YARA-Signator: Automated Generation of Code-based YARA Rules

Felix Bilstein[1], Daniel Plohmann[1]
[1] Fraunhofer FKIE

This paper was presented at Botconf 2019, Bordeaux, 4-6 December 2019, www.botconf.eu
It is published in the Journal on Cybercrime & Digital Investigations by CECyF, https://journal.cecyf.fr/ojs

Abstract

Effective detection and identification signatures are an important component in the toolkit for malware analysis. The creation of such signatures is still widely a manual task that requires notable experience and knowledge on the side of analysts. In this paper, we present YARA-Signator, an approach for the automated generation of code-based YARA rules. The method is based on the isolation of instruction n-grams that on the one hand appear frequently within a malware family and on the other hand are not found in any other family. Applying YARA-Signator to the Malpedia data set, we show that in fact on average 51.85% of the instruction n-grams of length 4 and higher are only found in the respective family. The rules produced by the system using this data set achieve an overall F1 score of 0.983 and cause only very few false positives in a sanity check against a large goodware data set. YARA-Signator is made available as open source and a periodically updated reference rule set is provided for free through Malpedia.

1 Introduction

Malicious software (short malware) remains to pose a significant threat to the security and integrity of computer systems. To effectively and rapidly triage malware, analysts make frequent use of a variety of tools and systems. A cornerstone in the initial assessment of suspicious files are syntactic signatures that already have a long-standing tradition in anti-malware efforts. These signatures primarily enable detection and identification of malware families, which helps to speed up analysis procedures by making use of previous knowledge for these families. One of the most important and popular tools in this context is YARA.

YARA is a highly efficient pattern matching engine, accompanied with a very accessible rule description language. This has lead to YARA becoming a quasi-standard with wide adoption among practitioners and many rules being shared openly or in private threat hunting groups.

However, crafting rules that generalize well while avoiding misclassifications still remains a challenge. This process is often carried out manually, requiring knowledge and experience on the side of the analyst. Effective rules should ideally aim for stable and characteristic elements of malware, similar to the upper regions in the "Pyramid of Pain" [1] when thinking about attackers. One way to interpret this is trying to avoid potentially volatile or easily changed elements such as strings and instead aim for the code itself.

Previous works, e.g. by Blichmann [2] or Zaddach and Graziano [3], have already successfully demonstrated that the automated generation of code-based rules is possible. These approaches are based on the heuristical identification of longest common subsequences (LCS) that isolate code patterns in the form of instruction sequences that are found in all files of the input data. One drawback of the demonstrated approaches is their dependence on proprietary components (such as IDA Pro) and potential limitations in scalability.

In this work, we present our approach *YARA-Signator*, which follows the underlying idea of the previously presented approaches but transfers it to instruction sequences of fixed size, so-called n-grams. Opposite to the other works, YARA-Signator processes all malware families in a given input data set in paral-

lel. This allows us to execute aggregation operations that have the following benefits. First, we can eliminate code sequences that are found in multiple families, which are most likely instances of shared code undesired to become part of signatures, e.g. libraries. Second, by counting and ranking the number of appearances of n-grams in samples of the same family, we achieve an approximation of the LCS identification.

We propose a prototype implementation of YARA-Signator depending only on open source components and apply it to Malpedia [4], a community-curated corpus of cleanly labeled, unpacked malware samples covering more than 1,500 malware families. On this data set, the rules produced by the system achieve an overall F1 score of 0.983 with a high precision of 0.995. We additionally test the rules against a corpus containing 10 TB of benign software, on which 70 out of 992 rules produce a total of 13,879 false positives. While seemingly large, these numbers are however drastically driven by very few outliers, as 10 of these rules account for more than 92% of the FPs, showing that the rules are generally indeed very accurate.

In summary, our paper makes the following contributions:

- We present YARA-Signator, a method for the automated generation of code-based YARA signatures.
- Using the disassembly for 992 malware families from the data set Malpedia [4], we show that on average more than 51.85% of instruction n-grams of size 4 and larger are intrinsic for the respective families, i.e. only found in these and thus serve as good candidates for rule creation.
- We provide an open source implementation of YARA-Signator and make a periodically updated reference rule set for all processable families found in Malpedia publicly available.

The remainder of the paper is structured as follows. We first provide background information to ease the understanding of the proposed methodology and discuss related work to give a thematic overview of the topic. We then introduce our approach YARA-Signator and explain the workflow and components of the system. Afterwards, we examine the general viability of the method by providing a detailed statistical evaluation of the data set. This is followed by an evaluation of the classification performance of the rules generated using this data set and a false positive analysis against a large goodware corpus. We conclude with a discussion of limitations and future work.

2 Background

In this section, we discuss a number of aspects relevant for the understanding of the method proposed in this paper. We first discuss pattern matching and YARA in particular, before giving a short overview of the Intel instruction syntax and the concept of n-grams.

2.1 Pattern Matching and YARA

Pattern matching is a popular methodology that is also widely adapted in the context of threat detection, e.g. when monitoring network traffic or scanning files for malicious content. It typically uses a signature that consists of one or more known patterns associated with a threat to be evaluated against data of interest. In this regard, it is also used to detect or identify malicious software.

Apart from ClamAV [5], YARA [6] has become the de-facto standard for pattern matching in malware analysis. Its syntax is simple yet powerful, which makes it very popular among practitioners. As a result, there are many resources available where detection and identification signatures using the YARA format are shared.

Figure 1 shows an excerpt of a YARA signature. This particular signature is also an example of the automatically generated rules produced by the approach proposed in this paper: YARA-Signator.

All YARA signatures contain at least one mandatory part. A `condition` that describes what is necessary to trigger a detection when using this rule. This is a logical expression that can optionally address file meta data or content (e.g. `filesize` as shown in Figure 1) but will typically reference sequences defined in the `strings` environment. These `strings` can be defined as printable character sequence, i.e. text string, as hex string, or as regular expression. Optional keywords can modify the condition under which they evaluate as a match, e.g. `ascii` or `wide` for text strings, controlling the encoding for which the strings are defined. A third environment is also possible for YARA signatures: a collection of `meta` fields. These allow to annotate a signature with additional information, such as author names, creation date, or sharing restrictions.

2.2 Intel Instruction Syntax

Intel x86/x64 machine code instructions [7] are variable in length between 1 and 15 bytes and structurally encoded as a sequence of 6 fields:

Legacy Instruction Prefix: An instruction may be prefixed with zero to four instruction behavior modifiers, indicating exclusive use of shared memory (`LOCK`), conditional instruction repetition (`REP`), as well as segment, operand size, and address size override switches.

(Prefixed) Opcode: The core of the instruction is a 1- to 3-byte field that defines the actual opcode (cf. Figure 6, which gives a visual overview for all 1-byte opcodes in x86). Under certain circumstances, the opcode can optionally be prefixed, e.g. with a REX prefix when operating under 64-bit and wanting to access extended registers such as R8 to R15.

```
rule win_citadel_auto {

    meta:
        author = "Felix Bilstein - yara-signator at cocacoding dot com"
        malpedia_reference = "https://malpedia.caad.fkie.fraunhofer.de/details/win.citadel"
        malpedia_license = "CC BY-NC-SA 4.0"
        malpedia_sharing = "TLP:WHITE"


    strings:
        $sequence_0 = { 3bfe 7449 ff7508 e8???????? }
            // n = 4, score = 3500
            //   3bfe                 | cmp                 edi, esi
            //   7449                 | je                  0x4b
            //   ff7508               | push                dword ptr [ebp + 8]
            //   e8????????           |

        [...]

        $sequence_9 = { 8b0c0e 43 8901 8b470c 8bf3 }
            // n = 5, score = 3500
            //   8b0c0e               | mov                 ecx, dword ptr [esi + ecx]
            //   43                   | inc                 ebx
            //   8901                 | mov                 dword ptr [ecx], eax
            //   8b470c               | mov                 eax, dword ptr [edi + 0xc]
            //   8bf3                 | mov                 esi, ebx

    condition:
        7 of them and filesize < 1236992
}
```

Figure 1: Example for a YARA signature targeting the malware family Citadel, automatically generated using YARA-Signator.

ModR/M: A field that is required for some opcodes. If present, it encodes an extension, which defines which concrete registers or memory addressing mode should be used.

Scale, Index, and Base (SIB): Another field only required for some opcodes. If present, it will describe how exactly addresses are calculated and how the displacement may be used in this context.

Displacement: The displacement is a field containing a value of 1, 2, 4, or 8 byte length that is used as an offset for the calculation defined by the SIB field (if present). In case the displacement has a length of 8 bytes, no immediate may follow.

Immediate: Some instructions may use an immediate value, which can be 1, 2, 4, or 8 byte long, depending on what is defined by the instruction or ModR/M field. Similarly, an 8-byte long immediate is mutually exclusive with a displacement.

In the context of this paper, the Displacement and Immediate fields are of special interest. Because both fields may contain concrete addresses and offsets that are very specific to a given compiled program or a result of mapping and memory relocations, it is desirable to replace these concrete values with wildcards in certain cases to achieve signatures with better generalization. Figures 1 and 2 give an example of wildcarding in the context of YARA, in both cases removing the concrete value for a interprocedural, relative-offset call instruction.

2.3 N-gram Structure

N-grams are consecutive subsequences with a fixed length taken from a given sequence of items. The use of n-grams in the context of detection or identification is a common technique in the field of malware analysis. [8]

For code-based signatures, the interpretation of items could either mean taking a set number of bytes or instructions, which themselves usually consist of multiple bytes. In our case, we use instructions and Figure 2 provides an example for the derivation of n-grams given a stream of instructions.

There is a list of instructions given on the left hand side in Figure 2. These instructions are sequentially executed by the target architecture and therefore we have to keep this order. For a given length (four in this case), we derive four possible n-grams from the instruction list of length seven.

3 Related Work

In this section we provide an overview of related work. We focus on three categories: Frameworks for autonomous rule generation, tools supporting manual creation of rules, and YARA rule archives.

With regard to full systems for rule generation, Blichmann recently published VxSig [9], a reference implementation for the seminal approach published in his diploma thesis [2]. VxSig allows the automated generation of signatures in the YARA and ClamAV format from sets of previously grouped, similar binaries. Input files are processed using BinExport [10] and BinDiff [10] to locate and isolate common code fragments. BASS [3] is a framework published by Zaddach and Graziano for the automated generation of ClamAV signatures over previously identified malware clusters. As noted by the authors, their method has strong sim-

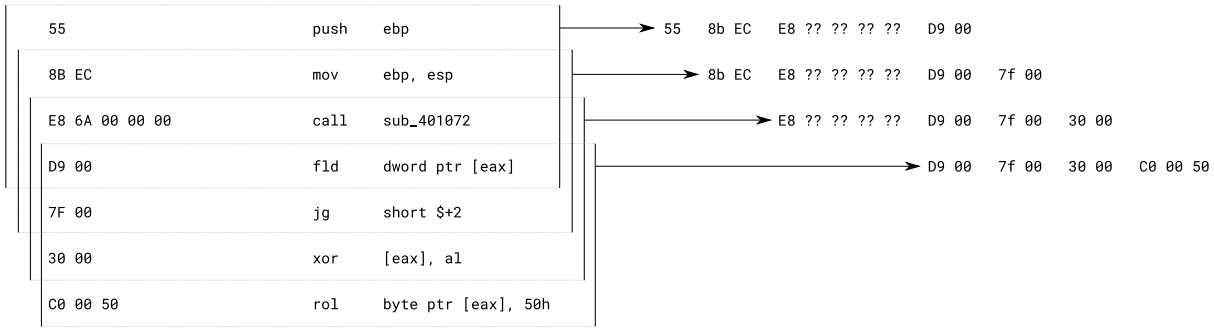

Figure 2: N-gram derivation from a given sequence of x86 instructions to 4-grams.

ilarities to Blichmann's approach [2] but aims for high scalability and additionally includes a method for filtering out code sequences from known goodware, using Kam1n0 [11]. Roth published yarGen [12], a tool that enables the automated generation of YARA rules based on one or more input files. It can process both strings and code and can optionally include blacklist information from databases, e.g. to further enhance the rule creation procedure by removing all strings that also appear in known goodware. Doman published YaBin [13], a tool that creates YARA signatures from code sequences that are automatically extracted from its input programs. The concept of YaBin is based on a heuristic search for common function prologues, e.g. "55 8B EC" (`push ebp; mov ebp, esp`) and discrimination against a whitelist of sequences from a collection of non-malicious software (about 100 GB in size). Heuristics for prologues cover the compilers MS Visual C, Borland, and MinGW.

The following projects aim at improving the workflow for manual creation of YARA rules. Yi published Hyara [14], a plugin for IDA Pro and BinaryNinja that allows to highlight code regions and strings that can then be quickly turned into YARA rules. KoreLogic published pat2yara [15], a helper script that allows to convert rule files generated using IDA Pro's FLAIR engine into YARA rules. Ballenthin created "YARA-FN" [16], a script for IDA Pro that creates a YARA rules from all basic blocks of the currently shown function that is also capable of wildcarding relocations and jump instructions.

Multiple notable public collections of YARA rules exist. The YaraRules Project provides a large collection of community-collected YARA rules that is managed as a Github repository [17]. Roth provides an extensive and frequently updated set of free YARA rules called "signature-base" [18], which is the default rule set used by his free scanning tools. Worth offers a curated collection of YARA rules called "Open-Source-YARA-rules" [19] sorted by their creator, covering 126 entities with 1,711 rules. Wesson provides a set of rules via "Project Icewater" [20], which produces rules that are automatically derived based on clustering over similarity.

4 YARA-Signator

In this section we introduce YARA-Signator, our framework for the automated generation of code-based YARA rules. The framework is supposed to generate YARA signatures based on a given set of disassembly reports by processing them in multiple steps that we explain in this section. We start by providing a general overview of the approach in Section 4.1 and then present each of the processing steps of our framework in more detail in Section 4.2. The structural details of the implementation, i.e. its components and dependencies, are discussed in Section 4.3.

4.1 Approach

Our approach can be summarized as the task to identify fragments of code that are found only in representatives of the same malware family, while being absent in all others. These specific fragments by definition are characteristic for the respective family and thus should serve as good components for a detection signature. Since malware has to be considered simply a special category of software we assume that underlying development processes are very similar or comparable to the processes used for regular software. This means that we generally expect most software projects to have a somewhat stable code base that usually does not change too drastically between its versions. We can furthermore take advantage of this assumption by preferably selecting the code fragments that appear in as many versions of the family as possible, which can be seen as a sign for their significance. In consequence, we expect that combinations of such fragments will yield reliable YARA signatures with good potential for generalization.

Since YARA signatures can consist of strings, byte sequences, and regular expressions, we choose byte sequences as they are best suited to represent code fragments. Because code is naturally structured in (machine) instructions, we assume that start and end markers of common code fragments will potentially fall together with instruction borders.

To avoid having to find common code fragments of maximum length (cp. Blichmann [2]), we instead

decide to work on n-grams of instructions. These n-grams are derived from disassembly reports, which serve as the input data format. All n-grams for one family are aggregated into a common pool. Given k malware families, the resulting task of isolating the characteristic code fragments essentially can be understood as a set operation in which for each of the family's n-gram pools all n-grams of the $k-1$ other families are removed. Figure 3 illustrates this part of the approach. Following this procedure, the remaining n-grams in each pool are ranked and a selection of candidates is composed into YARA signatures which are then optimized for coverage.

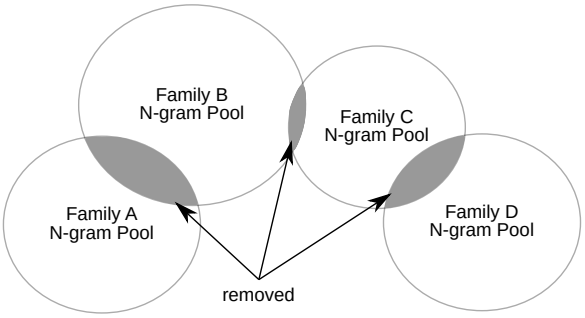

Figure 3: N-gram filtering.

Overall, the procedure to automatically produce code-based YARA signatures can be seen as a workflow consisting of four stages:

1. Data ingestion
2. Unification and Filtering
3. Rule Generation
4. Iterative Improvement

During the first step, data ingestion, all reports are parsed by framework and linearized into n-grams. These n-grams are unified and filtered by performing data aggregations. The goal is to find n-gram candidates with a high coverage for a given malware family that are not overlapping with the code of other families. On the basis of these candidates we apply several filters to find the most suitable candidates. These candidates are written into YARA signatures which are evaluated. Then, we iteratively improve the generated YARA signatures by re-validating them in every step, potentially increasing their precision and coverage. These stages are explained in detail in the following section.

4.2 System Details

In this section we present the different stages of our approach in detail. Figure 4 illustrates the four primary stages of our framework.

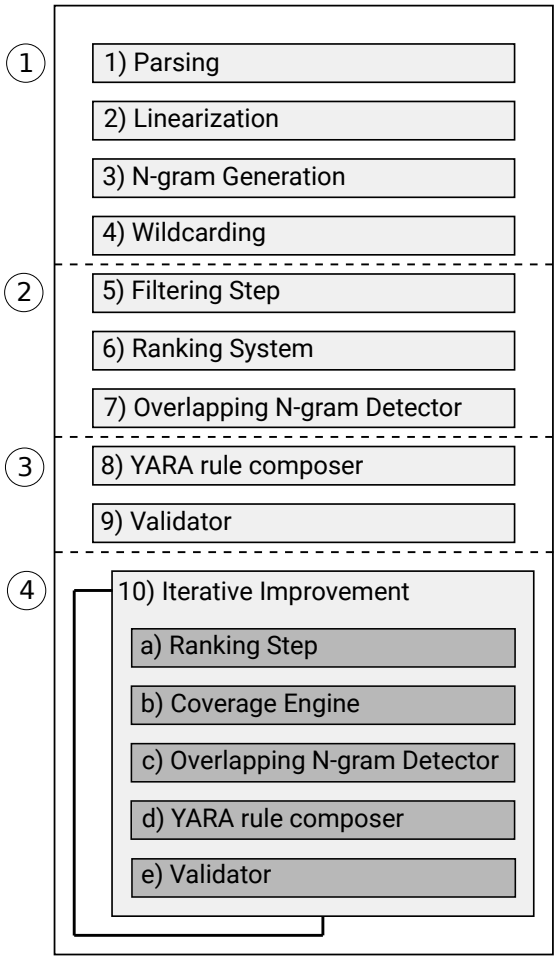

Figure 4: **YARA-Signator** and its processing steps.

Data Ingestion is the first phase of our approach. We process given disassembly reports for a set of malware samples that we want to create YARA signatures for. The malware samples have to be unpacked and pre-clustered beforehand, so that samples are grouped with their respective family. Disassembly reports are parsed and instruction n-grams are extracted.

The second phase operates on the normalized data that we created in the first phase. We filter all duplicate n-grams between families and rank candidates for each malware family. Overlapping n-grams are removed to sanitize the candidate pools for each family.

A first set of YARA signatures are created within the third phase of our approach. After composing the YARA rules, we validate them against the input corpus to evaluate the quality of the generated signatures.

The last phase is an iterative improvement phase where problematic YARA signatures are re-generated and re-evaluated to provide an improvement of the different previously created signatures over time.

4.2.1 Data Ingestion

Parser. As an initial step, disassembly has to be parsed. We use SMDA [21] as for this, because it is an open-source recursive disassembler built on top of

Capstone [22]. It is convenient to use, producing JSON files as output and has the capability to reliably reconstruct and extract code from memory dumps. Note that the approach (and the implementation) is generally independent from the choice of disassembler as it could be trivially adapted to any other data input format.

Linearization. An advanced disassembler will produce fully reconstructed CFGs that divide the identified code into functions. Because YARA operates on byte sequences, we need to flatten the Control Flow Graph into a linear sequence of instructions that resembles how code is encountered in the binary. Using address offsets and the individual instructions' sizes, we can furthermore split the linearized stream into consecutive chunks whenever a gap is encountered.

N-gram Generation. In this step, we produce the data points YARA-Signator actually operates on. We derive all possible n-grams of a pre-defined size from the chunks resulting from Linearization. In the context of this work, we use sizes of 4 to 7 instructions per n-gram based on previous findings [23].

Wildcarding. Code may generally contain absolute virtual addresses or offsets, like memory pointers to code or data. These may even be shifted due to relocations while mapping a binary. Using sequences with these absolute values in place for signature generation could lead to false negatives. To avoid this, we perform additional abstraction and wildcard all occurrences of such pointers using absolute addresses. In fact, we even wildcard all relative references pointing outside the scope of the function we generate n-grams for as well, e.g. inter-procedural calls and jumps. Additionally, we also wildcard immediate values that could be interpreted as addresses within the mapping of the given binary when mapped. For this, we inspect the Displacement and Immediate fields of all instructions (cf. Section 2.2). This procedure is equivalent to the wildcarding applied by Cohen and Havrilla [24] for their technique of creating position-independent code (PIC) hashes of functions. We expect this to additionally benefit rule generation as it may help make signatures more robust against code reording that can happen due to an author's refactoring or compiler effects.

4.2.2 Unification and Filtering

Filtering Step. This step implements the actual idea, as initially described in the beginning of Section 4.1. By aggregating identical n-grams across all ingested samples and families, we can filter out all n-grams that occur in more than one family. While doing so, we additionally track in how many different samples the family-unique n-grams occur as this will help identifying representative n-grams in the next stages. All remaining n-grams are considered potential candidate n-grams for rule generation.

Ranking System. We developed the ranking system to allow flexible configuration by the user. It consists of individual ranking functions that generate a score for a given n-gram. Multiple ranking functions can be chained to incorporate multiple semantics into the overall rating of an n-gram. Example metrics used for ranking in our reference implementation are the number of occurrences in different samples and the types of instruction (e.g. memory-access, or logic/arithmetic) found in the n-gram. After the ranking, a selection of highest-ranked candidates is selected per family (in our configuration, 10).

Overlapping N-gram Detector. When n-grams are aggregated across samples, the information about the relative position of n-grams to each other is lost. As Ranking is applied for individual n-grams in a procedure not considering other n-grams, it may rank several n-grams similarly well due to characteristics they share. This may potentially be a result of them overlapping or even being contained within each other, e.g. ABCDEF and ABCD or ABCDEF and CDEFGH (with each letter representing an instruction at a certain offset). As it may be favorable to have signature contents being spread over the target or at least not being redundant, this stage ensures that no excessive overlapping exists between n-grams selected for the signatures.

4.2.3 Rule Generation

YARA Rule Composer. Given a collection of n-grams, this step uses a rule template to construct a functional YARA rule, updating meta data information such as a date and input data used. In the workflow of the framework, the n-gram candidates per family are used to compose a rule. We also include a filesize cap for each family's rule that is calculated as twice the size of the biggest input sample.

Validator. The Validator performs an evaluation of all rules created against the data they were generated from. The desired result is obviously full coverage with no false positives. However, since rules are only derived from parts of the input binaries (disassembled code), false positives may still occur. Due to the initial selection of n-grams, false negatives may also occur. The resulting evaluation report is used to trigger an Iterative Improvement phase that is applied to all rules that do not have full coverage without false positives yet.

4.2.4 Iterative Improvement

The Iterative Improvement aims at optimizing the YARA signatures through additional rounds of refinement. Every iteration can be controlled independently by using a different configuration for each cycle. One iteration cycle has five different steps: Ranking, Coverage Engine, Overlapping N-gram Detector, YARA Rule Composer, and Validator. All steps are similar to their equivalent described before, except that the Ranking step can be configured for each iteration independently and additionally Coverage Engine is executed.

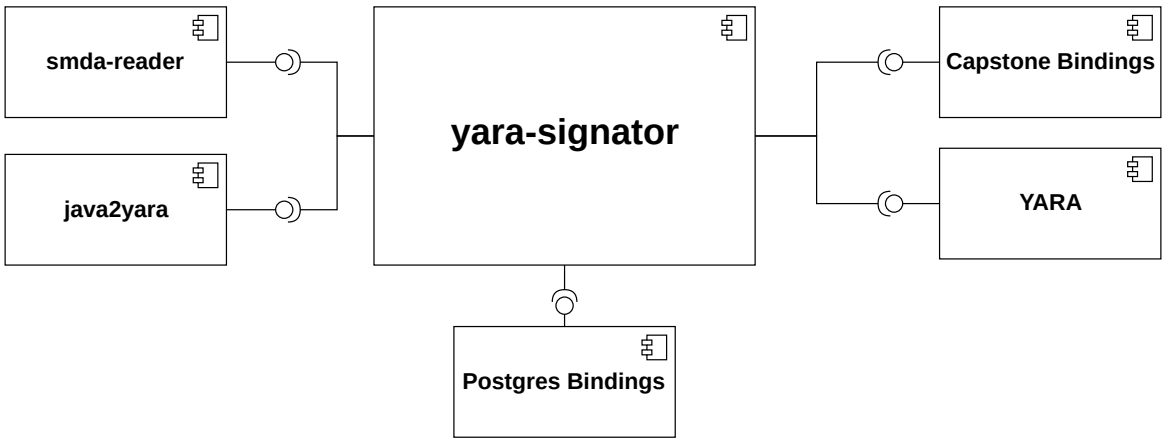

Figure 5: **YARA-Signator** and its components: The two JAVA libraries *smda-reader* and *java2yara*, its bindings to *PostgreSQL*, to the *Capstone* disassembly engine and *YARA*.

Information about false positives is used to blacklist n-grams from their use in rules. The Coverage Engine is then applied to all rules that did not have optimal coverage yet. Given the information about which n-grams cover which samples, the problem of achieving a minimal coverage of all samples is an instance of the Set Cover problem [25] and in theory NP-hard. We use a greedy approximation [25] that performs in polynomial time and exceeds the optimal solution by no more than the nth harmonic number in ratio. This suffices for our use case as we look at a few hundred n-grams as input at most (example harmonic numbers: $H(100) = 5.19$, $H(1000) = 7.49$). The algorithm iteratively selects an n-gram that achieves the highest coverage gain, i.e. covering additional uncovered samples, until all samples are covered. The Overlapping n-gram Detector again ensures that the coverage is additionally spread over the code. Validation rounds are used to update the blacklist with potential iteration until a satisfying result in rule output is achieved.

4.3 Implementation

We now discuss the implementation of our approach. We created a framework around our core tool YARA-Signator to provide a full toolchain enabling automated generation of YARA signatures. Figure 5 illustrates the core and relationship of the different modules.

We implemented the library `smda-reader` as a means for ingesting disassembly reports generated using SMDA [21] as described in Section 4.2.1. Technically, `smda-reader` parses the reports provided in JSON into Java objects. As of now, we only support SMDA as a disassembler but since the data ingestion is handled through an interface and normalized objects, we are not limited to a single technology with our approach. An adaption of other third-party software like IDA Pro or objdump as an input provider would be trivial.

Because processing the disassembly for rule generation requires space and we want a performant procedure, we decided to use a database as backend. Given several databases to choose from, we decided to use the relational database PostgreSQL [26]. This database management system natively supports various techniques that can be used to efficiently implement our approach. This includes aggregations for filtering data and a range of performance tuning techniques such as indexing and partitioning. We simply implemented a wrapper to communicate with the database driver to access and persist data.

Since we create YARA signatures we needed a library to build YARA rules from JAVA programs. We implemented `java2yara` as a library with which signatures can be created from a collection of strings and given meta data. As we want to enrich the rules with additional information about the instructions used in the signature strings, we also need a disassembler. Again, having different options to choose from, we went with capstone [22] because it is open-source and also the basis for SMDA.

Finally, for the validation of the YARA rules we use YARA itself as an external program. The results are parsed from its output and processed by our framework. The scan reports generated this way are used to evaluate the rules against the input data set and an important element to steer the iterative improvement process.

5 Statistical Analysis

Before we conduct a performance evaluation of the rules produced by YARA-Signator, we first want to get a better understanding for the general viability of signature generation approaches based on code n-grams. For this reason, we perform a statistical analysis of the primary data set used in this study: "Malpedia" [4].

After a short introduction of the data set, we will examine different distributions, e.g. amounts of code found in malware families as well as the individual instructions in the corresponding disassembly reports. We then continue by further analyzing n-gram distributions and uniqueness, which we obtain as an intermediate result in the procedure of applying YARA-Signator on the data set.

5.1 Data Set

Given the design of the approach as described in section 4, we note that one requirement is that the files of the input data have to be grouped already, e.g. by malware families. A data set that is well suited to test our approach on is the Malpedia [4] corpus, a community-curated malware corpus of (unpacked) reference samples including public analysis references for as many families as possible. In this study, we use Git commit `d9bc781` from February 25th, 2020 as a baseline snapshot. At this time, Malpedia consists of 4,469 samples for 1,573 malware families, which accumulate to a total of 8,939 files.

Not all of the files found in Malpedia can be processed by YARA-Signator. Because the framework currently operates on x86/x64 exclusively and we intend to only process unpacked or dumped files, we need to filter the data set before we disassemble the files. This reduces our input data to 3,313 processable samples from 1,150 families.

Among these are still families that consist of non-native code because they are written in other programming languages, e.g. those created using the .NET framework. Filtering out all files that do not fulfil the native-code requirements, we now use the SMDA disassembler [21] to process the input files. Ultimately, this leaves us with disassembly reports for 3,039 samples from 1,022 families. These amount to a total of 4,150 input files because sometimes more than one unpacked or dumped representation is associated with one sample, e.g. because of a 32bit and 64bit payload, or additional modules.

Per Family	Files	Functions	Instructions
Minimum	1	1	2
25%	1	138	7,087
50%	2	412	20,923
75%	3	1,135	52,133
Maximum	121	18,213	931,948
Total	4,150	3,733,355	195,422,329

Table 1: Statistics for the processed input data. Functions and Instructions have been averaged per family before aggregation.

5.2 Disassembly Overview

In this section, we provide a characterization of the disassembled data in general.

We first address bitness. Out of the 4,150 files, 91.78% are 32bit and 38.22% are 64bit. With respect to families, we find that 97 feature 64bit code only, and 74 have code both in 32bit and 64bit, while the 826 remaining families are 32bit only.

In total, SMDA identifies 3,733,355 functions with 195,422,329 instructions in the used portion of the data set. Table 1 provides further insight into how the functions and instructions are distributed across the samples in the families. Interestingly, these numbers are very much in line with our earlier reports about these statistics [23]. We note that the prototypical function has about 8-10 basic blocks and consists of round about 50 instructions, which is fully sufficient to apply our proposed method on.

We now use the wildcarding method by Cohen and Havrilla [24] as explained in Section 4.2.1. This allows us to determine position-independent code (PIC) hashes for all functions. In our implementation, we use MD5 over the concatenation of all wildcarded instructions in their hexbyte representation, sorted by address. This leaves us with 947,421 unique hashes for functions, out of which 848,783 (89.59%) only appear in one family each. Our number is higher than the 81% reported by Cohen and Havrilla but likely explained by their more diverse data set for which we would expect a wider presence of library code. In any case, this is a positive result as it indicates that we can definitely expect to find significant amounts of code being unique per malware family which will benefit the generation of rules.

	Mnem	32bit	64bit
1	mov	49,890,410 (28.17%)	6,144,638 (39.76%)
2	push	26,770,256 (15.12%)	274,490 (1.78%)
3	call	14,704,502 (8.30%)	1,347,419 (8.72%)
4	pop	8,548,750 (4.83%)	273,385 (1.77%)
5	cmp	8,060,341 (4.55%)	770,526 (4.99%)
6	lea	7,570,190 (4.27%)	1,147,978 (7.43%)
7	add	6,580,883 (3.72%)	557,804 (3.61%)
8	je	6,371,325 (3.60%)	581,553 (3.76%)
9	dec	6,064,865 (3.42%)	41,810 (0.27%)
10	test	5,711,807 (3.23%)	585,390 (3.79%)
11	jmp	5,184,997 (2.93%)	541,978 (3.51%)
12	xor	4,934,907 (2.79%)	618,684 (4.00%)
13	jne	4,525,392 (2.55%)	437,352 (2.83%)
14	ret	3,481,393 (1.97%)	246,005 (1.59%)
15	inc	2,595,499 (1.47%)	109,312 (0.71%)
16	sub	2,485,040 (1.40%)	322,980 (2.09%)
17	and	1,863,284 (1.05%)	196,456 (1.27%)
18	movzx	1,577,742 (0.89%)	216,612 (1.40%)
19	or	1,352,995 (0.76%)	112,826 (0.73%)
20	shr	667,087 (0.38%)	69,654 (0.45%)
21	jb	616,242 (0.35%)	58,695 (0.38%)
22	shl	571,271 (0.32%)	59,692 (0.39%)
23	nop	557,776 (0.31%)	57,045 (0.37%)
24	jle	471,338 (0.27%)	40,606 (0.26%)
25	jl	461,572 (0.26%)	33,305 (0.22%)
Total		171,619,864 (96,91%)	14,846,195 (96.30%)

Table 2: The 25 most prominent instruction mnemonics for 32bit and 64bit.

5.3 Instruction Occurrence Frequencies

We now have a closer look at the individual instructions contained in the disassembly reports. Looking at the 25 most popular instructions mnemonics (cf. Table 2), we can see that they aready account for more than 96% of all mnemonics. We further notice significant differences for a number of mnemonics between 32bit and 64bit. First, for 64bit, `mov` instructions are about 10% more frequent. At the same time, the stack operations `push` and `pop` together reach only 3.5%, opposed to 20% for 32bit. A primary reason for this is that 64bit function calls are often carried out using the __fastcall calling convention, passing arguments via registers instead of using the stack.

Similarly, `inc` and `dec` are found way less often for 64bit code. While we did not investigate this in-

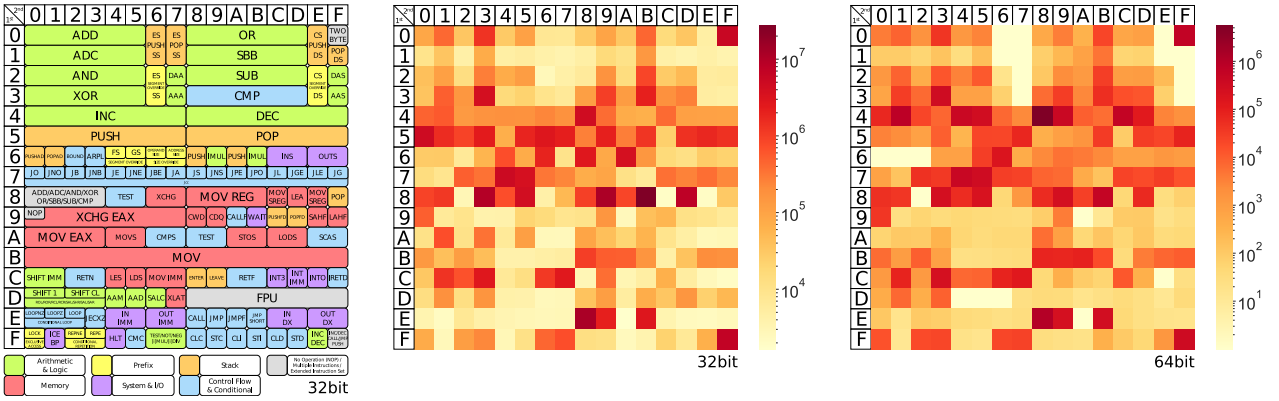

Figure 6: First byte occurrence distribution among the 195,422,329 instructions, separated by bitness. A major difference is the increased used of `0x40-0x4F` bytes in 64bit (REX prefix), and reduced use of stack operations.

depth, we believe that this is connected to instructions starting with `0x40-4F` (`inc/dec <register> under 32bit`) being repurposed as REX prefix under Intel x64 [7]. We have also rendered heatmaps of the first byte instruction distribution in Figure 6, along with a reference for 32bit instructions and their semantic context.

5.4 N-gram Occurrence Frequencies

After the examination of distribution properties for individual instructions, we now look at sequences of instructions, i.e. n-grams, as used by YARA-Signator. We are interested in two statistics particularly: Uniqueness of n-grams across families overall and with respect to the families they originate from. Both of these values provide insight in the general viability of our outlined approach.

occurrences	N-gram size 4	5	6	7
1	84.94%	86.51%	87.47%	88.10%
2	7.70%	7.09%	6.68%	6.34%
3	3.06%	2.73%	2.53%	2.39%
4	1.22%	1.05%	0.96%	0.46%

Table 3: Occurrence counts of n-grams in different malware families.

First, we look at the occurrence frequency of n-gram uniqueness across families. The results are listed in Table 3. We count a total of 187,800,586 unique n-grams for all lengths combined. With regard to their relative uniqueness, we see that even for instruction n-grams of length 4, already 84.94% of these n-grams appear only in a single family. For two and three families, we note a steep decline with 7.70% and 3.06% respectively, summing up to 95% of all n-grams. Expectedly, for longer n-grams, these numbers lean even more towards an occurrence of one time across all families only. For n-grams of length 7, the 88.10% of family-unique n-grams are also very close to the observed 89.59% for family-unique PIC function hashes as discussed in Section 5.2. Overall, these statistics are good news as they imply that a vast majority of n-grams can be used for signatures without causing false positives on the data set. However, we do not know yet how these unique n-grams are distributed across families.

Per Family	4	5	6	7
Minimum	0.00%	0.00%	0.00%	0.00%
25%	20.68%	23.60%	25.53%	26.77%
50%	45.21%	51.11%	53.61%	55.88%
75%	68.68%	78.78%	83.97%	87.24%
Maximum	100.00%	100.00%	100.00%	100.00%
Average	45.86%	51.15%	54.23%	56.14%

Table 4: Relative amount of unique n-grams per family.

Therefore, we now inspect the percentage of family-unique n-grams for all families. The results are shown in Table 4. For a total of 5 families (with one sample each), YARA-Signator did not identify unique n-grams. An inspection of these shows that 4 samples were misclassified in Malpedia because of different aliases referring to them in the referenced reporting, while one family was a .NET sample that was not filtered before. For each of the remaining 992 families, a number of n-grams sufficient for rule generation is found. Not only this, for the median family, between 45.21 and 55.88% of n-grams are unique to that family depending on n-gram length. Similar to what was observed before, longer n-grams lead to higher relative uniqueness. Overall, we find that basically every family contains some portions of unique code that can be automatically identified and used to target it in a signature. The average percentage of unique n-grams across all n-gram lengths and families is 51.85%.

Deeper investigation of the results allows us to make more interesting observations. For example, families that stick out with a high n-gram but low unique n-gram count are for example `win.combojack` [27] (520,891 n-grams total but 2.40% unique) and `win.shurl0ckr` [28] (1,441,625 n-grams but 3.25% unique). Both are compiled with frameworks that make use of excessive static linking, in these cases Delphi and Go respectively.

In a few cases, we observe a similar phenomenon for families compiled with the much more popu-

lar MS Visual Studio. Here, we find families that have a lower number of total n-grams but still a low number of unique n-grams. An example would be `win.carrotbat` [29], a simple downloader used by a threat actor in campaigns targeting Southeast Asia. For this family we count 40,295 n-grams among which 6.5% are considered unique.

On the other end of the spectrum, we can find families such as `win.locky` [30] and `win.nymaim` [31]. These families employ custom obfuscation schemes that lead to a high n-gram count out of which the vast majority is also unique. For example, in `win.nymaim` we find 2,335,906 n-grams out of which 99% are unique across all families.

6 Evaluation

In this section we present the results of our evaluation of the generated YARA signatures. We first explain the data sets used and then describe the rule generation process and structure of produced rules. Afterwards, we discuss the experiments to measure classification performance.

Phase	Time in hours
Disassembly	2
Data Ingestion	8
Filtering	2
Initial Rule Generation	2
Iterative Optimization	1
Total	15

Table 5: Duration of a full run of YARA-Signator.

6.1 Data Set

In the evaluation, we use the same data set as described in Section 5.1: a snapshot of Malpedia (commit: `d9bc781`) [4].

However, since previous works [4] showed that almost 80% of the analyzed 443 malware families for Windows in that study had been created using Visual Studio, we take this into concern for blacklisting. For this purpose, we use the data provided by the Empty MSVC Project [32]. As the name implies, this project contains empty projects compiled with all available versions of MS Visual C in all major compiler settings (dynamic and static linking, debug and release builds). This way, the resulting programs contain only code that we would expect to be inserted by Visual Studio, which is likely shared and should not become part of YARA signatures. We also add a number of goodware that we identified being prone to false positives in previous experiments, among them MFC libraries and network libraries from Internet Explorer and Firefox.

To further test the robustness of the rules produced by YARA-Signator with regard to FPs on benign software, AVAST kindly ran our rules against their corpus containing 10 TB of goodware.

	SeqCount	N-gramLen	SeqLen	SizeCap
Minimum	5	4	4	24,576
25%	10	5	14	188,416
50%	10	6	18	402,432
75%	10	7	23	1,040,384
Maximum	220	7	77	35,323,904

Table 6: Statistics that describe the characteristics of the output rules. SeqLen and SizeCap in bytes.

6.2 Rule Generation

For rule generation, we use a system with the following specifications: Intel I7 with 32GB RAM, an SSD as system and a HDD to host the data partition. With respect to the different processing phases described in Section 4.2, we note the processing times as shown in Table 5, taking about 15 hours for the full procedure.

The outcome of this procedure is a rule set for 992 of 997 families that we used as input. For 5 families, no unique n-grams could be identified and thus no rules were generated. We now further characterize these rules, a summary is given in Table 6.

First, we inspect the number of sequences used per rule. Only the rule for `win.poisonivy` consists of 5 sequences (all others have 8 or more) and the largest is `win.isfb` with 220 sequences that originate from 121 input files. In fact 763 (78.23%) have exactly 10 sequences, which is a result of the configuration chosen in Section 4.2.2 and the fact that 482 families from the input data are only represented by a single file. The total number of sequences is 12,542, out of which 6,430 (51.27%) contain one or more wildcards.

With regard to the distribution of n-gram lengths within these sequences, we can see that they are skewed towards longer sequences and there are in fact 5,028 n-grams with 7 instructions, which are 40.01% of all sequences. This is primarily caused by the Overlapping Detection, which discards shorter n-grams in favor of longer n-grams containing them. All n-grams combined contain 72,954 instructions, out of which 10,004 (13.71%) are wildcarded.

For sequence lengths (SeqLen), we notice that half the sequences are between 14 and 23 bytes. Raff et al. [33] recently showed in a study on code reuse identification with large n-grams of up to 1024 bytes length, that shorter n-grams of $n < 32$ generally provided better accuracy.

Finally, we look at the values used to cap file sizes as explained in Section 4.2.3. As this value is twice as large as the largest input file per family, we note a range from 24 KB up to 35 MB. The smallest files are very simple downloaders that only consist of a few functions while the largest is `win.rms`, a remote administration toolkit that was observed being used in targeted intrusions by threat actor TA505.

	True	False
Positive	4,035	22
Negative	3,459	115

Table 7: Classification results of running the 992 YARA rules against the input data set. In addition to the 4,035 True Positives, another 1330 hits on files of the respective family were registered.

6.3 Classification Performance

After inspection of the generated rules, we now want to evaluate their performance with regard to detection capabilities.

We first apply the rules against the Malpedia data set with which they were generated. The results are shown in Table 7. Overall, all except 115 files were positively classified, which results in a Recall of 0.972. Interestingly, 1,329 additional files were correctly classified by the respective rule corresponding to their family, which indicates the generalization potential of the used n-gram and wildcarding method. With just 22 false positives, the rules have a very high Precision of 0.995. The overall F1 score is 0.983.

Looking closer at the rules, we find that 977 rules did not produce false positives and 923 rules did not have any false negatives. Combined, 916 rules are considered clean in that they did not cause any misclassifications.

We next investigate these misclassifications in detail. First off, false positives typically have to be considered as a direct result of disassembly errors. If all disassembly was exact, the sequences causing FPs would have been sorted out during the aggregation and filtering stage. The following scenarios can occur. First, if code is disassembled correctly in one family but missed in another, this may result in n-grams that lead to false positives. Otherwise, if disassembly is produced "wrongly" for a family, this may lead to wrong instruction borders and thus n-grams that will still detect the same byte sequences in other families.

With this in mind, we now first focus on the false positives that occurred. For at least 9 out of 22 hits, we assess that they are caused by actual contextual relationship between the families. For example, the YARA rule for `win.isfb` detects `win.dreambot`. Both families are based on the leaked `gozi` source code [34], with Dreambot e.g. being able to use Tor. Other overlap that is similarly explainable is found e.g. for `win.dropshot`, `win.shapeshift`, and `win.stonedrill`. The rule for `win.reactorbot` also causes hits in `win.rovnix`, because this protector/rootkit has been used in conjunction with `win.reactorbot` and the samples in Malpedia for `win.rovnix` contain the `win.reactorbot` payload. One hit is also caused by a binary duplicate stored under two family names in Malpedia (that has since been resolved). For the remaining 13 hits, we could not find a better explanation than disassembly errors and potential library code overlap.

With regard to false negatives, we note that they are also the result of different effects. In the majority of cases, we note that disassembly errors may lead to situations where parts of a sample are missed that could otherwise be used as characteristic sequences for a given family. This naturally causes a situation where not enough sequences for a sample are extracted and incorporated into the rule, causing it to miss the sample. We found that this particularly affects samples that already have a very small number of functions. In a number of cases, we also found that a sample sorted into the wrong family resulted in elimination of many otherwise possible sequences from rules in the filtering stage, leading to an insufficient number of sequences to trigger on the sample. This had the positive side effect that we could optimize the corpus and correct these wrongly classified samples in the data set. In few cases, we also noticed that this happened in legitimate cases, especially when a family as itself is used as a "module" in another family.

6.4 False Positive Analysis

We now conduct an analysis of false positives on a second data set. For this analysis, AVAST kindly ran our rules against their clean data set and provided the detection results back to us. The data set comprises of about 10 TB of data and any hits can be safely assumed to be undesired as it only consists of known benign software.

We register a total of 13,879 hits caused by 70 of the 992 rules. While this seems initially like a large number, the distribution is highly skewed. The rule with the most hits alone is responsible for 8,206 hits (59.13%) and targets the ransomware `win.scarabey` [35]. We analyzed the rule composition and malware, noting that the malware makes extensive use of Application Framework Extensions (AFX), a predecessor of Microsoft Foundation Classes (MFC), typically used to create GUIs. Smaller portions of AFX code fragments are only found in 6 other malware families. Because AFX was not added to the blacklist beforehand, this leaves enough n-grams to be considered "unique" across the malware in Malpedia. However, since lots of benign software also make use of AFX, this immediately explains the amount of FPs caused by this.

Looking at the next rules in the top five, we find 1,258 (9.06%), 957 (6.90%), 914 (6.59%), and 370 (2.67%) hits. Only 6 other rules have more than 100 hits and all of them together are responsible for 92.96% of false positives. For the remainder, there are 31 rules with between 10 and 100 hits, while 28 rules produce less than 10 hits.

It is also notable that 4 out 24 rules for macOS malware produce false positives. This is explained with the fact that with such few families in Malpedia generally the expected code elimination effect is minimal compared to Windows PE files and that the blacklist data did not specifically target macOS.

7 Limitations and Future Work

We now discuss limitations and ideas for future improvements of YARA-Signator.

Right now, YARA-Signator is only capable of ingesting disassembly reports created using SMDA. The primary limitation of this is that only input binaries consisting of x86 and x64 Intel machine code can be processed. Ways to improve this situation would be to provide additional or a more generic interface for data ingestion. For example, YARA-Signator could provide an interface to parse the output of other disassemblers with wider architecture coverage such as IDA Pro, Ghidra, or radare2. Otherwise, it could also provide an interface to ingest pre-processed data sequences as a full replacement of the current first phase. This would open it to e.g. working on raw byte sequences of arbitrary length.

While providing rules that lead to few false positives already, further improvement could be achieved by providing a more comprehensive blacklist of code found in benign programs. For further reduction of false negatives, the parameters of the system (n-gram size, n-grams per family and hit condition, ...) could be evaluated to find an optimal configuration.

We also believe that there is room for further research and improvement in the sequence selection process.

8 Conclusion

In this paper we presented YARA-Signator, a framework for the automated generation of code-based YARA rules.

First, we outlined the general idea of isolating characteristic code sequences unique to a family through filtering and data aggregation. We explained the processing stages including n-gram deriviation and code wildcarding up to rule creation and iterative improvement, as well as the implementation of the framework in detail.

Next, we performed a statistical analysis of the comprehensive malware corpus Malpedia, showing that the general idea of finding unique code sequences for malware families is very viable. With on average 51.85% of n-grams being unique to a family, this leaves significant amounts of code to pinpoint and base signatures on, e.g. using YARA.

We then used YARA-Signator to produce YARA rules for 992 processable malware families in Malpedia and evaluated the detection performance of these rules against the input data set and a collection of benign software. The results are very positive, with a F1 score of 0.983 against the input data set and just 13,879 false positives from 70 rules (out of which only 11 cause more than 100 FPs) on a goodware corpus spanning more than 10 TB of data.

We provide the code for YARA-Signator as open source via the GitHub repository: https://github.com/fxb-cocacoding/yara-signator, and the frequently updated YARA rules created from the Malpedia corpus can be freely accessed using the REST API: https://malpedia.caad.fkie.fraunhofer.de/api/get/yara/auto/zip.

Acknowledgments:

The authors would like to thank the anonymous reviewers of Botconf for their valuable feedback.

We would also like to thank Jakub Křoustek and his team at AVAST for their kind support of running our rules against their goodware data set.

Author details

Felix Bilstein

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
felix.bilstein@fkie.fraunhofer.de

Daniel Plohmann

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
daniel.plohmann@fkie.fraunhofer.de

References

[1] D. Bianco, "The pyramid of pain." Blog post: http://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html.

[2] C. Blichmann, "Automatisierte Signaturgenerierung für Malware-Stämme," 2008. Diploma Thesis.

[3] J. Zaddach and M. Graziano, "Bass - bass automated signature synthesizer," 2017. Github repository: https://github.com/Cisco-Talos/BASS.

[4] D. Plohmann, M. Clauß, S. Enders, and E. Padilla, "Malpedia: a collaborative effort to inventorize the malware landscape," *Proceedings of Botconf*, 2017.

[5] L. Gibelli, T. Edvin, T. Kojmnet, A. Wu, and N. Horne, "Clamav - open source anti virus engine," 2004. Website: https://www.clamav.net/.

[6] V. M. Alvarez, "Yara - the pattern matching swiss knife for malware researchers," 2014. Website: http://virustotal.github.io/yara/.

[7] I. Intel, "Intel-64 and ia-32 architectures software developer's manual," 2013.

[8] R. Edward, Z. Richard, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, 2016.

[9] C. Blichmann, "vxsig - automatically generate av byte signatures from sets of similar binaries.," 2019. Github repository: https://github.com/google/vxsig.

[10] T. Dullien, E. Ventura, S. Meyer-Eppler, T. Kornau, C. Blichmann, and J. Newger, "Zynamics," 2004. Website: https://www.zynamics.com/software.html.

[11] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 461–470, Association for Computing Machinery, 2016.

[12] F. Roth, "yarGen," 2013-12-18. Github Repository: "Github repository: https://github.com/Neo23x0/yarGen.

[13] C. Doman, "Yabin," 2018. Github repository: https://github.com/AlienVault-OTX/yabin.

[14] H. Yi, "Hyara (ida plugin)," 2018. Github repository: https://github.com/hy00un/Hyara.

[15] KoreLogic Security, "Converting ida pat to yara signatures," 2013. Blog post: https://blog.korelogic.com/blog/2013/11/15/pat2yara.

[16] W. Ballenthin, "Yara-fn," 2019. Github repository: https://github.com/williballenthin/idawilli/tree/master/scripts/yara_fn.

[17] J. Martin, j0sm1, jovimon, and mmorenog, "Yara rules," 2018. Github repository: https://github.com/Neo23x0/signature-base/tree/master/yara.

[18] F. Roth, "Yara rules from signature base," 2018. Github repository: https://github.com/Neo23x0/signature-base/tree/master/yara.

[19] M. Worth, "Open-source-yara-rules," 2018. Github repository: https://github.com/mikesxrs/Open-Source-YARA-rules.

[20] R. Wesson and SupportIntelligence, "Project icewater," 2018. Github repository: https://github.com/SupportIntelligence/Icewater.

[21] D. Plohmann, "SMDA - a minimalist recursive disassembler library for x86/64.," 2018. Github repository: https://github.com/danielplohmann/smda.

[22] N. A. Quynh, "Capstone: Next-gen disassembly framework," 2014. Website: http://www.capstone-engine.org/BHUSA2014-capstone.pdf.

[23] F. Bilstein, "Automatic generation of code-based yara-signatures," 2018. Bachelor Thesis.

[24] C. Cohen and J. Havrilla, "Function Hashing for Malicious Code Analysis," tech. rep., SEI, CMU, 2009.

[25] V. Chvatal, "A greedy heuristic for the set-covering problem," *Math. Oper. Res.*, vol. 4, p. 233–235, Aug. 1979.

[26] M. Stonebraker, "Postgresql," 1989. Website: https://www.postgresql.org/.

[27] B. Levene and J. Grunzweig, "Sure, I'll take that! New ComboJack Malware Alters Clipboards to Steal Cryptocurrency." Blogpost: https://researchcenter.paloaltonetworks.com/2018/03/unit42-sure-ill-take-new-combojack-malware-alters-clipboards-steal-cryptocurrency/.

[28] T. Micro, "ShurL0ckr Ransomware as a Service Peddled on Dark Web, can Reportedly Bypass Cloud Applications." Blogpost: https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/shurl0ckr-ransomware-as-a-service-peddled-on-dark-web-can-reportedly-bypass-cloud-applications.

[29] J. Grunzweig and K. Wilhoit, "The Fractured Block Campaign: CARROTBAT Used to Deliver Malware Targeting Southeast Asia." Blogpost: https://unit42.paloaltonetworks.com/unit42-the-fractured-block-campaign-carrotbat-malware-used-to-deliver-malware-targeting-southeast-asia/.

[30] M. Talbi, "De-obfuscating Jump Chains with Binary Ninja." Blogpost: https://thisissecurity.stormshield.com/2018/03/20/de-obfuscating-jump-chains-with-binary-ninja/.

[31] D. Plohmann, "Patchwork: Stitching against malware families with IDA Pro." Presentation for SPRING2014: https://public.gdatasoftware.com/Web/Landingpages/DE/GI-Spring2014/slides/004_plohmann.pdf.

[32] D. Plohmann, "Empty msvc," 2019. Github repository: https://github.com/danielplohmann/empty_msvc.

[33] E. Raff, W. Fleming, R. Zak, H. Anderson, B. Finlayson, C. Nicholas, and M. McLean, "Kilograms: Very large n-grams for malware classification," 2019.

[34] gbrindisi, "Gozi ISFB Sourceccode." Github Repository: https://github.com/gbrindisi/malware/tree/master/windows/gozi-isfb.

[35] A. Ivanov, "Scarabey Ransomware." Blogpost: https://id-ransomware.blogspot.com/2017/12/scarabey-ransomware.html.