universitätbonn

Improving YARA-Signator for effective Generation of code-based YARA-Signatures

Felix Bilstein

Lab Report

Examiners: Prof. Dr. Peter Martini
Advisor: Daniel Plohmann
Institute for Computer Science IV
Work group Communication Systems
University Bonn

Statement of Authorship

I hereby confirm that the work presented in this lab report has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

Bonn, August 27, 2019

Felix Bilstein

Abstract

The classification of malware is typically a time-consuming, often manually executed procedure which requires expert knowledge and experience leading to an expensive process. Attacks using malware, be it for monetary gain or even driven by nation-state actors, have become a reality and reached volumes that require significant automation for their handling. The malware classification tool *YARA* [IBM+18] has evolved as an industry-wide de facto standard. We previously created the framework *YARA-Signator* [Bil18b], which is capable of generating code-based YARA signatures in a fully automated fashion.

In this work, we will improve the proof of concept of our previous work as documented in "Automatic generation of code-based YARA-signatures" [Bil18a] and the tool YARA-Signator by developing a more mature algorithm for YARA rule generation. Apart from improvements to the overall codebase to achieve better scalability, we were able to increase the quality of the automatically generated YARA signatures by applying our new, iterative approach taking advantage of the result feedback provided by YARA to our framework by testing the generated YARA rules against the malware corpus. The f-score of the YARA signatures of our previous approach was 0.86 and using the new methodology the f-score could be increased to 0.98. The false negatives could be reduced by 90% from 820 to 84. The new version of YARA-Signator was able to reduce the false positives from 139 to 83.

Contents

1 Introduction **1**

2 Related work **2**

3 System Analysis and Improvement **3**
3.1 Prior Work and Current Status . . . 3
3.2 Analysis & Proposed Improvements . . . 4
3.2.1 Refactoring YARA-Signator . . . 5
3.2.2 Wildcarding of Instructions . . . 6
3.2.3 Iterative Approach . . . 7

4 Evaluation **9**
4.1 System properties . . . 9
4.2 Previous statistical evaluation . . . 9
4.3 New statistical evaluation . . . 10

5 Summary **12**
5.1 Future work . . . 12

6 Bibliography **14**

List of Tables

1 Results of YARA-Signator version 1 using the Malpedia repository from the Bachelor Thesis. . . . 10
2 Results of YARA-Signator version 2 evaluated against the current Malpedia version (commit ”24837c“, date ”02.07.2019“, 1320 families), testing the performance of different wildcarding versions and the new iteration step. . . . 10

1 Introduction

Today, systems are attacked by different types of threat actors on a daily base. There exist plenty of infected systems, compromised by *malicious software* (malware). This malware is usually packed, obfuscated, or protected to prevent static analysis by researchers.

Manual unpacking by analysts requires expert knowledge and is typically very time-consuming. Modern approaches of unpacking malware can approximate the results of malware analysts via *Run&Dump* practices, i.e. via sandboxing techniques. For further investigations the dumped samples need to be classified for advanced post-processing. To date, YARA signatures are manually or rather tool-assisted created. Current solutions of YARA rule generation support at most partially automated derivation of rules based on strings. Our concept targets the code of the malware or more concrete, instructions recovered by a disassembler which are merged into YARA signatures.

We focus on generating rules for the curated malware repository *Malpedia* [PE18], offering over 1000 malware families and more than 2500 malware samples of already categorized and labeled data. Although we work with Malpedia the approach is generally valid for every pre-categorized malware repository.

In our prior work we introduced *YARA-Signator*, a framework for automatic generation of code-based YARA signatures. The framework was able to generate high-quality rules for a certain subset of malware samples provided by Malpedia focusing on the Microsoft Windows platform. It was possible to generate YARA rules for over 600 malware families and we showed that it is possible to generate YARA signatures automatically on the basis of a complete malware corpus.

In this work, we present an improved approach of automatically generating YARA signatures using different ranking algorithms and filtering steps as well as a new, iterative approach to generate high-quality YARA rules effectively.

We successfully increased the code quality and maturity of the complete framework. The project is highly configurable via a new configuration files format. Additionally we provide interfaces for new plugins by exposing core functionalities such as the ranking system or the iterative operator factory. We improved our current approach and the framework for automatic YARA rule generation. The quality of the automatically generated YARA signatures was increased significantly to over 97% recall and 97% precision at the same time.

2 Related work

In this section various projects and related work will be described and discussed. There exist a wide range of projects which are capable of generating signatures or help malware analysts in their manual YARA rule creation process.

The project *yarGen* [Rot18] is a framework for automatic generation of YARA signatures by Florian Roth. Following a string-based generation approach, the project is able to build YARA signatures based on the strings detected in malware samples. Those strings are checked against a goodware-database to ensure that only strings are used for the rules if they do not appear in legitimate software to avoid false positive detections. *YaBin* [Dom18] is a project generating YARA rules focusing on executable code instead of strings. The project uses a goodware database to distinguish between legitimate executable code and malicious code within provided samples.

The following projects are tools which help analysts in their manual creation process. While they do not cover our approach directly these tools help to understand the analyst's toolchain and workflow which we can adapt in our framework.

Hyara [Yi18] is an IDA Pro plugin to help during binary analysis of malware by providing a graphical interface for interactive YARA rule creation by using strings, instructions and other characteristics. Furthermore the software provides basic automation of YARA rule creation by deriving a YARA signature from previously selected strings or instructions.The script *pat2yara* [Sec13] converts FLAIR pattern based PAT signatures from IDA Pro into YARA signatures. *yara-fn* [Bal16] is a Python script for automatic YARA rule generation based on instruction sequences within basic blocks and the tool is capable to mask addresses. Addresses containing references or jumps can be masked to provide more abstract YARA rules targeting families and samples without absolute addresses which might change every execution or compilation.

Christian Blichmann developed a framework for automatic generation of signatures for malware classification in his Diploma Thesis [Bli08]. The approach described in his thesis focuses on dumped malware, generates disassembly of the memory dumps, analyzes and classifies it. The generated signatures support ClamAV [Tea04]. The project *bass* [Gro17] is an open source project by Cisco Talos Intelligence Group based on the algorithm by Blichmann as described in his thesis [Bli08]. Based on his prior work the tool is capable of automated generation of ClamAV rules via a framework of different docker containers. Currently the project is considered alpha and therefore unstable. Another tool, *vxsig*, is internally developed since 2011 at Google and publicly available since 2019 [Inc19]. It is based on the approach by Blichmann, too. The project is developed in C++ and is considered stable for productive use [Inc19].

3 System Analysis and Improvement

In this section we provide a summary of the state of the prior work as well as improvement strategies for evolving the approach. The prior framework (in the following referred to as version 1) provided a processing chain for automatic generation of YARA signatures. In section 3.1 we give an overview of this implementation focusing on the relevant details, section 3.2 provides an analysis of its shortcomings as well as three concrete ideas of optimization that have been executed to improve YARA-Signator.

3.1 Prior Work and Current Status

In this section we discuss the concrete implementation improvements after a short summary of the prior work and approach. Figure 1 shows the different sequential steps of the prior implementation:

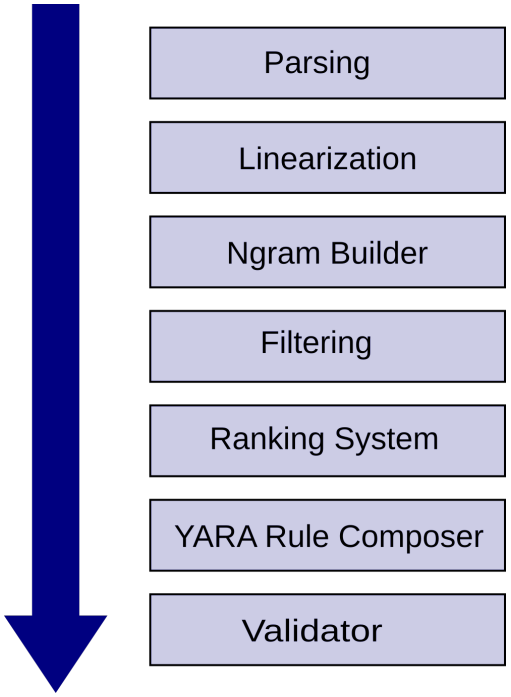

Figure 1: *Iterated modules of the processing chain.*

The prior implementation is a sequential procedure of several steps. First, the provided data is parsed and converted into linearized disassembly. The data provided by SMDA contains originally a complex structure offering the full control flow of the disassembled software and various meta information. This data is transformed into sequences of linear disassembly since YARA identifies strings by searching

sequentially through binary files. On the base of this output data, ngrams are derived for the further processing. Ngrams are consecutive subsequences of elements with a fixed length of *N*. The ngrams are the smallest entity YARA-Signator operates with. All ngrams are further handled by the remaining steps within the processing chain.

The Filtering step applies different filters to all ngrams previously generated. Reducing the data by applying filters to the ngrams is an essential step of the signature generation. We remove all ngrams which occur in more than one family and store the remaining ngrams in a database. On the base of this output we rank the remaining entries by counting how many samples of a malware family are covered by the respective ngram. This ranking step is important to be sure that only the best candidates are chosen for a YARA signature. It is also possible to add certain ranking parameters to the YARA rule generation framework, i.e. to punish potential padding bytes like "CC" or "00".

The YARA rule composer builds the resulting YARA signature. The ngrams are converted into a detection rule which can be further processed using YARA. The full implementation details of this approach can be found in the report of the previous work [Bil18a].

3.2 Analysis & Proposed Improvements

The prior YARA rule generation system provides YARA rules which cover most (534 out of 636 families, 84%) of the malware families within Malpedia precisely. However, besides false positives, there are still a significant number of false negatives which we targeted with a more powerful generation approach. We address three different aspects of *YARA-Signator* with several improvements. To improve our YARA rule generation framework we want to cover the complete Malpedia database including malware targeting the platforms Mac OS and Linux to increase the amount of resulting YARA signatures. In the following paragraphs we discuss the chosen improvements of our prior work.

Refactoring While the prior approach offers already precise rules for the Malpedia corpus there are still several aspects which can be improved. The generated rules still suffer from a high number of false negatives and many false positives [Bil18a]. We refactored the source code of *YARA-Signator* to advance the prototype into a productive software project. We want to improve several aspects of the prior software that was considered a proof-of-concept state. The current stage of development has been enhanced to provide a first release candidate. The database backend changed from MongoDB to PostgreSQL in expectation of performance improvements as well as various Quality of Life enhancements we describe in section 3.2.1.

Wildcarding To generate more abstract YARA rules covering a broader set of samples of a family we improve the wildcard operator which can mask certain intructions or references. By masking addresses in code sections the algorithm is able to generate YARA signatures which are position-independent. This is important to deal with effects like *Address Space Layout Randomization* (ASLR) or relocations when the malware is executed. Turning addresses into wildcards as shown in section 3.2.2 will allow to cluster many different ngrams by the implied generalization of code. This should lead to a smaller number of unique ngrams within malware families which might result in more precise ngrams that can be used for the generation of more effective detection signatures.

Iterative Approach For more precise rules covering the remaining false negatives we implemented a new, iterative approach to rebuild rules which do not perform well. The current algorithm contains the step *Validator* which did not contribute to creation of YARA signatures but was used for evaluating the performance of the rules created. The new concept includes the development of an iterative YARA rule

generation process, rebuilding underperforming rules and therefore modifying the current approach by including the information given by the validation step. The prior approach supports one sequential procedure to generate YARA rules. This can be a problem when the validation step 1 detects outliers or imprecise rules. To fix these "broken" rules we designed a new, iteratively YARA rule generation approach which is called *YARA-Generator Next-Generation* (YG-NG). The approach is discussed in section 3.2.3 in detail.

Figure 2 shows the enhanced approach. The first three steps of the procedure, *Parsing*, *Linearization* and *Ngram Builder* as well as the *Filtering* step and the *Ranking System* remain unchanged in the processing chain.

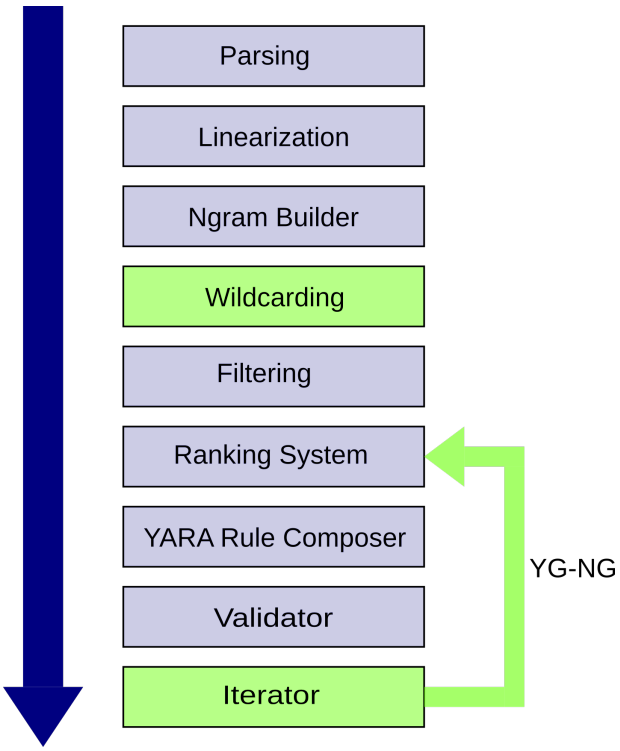

Figure 2: *Iterated modules of the processing chain. The green elements are new to our approach.*

This approach gives us the possibility to generate YARA rules iteratively by processing the information the Validator provides to the framework. After each iteration the validation step checks against false positives or false negatives. If such flaws are detected by the program, *YG-NG* tries to generate new YARA signatures based on the ngram database.

3.2.1 Refactoring YARA-Signator

The codebase of the YARA rule generation framework is improved in many features. The complete program is now controlled via a configuration file offering a better support of different setups without recompilation of the source code.

The database backend was initially implemented using MongoDB [Inc18b], a NoSQL (Not Only SQL) database management system. We decided to use MongoDB as our database system because of its ability to handle data in a document-based storage which allows faster development than traditional database management systems like i.e. relational database systems. Document-based databases allow storing data

structures without a previously designed database schema. Since we were able to reduce our schema into a rigid structure we changed the database system into a relational database management system. This allows us to take advantage of the features relational database management systems provide since our framework does not need the additional features and flexibility NoSQL databases offer. We changed the database backend from MongoDB to PostgreSQL to take advantage of various PostgreSQL features such as table partitioning. Figure 3 shows the high-level concept of table partitioning in PostgreSQL.

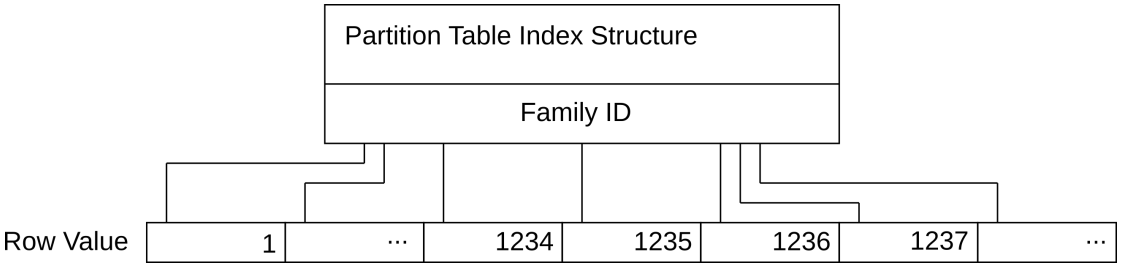

Figure 3: *Table partitioning in PostgreSQL used by YARA-Signator. The malware* family_id *field is partitioned for faster access to database tables of a certain malware family.*

Table Partitioning divides the data into several partitions to increase the access speed during database queries. To reduce the duration of database queries we partitioned the tables to build a single partition for each family as shown in Figure 3. A query requesting a certain family via the family_id field can be handled by the database by requesting a single partition table only instead of running a full table scan or querying through an index structure. The partition can outperform the index in a case like ours because an index needs to be loaded into memory to query it while the partition lookup table is an easier structure. PostgreSQL implements partition tables just like regular tables, allowing to query the correct table directly. Since we have around 880 malware families in the current Malpedia release we target with YARA-Signator, table partitioning reduces the search space from around 880 candidate tables to one.

3.2.2 Wildcarding of Instructions

In this section we present the wildcarding step of our processing chain mentioned in Figure 2. Wildcarding describes in our context the exchange of certain instructions or opcodes in ngrams into YARA wildcard characters. Wildcarding has the potential to give us multiple advantages. Overwriting addresses, pointers, and references with wildcard symbols might increase the quality of YARA signatures in the context of their abstractness. Many instructions in assembly language contain addresses or fixed relative addresses such as jump and call instructions. YARA signatures which contain sequences targeting values like (relative) addresses might perform well on our test data but not necessarily on real world data. Since absolute addresses might change every time when the malware is dumped or relative addresses might change when the malware is compiled, abstract rules are still able to cover samples with changed addresses.

There are several approaches for removing certain instructions from the ngram candidates. We created two new wildcard operators which mark certain instructions in ngram candidates. Wildcarding helps us to create more generic YARA signatures since the ngram candidates differ no longer due to different addresses in the listed in of instructions. Those candidates can be aggregated which shall reduce the candidate space.

```
rule example_wildcards {

[...]
$sequence_wildcarding_disabled = { 0fb706 8b5dfc 50 e8c1feffff }
    // n = 4, score = 4000
    //   0fb706          | movzx               eax, word ptr [esi]
    //   8b5dfc          | mov                 ebx, dword ptr [ebp - 4]
    //   50              | push                eax
    //   e8c1feffff      | call                0xffffffec6

$sequence_wildcarding_jumps = { 0fb706 8b5dfc 50 e8???????? }
    // n = 4, score = 4000
    //   0fb706          | movzx               eax, word ptr [esi]
    //   8b5dfc          | mov                 ebx, dword ptr [ebp - 4]
    //   50              | push                eax
    //   e8????????      | call                ????????
[...]

}
```

Figure 4: *Excerpt of YARA signatures with different wildcard operators. The first one is generated without wildcarding enabled, the second one with jump wildcarding enabled.*

The first wildcard operator marks the control flow instructions such as *jump* or *call*. These instructions operate with fixed addresses (absolute or relative) which might change very often due to address space randomization of operating systems or recompilation of the malicious software. This wildcard operator is called wildcard_jumps. The other wildcard operator masks data references such as pointers due to the same reason.

Figure 4 illustrates wildcarding in a concrete example. Ngrams containing e.g. jump conditions like e8c1feffff are masked using the YARA wildcard character ? to e8????????. We only mask full bytes and not nibbles because that would decrease the performance of YARA significantly. The wildcarding step is applied to the ngrams before aggregating them in the filtering step leading to a smaller candidate pool and . The details of how the wildcard operator improves the resulting YARA signatures are provided in section 4.

3.2.3 Iterative Approach

The iterative approach is a new feature to improve the generation of YARA signatures by taking advantage of the knowledge gained from the Validation step. Previously generated YARA signatures by our sequential algorithm contained N YARA sequences and a condition of the type “K of N sequences have to be detected” which has to be fulfilled that the YARA rule assigns a sample to the respective malware family.

We implemented two techniques to generate YARA signatures from the ngram candidate pool in our framework. The sequential approach derives ten YARA sequences into a YARA rule with a “7 of 10 sequences have to be detected” condition (the same condition we used in the Bachelor thesis) directly after applying the different ranking criteria by the ranking system. Our iterative approach, on the other hand, generates seven sequences for each sample within a malware family.

The iterative approach covers only the malware families which were generated previously by the sequential procedure but turned out to have an insufficient composition. Only the YARA rules which do not cover all samples of a family (false negatives) or which do cover samples from other families (false

positives) are recreated by the iterative approach. Although this approach improves the correct detection of many YARA signatures, false positives can still occur. If a sample has only very few instructions or a malware family has only few samples then it is not possible to filter for large pieces of shared code. This can be specially the case when dealing with 64-bit malware samples since most of the archived malware in Malpedia targets x86 32-bit architecture.

Figure 5 illustrates the selection process of the iterative approach. For each malware family that was detected by the Validator, new ngrams are derived from the database.

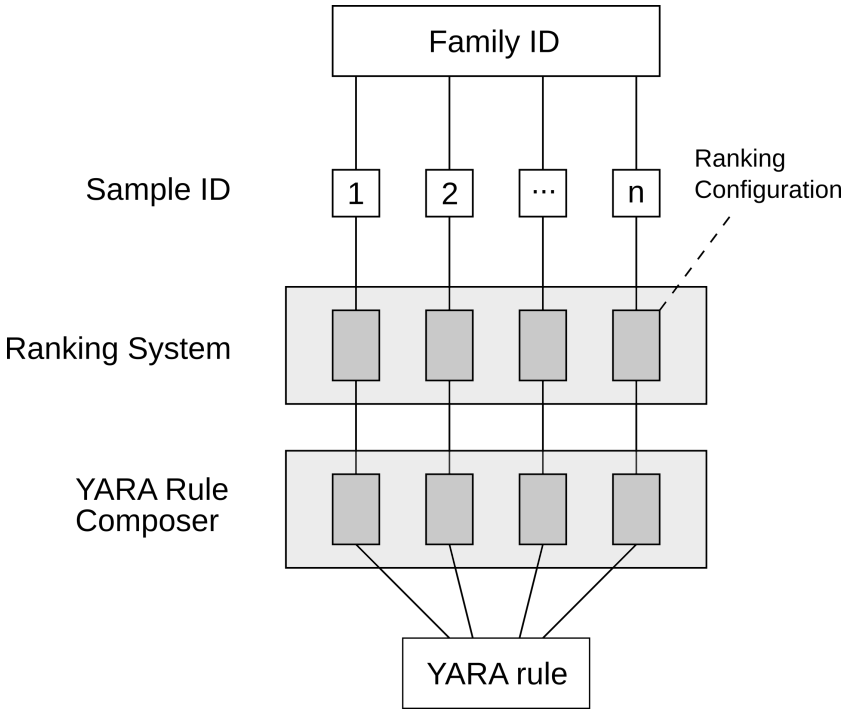

Figure 5: *The iterative approach to generate YARA signatures. The* family_id *is known from the Validation Step for every family.*

Every sample gets seven ngrams assigned from the database. If samples get already ngrams due to code sharing between samples, i.e. they have the same ngrams, they are counted together as an ngram for each of the samples to take advantage of shared code between them. All ngrams are rated by the ranking system with a previously selected ranking configuration. It is possible to select multiple configurations although we currently selected one default configuration only. Then, the resulting YARA sequences are merged into one YARA rule which shall cover the complete malware family without detecting samples from other families. To validate our new implementation against the prior one we evaluated the generated rules in section 4.

4 Evaluation

In this chapter the generated YARA signatures are evaluated against the YARA rules created using the previous approach as described in [Bil18a]. First, we present the system properties we used for our evaluation to provide reproducible results. They are followed by a comprehension of the YARA signatures which were generated by YARA-Signator version 1 and those by version 2.

4.1 System properties

To improve the reproducibility of our approach and the generated YARA signatures using YARA-Signator, we provide the specific versions used to build the framework. The following dependencies were used to build and run YARA-Signator 0.2a and the corresponding framework of java2yara, smda-reader and capstone_server:

- Apache Maven 3.6.1
- OpenJDK 11.0.3_p7
- GCC 8.3.0-r1 p1.1
- CMake 3.14.3
- PostgreSQL 11.3

As an operating system we used gentoo Linux [Inc18a], a source-based Linux distribution in a rolling release upgrade system without a fixed distribution version.

4.2 Previous statistical evaluation

Table 1 provides the best calculated statistical data for the YARA rules generated with the basic ranking system of the prior approach using the Malpedia snapshot Malpedia snapshot from June 2018 (commit “5e9ac34d”, date “Thu Jun 7 12:49:02 2018”). The malware families we processed were reduced to the Windows platform although the approach would have been able to build rules for different platforms such as MacOS and Linux. We focused on the f-score, precision and recall to evaluate how the generated rules would match the target malware sets. The precision of the YARA signatures provides information about the correct detection of malware families while the recall detects how the malware samples or families were covered by their YARA signature. The f-score on the other hand shows the combination of correct coverage of a malware family or a single sample and the completeness of the coverage by the YARA rule. Then we compared the evaluated data with a focus on three different criteria.

The ”samples“-criteria defines the coverage of all the samples in the initial data set. A sample covered by a rule counts as a true positive, while wrongly assigned samples count as false positives. False negatives are all the samples which are not covered by the signature for the certain malware family. On the other hand, the ”families“-criteria is a stricter criteria. Results are marked as false positives if at least one sample of the family is detected by a YARA signature of another family. If one sample of a

family is not covered by its YARA rule, then the family counts as false negative, and only if a family is completely covered by the YARA signature it is marked as a true positive. The YARA signatures were also tested against a newer Malpedia version (commit "806ffe82", date "Fri Oct 26 02:41:04 2018") to give an estimation about the coverage of samples which were not processed by YARA-Signator previously.

test data	false pos.	false neg.	true pos.	precision	recall	f-score
samples	1	219	1564	0.999	0.877	0.934
families	1	92	534	0.998	0.853	0.920
Malpedia	88	1001	2786	0.969	0.736	0.837

Table 1: *Results of YARA-Signator* version 1 *using the Malpedia repository from the Bachelor Thesis.*

The previous results show that we already achieved YARA signatures which are not only automatically generated but offer high quality in regard to precision and coverage. But the YARA rules still lack in coverage of certain malware families. To increase the recall we implemented a new algorithm to increase the performance of the generated YARA signatures in an iteration schema as described in section 3.2.3. The results of our new design are evaluated in the following section.

4.3 New statistical evaluation

In this section we present the results of our new, iterative approach. We compare both strategies: the sequential and the new, iterative approach. The iterative approach is also tested against different wildcarding filters to evaluate their performance. Table 2 shows the results of the different wildcarding filters and strategies. In addition to the system properties we mentioned in section 4.1 we used the following hardware and components during the generation of YARA signatures using the YARA-Signator framework:

- Gentoo Linux and Linux Kernel 4.19.37-gentoo
- Intel Core i7-3770 CPU @ 3.40GHz
- 32 GB RAM

YARA-Signator was run on hard disk drives in a Linux Software Raid 1 and the YARA rules were generated within 20 hours. We generated YARA signatures on the base of SMDA disassembly reports of the Malpedia repository from 18.06.2019 (commit 6fca97, 887 usable malware families) and evaluated against Malpedia from 02.07.2019 (commit 24837c, 1320 malware families).

strategy	sequential	iterative			
wildcarding	none	none	jumps	datarefs	both
FP	139	80	84	73	83
FN	820	87	85	86	84
TP	2886	3619	3618	3617	3619
precision	0.954	0.978	0.977	0.980	0.978
recall	0.779	0.977	0.977	0.977	0.977
f-Score	0.858	0.977	0.977	0.978	0.977

Table 2: *Results of YARA-Signator* version 2 *evaluated against the current Malpedia version (commit "24837c", date "02.07.2019", 1320 families), testing the performance of different wildcarding versions and the new iteration step.*

The generated YARA signatures by YARA-Signator version 2 show that the iterative approach outperforms our sequential algorithm. We increased the f-score from 0.86 to 0.98 and we were able to reduce the false negatives by 90% from 820 to 84. The false positives were reduced from 139 to 83.

While the iterative approach is more effective than the sequential, the various wildcarding filters did not impact the results significantly. It should still be noted that these results might vary if the test data where the YARA signatures were applied to malware which was not previously processed by our framework. The malware samples were dumped on a system with disabled ASLR. Address Space Layout Randomization (ASLR) is a technique of modern operating system to mitigate exploitation of vulnerable applications. ASLR randomizes the base addresses of the sections of the process which can break YARA signatures that contain addresses. Since ASLR is activated per default on every modern Windows version it might be possible that YARA signatures with wildcarded addresses perform better than YARA rules containing unmasked addresses.

5 Summary

In this work we presented several optimizations and improvements for automatic generation of code-based YARA signatures. The creation of YARA rules is typically a time-consuming task that requires the knowledge of experts and a lot of experience. There are several research projects covering the generation of YARA rules on the base of strings and tools helping researchers to generate YARA signatures semi-automatic. In this work we provide several improvements for our initial approach presented in the Bachelor Thesis "Automatic Generation of code-based YARA-Signatures" [Bil18a].

We generated YARA signatures for the platforms Linux and Mac OS using our new YARA-Signator version. To support YARA rules that are more abstract we implemented a new wildcarding filter interface to allow the usage of miscellaneous filters such as masking opcodes of pointers or references. We improved the core structure of the framework in general and added various features to build a more mature version of the project. The database backend is switched from MongoDB to PostgresQL to take advantage of various features from relational databases such as table partitioning. The configuration files offer more flexibility to be able to change the framework without recompiling it.

The most important improvement is the new approach to generate YARA signatures using an iterative instead of a sequential algorithm. The new approach increased the precision and recall of our resulting YARA signatures from an f-score of 0.86 to 0.98 points.

The new YARA signatures offer an increased quality than the ones which were generated using the previous approach. We improved the framework with regard to the usability, effectiveness and enhanced the YARA signatures. Although the rules offer a high quality there are still various ways to improve the YARA rules and the YARA-Signator framework.

5.1 Future work

The YARA-Signator framework is able to generate high quality YARA rules. We were able to improve our framework with regard to miscellaneous aspects. There are still several features of the framework which can be improved or extended.

The generated YARA signatures still detect samples as false positives and false negatives. An important improvement can be a goodware (legitimate software as opposed to malware) database as similarly used by the project "yarGen" by Florian Roth. The current approach of YARA-Signator could compare ngram candidates from the disassembly reports against ngrams from a recent database of goodware. This could help to detect candidates which might lead to false positives. YARA rules that contain ngrams of instructions that are used in common software or libraries can result in a higher increase of false positives. A very common example are shared libraries such as *openssl* or compiler-specific code such as *MSVCRT.dll* from Microsoft Visual Studio. A goodware database could be used by our framework to detect code pieces which belong to legitimate software or common libraries and reduce the remaining false positives.

Some of the false positives of the YARA signatures are the result of missing other samples for better filtering of shared code. Since Malpedia has only very few 64-bit malware samples and families, the ngrams containing 64-bit instructions look unique in the corpus but are common code which is just not available in the corpus. But this also means that an increased usage of 64-bit malware leading to more 64-bit samples in Malpedia can be taken as an advantage by our approach. Another fundamental problem are ngrams which overlap themselves. A YARA signature that contains ten sequences and detects samples if seven of those sequences are found, then the probability to find those is disproportionately greater if some of the sequences are subsequences of other sequences. For example a sequences derived from a 7-gram can contain a 6-gram, a 5-gram and a 4-gram in the worst case or 6-grams which contain several overlapping elements covering the same code sequence. This problem might be mitigated by enforcing a certain step size ngrams need to have between them when integrated into a YARA signature.

The Code quality and usability of the framework could be improved in general. While the project offers configuration files for users and a documented building process it is still considered unstable and not to be used in a productive environment. Enhanced testing and code safety features can increase the code quality and help growing the project into a productive usable framework for industry actors. The project might evolve into a full processing chain for malware dumping and clustering as a complete software solution.

Such an analysis chain could consist of various steps to generate YARA rules completely automatic without invoking substeps manually. A monitoring daemon for Malpedia could detect changes to the Malpedia database to invoke the start of such an analysis chain. Then disassembly can be generated using a disassembler like SMDA [Plo18] to provide SMDA reports for a new run of YARA-Signator. The changes of the Malpedia repository could be applied to the database backend and the new run of YARA-Signator would derive new YARA signatures.

6 Bibliography

[Bal16] W. Ballenthin. Yara-fn. https://gist.github.com/williballenthin/3abc9577bede0aeef25526b201732246, 2016. [Online; accessed 03 November 2018].

[Bil18a] Felix Bilstein. Automatic generation of code-based yara-signatures, 2018.

[Bil18b] Felix Bilstein. Yara-signator: Automatic yara rule generation for malpedia. https://github.com/fxb-cocacoding/yara-signator, 2018. [Online; accessed 16 July 2019].

[Bli08] Christian Blichmann. Automatisierte signaturgenerierung für malware-stämme, 2008.

[Dom18] C. Doman. Yabin. https://github.com/AlienVault-OTX/yabin, 2018. [Online; accessed 03 November 2018].

[Gro17] Cisco Talos Intelligence Group. Bass - bass automated signature synthesizer. https://github.com/Cisco-Talos/BASS, 2017. [Online; accessed 15 June 2019].

[IBM+18] Google Inc., Heiko Bengen, Joachim Metz, Stefan Buehlmann, Viktor M. Alvarez, and Wesley Shields. Yara — virustotal.github.io. https://virustotal.github.io/yara/, 2018. [Online; accessed 18-June-2018].

[Inc18a] Gentoo Foundation Inc. Gentoo linux. https://www.gentoo.org/, 2018. [Online; accessed 03 November 2018].

[Inc18b] MongoDB Inc. Mongodb. https://www.mongodb.com/, 2018. [Online; accessed 03 November 2018].

[Inc19] Google Inc. vxsig - automatically generate av byte signatures from sets of similar binaries. https://github.com/google/vxsig, 2019. [Online; accessed 15 June 2019].

[PE18] D. Plohmann and S. Enders. Malpedia. https://malpedia.caad.fkie.fraunhofer.de/, 2018. [Online; accessed 03 November 2018].

[Plo18] D. Plohmann. Smda. https://github.com/danielplohmann/smda, 2018. [Online; accessed 03 November 2018].

[Rot18] F. Roth. yargen. https://github.com/Neo23x0/yarGen, 2018. [Online; accessed 03 November 2018].

[Sec13] KoreLogic Security. Converting ida pat to yara signatures. https://blog.korelogic.com/blog/2013/11/15/pat2yara, 2013. [Online; accessed 03 November 2018].

[Tea04] The ClamAV Team. Clamav net. https://www.clamav.net/, 2004. [Online; accessed 15 June 2019].

[Yi18] H. Yi. Hyara (ida plugin). https://github.com/hy00un/Hyara, 2018. [Online; accessed 15 October 2018].