4 bytes, this should not overdraw 16 atoms theoretically. However, there are still other options, such as *wide* (strings encoded with two bytes per character), or metacharacter dot '.' (all ASCII characters except a new line), increasing the overall number of atoms very quickly.

The last, but not least, specific aspect in YARA is how we access the Aho-Corasick machine during the matching process. The whole Aho-Corasick machine is stored in the interleaved state-transition matrix [11]. The main advantage is we have a constant access time to get the next state for the current symbol in the input text.

The matrix `M`, which represents the machine's transition table, is an array of 32-bit integers. Each state has its index `I`, starting with the root state on the position `M[0]`. In the position `M[I]`, the state's failure function is stored. If there is no transition for the current input byte, the failure function will be used. The next state transitions are inserted on positions `M[I+1+B]`, where the B is the ASCII value of the input byte. The position contains the next state's index if the transition function defines any next states; otherwise, it contains zero.

To allocate 257 ints long array for one state (1 failure link plus 256 possible input characters) is not practical because, in most cases, the transition is defined only on a couple of symbols. For this reason, the states' arrays are interleaved, meaning that the transitions from several states can be next to each other. To be able to distinguish between state *s1* on `M[I+1+'b']` and *s2* on `M[(I+1)+1+'a']` the lower 9 bits are used for storing the offset of the index relative to its owner state (value of character plus 1). For this case, if the value `1+'b'` is encoded, the transitions belong to state *s1*. The next state index is encoded using higher 23 bits of an integer number. Each transition is therefore connected to its state, even if the slots are interleaved.

All these changes and improvements are designed to increase the scanning speed and eliminate the selection of atoms that are inefficient (e.g. too general). However, these changes do not always work the best way, mainly when the strings are defined as regular expressions.

IV. IMPROVED AHO-CORASICK ALGORITHM IN YARA

Regular expressions are a very disputable topic. The most common critique is they are overused and the authors of this paper find this statement mostly correct. They are often used in languages that are stronger than the class of regular expressions, for example, for context-free languages. These cannot be correctly accepted by regular expressions, which leads to several problems, even including security vulnerabilities. On the other hand, there are many cases where regular expressions are more practical and simpler to use. A typical example is malware analysis, where analysts want to detect e-mail addresses, IP addresses, etc. However, they do not need to validate the correctness of found strings.

In YARA, there is a massive problem because it is not simple to write a regular expression that will not warn us about slowing down the scanning process. This is more than a symbolic issue because systems like VirusTotal Hunting do not accept rules with warnings, eliminating the vital use of YARA rules. Some regular expressions can indeed slow down the whole process, but avoidance is not a solution either. In this paper, we tested several ways of describing the regular expressions, and we looked up into YARA implementation, which creates the slowing down warning.

Our proposed version is focused on three main changes that are presented in this section. Firstly, instead of working with substrings, we create atoms as stand-alone regular expressions capable of storing more information than their previous implementation. This will allow for subtler manipulation with strings defined in YARA, and as we present in Section V, this can significantly improve the scanning performance.

Secondly, we present the improved Aho-Corasick machine that can match a new atom's form. These include creating the machine on the set of regular expressions as well as additional steps for calculation of deterministic goto and failure functions.

The third step contains creating the interleaved state-transition matrix that will reflect the changes made in the previous two steps, with the matrix used more effectively as each state will hold more input symbols per state.

A. SELECTION OF ATOMS

In the Aho-Corasick algorithm's proposed version, the atoms are still being selected. However, instead of working with

TABLE 2. The original (YARA) and proposed (MYARA) versions create different atoms (substrings) for the Aho-Corasick machine.

String	YARA	MYARA
"abc" nocase	8: abc, Abc, ...	1: [Aa][Bb][Cc]
/[a-z]/	1: ""	1: [a-z]
/[a-z][0-9]/	1: ""	1: [a-z][0-9]
/a.c/	256: aac, abc, ...	1: a.c
/a.c/ ascii wide	512: aac, a0a0, ...	2: a.c, a0.0
/a.c/ nocase	1232: aac, Aac, ...	1: [Aa].[Cc]
/a.c/ ascii wide nocase	1848: aac, A0a0, ...	2: [Aa].[Cc], [Aa]0.0

atoms as text literals, where one substring can generate multiple text atoms, we propose working with atoms almost like with regular expressions. Every byte in an atom that can be one of three categories: a literal (ASCII character), dot metacharacter (representing all ASCII characters), or *class*.

The class, or *character class* as defined in [18], is the list of characters we want to match. The class `[a-z]` matches every lower-case letter from the English alphabet, while `[0-9]` matches digits.

The resulting differences can be seen in Table 2. The YARA version represents the original version, MYARA, the proposed one. Note that in the case of `[a-z]`, the original version will select one atom of the length zero (e.g. empty string).

In the new version, every byte is encoded as a bitmap. Adding a character or testing its presence in the bitmap is a simple task checked with fast bitwise operators. This enables us to store otherwise large amounts of atoms in one, as shown in the MYARA version (Table 2). Moreover, the selection of substrings for atoms is also different, because in this case, the classes and dot metacharacter have a much more important role. In instances where YARA is working blindly with empty substrings, and this results in a naive pattern matching algorithm searching for a match byte by byte, MYARA is strategically selecting atoms to provide a more specific list of potential matches. As will be shown in Chapter V, this will lead to significantly faster scanning, mostly in cases where short atoms were selected in the original version.

B. AHO-CORASICK MACHINE

The proposed version of the Aho-Corasick machine changes the limitation on how many transition characters one state can have. Only one input character is allowed in the original version, which means the algorithm can make a transition into the state based on that single character. The proposed version enables a set of characters. This means the input function accepts multiple characters per state. This is also in correlation with atoms so that the machine can make the transition with a literal, dot metacharacter, or class. Every variant is stored in a bitmap with the type of atom and byte value that is useful when working with the literal atom.

This also leads to a more compacted machine than in the original version. However, there is an additional step that is needed to be done before the failure function calculation. Because several symbols are allowed in one transition, the Aho-Corasick machine is not necessarily deterministic for the failure function. This is true even when the machine is built with a deterministic transition function. With the original version, the history of already read symbols was implicitly stored. That helped the failure function. But by allowing the wildcard symbols and classes, several variants are possible. With the string `a.`, the `ab` could be read, as well as `az`. For this reason, ambiguous cases as these have to be eliminated. For an example with strings `a.` and the `ab`, we need to change `a.` to `a[^b]` to create a distinction between found strings (`[^b]` which means every character except b).

The process is similar to the calculation of the failure function. During the construction of the failure function, we use the term *depth of the state s*, which is the length of the shortest path from root state *s* [7]. The failure functions are computed for all states with depth one, then two, and so on. The root state itself does not have a failure function. The states of depth one can only return to the root state. For other states, the algorithm is starting from the root state and is checking if there is a transition function defined with the same input symbol. When in a state where the symbol *a* from input was read, the failure function points to previous examples where *a* could also be read.

During the failure function determinization process, the Aho-Corasick machine is processed in the same way as in the original algorithm. The goal is to have only one state that is the output of the failure function. In every state, all possible transitions are compared. If the sets of symbols are conjunctive, the sets are adjusted. This can include the creation of new states, symbols withdrawal from a set, or even deleting states that become redundant.

In the worst-case scenario, the algorithm for deterministic failure function requires time quadratically proportional to the number of states in the Aho-Corasick machine. In the best-case scenario, the time complexity is linear. It should be noted that the limited size of atoms (up to 4 bytes) restricts the machine's depth.

C. INTERLEAVED STATE-TRANSITION MATRIX

Based on the different nature of the Aho-Corasick automaton, the interleaved state-transition matrix scheme has also changed. The general algorithm remains. Each state has its slot – part of the matrix starting at position `M[I]`. In this position, the failure function is stored. All next states are located on position `M[I+1+B]`, where *B* is the *ASCII* value of the current character from the input text. Because of more input symbols per state, the slots are more densely occupied. The determinization process will also lower the chances of conflicts while selecting the base index for the state.

V. EXPERIMENTS

Tests were run on the server with the Debian operating system in release 4.9.168 on 64-bit architecture with a processor Intel Xeon E5-2697 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.

TABLE 3. The minimal average speed for selection of rules.

Rule	YARA [s]	MYARA [s]	MYARA/YARA
Bitcoin1	169.700	17.623	0.104
Bitcoin2	23.787	17.564	0.738
Bitcoin3	23.980	24.032	1.002
Emails1	10.510	13.147	1.251
Emails2	10.456	13.070	1.250
Emails3	15.378	11.094	0.721
Ini	10.234	10.110	0.988

The main goal of testing was to compare the original VirusTotal implementation and the implementation of the proposed new algorithm (both in version 4.0.2) in their performance. No additional implementation of YARA was chosen. As we mentioned in Section I, there is a version optimized for FPGA [6], but the comparison on the CPU system would not be comparable. Additionally, we exclude alternative implementations of Aho-Corasick algorithm and pattern matching tools such as Snort, HyperScan,[6] or RE2.[7] They do not provide an equivalent substitution for YARA in current use cases.

Three sets of YARA rules were chosen for this purpose. The first ruleset, containing five rules, will be described in detail later. The second one was obtained from the public repository by ReversingLabs [19]. It is a more extensive ruleset, including 139 files and 371 rules. ReversingLabs repository represents real-world examples of rules used for malware analysis and detection with various strings.

For both rulesets, tests were repeated over six days, where every rule (or file in case of ReversingLabs ruleset) was run 15 times. The minimal values from each day were selected, and the average was calculated. As such, the impact of the operating system or memory caching should be minimized.

Scanning was done on the publicly available data corpora representing real data. The package, 8.54 GB of data in total, contains the following data formats: bitcoin historical data, e-mails, and INI format files. Information about sources, rulesets, and others can be found on the public repository.[8]

The results of the ruleset number 1 can be seen in Table 3. The ReversingLabs ruleset is shown in Table 4.

A. BITCOIN ADDRESS DETECTION

The addresses of cryptocurrencies are one instance of strings that analysts may want to detect in the malware. Similarly to the file formats, the addresses also have standardized forms. For the sake of simplicity, we consider only Bitcoin addresses in the P2PKH and P2SH formats [20]. These start with the number 1 or 3. The length of the address can vary, from 26 up to 34 characters. The selected two formats are case-sensitive. The address is a random sequence of numbers and letters in both lower and upper case. Some letters and numbers are never used, like number 0 and upper case letter O to prevent visual ambiguity [20].

[6]https://www.hyperscan.io/
[7]https://github.com/google/re2
[8]https://github.com/regeciovad/yact20

```
rule contains_btc_address_version_1
{
  strings:
    $btc = /[13][a-km-zA-HJ-NP-Z1-9]{25,34}/
           fullword ascii wide
  condition:
    $btc
}

rule contains_btc_address_version_2
{
  strings:
    $btc1 = /1[a-km-zA-HJ-NP-Z1-9]{25,34}/
           fullword ascii wide
    $btc3 = /3[a-km-zA-HJ-NP-Z1-9]{25,34}/
           fullword ascii wide
  condition:
    $btc1 or $btc3
}

rule contains_btc_address_version_3
{
  strings:
    $btc = /(1|3)[a-km-zA-HJ-NP-Z1-9]{25,34}/
           fullword ascii wide
  condition:
    $btc
}
```

FIGURE 8. The YARA rules for the detection of Bitcoin addresses.

In Figure 8, there are three ways to describe Bitcoin addresses with regular expressions and how it will be shown, all of them result in very different atoms, and Aho-Corasick machine afterward. Note that keyword *fullword* means we are looking for address delimited by whitespace characters, *ascii* means we search for characters per byte, and *wide* is a subclass of UTF-16, where the characters are encoded as 2 bytes. However, in YARA, the first nibble is always 0. UTF-16 encoding family is not supported for that reason.

Version 1 (see Figure 8) is defined as a concatenation of two classes. This is the simplest notation of these three, but also the most problematic. The original version of YARA actually ignores the classes in regular expressions. In the process of choosing atoms, not a single byte is selected, and the Aho-Corasick automaton matches everything because the accepting state is also the root state. Of course, this will slow down scanning.

The proposed version, MYARA, is, on the other hand, working with classes more intuitively, as shown in Figure 9. Two atoms are generated:

`[13][a-k...1-9][a-k...1-9][a-k...1-9]` and `[13]0[a-k...1-9]0` (zeros between the original classes are for the wide option). This approach requires a little bit more preprocessing time, but the results speak for themselves: during testing, the YARA needs, on average, 169.7 seconds. The MYARA needs only around 17.6 seconds for the same dataset (Table 3).

Version 2 of the Bitcoin address (see Figure 8) is broken down into two strings with prefix numbers *1* and *3*. The YARA is selecting these prefix numbers as atoms (Figure 10). These atoms are still not sufficient enough for matching, and they still yield slowing down warning. However, this simple change radically decreased the time necessary for

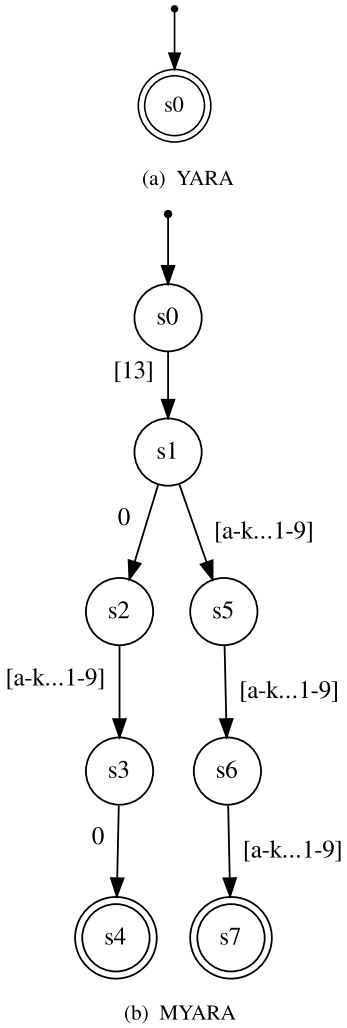


FIGURE 9. Aho-Corasick machine for Bitcoin addresses version 1.

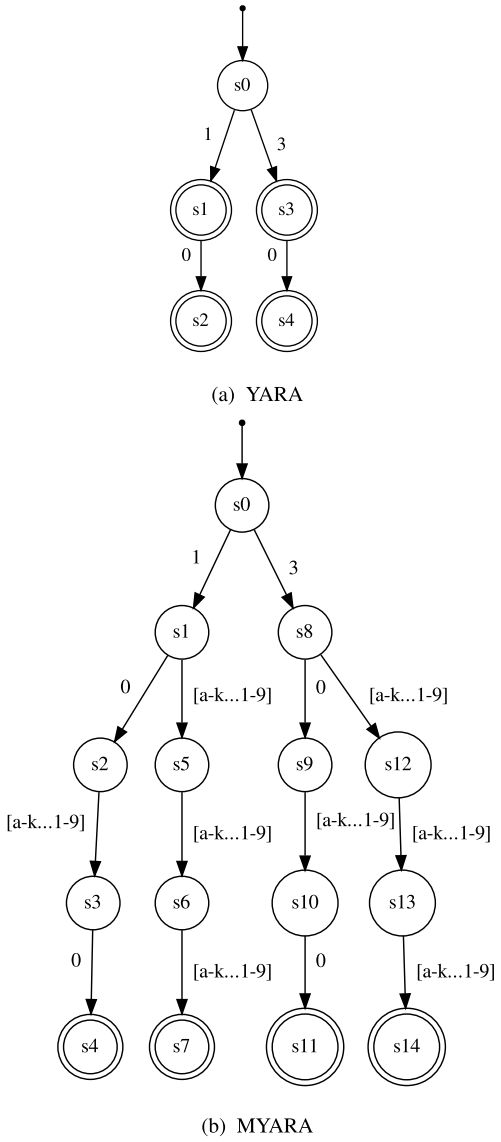


FIGURE 10. Aho-Corasick machine for Bitcoin addresses version 2.

scanning. The version 2 of the Bitcoin rule needs only around 23.8 seconds. The MYARA is still faster; it needs around 17.56 seconds. The Aho-Corasick is, in this case, bigger, but it does not impact the time negatively.

Version 3 of Bitcoin address (see Figure 8) contains an alternation. This generates for both YARA and MYARA the same atoms: `1` and `3` (as YARA version in Figure 10). In this case, the alternation is problematic, as YARA separates it from the rest of the string when searching for atoms. This can also lead to slower scanning, as in the case with classes. In future work, this will be reflected as well. The time for scanning is, in this case, similar, YARA is about 0.2% faster.

We would prefer version 1 or 2 for MYARA and version 2 for YARA in our proposed work. In the case of YARA, the dramatically slowest one is version 1. For MYARA, version 3 is the slowest, but not so significantly.

B. E-MAIL ADDRESS DETECTION

The e-mail addresses represent other typical patterns in malware analysis and are a good candidate for regular expressions. One possible way to describe the e-mail address as shown in Figure 11, version 1. Note that the *i* after string means the case-insensitive option, which can also be specified with *nocase* option behind string, as shown in version 2. Atoms selected are shown in Figure 12. In only this case from the first ruleset, the YARA version was faster (MYARA was slower by about 25%). When inspecting the reasons, we discovered the short atoms '@' and '@0' were effective enough because, in a dataset containing e-mails, the @ is preciously pointing to the e-mail addresses. To verify this, we ran additional tests without @, and the results showed that this is indeed this case. In version 3 (11, 13) MYARA is about 28% faster. It should be noted that versions 1 and 2 yield the same results for both YARA and MYARA.

C. INI FILE FORMAT DETECTION

An INI file is a configuration file for computer software that consists of a text-based content with a structure and syntax comprising key-value pairs for properties and sections that

```
rule contains_emails_address_version_1
{
  strings:
    $email =
    /[-a-z0-9._%+]@[-a-z0-9.]{2,10}\.[a-z]{2,4}/i
    ascii wide
  condition:
    $email
}
rule contains_emails_address_version_2
{
  strings:
    $email =
    /[-a-z0-9._%+]@[-a-z0-9.]{2,10}\.[a-z]{2,4}/
    nocase ascii wide
  condition:
    $email
}
rule contains_emails_address_version_3
{
  strings:
    $email = /[-a-z0-9.]{2,10}\.[a-z]{2,4}/
    nocase ascii wide
  condition:
    $email
}
```

FIGURE 11. The YARA rule for the detection of e-mail addresses.

organize the properties [21]. There are two primary components of INI files: keys (properties) and sections (Figure 14). A key is a pair of names and values delimited by '='. The section is a set of keys grouped by the name of the section. In this particular case, only strings in lowercase are matched. This is just an illustrative example; for real-world use, the case-insensitive strings are recommended, as the case-sensitive strings can eliminate many potential matches. It should be noted that we are not allowing any whitespace characters between key names and values as specified in the original rule from which we took inspiration. Figure 15 shows that the MYARA is generating more complex atoms with classes. The results are, however, very similar: both versions took a little bit over 10 seconds. The possible explanation is that the data set includes a small percentage of valid INI file data. Hence, the set of potential false-positives in the YARA version caused by shorter atoms is not that significant.

D. TESTING WITH ReversingLabs RULESETS

The public repository of ReversingLabs company [19] provides an ideal set of YARA rules for testing. The ruleset is a practical example of rules actually used for malware detection. The comparison of the proposed new implementation with the original version will provide more information about the impact on performance. It contains 139 files and 371 rules. These rules contain 896 hexadecimal strings and 2,876 text strings. None of these rules contains regular expressions. This, however, does not mean the set is not suitable for testing. The variety of strings can vary based on searched malware and the results are even more interesting because these strings are not targeted with proposed changes. The previous ruleset (Section V-A) showed the impact on regular expressions, and this set will provide more information on other types of string available.

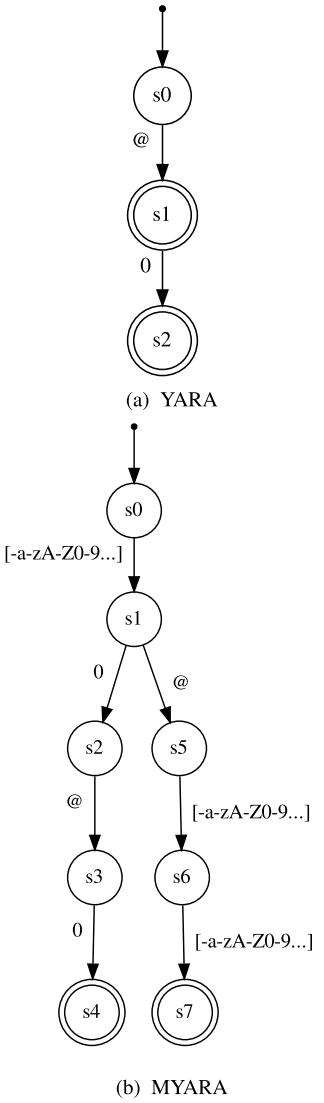


FIGURE 12. Aho-Corasick machine for e-mail addresses version 1 and 2.

The results are presented in Table 4. The time differences are on the border of desirability. The average of minimal values necessary for scanning all data is, in both cases, almost identical. The ruleset was divided into two sets: the first set contained files for which the YARA variant was faster, the second was the opposite. The average difference was calculated, as well as the overall time. There was also almost no difference and there is no single file that would yield significantly different numbers for these two versions. Our conclusion is that both versions behave comparably.

E. TESTING WITH AVAST RULESETS

One additional ruleset was tested with kind permission from Avast company. They allowed us the use a subset of their private repository that is used for malware detection in their systems. The selected ruleset contains 1,512 files with 13,026 rules, and it has around 6.59 MB. It contains 24,784 string definitions where 71.7% are text strings, 26.2% hexadecimal strings, and 2.1% regular expressions.

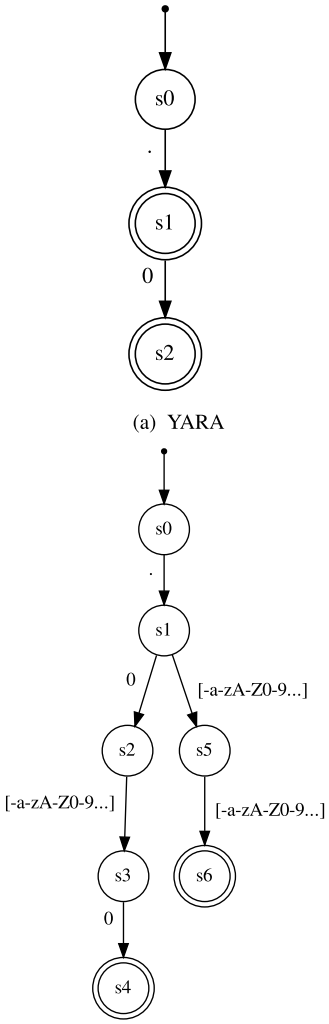


FIGURE 13. Aho-Corasick machine for e-mail addresses version 3.

```
rule ini_file_format
{
  strings:
    $re1 = /\[[a-z0-9]+\]\n/
    $re2 = /[a-z0-9]+=[a-z0-9]{2,}\n/
  condition:
    $re1 or $re2
}
```

FIGURE 14. The YARA rule for the detection of INI file format.

TABLE 4. ReversingLabs rulesets show just a small difference between version YARA and MYARA. ReversingLabs rulesets were divided into 3 sets: the first set contains all files where YARA was faster, the second where it was slower, and the third contains all files together.

Set	YARA [s]	MYARA [s]	MYARA/YARA
Set 1 (75 files)	824.324	829.233	1.006
Set 2 (64 files)	679.329	675.666	0.995
All files (139 files)	1503.653	1504.898	1.001

Tests were run on the data set presented in this section. YARA needed 21,831.62 seconds to scan all files, MYARA 21,635.49 seconds. Even with a low percentage of regular expressions that are the main targeted types of string, the proposed version was faster, at about 1% of total time.

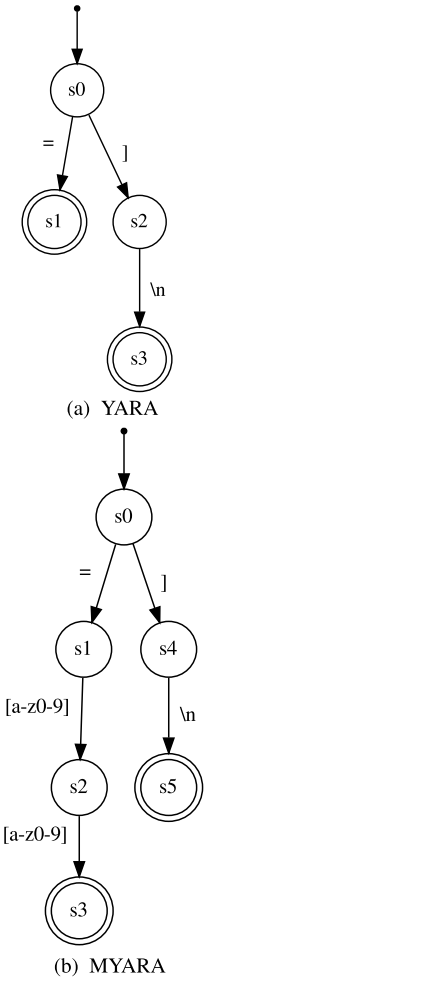


FIGURE 15. Aho-Corasick machine for INI file format.

As we saw in this ruleset and ReversingLabs company's ruleset, the regular expressions are used sparingly. In the original version, they are often raising the slowing down warning, and they are challenging to use for that reason. Because the proposed changes target the regular expressions, they do not improve the speed of rulesets specialized for the current version. However, the way we proved with these tests, they do not slow it down either.

The authors also believe that the proposed version can increase the numbers of regular expressions in YARA in the long run because analysts will have more options on how to define strings without worrying about scanning slowing down warnings.

VI. CONCLUSION

YARA is a well-known tool in malware detection and it becomes a vital part of threat intelligence infrastructure in many companies worldwide. We set the goal to explore this tool's power, mainly focusing on regular expressions, as we found them fundamental for description and classification purposes.

The tool is already using several methods and heuristics to speed up the scanning, with a few of them mentioned in this paper. The regular expressions, particularly these with a set of the classes are, on the other side, handled as ineffective, and they are overlooked in the matching process.

We introduced a different approach. We proposed a new version of the Aho-Corasick algorithm that improves the scanning speed when using regular expressions with classes and metacharacters. In selected cases, this version was about 27% faster than the original version. We proved that the selection of so-called atoms, the initial substring for the Aho-Corasick machine, can dramatically influence the scanning speed. With improved Aho-Corasick machine and interleaved state-transition matrix that is able to store multiple characters per state, the tests showed an increased speed within targeting patterns while maintaining the pace on the example of real rules used in malware analysis. These changes in YARA implementation will create new possibilities for defining strings in YARA rules and eliminating the warnings about scanning slowing down.

ACKNOWLEDGMENT

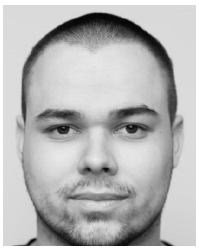
Dominika Regéciová would like to thank Avast Software s.r.o. and Jakub Křoustek for supporting the work.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Pressn, 1989.
[2] D. P. F. Bilstein, "YARA-signator: Automated generation of code-based YARA rules," *J. Cybercrime Digit. Invest.*, vol. 5, no. 1, pp. 1–13, 2019.
[3] N. Naik, P. Jenkins, R. Cooke, J. Gillett, and Y. Jin, "Evaluating automatically generated YARA rules and enhancing their effectiveness," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Dec. 2020, pp. 1146–1153.
[4] E. Raff, R. Zak, G. L. Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt, "Automatic YARA rule generation using biclustering," in *Proc. 13th ACM Workshop Artif. Intell. Secur.*, Nov. 2020, pp. 71–82.
[5] T. H. Dadi, E. Siragusa, V. C. Piro, A. Andrusch, E. Seiler, B. Y. Renard, and K. Reinert, "DREAM-YARA: An exact read mapper for very large databases with short update time," *Bioinformatics*, vol. 34, no. 17, pp. i766–i772, Sep. 2018.
[6] S. G. Singapura, Y.-H. E. Yang, A. Panangadan, T. Nemeth, P. Ng, and V. K. Prasanna, "FPGA based accelerator for pattern matching in YARA framework," in *Proc. Int. Symp. Appl. Reconfigurable Comput.*, 2016, pp. 1–11.
[7] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.
[8] *Corasick*. Accessed: Jul. 21, 2020. [Online]. Available: https://upclosed.com/people/margaret-j-corasick/
[9] M. Norton. (2004). *Optimizing Pattern Matching for Intrusion Detection*. System. [Online]. Available: https://bit.ly/2ZTWLP2
[10] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2628–2639.
[11] M. Sandberg. *ACISM: Aho-Corasick Interleaved State-Transition Matrix*. Accessed: Aug. 7, 2020. [Online]. Available: http://goo.gl/lE6zG
[12] *YARA Github Repository*. Accessed: Aug. 7, 2020. [Online]. Available: https://virustotal.github.io/yara/
[13] *YARA is a Tool Aimed at Identify and Classify Malware Samples—Interview With Creator Victor M. Alvarez*. Accessed: Aug. 7, 2020. [Online]. Available: https://pentestmag.com/yara-is-a-tool-aimed-at-identify-and-classify-malware-samples-interview-with-creator-victor-m-alvarez/
[14] *Malpedia*. Accessed: Aug. 7, 2020. [Online]. Available: https://malpedia.caad.fkie.fraunhofer.de/
[15] *VirusTotal Hunting*. Accessed: Aug. 7, 2020. [Online]. Available: https://www.virustotal.com/gui/hunting-overview
[16] J. D. Murray and W. V. Ryper, *Encyclopedia of Graphics File Formats*, 2nd ed. Sebastopol, CA, USA: O'Reilly & Associates, 1996.
[17] *YARA Documentation*. Accessed: Aug. 7, 2020. [Online]. Available: https://yara.readthedocs.io/en/v3.10.0/
[18] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed. Sebastopol, CA, USA: O'Reilly, 2006.
[19] *Reversinglabs: Public Reversinglabs-YARA-Rules Repository on GitHub*. Accessed: Sep. 27, 2020. [Online]. Available: https://github.com/reversinglabs/reversinglabs-yara-rules/tree/%5581f4a04e8c3cc6304db5e215cb1fc48a00ceb8
[20] A. M. Antonopoulos, *Mastering Bitcoin*. Newton, MA, USA: O'Reilly Media, 2014.
[21] B. Allbee, *Hands-on Software Engineering With Python*. Birmingham, U.K.: Packt Publishing, 2018.

DOMINIKA REGÉCIOVÁ was born in Vyškov, Czech Republic, in 1992. She received the bachelor's degree in information technology and the master's degree in information technology security from the Faculty of Information Technology, Brno University of Technology, in 2016 and 2018, respectively. Since 2018, she has been a Ph.D. Student and a member of the Formal Model Research Group, Faculty of Information Technology, Brno University of Technology. Her research interests include formal models and compilers, and their use in computer security.

DUŠAN KOLÁŘ received the Ph.D. degree from the Brno University of Technology (BUT) devoted his strengths to teaching courses and research dealing with formal languages and automata, compilers, database systems, and reverse engineering. Finally, he has extended his focus on security and safety issues in a broader way applying additional areas of computer science. Moreover, he has participated in several commercial projects involving development of tools offering rapid application development.



MAREK MILKOVIČ received the master's degree in information technology security from the Faculty of Information Technology, Brno University of Technology, in 2017. He is currently working with Avast Software s.r.o. as a Lead Software Engineer, where he leads a team building threat intelligence systems. He is interested in compilers and their practical design, reverse engineering, and container technologies.