GenRex Demonstration: Level Up Your Regex Game

Dominika Regéciová
Gen Digital

Abstract

GenRex is a unique tool for detecting similarities in artifacts from executable files and the generation of regular expressions.

This paper demonstrates how to use GenRex to maximize the usage of regular expressions automatically created from behavioral reports and other potential use cases.

GenRex is open-sourced, and additional resources, such as a dataset of behavioral reports and an extension to the YARA tool, are provided.

Keywords: GenRex, malware detection, pattern matching, regular expressions, YARA.

1 Introduction

GenRex was first introduced in the 2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) in the paper *GenRex: Leveraging Regular Expressions for Dynamic Malware Detection* [1]. The paper presented a deep dive into the internals of this tool, as well as the results of experiments, which demonstrated that GenRex is a relatable instrument for regular expression generation.

At Botconf, the main focus is to show how GenRex can be used in practice by malware analysts and by others who have a long list of strings with some level of similarities, such as named objects created by malware.

2 GenRex

GenRex is available online on the GitHub page: `https://github.com/avast/genrex`, where the instructions for installation and usage can be found.

GenRex is a Python library that can be used, as shown in Figure 1.

```
import genrex

results = genrex.generate(
  input_type=genrex.InputType.MUTEX,
  source={
    "source1": [
      "aabcmalware7992",
      "adeemalware3022",
      "aefdmalware1896"],
    "source2": [
      "bfbcmalware5996",
      "bbcamalware4508"],
  })

print("Results:")
for result in results:
  print(result)
```

Figure 1: GenRex example

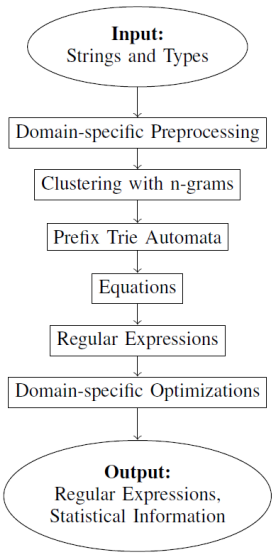

Figure 2: Model architecture for GenRex

The input processing for the results will be briefly explained, split into several logical steps, as illustrated in Figure 2.

This will help the users understand how the results were created, which could help them to evaluate the quality of the results.

The input is defined as a list of inputs and their resources and their type (mutex, register, etc.). The reason is that both pieces of information are influencing the evaluation.

Domain-specific preprocessing is the first step, where stings are converted to a more general format that would be more usable in YARA rules and other use cases. The usernames should be removed, for example, as they are too specific. Strings containing only GUID are also not suitable, as they are similar to hashes - unique to each sample.

The clustering phase emphasizes speed rather than precision. The similarity based on n-grams was used. The length of n-grams is calculated from input, and based on created clusters, we obtain statistical information that can be used for detection. In the presented example, one cluster containing all input strings is created, detecting the common middle part of the mutexes.

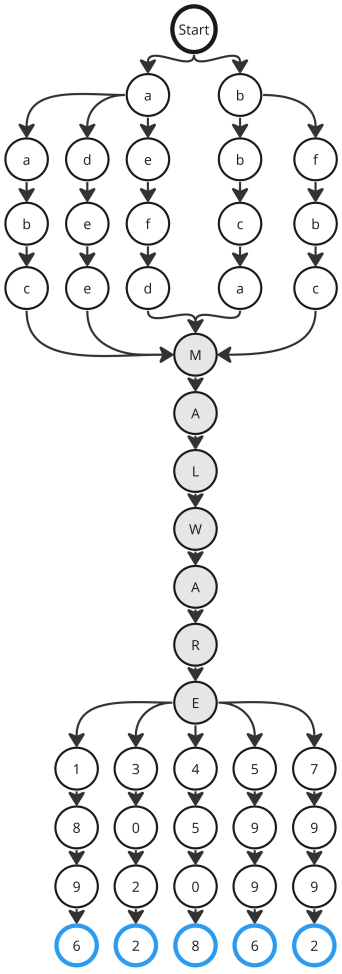

Figure 3: The prefix tree

For each cluster, a set of similar strings, a prefix trie is created. To detect similar subparts of these strings, the n-gram that leads to the cluster creation is used. Then, the rest of the trie is made to detect the common parts not only for prefixes (as a character a on the picture) but also parts in the middle of strings. The trie is then minimized, improving the detection of common parts even further, as demonstrated in Figure 3.

Then, two matrices are created – one for each trie state and the second for the final states. With the simplified Brzozowski algebraic method, the trie is transformed into a set of equations. By solving them, an internal representation of regular expressions is created using two operations – concatenation and union.

```
for n = number_of_states decreasing to 1:
  for i = 1 to n:
  B[i] += A[i,n] . B[n]
    for j = 1 to n:
      A[i,j] += A[i,n] . A[n,j]
```

Figure 4: Brzozowski algebraic method

The last step is domain-specific optimization heuristics, which ties everything together. The aim is to create readable, effective, and usable results.

An example of the result is shown here. The common part was detected, as well as variable prefixes and suffixes. The result was generalized based on the nature of the input. These results, with statistical information from clusters, can be used for automatic YARA rules generation and many other tasks. More details about the algorithm and each step can be found in the previously mentioned paper [1].

```
Results:
Regex: (^|\\)[0-9a-f]{4}malware[0-9]{4}$
Ngram: malware
Unique: 5
Min: 2
Max: 3
Average: 2.5
Resources: ['aabcmalware7992',
'adeemalware3022', 'aefdmalware1896',
'bbcamalware4508', 'bfbcmalware5996']
Originals: []
Named object type: mutex
Hashes: ['source1', 'source2']
```

Figure 5: The resulting regular expression

3 Demonstration

The main focus will be a practical demonstration of the GenRex on an experimental dataset of behavioral reports. Publicly available resources will be introduced, as well as illustrations of their usage.

3.1 YARA

YARA[1] is a tool for pattern detection, widely used in threat intelligence.

In so-called YARA rules, malware can be described based on both static characteristics and its behavior. The dynamic characteristic is evaluated based

[1] github.com/VirusTotal/yara

on parsing reports from sandboxes and emulators in JSON format.

This is done through modules—extensions of the core YARA code that can be easily modified and expanded for the specific needs of users. In this paper, the Cuckoo module will be discussed, which accepts reports in JSON format from the Cuckoo sandbox, a commonly used sandbox for malware detection[2]. The example in Figure 6 shows a simple YARA rule named *malware_mutex*:

```
import "cuckoo"

rule malware_mutex
{
 condition:
 cuckoo.sync.mutex(/(^|\\)EvilMutex[0-9]$/)
}
```

Figure 6: An example of a rule in YARA.

This rule contains a function for the Cuckoo module that scans the Cuckoo reports to find all samples that open or create a mutex matching the provided regular expression.

The prefix *(^/\\)* and suffix $ allow us to detect named objects in a more general form with prefixes.

YARA will accept Cuckoo reports as an argument, the rule, and the sample in format:
yara -x cuckoo=behavior_report_file rules_file sample_file.

The sample itself scans the static part of the rules (called the strings section, missing in our case), while behavioral conditions are searched in the report. In both cases, we scan the data without actually running the samples.

3.2 Extended Cuckoo Module

```
import "cuckoo"

rule test_cuckoo
{
 condition:
 cuckoo.genrex.api_call(/GetProvider/) >= 3 or
 cuckoo.genrex.atom(/rOBDoI/) >= 3 or
 cuckoo.filesystem.file_access(/(^|\\)xyc/) or
 cuckoo.registry.key_access(/(^|\\)AppXYZ/) or
 cuckoo.sync.mutex(/kzyyjqyi/) >= 1 or
 cuckoo.genrex.resolved_api(/xx.dll/) >= 3 or
 cuckoo.genrex.semaphore(/LJpEx8rffiNY/) >= 2
}
```

Figure 7: An example of YARA rules created based on the new version of the Cuckoo module.

The extended version of the Cuckoo module will be used in the following examples. The code is available online, and contains additional methods and allows testing of how many times the expression was matched in the report.

An example of a rule is shown in Figure 6. New methods are named Cuckoo.genrex.named_object, but this naming can be changed based on the users' preferences.

3.3 Dataset

For a demonstration of YARA rules generation, a publicly available dataset created by Avast Software and Czech Technical University was used [2].

The dataset contains behavioral reports from the CAPEv2 sandbox, one report for each sample. CAPEv2[3], or *Config And Payload Extraction*, is a malware sandbox derived from Cuckoo. YARA can work with CAPEv2 reports, as it does with Cuckoo reports. From malware, there are ten families of trojans, worms, spyware, and bots representing various malware.

For the newer version, the labeling was updated, cleanware samples were added, and reports to the newer version of CAPEv2 were re-created, and it is also available online [3].

To filter so-called clean strings, strings that were created by the setup of the sandbox, a list of clean strings from the newer version of the dataset was also published.

3.4 YARA Rules Generation With GenRex

The GenRex results can be used in automatic YARA rules creation or updates. To demonstrate the capabilities of GenRex, a selected number of named objects, such as mutexes, API calls, and resolved APIs were used.

For each malware family from the dataset, a YARA rule was created, and then the precision of them was evaluated. The dataset was split for this purpose in half – the first half was for creating YARA rules, and the second was for matching.

To create YARA rules for each malware family from the pre-known dataset, the following process was used:

1. Select 100 samples that have SSDeep hash similarity less than 50 as the input set
2. Create an empty YARA rule
3. Repeat until you cover all samples with the pre-known dataset, or you can not extend the YARA rule any further:
   (a) Filter clean strings from the input set
   (b) Generate regular expressions for each string category from the input set with GenRex

[2] yara.readthedocs.io/en/stable/modules/cuckoo.html

[3] github.com/kevoreilly/CAPEv2

(c) Add regular expressions that cover most samples from the input set and do not generate false positives in the pre-known dataset to the YARA rule

(d) Select the samples that did not match the YARA rule from the pre-known dataset to the input set

The created rules with additional information can be found online[4].

3.5 Evaluation

To evaluate how precise the created YARA rules are, they were run over the dataset to calculate the number of correctly classified malware samples, as well as incorrectly classified samples. The second case did not consider whether the YARA rule matched the clean sample or just a different malware family. The results can be shown in Table 1.

Table 1: The results of malware families' detections. The table shows the true positives and true positive ratio, the number of correctly matched samples, and false positives and false positive ratio, meaning the rule wrongly matches a sample outside of the family class.

Family	TP	TPR [%]	FP	FPR [%]
Adload	665	94.33	0	0
Emotet	13,939	96.63	0	0
HarHar	655	100	0	0
Lokibot	3,700	88.28	2	0.004
njRAT	2,460	73.04	2	0.003
Qakbot	4,857	99.00	0	0
Swisyn	12,571	99.83	1	0.002
Trickbot	3,166	75.33	1	0.002
Ursnif	767	57.41	0	0
Zeus	2,444	94.22	1	0.001
All families	**45,224**	**92.34**	**7**	**0.01**

The generation of the YARA rules was successful, with high true positives and low false positives. With the help of a generation algorithm for YARA rules, GenRex generated good-quality regular expressions that led to precise classification.

4 Additional Tips and Tricks

GenRex is relatively easy to use, but there are some issues you can encounter while working with this tool. The following section will provide tips and tricks to help you achieve the best results possible.

4.1 Too Short Strings

The first problem you can face is that GenRex is not returning any results at all. Do not panic, as this is completely good behavior. The goal is not to create regular expressions for all costs. One of the reasons the input is not generating regular expressions could be that the input strings need to be longer. The minimum string length for GenRex is four characters; shorter than that, GenRex filters out. While the input *["abc", "abd", "abz", "aby", "abc"]* will not give you any result, the input *["prefix-abc", "prefix-abd", "prefix-aby", "prefix-abz"]* will result into the regular expression *(^|\\)prefix-ab[a-z]$*.

```
import genrex

results = genrex.generate(
  input_type=genrex.InputType.MUTEX,
  source={
    "hash1": [
        "abc",
        "abd",
        "abz",
        "aby",
        "abc",
    ],
  })

print("Results:")
for result in results:
  print(result)

# This will not generate any results
```

Figure 8: GenRex example: too short strings

```
import genrex

results = genrex.generate(
  input_type=genrex.InputType.MUTEX,
  source={
    "hash1": [
        "prefix-abc",
        "prefix-abd",
        "prefix-abz",
        "prefix-aby",
        "prefix-abc",
    ],
  })

print("Results:")
for result in results:
  print(result)

# This example will generate the result
(^|\\)prefix-ab[a-z]$
```

Figure 9: GenRex example: added prefix

4.2 Why Notate Sources?

Many users in the past had questioned why we need to have differentiation for the sources of the strings. You can list all your input strings into one set, but if you are working with more sources (typically from multiple samples), it can be handy to use this information and help GenRex create better results.

For example, based on the input from Figure 1, if you have an additional string *"hello234"* in set *["aabcmalware7992", "adeemalware3022","aefdmalware1896", "bfbcmalware5996",*

[4] github.com/regeciovad/GenRex-demo/

"bbcamalware4508", "hello234"], the solely appearance will not be enough to produce result by itself. However, if you note that this string is in more than one source, the importance of this string will be higher, and it will produce two regular expressions: *(^|\\)[0-9a-f]4malware[0-9]4$* and *(^|\\)hello234$*.

```
import genrex

results = genrex.generate(
  input_type=genrex.InputType.MUTEX,
  source={
    "hash1": [
      "aabcmalware7992","adeemalware3022",
      "aefdmalware1896","bfbcmalware5996",
      "bbcamalware4508","hello234"],
  })

print("Results:")
for result in results:
  print(result)

# This example will generate only the result
(^|\\)[0-9a-f]{4}malware[0-9]{4}$
```

Figure 10: GenRex example: only one source

```
import genrex

results = genrex.generate(
  input_type=genrex.InputType.MUTEX,
  source={
    "hash1": [
      "aabcmalware7992","adeemalware3022",
      "aefdmalware1896", "hello234"],
    "hash2": [
      "bfbcmalware5996", "bbcamalware4508",
      "hello234"],
  })

print("Results:")
for result in results:
  print(result)

# This example will generate both results
(^|\\)[0-9a-f]{4}malware[0-9]{4}$
and (^|\\)hello234$
```

Figure 11: GenRex example: two sources

4.3 Test Your Rules

And finally, we advise checking the results before putting them into the rules. No system is perfect, no results are absolute, and it is always a good idea to have a system of checks against false positives, mainly on your cleanset.

5 Conclusion

In this paper, we demonstrated the practical use case of GenRex. With CAPEv2 behavioral reports as input, we created a set of YARA rules and tested the correctness and precision of the rules.

Based on an archived high true positive rate of 92.34% and a low false positive rate of 0.01%, we demonstrated that the tool could reliably describe the nature of named objects for the given malware family. We also provide access to the project and additional resources for easier use.

References

[1] D. Regéciová and D. Kolář, "GenRex: Leveraging Regular Expressions for Dynamic Malware Detection," *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2023. DOI: 10.1109/TrustCom60117.2023.00123.

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

[3] "CAPEv2 dataset v2." `github.com/regeciovad/avast-ctu-cape-dataset/tree/reports_min`.

Author details

Dominika Regéciová

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

Dominika Regéciová received a bachelor's degree in information technology and a master's degree in information technology security from Brno University of Technology, Faculty of Information Technology, in 2016 and 2018. Since 2018, she has been a Ph.D. student and a member of the Formal Model Research Group at Brno University of Technology, Faculty of Information Technology. Her research includes formal models and compilers and their use in computer security. She currently works as a Senior Researcher at Gen Digital.