universitätbonn

Automatic Generation of code-based YARA-Signatures

A practical approach for the generation of high-quality YARA-rules

Felix Bilstein

Bachelor Thesis

Examiners: Prof. Dr. Peter Martini, Prof. Dr. Michael Meier
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 bachelor thesis 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, November 5, 2018

Felix Bilstein

Abstract

While detection of malware became more mature over many years, classification of malware is still a challenge. By analyzing unpacked malware (e.g. after dumping memory), various methods for identification can be applied, e.g. YARA signatures in order to determine the malware family. Clustering the malware samples into families can be automated by applying different YARA rules to a malware pool or in an analysis toolchain. However, composing effective YARA rules is typically a time-consuming task that requires extensive domain knowledge.

In this work, we will investigate the effective automation of generation of YARA signatures, building on the disassembly of malware found in memory dumps and translating analysts' common best practices into an automated method.

We present a flexible processing chain which has the potential to create YARA rules of high quality. The concept is implemented and developed JAVA software, containing different libraries and applications which consist of highly modularized components offering a configurable framework. Moreover it was possible to create YARA rules using the developed framework which classify more than 85% of previously learned malware families correctly with almost perfect accuracy.

Contents

List of Tables

1 Introduction

Nowadays, information systems are attacked by various different attacker types on a daily basis. Many systems are compromised through the use of malware, but the malicious code is usually packed. Packing is a very common technique among anti-virus evasion strategies, malware authors allowing to bypass anti-virus solutions for some period. Packing increases the difficulty for detection, especially the identification of malware families becomes a much more complex task. Because different families may be using the same packers, those binaries have a large code base in common, even if their actual payload is completely different, since they use the same unpacking routine.

The YARA [IBM+18] framework has emerged as a de facto standard in malware classification. YARA is a highly efficient pattern-matching engine, most commonly used to scan large amounts of data. To use YARA, one must have a set of YARA rules, which are applied to the malware samples. A YARA rule is a text file containing a set of detection strings (printable strings, byte sequences, or regular expressions) and a matching condition. YARA parses these rules and the scanner application produces a binary matching result, using previously defined logical expressions like *and*, *or*, etc.

The creation of YARA rules is an expensive task that requires experience in malware analysis and expert knowledge. It is a very time consuming and complex process, especially maintaining and updating previously created YARA rules.

The aim of this work is the automatic generation of code-based YARA rules which are as concise and simple as possible, maximizing coverage for a given family while avoiding false positives. To apply YARA for static analysis approaches the usage of already unpacked samples is necessary.

In this thesis, we will investigate if high-quality YARA rules can be automatically generated when using disassembly as a basis. By generating YARA rules for malware families, the quality of the results may be increased when utilizing disassembled memory dumps of the original malware samples, because the disassembly contains information about functions, blocks, and the complete control flow graph.

Generally, it is save to assume that malware families have a very stable code base along their versions, potentially a result of incremental development. This might be exploitable to detect and cluster new versions of previously analyzed malware by deriving code-based YARA signatures that focus on the most stable parts of the family.

By using automation, these rule sets shall be created faster and unsupervised in future. The generated YARA rules have to be based on the code base to be utilized for classification of malware samples in future versions.

But since many malware families may share similar third party code (statically linked libraries such as compression or encryption algorithms), finding the right code sections for the rules becomes more challenging. For example the Zeus source code was leaked [Vis14] and after that, many subfamilies emerged, sharing a significant amount of the original Zeus code [Plo18a]. This can cause difficulties, because using this approach we will have to find the unique fingerprint of a family's stable code base and factor in source code leaks as well as shared code with different malware families.

To meet these challenges, we try to focus on the immanent, common code base within a malware family while excluding code which is used in program code of other families. Our approach consists of a process flow containing various code detection and coverage validation techniques.

In summary, we make the following contributions:

- A library for parsing SMDA [Plo18c] disassembly reports into JAVA objects, »smda-reader«
- A library for creating YARA rules with JAVA, »java2yara«
- A configurable framework for automatic YARA rule generation, »yara-signator«
- A collection of YARA signatures which classify several malware families from Malpedia [PE18] with high accuracy

More concrete the process chain includes the processing of disassembly reports, linearization of the control flow graph and building different-sized ngrams out of these data. Therefore, followed by filtering and ranking of the candidates, ending up the process with the generated YARA signatures. The process chain is basically provided by a framework of different developed JAVA programs and libraries called »smda-reader«, »java2yara« and »yara-signator«, the core component of the implementation. Each part of the framework is introduced and described by defining their exact place in the process chain, followed by a detailed specification of their exact function.

Afterwards, a set of generated YARA signatures by different ranking systems is evaluated, followed by the conclusion and a subsumption of the created result during this thesis. Finally, the results are analyzed and potential challenges as well as promising approaches are discussed further.

2 Basic Notions

This chapter describes basic techniques and terms that are highly relevant in the context of this thesis. These terms cover the general concept of program compilation and analysis using disassembly, advanced analysis of malicious software (short: malware) in general and approaches for the detection of malware based on signatures.

2.1 Compilation and Disassembly

Software development in system-programming languages like C or C++ requires following steps from source code to an executable program. When software is compiled, multiple steps are sequentially performed. Usually this is done with a front-end of modern compilers performing all tasks automatically.

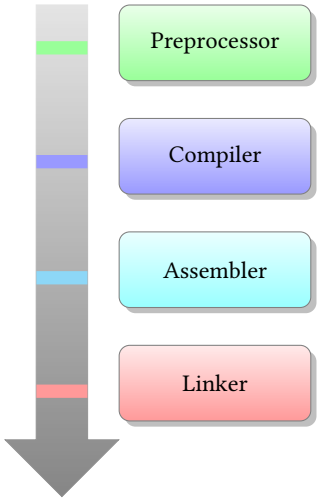

Figure 1: *Illustration showing the compiling process*

First, the preprocessor statements are executed and applied to the source code. After preprocessing, the source code gets compiled. The compiler generates code in assembly language for the target architecture, e.g. x86 or MIPS. When the Assembler is invoked object files are generated in a valid binary format, e.g. PE or ELF. After linking, all addresses are set up in the object file and the resulting binary file finally contains all information necessary for execution.

When performing program analysis on compiled code, the analyst is faced with machine-level information. An essential tool for this is a disassembler, which can reverse this process up to a certain point, i.e.

until the after-compile stage. However, it is a common situation that a lot of high-level information that was available in the source code can not easily be recovered, such as labels for functions and variables. On top of that, obfuscation techniques such as self-modifing code and code generation during runtime can make the process very complex.

For disassembling, there are two general approaches: linear and recursive disassemblers. While a linear disassembler like objdump [Fou18] reads the executable only as a linear sequence of instructions, a recursive disassembler like IDA Pro [HR18] or SMDA [Plo18c] can obtain information about the control flow while processing the binary. This is usually achieved by reading the executable multiple times and constructing a control flow graph on the base of queued jump- and call-instructions.

A linear disassembler is only of limited use in the context of malware analysis, because malicious code allocates often memory at different locations between data sections. The distinction between code and data is complex and linear disassemblers are typically not capable of treating these situations appropriately. As a contrary example, SMDA is specialized to read memory dumps and corrupt binary files (e.g. read without PE-Header). In this work we use SMDA reports for a set of binaries from the Malpedia [PE18] database which were previously dumped from memory.

2.2 Malware analysis and Anti-Analysis techniques

The following section provides basic information about malware analysis techniques and common challenges the malware analyst is frequently confronted with. While malware is a well-known problem, challenges in the analysis are still relevant today. Malware has become an important element in the growing market in organized crime [Dur14]. Due to the strong increase of the amount of malware and the professionalized, improved malware techniques realized with strong increased resources and financial sources, malware analysts are faced with major challenges. Even nation-state attackers with government-backed financing lead to an increasing participation in the modern malware area [CLR17].

2.2.1 Malware

Malware (malicious software) is an umbrella term for software referring to “any software designed to cause damage to a single computer, server, or computer network” [Cor03]. It describes malicious code in general, which is clustered into different subcategories based on the behavior of the malware, for example backdoors, viruses, rootkits, bootkits, ransomware, trojans, etc. In this work we will focus on malware as a general term as the approach is more abstract and should work independently of the malware category.

Currently the impact of malware is continously growing and specially for nation-state attackers an attractive way to achieve the actor’s goal. The attribution in general is very hard due its plausible deniability and since there are only a few examples for retribution, the risk-reward ratio is very attractive for attackers. Even if a successful attack is detected, attribution to concrete nation-state attackers with high certainty is a challenging task. There are currently over 30 different countries which develop offensive cyber-security weapons, tools and techniques. [CLR17] [Ran17]

2.2.2 Malware Analysis

Analyzing malware is generally categorized into two complementing techniques, static and dynamic analysis. In static analysis, malicious code is generally not executed natively, while dynamic analysis focuses entirely on runtime-analysis.

Within a static analysis of a malware sample the usual way of processing is generating indicators of malicious software at first. Generating and comparing the hash (typically SHA1 or SHA256) of the sample in common malware databases is an efficient way that allows the malware analyst to find out if the sample was already seen or analyzed by other researchers. Extracting the strings of a sample can lead to first indicators of any malicious nature of the regarding sample. The header of the executable binary file (e.g. PE, ELF) can give certain hints that the file is not an ordinary executable, for example it may has a corrupt checksum.

Evaluating the entropy of the sample can suggest the analyst the assumption that the sample is packed, encoded or encrypted in any way. A nearly empty import table may hint that there is some functionality hidden, especially if the malware sample calls APIs of the operating system which load dynamically linked libraries or shared objects.

Further in-depth static analysis requires disassembling and time-consuming, detailed analysis of the code structures in assembly language. After disassembling the malware sample and generating a control flow graph, the structure of the malware becomes human-readable. One major disadvantage when analyzing malware statically are missing label information of functions and variables as well as memory and data run-time information. While one is able to find strings, various information referring to the platform and following the control flow graph, the inspection of a sample becomes very complex if the sample changes itself, for example by generating new instructions into memory. Some malware changes only strings in run-time by applying XOR-based encryption, others decrypt their instructions during run-time. While the former is still easy to detect and analyze statically, the latter is very hard to analyze without executing the sample.

Dynamic analysis can take advantage of the comprehensible run-time information and data allocation in memory during run-time while it restricts the analyst to the given execution path. Important for run-time malware analysis is a contained environment. Since the malicious code is executed the analysis system will be compromised by the malware. Analysts try to counter this by using virtualization software. Another risk are possible anti-debugging techniques and anti-virtualization mechanisms which can lead to different behavior. Malware samples try to detect the fact that it is executed in an virtualized environment, anticipate this and behave differently than on actual compromised systems of their victims. For this reason malware is usually dumped after execution to get a decoded and less protected version of a sample.

Running the malicious software to analyze it during run-time is known as dynamic analysis. There are several tools available to get detailed information about changes towards the underlying operating system when malicious code is running, for example watching the windows registry, file I/O, process and network monitoring, etc. These details of running processes can be also obtained through automation using sandboxing tools like Cuckoo [GTBo18].

Especially debugging of the malicious software lets one inspect detailed information about the processes of a malware sample during run-time. Debugging offers the possibility to watch the running program with all its internals and state changes, such as instructions, cpu registers, memory allocation on stack and heap, loading of external libraries and every other possible action. This being a powerful weapon against hiding malicious actions, a major drawback is the complexity of the debugging due to the nature of assembly language and especially the instruction speed which makes debugging a very time-consuming process that requires the knowledge of an expert. Breakpoints allow to stop the running process at certain points of interest for various analysis purposes.

2.2.3 Packing and Protecting

Packing is a technique used by malware authors to protect their binaries from detection and analysis. There are several tools available in public domain, such as UPX [OM17], AsPack [Inc18f], Yoda Crypter [ash13], and many more. However, many malicious actors will develop or rely on custom protection software to ensure that they remain undetected as long as possible [Lab17]. Before one can analyse the payload of a packed malware sample, the sample has to be dumped or unpacked. For some packers standard unpackers are available, for example the tool UPX has a built-in unpacking option. If the sample is protected with a packer for which no unpacking tool exists, one way to speed up the process and circumvent the packing is to create a memory dump during runtime once the payload is unpacked in memory as shown in Figure 2.

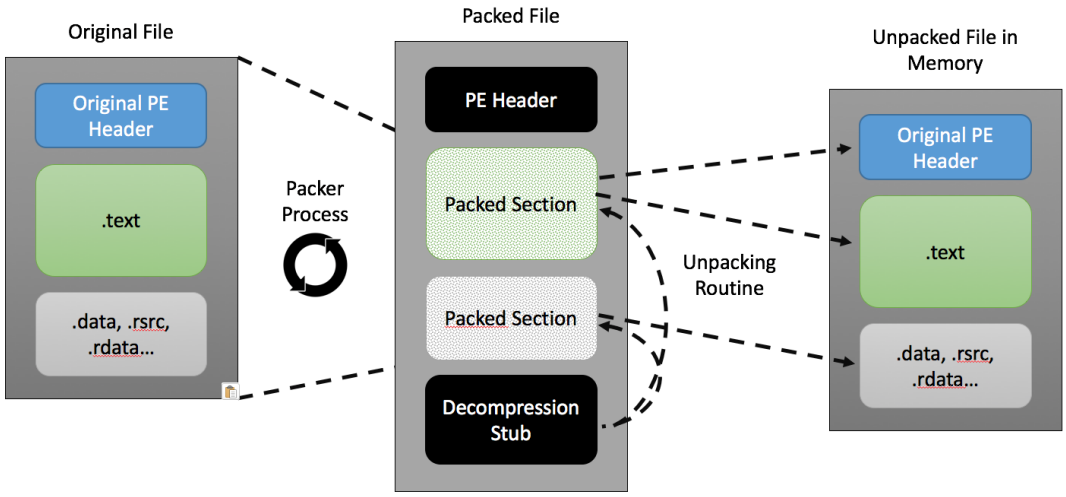

Figure 2: *Illustration showing a common packing technique [McA18]*

The decoded file in memory can be stored for further analysis purposes. In general these saved files can be made executive again by reverting several modifications applied during the loading of the program, e.g. reconstructing offsets according to the section table, rebuilding the import table, and removing address relocations. There exist several packers which do not embed the original PE header of the binary since the header is not needed during execution.

While packed binaries are difficult to inspect statically, there are various techniques which make the unpacking process harder or even impossible. One approach is to decode special functions into a virtual machine instruction set and ship the malicious software with a virtual machine interpreter.

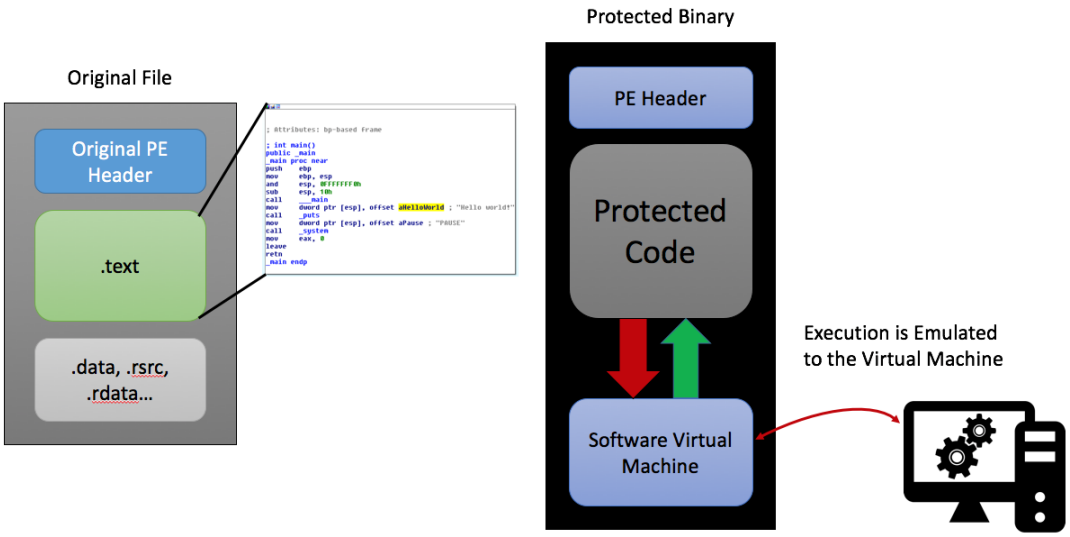

Figure 3: *Illustration showing a protected binary using virtualization techniques [McA18]*

A "code-virtualized" binary like shown in Figure 3 cannot be unpacked easily because the original instructions are encoded into the virtual instruction set before shipping the malware to the victims. A memory dump would show the same instructions as the file on disk before execution. To analyse a sample protected by a virtual machine the analyst needs expert knowledge of the underlaying virtualization technique for further inspection.

2.2.4 Memory Dumping

Memory Dumping is a technique to save the memory of one or many processes to analyze the process layout statically after dumping. Many malware families have different protection mechanisms against static analysis. A common procedure to evade these protection mechanisms is generating a memory dump of the running malware by saving the memory of active and suspicious processes. There are two methods of memory dummping, physically and process-based memory dumping from virtual process memory. Physical memory dumps can be performed inside a running system (ReKall [ACC+18]) or with special malware analysis systems, for example Cuckoo Sandbox [GTBo18]. The mapping between physical and virtual memory is transfered automatically or with supporting software for manual interaction with, for example, Volatility [ACL+18]. The resulting memory dump can be disassembled and statically analyzed afterwards to save the analyst's time towards manual debugging. To execute a memory dump, the structure has to be converted into a for the target operating system valid executable.

2.3 Ngrams

Ngrams are defined as consecutive, fixed-length subsequences taken from a given ordered list of items. Building ngrams in the context of malware detection and classification is a common technique used by malware researchers [ERC+16]. Figure 4 shows an example how ngrams of the size four using words as unit can be created for a popular sentence known from typography testing.

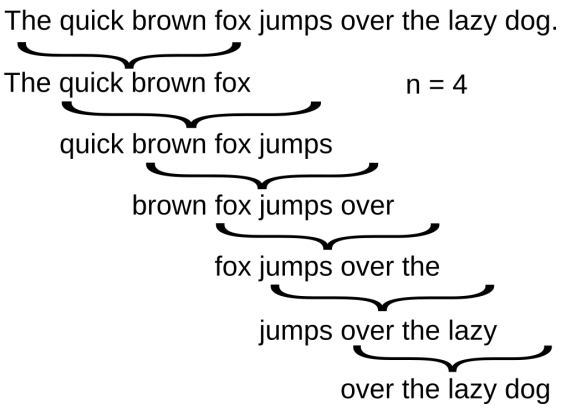

Figure 4: *Demonstration of the ngram building procedure.*

Ngrams are often used in analysis of byte streams and data processing. For this work, ngrams of instruction bytes will be essential for generating YARA-rules as signatures for malicious code.

2.4 Signatures

Signatures are descriptions which can detect certain characteristics in a pre-specified syntax. They can be designed for example as a group of byte sequences. Signatures are used in our approach to identify binary blocks of code or data within a specified area. They are widespread in the context of malware detection and identification, packet inspection within information technology networks, e-mail appendix scanning and various other.

Distributing YARA rules is a popular approach to collaborate with fellow researchers and security analysts via an open and standardized format for signatures. The corresponding signature is translated into a pattern-matching state machine which is applied on data trying to match them.

A YARA rule is a plain text structure which can be processed by YARA [IBM+18] for scanning files, directories, or processes. They usually contain a “meta”, “strings” and “condition” section. Meta information in the “meta” section can offer various information regarding to the processed data set, author, date and other data although this section is not mandatory. The “string” section contains bytes or strings which are used for classification and the “condition” part is the decision rule. An example for a YARA rule is shown in Figure 5:

```
rule win_xtunnel {

    meta:
        Author = "Felix Bilstein - fxb at cocacoding dot com"
        Date = "2018-10-11"
        Description = "autogenerated rule brought to you by yara-signator
            "
        Reference = "Malpedia data set"

    strings:
        $ = { 8940088b4644c64014018b4e44c6411501 }
        $ = { 8940048b46448940088b4644c6401401 }
        $ = { 8b46448940048b46448940088b4644c6401401 }
        $ = { 0385440100008944241083bd5001000000 }
        $ = { 89008b46248940048b4624894008 }
        $ = { 014e0c014e102bc1894604f7870001000000020000 }
        $ = { 8b4e0c89480c8b56208950208b4e10 }
        $ = { 8b4e048948040fb75608668950088b4e18894818 }
        $ = { 03d3894424188b8424bc0000002b44241489442414 8b442418 }
        $ = { 833800751683c8ff }

    condition:
        7 of them
}
```

Figure 5: *Example for a YARA rule covering the xtunnel family.*

There exist various proprietary signatures for malware detection, common examples are malware databases from anti-malware software vendors.

While anti-virus signatures are a widespread technology there exist various other signature types to detect malicious activities, for example within the scope of network security for Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS) and mixed technologies.

3 Related work

Automation of YARA rule generation has been approached by several researchers so there are several concepts to be looked at closely. While some approaches are more mature than others, each covers special aspects in YARA rule generation which are worth to be mentioned. The complexity of the different projects varies a lot. Many are small python scripts for IDA Pro [HR18], while yarGen [Rot18b] and yabin [Dom18] are more advanced projects.

Many of these solutions still require manual analysis or significant expert knowledge in malware analysis. The proposed methodology presented in the following chapters of this thesis is an improvement and adjusted solution that has been derived by cherry-picking the most promising core ideas of the related work outlined here.

This chapter is divided into two sections, the signature generation projects and the malware signature collection projects. Some of the signature collections provide YARA signatures which were automatically generated in the first place while others are a set of contributed YARA rules created by individual analysts, researchers or IT security teams.

3.1 Signature generation

There exist various publicly available signature generation projects which offer full- or semi-automatic signature generation techniques. The projects mentioned in the following section are a selected set of actively developed and most popular projects.

3.1.1 yarGen

The yarGen project [Rot18b] is a YARA rule generator which is capable of creating YARA rules from a binary file database using string and byte sequences. Those YARA rules are derived from a data pool containing binary files, e.g. PE-files. With a focus on strings these rules will be able to detect the prior processed families.

Generalization is important to more accurately detect future versions of the malware which makes a generic rule better fitted for this approach. While yarGen is optimized to work on binary files, the approach to be developed in this thesis will be optimized for working on disassembly. By using this new approach the generated YARA rules can be more precise, since the disassembly of unpacked malware samples contains more useful information provided to the tool.

3.1.2 YaBin

YaBin [Dom18] is a project using another approach for automatically creating YARA signatures. It searches for function prologues and stack frame creation to detect the beginning of a function, for

example “55 8B EC” which translates to “push ebp; mov ebp, esp”, saving the current stack frame base pointer to initialize a new stack frame. This is part of a calling convention which is generated automatically for most functions by Microsoft Visual Studio. After that, each sequence is validated against a white list containing over 100 GB goodware to examine that the selected functions from the malware sample are not part of common libraries.

Analyzing these function can lead to unique YARA rules for the detection of malware families. In our approach, we will follow a similar idea to identify relevant functions, but more differentiated since the disassemblies already contain a complete function graph of the malware sample, effectively providing a far more complete basis to work with in signature generation.

3.1.3 pat2yara

The project Pat2YARA [Sec13] tries to solve the problem of analyzing stripped ELF-Files. In a stripped and statically linked ELF-binary there are no debug symbols left which makes reverse engineering of such files very time-consuming. Pat2YARA is basically a conversion tool to port Pat-files to YARA signatures. The Pat files are generated with FLAIR and IDA Pro by searching in the binary for specific pattern of known libraries.

Those files are converted to YARA files and after this step one can use these YARA rules to identify used libraries in the stripped binary. This idea can be useful to detect common library functions in ELF but also in PE-files. This can help to figure out which code fragments that are commonly generated by compilers or parts of shared libraries that should be avoided in signature composition.

3.1.4 YARA-FN

YARA-FN [Bal16] is a project which can generate a YARA rule from basic blocks of a function. It has the ability to wildcard addresses and ignore jump instructions. This is a common used technique in signature generation since addresses of the binary can change heavily between different execution environments. This problem affects jump instructions since they can be absolute which means they contain absolute addresses. These aspects are covered within our YARA rule generation processing chain in chapter 4 and in detail in subsection 4.2.5.

3.1.5 Hyara

Hyara [Yi18] is an IDA Pro plugin for supporting the process of YARA-rule creation. It is designed to accelerate building and improving YARA rules by the malware analyst. If the analyst provides start and end addresses the tool can create YARA rules automatically. The project offers various supporting features such as string extraction which can help the analyst to create YARA rules covering unique strings of the malware sample.

3.2 Signature Collections

Among YARA signature generation projects there exist various collections of YARA rules. In this section we will cover some selected, public available YARA signature collections. It shows the lack of powerful and normalized malware data set in conjunction with high-quality malware signatures which we try to address in this work.

3.2.1 YARA rules

The public available YARA rule repository »YARA rules« [Mjjm18] contains several signatures to detect malicious software but also packer and protector, malicious documents, web shells, emails, exploit kits, and others.

The project itself is community driven while the quality is assured by four project maintainers. It offers over 500 YARA rule files containing over 10000 rules for detection in total. Due its size, the project is categorized into different subsections:

- Anti-debug and Anti-VM
- CVE Rules
- Cryptographic
- Exploit Kits
- Malicious Documents
- Malware
- Packers
- Webshells
- E-Mails
- Mobile Malware

Anti-debug and Anti-VM is about covering and detecting miscellaneous anti-debugging and virtual machine detection routines within malicious software. Common Vulnerabilities and Exposures (CVE) signatures try to identify CVE-based attributes within samples. The Cryptographic Engine detection helps analysts to detect a malware's capability of using cryptographic routines or public available implementations.

The YARA signatures for packers try to detect known packers and can help analysts in the unpacking process. The rules for malicious documents cover various malware types and samples which are embedded into documents. The most relevant component of the YARA rules collection for our topic is the malware section. It contains over 1800 YARA rules to detect malware families, miscellaneous loaders and plenty malicious software modules.

3.2.2 YARA rule signature base

The public available YARA rule repository »signature base« by Florian Roth [Rot18a] contains over 380 signatures to detect malicious software focusing on APT malware. The collection is based on manually created YARA rules by malware analysts and offers a higher quality than the much larger YARA rules project mentioned in the previous section.

3.2.3 Project Icewater

The YARA signatures from Project Icewater [WS18] are generated by an AI software based on the genetic derived from the human DNA concept from SupportIntelligence [Sup18]. The project itself contains over 16000 YARA rules, although the rules are sorted on the base of the clustering by the generation software.

While the project collects a large number of YARA rules over clusters they use their own labeling system that makes interoperability between common malware labeling and Project Icewater difficult. Even though these YARA rules are public available many of them are declared as tlp:amber and protected with a custom license. The complicated labeling and the proprietary license bring the project in an outside position.

3.2.4 Open source YARA rules

The project »Open-Source-YARA-rules« [Wor18] by Mike Worth is a YARA signature collection of rules found in public domain. The data set is ordered by various companies in the IT security field which are the main sources of the collected YARA signatures. The project itself offers various meta information about the covered malware such as actor group name and family as well as fine granular YARA rules for each covered specific threat. Even YARA rules for advanced IT security tools like Metasploit and Rekall modules and malicious document files.

4 Concept

The automatic recognition of malware is an important step to defend against modern cyber attacks. While the number of recorded unique malicious files exploded over the years, many of these samples can be clustered into one of potentially just a couple thousand families. This clustering is very important to reuse already invested time in analysis to deal with the increased malware distribution of today. The analysis of malware allows us to defend advanced cyber attacks from cyber crime syndicates, APT groups and nation-state attackers. The purpose of this work is to target the increased modern cyber attacks in conjunction with malware campaigns by defining a automation technique to generate high-quality malware signatures in a shorter amount of time.

While there are generally previous works available (cp. Chapter 3), none of them uses the full potential of code-centric signature generation. The automation of YARA signature generation is important since there are many distinct or sometimes potentially related malware families and each family has typically multiple versions. Manual rule generation requires expertise and time, especially the implied frequent adjustments to keep signatures up to date. We therefore propose a method that is capable of generating signatures efficiently and with a high quality, i.e. avoiding false positives and negatives as good as possible.

Obviously, YARA rules which cover a single sample are trivial since they can be arbitrarily expressive and long. Our approach instead aims at understanding the degree of uniqueness of code fragments across a wide range of families first. Then, based on this information, rules that consist of a set of these fragments are derived that are potentially capable of even forward-detecting previously unseen versions of a given malware. Our approach instead is able to generate short and high quality YARA signatures for complete families which can even detect samples of a family that wasn't provided or computed in our processing chain previously while minimizing false positive rates as shown in the evaluation (cp. Chapter 6).

4.1 Proposed Method

Malicious software is a very inhomogeneous field. There are miscellaneous malware families which differ widely in their appearance. We establish a modular, adjustable processing chain to gain the needed flexibility covering these aspects. The key components (filtering, ranking system) are configurable to offer high flexibility.

Several aspects of the related work towards this topic are covered to generate YARA rules. In the context of this work, we will make heavy use of the malware corpus Malpedia as a baseline for our method. A major reason for this is the already ordered and aligned data set of over 600 malware families with over 1800 samples. The division of the data set into distinct families lets one easily compare instructions within families and above this range to other families. We expect special challenges with families that share a significant part of their code, such as families derived from the leaked Zeus source code [Plo18a].

To apply our method it is required that samples are already sorted into families. In this thesis we will use Malpedia as it is the largest public available resource for organized and clustered malware of this kind. This is not a drawback since it is possible for many cases to generate YARA rules automatically for new inserted malware in this corpus by applying all rules detecting the sample's family without manual analysis. To be more concrete, the processing chain needs already unpacked or dumped malware samples and their disassembly reports. We use the disassembly output of SMDA [Plo18c] as it is a recursive disassembler specialized on memory dumped software. These steps are automatable as well but this is beyond the scope of this work. The method we propose is a generic approach to generate detection signatures starting on a already clustered malware pool offering a fast recognition of new malware samples and potentially add them to the base pool to increase the precision of the signatures to cover new samples.

A detailed explanation of the processing chain and the concrete design of the steps are described in the following section 4.2.

4.2 Processing Steps

The general idea is based on the assumption that when considering a malware family to rely on a commmon code base, at least a certain fraction of its code will remain constant across a number of versions and that this fraction is also unique to this family. We therefore propose a code-based method for rule generation that can be divided into a range of consecutive, modularized steps. To generate and analyze those data, a disassembly report for each sample is used to identify the instructions used in the respective specimen. These reports are parsed into a machine readable format to obtain further information for analyses steps as Figure 6 shows in the first step.

The instructions from a sample are converted into linearized assembly by flattening the control flow graph into linear and sequential instructions which is important as YARA performs a byte-wise check if the strings of the signature are in a malware sample. Linearized disassembly is transformed into ngrams with a customizable length. Using different ngrams at the same time can result in YARA signatures with better access to specificity. While under some conditions longer ngrams can be useful for detecting common instruction of malware families, shorter rules can be combined together for more precise results. Considering previous work [PCEP18] we estimate hundreds of millions ngrams in sum of all samples need to be filtered and expect lots of ngrams belonging to shared code between families which is counterproductive for reaching the target and the amount of data is too large to proceed without filtering. The main advantage is to detect immanent code within malware families while removing code which is shared between malware families. The non-immanent, shared code between different families should be avoided since it can lead to false positives in the created YARA rule which we try to reduce as much as possible.

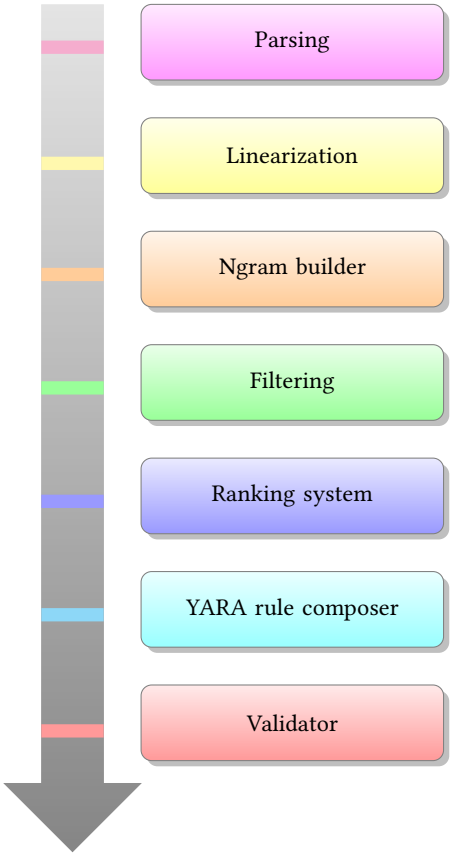

Figure 6: *Iterated modules of the processing chain.*

While the filtering system can reduce the amount of ngrams we cover, the resulting data will still be to big to create a single YARA rule out of it so the ranking system is another step to reduce the amount of data for signature generation. The ranking system itself is determining ranking scores for each ngram. Ngrams with many arithmetic or logical operations for example might be scored higher while instructions using memory might be ranked with a lower score. Finally, the resulting ngrams are converted into a valid YARA signature by the YARA rule composer as mentioned in Figure 6. Then, the YARA rules are validated against the malware base corpus to check if the generated rules are covering the respective samples. Although this step is part of our processing chain (cp. last step of Figure 6) it is not mandatory for signature generation. This step is useful for the review and validation of the generated YARA rules.

The following subsections provide detailed information about the generation steps of the processing chain in Figure 6 as mentioned before.

4.2.1 Parsing

The first step in the processing chain is to parse and normalize the input data to make it usable for the following steps. Most disassemblers provide reports that can be machine processed. The disassembly reports used as input data are converted into a machine readable format.

In this work, we will be using SMDA [Plo18c] as the disassembler of choice because it fits the input data best. However, it should be equally possible to implement the same methodology using e.g. IDA Pro or radare2. SMDA reports are JSON encoded files from the recursive disassembler SMDA. A report

contains several elements of descriptive meta information for example file name, hash, a summary of API calls, functions and other as well as a complete control flow graph of the dumped memory from an analyzed sample. Figure 7 shows an example SMDA report.

```
1  {
2   "architecture": "intel",
3   "base_addr": 268435456,
4   "bitness": 32,
5   "execution_time": 1.5418229999995674,
6   "family": "win.iap",
7   "filename": "ffcb650b28719d3bde1b032b14cfe7f5d7f2a73...",
8   "sha256": "d4a3c07c4fef7984e0ff1a63eaa0a4ad5b62c9ff0...",
9   "smtx_version": "0.1.0",
10  [...]
11  "timestamp": "2017-11-18T20-10-20",
12  "xcfg": {
13   "268439552": {
14    "block_refs": [],
15    "blocks": {
16     "268439552": [
17      [ 268439552, "ff742404", "push", "dword ptr [esp + 4]" ],
18      [ 268439556, "6a00", "push", "0" ],
19      [ 268439558, "ff3538d40010", "push", "dword ptr [0x1000d438]" ],
20      [ 268439564, "ff1504b00010", "call", "dword ptr [0x1000b004]" ],
21      [ 268439570, "c20400", "ret", "4" ]
22     ]
23    },
24    "num_inrefs": 34,
25    "offset": 268439552,
26    "outrefs": []
27   },
28   "268439573": {
29    "block_refs": [],
30    "blocks": {
31     "268439573": [
32      [ 268439573, "ff742404", "push", "dword ptr [esp + 4]" ],
33    [...]
34  }
```

Figure 7: *A typical machine-readable disassembly report, in this case generated using SMDA [Plo18c].*

The information in these files is varying since some files do not have a control flow graph or instructions since they were implemented in a intermediated language, for example ".NET" binaries. These files are skipped and not processed any further since they are out of scope for this work although they could be processed using other disassembler or analysis tools because SMDA is the limiting factor.

The SMDA files are processed from a top level view to bottom, first parsing the header, header summary and the control flow graph. The control flow graph is a high level structure containing functions. These functions contain an offset, blocks and miscellaneous other information. The blocks are an entity of instructions (cp. Figure 7, line 15). An instruction is a three-tuple containing offset, mnemonics and opcodes (cp. Figure 7, line 32). The offset is an integer value relative towards the base address. Mnemonics are the assembly instructions of the current instruction which were disassembled by SMDA and the opcodes are the bytes of the assembly instructions.

4.2.2 Linearization

Linearization is an important step required for rule derivation since these rules cannot capture interpreted information such as instruction control flow but are instead applied to byte sequences in the raw memory buffer. Therefore, the structure of basic blocks has to be simplified into a linear instruction sequence of the binary. First, all instructions are ordered by their offset. If the next offset is the opcode length added to the current offset then the next instruction is a seamless successor. Whenever gaps occur in this sequence, a code cave or data is found between the instructions which are skipped (cp. Figure 8).

The control flow graph is linearized and cut between code caves. Code caves can occur if the compiler creates padding bytes between instructions or the operating system loader maps instructions into different memory sections. Especially malware in the wild has often protection and obfuscation techniques which include code creation by loading hidden instructions into unusual memory sections. After dumping the memory of the malicious process the memory is very often not properly aligned, for example executable code generated on the stack or heap during run-time.

```
1  37497441    8bec         mov     ebp, esp
2  37497443    83ec10       sub     esp, 0x10
3  37497446    53           push    ebx
4  37497447    56           push    esi
5  37497448    8d75f4       lea     esi, dword ptr [ebp - 0xc]
6  37497451    83c8ff       or      eax, 0xffffffff
7  37497454    32db         xor     bl, bl
8  37497456    e8ad95ffff   call    0x23bc022
9  37497461    84c0         test    al, al
10 37497463    741d         je      0x23c2a96
11 37497465    ff75f4       push    dword ptr [ebp - 0xc]
12 37497468    8b5dfc       mov     ebx, dword ptr [ebp - 4]
13 37497471    8b4508       mov     eax, dword ptr [ebp + 8]
14 37497474    6800000200   push    0x20000
15 37497479    ff750c       push    dword ptr [ebp + 0xc]
16 37497482    e824ffffff   call    0x23c29b3
17 //code cave
18 37497494    5e           pop     esi
19 37497495    8ac3         mov     al, bl
20 37497497    5b           pop     ebx
21 37497498    c9           leave
```

Figure 8: *Linearization of a disassembly stub.*

Figure 8 shows an extract of linearized disassembly after the linearization process of a malware disassembly report. The left column contains the offset of each instruction, the middle column the opcodes of each instruction and the right column shows the assembly instruction in human-readable mnemonics. This resulting form allows the further processing to generate ngrams out of the linear disassembly.

4.2.3 Ngram building

The ngram generator creates ngrams of a predefined integer “n” from the previously generated linear disassembly. Ngrams are built containing n instructions of the sample without overlapping the code caves. Code caves can occur for a number of reasons.

Some compilers use padding to leave a place in the binary file for hot-patching in case of programming errors. Other code caves can occur when the malware is loaded by the operating system into the process memory which leads to a different memory alignment between the sections as in the binary file format which can lead to code caves between instructions especially in a malware dump. A code cave might was created in the binary for example while mapping the executable file into the process memory which might change from execution to execution.

If a code cave is found, the ngram building tool stops and leaves the last instructions out to not create any ngrams overlapping the code cave. Signatures containing instruction sequences over code caves would result in bad detection rates since the code caves have not always the same length in the context of malware dumps.

An instruction is defined the same way as in the smda reports to retain the information, containing the offset, opcodes and a instruction tuple. For the signature creation we remove the offset and instruction tuple keeping only the opcode bytes. Table 1 shows the creation schema exemplary for the ngram sized four and five based on the previous linearized disassembly shown in Figure 8.

n	line	ngrams derived from Figure 8
n = 4	1	#56#8bec#83ec10#53
	2	#8bec#83ec10#53#56
	3	#83ec10#53#56#8d75f4
	4	#53#56#8d75f4#83c8ff
	5	#56#8d75f4#83c8ff#32db
	6	#8d75f4#83c8ff#32db#e8ad95ffff
	7	#83c8ff#32db#e8ad95ffff#84c0
	8	#32db#e8ad95ffff#84c0#741d
	9	#e8ad95ffff#84c0#741d#ff75f4
	10	#84c0#741d#ff75f4#8b5dfc
	11	#741d#ff75f4#8b5dfc#8b4508
	12	#ff75f4#8b5dfc#8b4508#6800000200
	13	#8b5dfc#8b4508#6800000200#ff750c
	14	#8b4508#6800000200#ff750c#e824ffffff
	19	#5e#8ac3#56#c9
n = 5	1	#56#8bec#83ec10#53#56
	2	#8bec#83ec10#53#56#8d75f4
	3	#83ec10#53#56#8d75f4#83c8ff
	4	#53#56#8d75f4#83c8ff#32db
	5	#56#8d75f4#83c8ff#32db#e8ad95ffff
	6	#8d75f4#83c8ff#32db#e8ad95ffff#84c0
	7	#83c8ff#32db#e8ad95ffff#84c0#741d
	8	#32db#e8ad95ffff#84c0#741d#ff75f4
	9	#e8ad95ffff#84c0#741d#ff75f4#8b5dfc
	10	#84c0#741d#ff75f4#8b5dfc#8b4508
	11	#741d#ff75f4#8b5dfc#8b4508#6800000200
	12	#ff75f4#8b5dfc#8b4508#6800000200#ff750c
	13	#8b5dfc#8b4508#6800000200#ff750c#e824ffffff
	14	
	19	

Table 1: *Example of ngram creation derived from Figure 8*

The ngrams are builded from Figure 8 where each beginning line is mentioned in table 1. As one easily see the instructions interrupted by code caves are left out. After building, the ngrams are processed within a database to filter for promising characteristics and find possible candidates for the YARA signatures.

4.2.4 Filtering

The next step in the processing chain (cp. step four in Figure 6) is a filtering process to reduce the given input data from the previous step. First, the provided ngrams are clustered by their size. The meta data of the given input is clustered in mappings for their count, samples and families. An ideal way to provide the meta data to the processing chain would be a hash map data structure, offering fast insertion and access.

However, since the provided input contains more than 1800 sample and over 600 family meta data, we chose a database as a persistent storage for the large amount of data. A database offers the scalability and flexibility which is needed for the processing chain and its miscellaneous, configurable modules.

The first filter applied to the input data is a filter for unique ngrams to detect ngrams which occur in one malware family only while not covering others. This reduces the given data set and offers a good base for finding promising candidates.

The left data is randomized to increase the probability to cover different code sections of a sample. This minimizes the risk of using a dedicated code section only, which can lead to a less generic signature.

4.2.5 Ranking System

The input data from the filtering system will potentially still yield a large number of candidate ngrams for the YARA rules which is the reason why we need another data reduction step. The ranking system described in this subsection is used to rank each ngram candidate with a score and reduce the candidate pool by filtering out a configurable amount of ngrams with a lesser ranking score than other candidates.

The ranking system itself works by comparing and sorting all candidates by their individual score. Due to the modularized design different ranking modules can be applied in a sequence, leaving the best candidates in the ranking pool. The first step is setting a base score for each candidate which is left at this level in the processing chain. Then different ranking modules can be applied to each candidate, adjusting the given score of the ngram.

The score of an ngram can e.g. be reduced when detecting pointers to memory addresses or increased when detecting arithmetical or logical operations. While pointers to memory addresses may lead to YARA rules which detect only a single dump, the prior filtering should eliminate those candidates in advance if enough samples are in the pool set. A focus on arithmetical or logical operations may increase the quality of the YARA signatures since these instructions are very rare in comparison to data handling instructions as described in Section 6.2.

4.2.6 YARA rule composer

In this subsection, the YARA rule composer is described. The processing step contains the exportation of all ngrams left after the ranking system step into valid YARA rules. Before rated ngrams are inserted into the YARA rule templates to follow the special syntax of the YARA toolchain. The generated rules should be capable of detecting malware samples as well as recognition of their families.

4.2.7 Validator

The Validator evaluates the generated YARA rule set. Even though this step shown to be a part of the processing chain in Figure 6 it is not a mandatory processing step to create YARA signatures. The YARA rule validation tool executes YARA over the entire malware corpus and parses the output as generated during the matching procedure using YARA itself. This data can be used for further statistics and to review the quality of the generated YARA rules.

4.3 Discussion

The generated YARA signatures of the processing chain should be able to cover nearly all of the dumped malware samples from the provided input sources since they are derived from the input. Possible false positives may occur when provided data was not covered in the disassembly reports although this is very unprobabilistic because of the large ngram base and the filtering of immanent ngrams against shared code. More likely they are created by the ranking system trying to generalize YARA rules by choosing candidates which might occur in samples of other families as well.

However, the evaluation shows that it is unlikely to match wrong families of the provided malware base pool when using only ngrams which are unique within a malware family for YARA signature generation. More interesting is the validation of the YARA rules against the complete malware database not only containing previously processed samples.

The following Chapter 5 describes the realization of the concept which was described in this chapter as well as detailed design decisions of the implementation. The central idea introduced in Section 4.1 and more precisely shown in Figure 6 presenting the processing chain and how each step is implemented concretely will be discussed in Chapter 5.

5 Implementation

In this section, a description on how the concept presented in Chapter 4 has been implemented into a concrete prototype will be presented. As a general design choice, we use JAVA as a programming language.

This chapter contains an overview of the design choices for the project and implementation details where considered helpful. Generally, the project is split into two JAVA libraries and a single executable. For dependency resolution the build system Apache Maven [Tea18] is used to package the libraries »smda-reader« and »java2yara« into JAVA libraries. The main package is the executable project »yara-signator«.

To customize usage of the software one can define a setup in the configuration for the main method in »yara-signator«. It is possible to specify different lengths of ngrams, filters and ranking system configurations to create the YARA rules based on the purposes of the user.

Due to the flexibility by the design choices based on the implementation of patterns like facade, strategy and factory method it is possible to combine different choices within the generation steps. This leads to great flexibility within the framework while providing an automation process as well as letting malware analysts configure each step, combining their expert knowledge with a strong generation engine.

5.1 Project Overview

Figure 9 shows the matching between the concept and the implementation view. To keep the design modular and abstract, some functionality is split into the different libraries »smda-reader« and »java2yara«. The core of this framework is the executable program »yara-signator« which handles the communication and data flow between each processing step as well as the connection between the process and a database. We have chosen MongoDB as NoSQL database as it suits our needs well. It is a mature and scalable system and excels in rapid prototyping scenarios such as the context of this thesis.

To give a general outline, Figure 10 is a diagram that summarizes the relationship of all major components of the software. It is inspired by UML syntax, combining aspects of a packet- and a deployment diagram. The following Figure 10 gives only a short overview of the implementation of the concept.

The main project is the project »yara-signator« which generates the YARA rules from the provided smda reports. It depends on the two JAVA libraries »smda-reader« and »java2yara« which are developed in the context of this thesis as well.

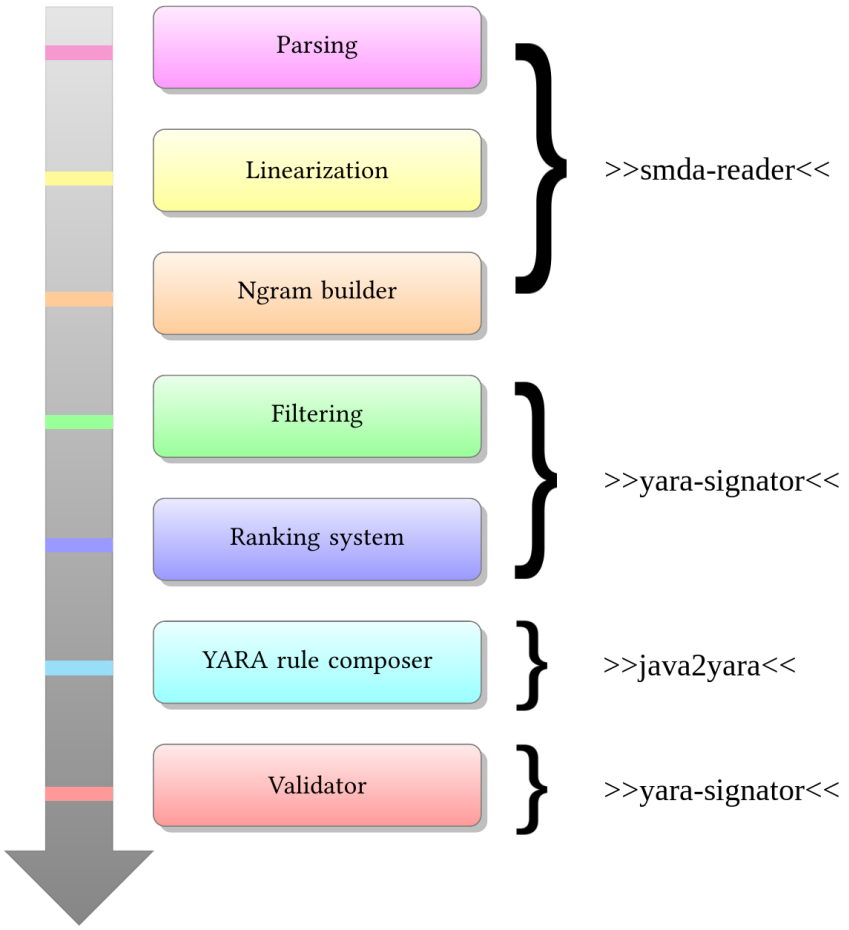

Figure 9: *Graphic showing the match between processing chain and implemented framework*

The SMDA file processing library »smda-reader« offers the possibility to read in the SMDA files for further processing. Its function consist of the function- and instruction parsing of the disassembly reports from SMDA, although other formats would be possible as well due to the modular design. The other functionality allows the configurable creation of ngrams and statistic evaluation functions for its processed data. It takes place in the processing chain in the steps “Parsing”, “Linearization” and “Ngram builder” of Figure 9.

Core component of the software suite is the JAVA project »yara-signator« which handles the steps “Filtering” and “Ranking System” of Figure 9 as well as the inter-process communication between the artifacts, libraries and database management system. Miscellaneous statistical evaluation functions are part of the project as well to provide access to an interface for the YARA rule quality measurement. The database binding is part of the program to provide a persistent storage of the data which allows working around potential RAM limitations.

The “YARA rule composer” shown in step six of Figure 9 is realized via the developed library »java2yara« within the processing chain. The library »java2yara« offers interfaces and methods for the creation of YARA rules by supplying the necessary information such as meta data like author or creation date and the covered YARA strings or byte sequences.

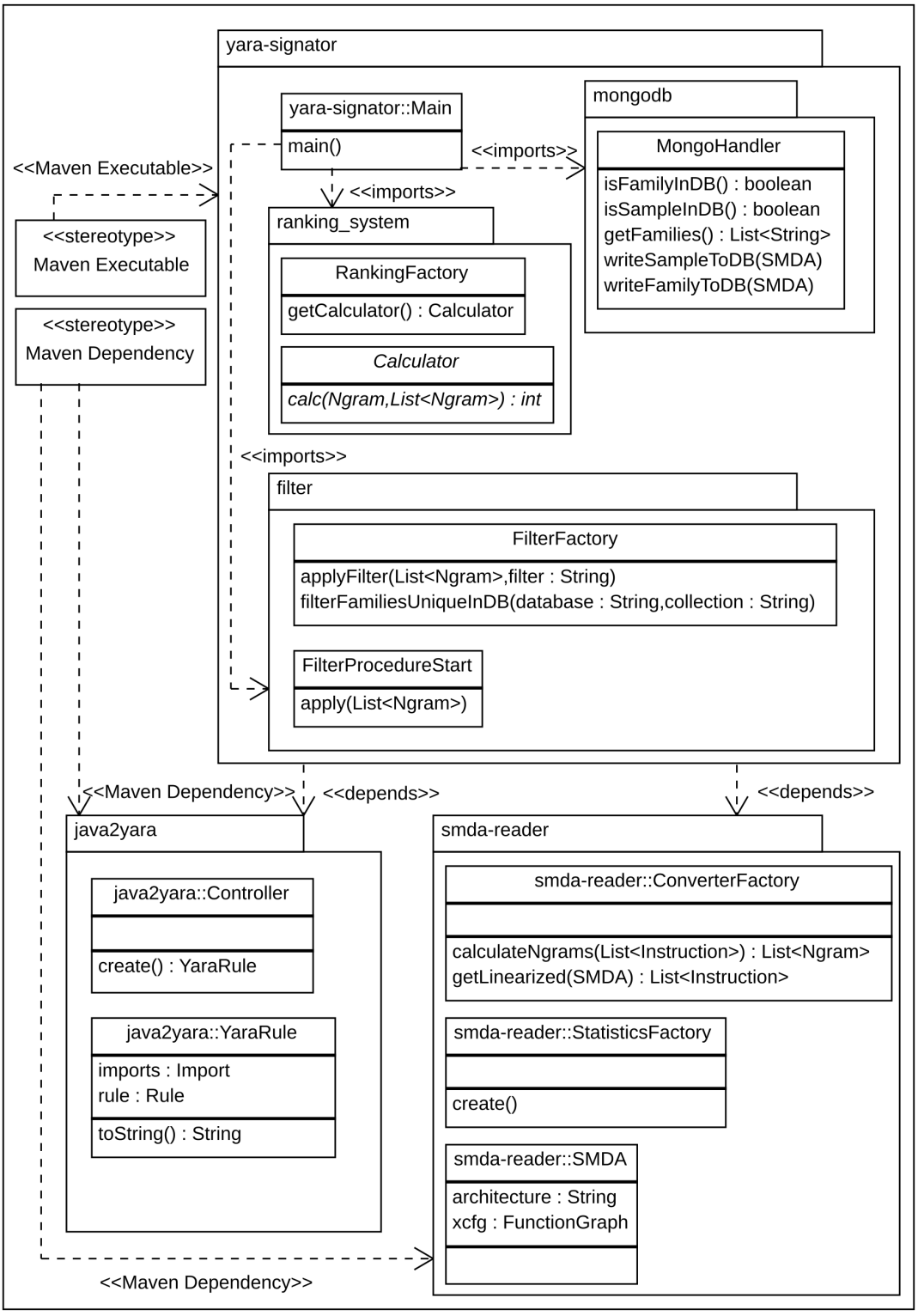

Figure 10: *Diagram showing the basic structure of the project*

The project is designed modularly to offer most flexibility for the evaluation. Therefore, each step in the processing chain is constructed within packages. Most packages have classes for the creation of each class inside the packages, usually called “Factory”. For interaction between packages, each package has an API object for access and interaction from the outside to encapsulate classes which are only needed inside their package (cp. Figure 10, MongoHandler, Controller and other). These “facade”-objects are in this framework usually identified with “Handler”, “Controller” and in some cases the “Factory” classes are constructed as API, too.

The »RankingFactory« class offers the most powerful ranking modules and therefore the biggest impact on the resulting YARA signatures, such as i.e. ranking based on the appearance of an ngram within a single malware family. The user can configure the software to ensure the YARA rules can be obtained with a focus on the user’s preferences.

5.2 SMDA-READER

In this section the JAVA library »smda-reader« is described - a library for processing SMDA reports in general and creating ngrams based on the processed instructions. The library is capable of reading every entry of an SMDA report and building a complete control flow graph of all instructions provided by it. This is the most important task in the parsing step (cp. Section 4.2.1).

Since the data in the SMDA reports is JSON formatted a JSON-parsing library is required for data processing. The library »gson« by Google [Inc18b] is a fast and stable JSON-parsing library for JAVA applications. Maintainers of the project offer an Apache Maven library which allows fast integration into the project. We chose »gson« because of its good documentation and the fact that it allows JAVA reflection-based object generation for predefined classes instead of using annotations which leads to an efficient and fast development process. The library is publicly available and continuously developed since 2008 [Inc18b] and licensed under the Apache License. It supports JAVA object serialization from JSON data which is an important step when storing persistent data in the database and restoring it back.

The control flow graph can be flattened into linearized disassembly which is the key point for the linearization step in Figure 9 in Section 4.2.2. It summarizes the code blocks into a list of instructions, which are a three-tuple of offset, opcode bytes and assembly language mnemonics. These blocks are split if a code cave occur, concatenating all instructions consecutively and appending them to the list else.

This linearized disassembly is the input for the next step of the processing chain, the ngram generation (cp. Section 4.2.2). Ngrams are created from the linearized disassembly by concatenating a predefined amount of »Ngram« objects containing the instructions. The amount of instructions is a by the user preconfigured integer “n”, defining the ngram size. This offers most flexibility and lets one create even different ngrams for the given disassembly report, i.e. in this work we use an ngram size from four to seven and use all those different sized ngrams for further processing.

The JAVA library »smda-reader« is built with Apache Maven [Tea18] support. This offers most interoperability with modern JAVA applications and frameworks. It is possible to exchange this library with i.e. a new library for processing other disassembly report types leaving the other parts of the framework untouched. Only the interfaces of »smda-reader« like the »Ngram« class and the »SMDA« container-class are further processed in the »yara-signator« tool. Any library providing these interfaces and classes can be used without too much effort instead of »smda-reader«, which offers great intercompatibility to process for example disassembly reports from objdump [Fou18] or radare2 [Alv18] in our processing chain.

5.3 YARA-Signator

The »yara-signator« is the core component and executable entry point of the YARA rule generation process. It manages the data flow from »smda-reader« into the database to operate on all objects and data while swapping out the large amount of data to persistent storage. It is written in JAVA and depends heavily on both libraries »smda-reader« and »java2yara« as well as on the MongoDB database. The filtering engine and the ranking system are the two core components which process and generate the candidates for the YARA rules. They are the most important steps since they have the biggest impact on the quality of the signatures and the largest as well as the most complex part of the framework. This section contains a top-level overview of the design decisions and concepts we followed for the implementation of our concept (cp. Sections 4.2.4 and 4.2.5).

While the first three steps of the processing chain (cp. Figure 9) are to a large extent covered by using the »smda-reader« library, after these steps the created data is stored in the database. The »smda-reader« reads the meta information and provides them to »yara-signator«, which stores them into the "families" and "samples" collection in the "caching_db" database.

All generated ngrams by »smda-reader« are stored in the "ngrams" collection. Then, the filtering process begins with calculating the immanent code sequences that overlap within malware families while not covering others. All ngrams are aggregated by their byte-sequences which has the same effect like the unification in a hash map. The remaining ngrams are stored in the "unique_samples" collection. Due to the aggregation, the left ngrams only occur in a single family to avoid shared code between malware families.

These ngrams are the base for the ranking system mentioned in step five of Figure 9 described in the following paragraphs. The resulting ngrams of the filtering procedure are shuffled for a better distribution of candidates within a sample and then further processed by the ranking system.

The ranking system is a wrapper class containing several ranking algorithms. It currently supports three modes of operation:

- no-operation mode
- basic ranking mode
- ranking prototype

The no-operation mode disables the ranking system by applying only a "dummy filter" which does nothing but reducing the ngram candidates to a preconfigured limit. For better results, the basic ranking mode offers a simple ranking algorithm based on the count of occurrences of an ngram in a sample within the family, as shown in Equation 5.1.

$$\text{base\_score} = \text{sample\_size} * 1000 \tag{5.1}$$

Each ngram gets a score assigned and based on the value of this score the ngrams are sorted and cut by the preconfigured value. The ranking prototype module calculates based on the previous base score as shown in Equation 5.1 by counting special instructions and absolute addresses. Arithmetical and logical instructions like "and", "neg", "or", "shl", "xor", "shr", "not", "div", "mul", "rol" and "ror" are being boosted while detected absolute addresses are punished with a reduced score as shown in Equation 5.2. The variable "instruction_counter" is a counter for instructions which include the previously mentioned ones, the variable "address_counter" in Equation 5.2 is increased if an absolute address was found.

$$
\begin{aligned}
\text{buster} &= 500 \\
\text{booster} &= 500 \\
\text{base\_score} &= \text{sample\_size} * 1000 \\
\text{boosting} &= \text{instruction\_counter} * \text{booster} \\
\text{punishment} &= \text{address\_counter} * \text{buster} \\
\text{ngram\_score} &= \text{base\_score} - \text{punishment} + \text{boosting}
\end{aligned}
\tag{5.2}
$$

The prototype module of the ranking system is a basic ranking option boosting rare instructions with logical or arithmetical purpose and punishing absolute addresses. It offers searching for rare instructions to find candidates which focus on special, cryptographic operations. Since we have reason to assume that malware authors tend to reuse code within their software we want to target complex parts of the sample which would be expensive in reimplementation.

Although the code-base is large we hope to generate better rules especially for families which have only one sample in the provided Malpedia excerpt. Families with just one sample are challenging because in the filter step of the processing chain we cannot filter for common code base between samples within a family but only filter for instructions which do not occur in other families. Since they miss the important data reduction step in the filtering process, it is important to use other techniques like the ranking prototype module in the ranking system to find promising candidates.

The ngrams generated by the ranking system are further processed with »java2yara« leading to valid YARA rules containing the left ngrams. However, the processed rules are the end of the actual generation chain described in Figure 9 although the last step is useful to examine if the produced results provide a good coverage of the given data set. In »yara-signator« are several statistical measurement algorithms implemented which can be found in the “statistics”-package.

5.4 java2yara

The small library »java2yara« is developed in JAVA to generate signatures in valid YARA format. It is an Apache Maven dependency of the »yara-signator« application controlling the generation of YARA rules based on previously generated ngrams. The wrapper class “Controller” offers the interface to create a new YARA rule. With the supporting “getter”- and “setter”-methods in the “YaraRule” class, the values of a YARA rule are configured and set by »yara-signator«. Using the “toString”-method of the “YaraRule” object, one can easily generate a YARA rule in a valid output format, which can be written into a file. These files are the main result of this thesis which can be processed by YARA in further steps. They can be easily shared within a community since YARA is the de facto standard format for collaboration between different organizations in the context of malware signatures.

5.5 Database Design

For quick development and fast-changing data structures during filtering we use the schemaless NoSQL-DBMS MongoDB [Inc18c]. Its concept differs widely from relational database solutions, databases in MongoDB do not contain tables but collections, which do not contain rows but documents. Documents are not declared before insertion and allow much more flexibility. MongoDB works natively with BSON,

a JSON-like format for its documents [1] which fit to the given input data from the JSON-encoded SMDA reports.

The data set for the given timestamp contains 600 different malware families and over 1800 samples. Therefore, we decided to store all meta information about the samples and families as well as the linkage between sample and family in collections for filtering purposes. This leads to an "ngram" collection of 306,450,486 ngrams derived from the disassembly reports and a "unique_sample" collection. The "unique_ngram" collection has 163,418,221 entries containing only ngrams which are unique within one malware family. The collections for families, samples, ngrams and "unique_ngram"s are stored in a MongoDB database called »caching_db« which has a size of 77.828 GB. A schematic overview of the database can be found in Figure 11.

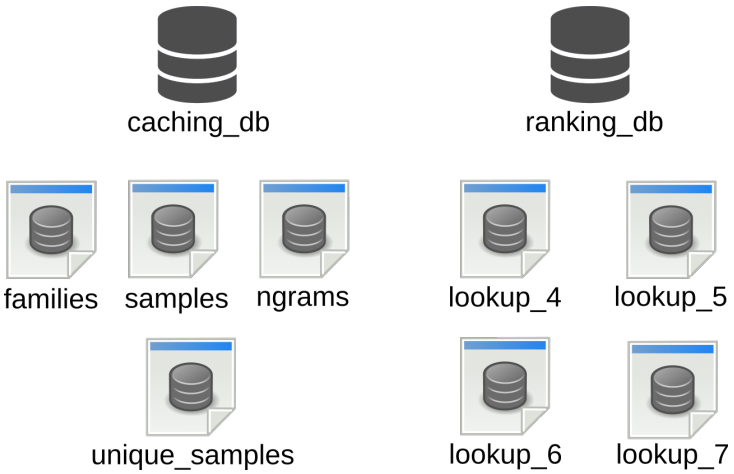

Figure 11: *Diagram showing the database design (Icons from third-party [2])*

.

The other database called »ranking_db« contains lookup-tables for the ranking system mentioned in step five in Figure 9 and is technically described in detail in Section 5.3. The database has a size of 67.441 GB and »ranking_db« contains a collection for each ngram size, i.e. "lookup_4" collection with all four-grams ordered as shown in Figure 11. The lookup table is used to search for the instructions behind opcodes of ngrams like "#53#56#57#8b45f8". These information about instruction details are not saved in the "unique_ngram" collection from the »caching_db« database for performance reasons. Instead each instruction is marked with a "#" to seperate the instructions in the encoded opcode bytes. They are looked up if they are needed by a module of the ranking system only and after significant data reduction to increase the performance of the ranking system. The lookup returns mnemonics in assembly language for each ngram candidate.

[1] The MongoDB Handbook states: "MongoDB stores BSON documents, i.e. data records, in collections; the collections in databases." https://docs.mongodb.com/manual/core/databases-and-collections/

[2] Database icon from Fabián Alexis, https://commons.wikimedia.org/wiki/File:Antu_application-x-kexiproject-shortcut.svg Collection icon from Everaldo Coelho, https://commons.wikimedia.org/wiki/File:Crystal128-database.svg

```
1   { "offset" : 32575514, "opcodes" : "53", "mnemonics" : [ "push", "ebx" ] },
2   { "offset" : 32575515, "opcodes" : "56", "mnemonics" : [ "push", "esi" ] },
3   { "offset" : 32575516, "opcodes" : "57", "mnemonics" : [ "push", "edi" ] },
4   { "offset" : 32575517, "opcodes" : "8b45f8", "mnemonics" : [ "mov", "eax, dword ptr [ebp - 8]" ]
      }
```

Figure 12: *Example result of a lookup process in the ranking system.*

An exemplary result for looking up the ngram candidate "#53#56#57#8b45f8" would return the following instructions as shown in Figure 12. These instructions are analyzed by the scoring algorithms of some of the modules within the ranking system, i.e. address detection.

6 Evaluation

In this chapter the proposed approach is evaluated towards a range of different aspects. First, the setup that was used in all experiments is described. Next the data set used in the evaluation is described in more detail, including a cursory structural analysis of the data found in the disassembly reports. After that, the methodology and goals for the evaluation of the approach itself are outlined. The chapter is then concluded by a performance evaluation itself and a discussion of the results. We provide an interpretation of the classification accuracy that can be achieved using the YARA rules generated by the software and address the key ideas in review.

6.1 Setup

The system we used to create the database and the YARA rules was a workstation with an Intel core i7-3770, 20 GB memory and Gentoo Linux [Inc18a] with a 4.12.12-gentoo kernel. We used MongoDB 3.4.10, 1.8.0_181 Oracle JAVA SE runtime environment, YARA 3.8.1 and Apache Maven 3.3.3 as a build tool. All software instances were running simultaneously on the work station. The system runs on hard disk drives in an RAID-1 compound using the Linux software RAID technology.

6.2 Data set

The data set provided for evaluation contains a snapshot of the Malpedia data set (commit "5e9ac34d", date "Thu Jun 7 12:49:02 2018"), accompanied with 1935 disassembly reports generated by SMDA for 692 malware families. Malpedia is a "centrally curated, free corpus" [PCEP18], offering over 3000 samples and more than 1000 different malware families [Plo18b] for various architectures and platforms.

The provided data base has eight SMDA reports that are incomplete which occurred due to a time out and 87 samples that do not contain any instructions. This might be the case when the samples which were disassembled using SMDA were created in .NET because this intermediate language is not processable by SMDA. This is not a limiting factor of our approach as described in Section 4.2.1 since the disassembly report processing method implemented in »smda-reader« is easily exchangeable. Figure 13 shows the distribution of the remaining 1927 SMDA reports of the samples.

Some of the SMDA reports are incomplete and do contain too few or not any instructions at all, resulting in the inability to generate valid YARA rules out of them. We ignore all SMDA reports that contain less than 10 instructions or do not have any instructions at all due to the lack of processing features in SMDA. This leaves 1828 SMDA reports for 636 families, 94.47% of the provided data set we will cover in this evaluation. The remaining 1828 reports contain 1784 reports which cover 32 bit samples and 44 reports which cover 64 bit samples. Our approach works for 32-bit as well as for 64-bit samples, although they contain different instructions, i.e. 64-bit registers and pointers which results in a smaller subset in the general ngram pool compared to 32-bit samples.

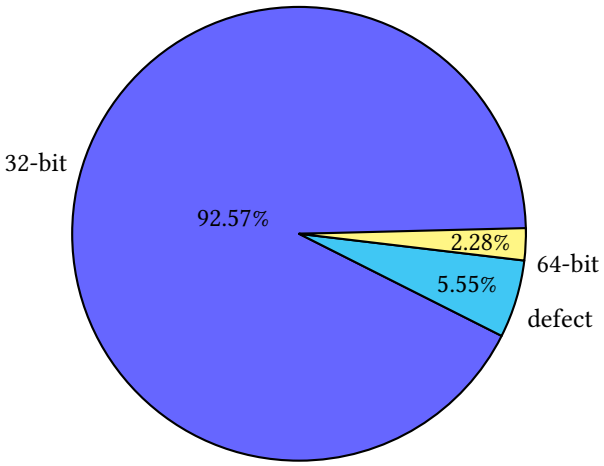

Figure 13: *92.58% of all provided SMDA reports are 32-bit x86 based, 2.28% are 64-bit x86 based and 5.55% are defect.*

For the statistical analysis, the disassembly reports are processed by the statistic evaluation modules in »yara-signator« and »smda-reader«. Both provide different interfaces for statistical data generation. The statistical results are described in the following sections.

6.3 Methodology

In this section the methodology is described and how we generate and evaluate the data using our approach on the described setup environment. The YARA rules are created by applying the processing chain built with the »yara-signator« software and the corresponding framework. Using our approach, the YARA signature generation is completely automated, only requiring the user to configure the tool.

The »smda-reader« generates statistics about the provided reports in general as well as information about the linearized disassembly. These data generated by processing the provided disassembly reports is presented and discussed in Section 6.4. Data for the section is generated by »smda-reader«, building a five-number summary in Table 2 and a table containing the top 25 most frequently occurred mnemonics (shown in Table 3).

In Section 6.5 first results are presented which were generated by analyzing all ngrams which can be built from the provided reports. These ngrams are created in step three of the processing chain (illustrated in Figure 6) and are not further processed. There are several statistical helper-classes within the »yara-signator« tool described in Section 5.3. They offer various information about the ngram distribution and occurrence as well as instruction ratio within all of the provided families.

Furthermore, the following Section 6.6 presents an evaluation of the generated YARA rules with our approach. The current system provides three different ranking system modules, providing a no-operation mode, a basic ranking mode, and an advanced ranking module prototype as described in Section 5.3. For better comparison, each YARA rule is generated with a "7 of them" condition (cp. Figure 5) and 10 YARA strings containing byte sequences of the corresponding ngram candidates. All YARA signatures were generated by the same configuration except the chosen ranking system. The ranking module "rankingPerNgramScore" has a configured limit of 1000 candidates which are processed by the "rankPrototype" module, limiting the resulting candidate pool to 10 ngrams which were afterwards written to YARA rules.

For quality assessment of the generated YARA signatures the F1 score as defined in Figure 14 is calculated, a common technique to evaluate the quality of data combining the precision and recall [Bei06]. The respective results can be found with further descriptions in Section 6.6.

$$\text{precision} = \frac{\text{true\_positives}}{\text{true\_positives} + \text{false\_positives}} \tag{6.1}$$

$$\text{recall} = \frac{\text{true\_positives}}{\text{true\_positives} + \text{false\_negatives}} \tag{6.2}$$

$$\text{F1 score} = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{6.3}$$

Figure 14: *F1 score calculation*

The F1 score is a calculation based on classification precision and classification recall as defined in Figure 14 by building a harmonic mean of them [Bei06] as shown in Equation 6.3. It describes the ratio of combining the amount of relevant samples as well as the coverage of relevant elements. Balancing between the precision and the coverage, the F1 score is an indicator describing the trade off between false positives and false negatives we face in the YARA rule generation process precisely. The following sections contain various statistical data, generated with different tools of the developed framework provided by »smda-reader« and »yara-signator«.

6.4 Disassembly reports statistics

This subsections contains a statistical overview of the provided SMDA reports. Table 2 shows a five-number summary of the disassembly results extracted from the provided malware samples sorted in ascending order. As one can easily see almost 75% of the provided disassembly reports contain less than the mean amount of instructions. Comparing the mean and median this way, it shows that there appear to be a few reports that are exceptionally large, introducing a noticeable bias towards these large reports in the mean value.

quantile	instructions	basic blocks	functions	func. calls	API calls	leaf func.	rec. func.
min	10	4	0	0	1	1	0
25%	9396	1331	201	777	0	54	0
50%	22430	3848	415	977	563	56	0
75%	47776	5474	1526	5208	0	739	0
max	1858338	337962	26438	114719	1810	6719	249
mean	44720.0	7750.19	902.50	3299.38	319.29	283.17	6.0

Table 2: *Five-number summary of the provided disassembly reports of the malware families.*

The maximum is a very uncommon example for big malware, it is a sample of the “win.mokes” family and contains over 1.85 million instructions. The sample is almost twice of the size in comparison to the second biggest report, a sample from the “win.micropsia”-family which contains around 995,700 instructions.

Overall, in the 1828 malware samples there are 653 unique mnemonics. Table 3 contains the top 25 mnemonics in average of the malware samples provided offering interesting information on which

mnemonics the framework might focus in data generation. There is a complete table of all mnemonics found by SMDA and their frequency attached in Appendix 9.

avg. frequency rank	assembly mnemonic	avg. instruction occurrence	ratio to avg. (44720)
1	mov	12134.97	27.14%
2	push	7137.75	15.96%
3	call	3904.82	8.73%
4	pop	2223.03	4.97%
5	cmp	2107.67	4.71%
6	lea	2055.18	4.60%
7	je	1691.50	3.78%
8	add	1674.15	3.74%
9	test	1476.39	3.30%
10	jmp	1303.77	2.92%
11	xor	1255.45	2.81%
12	jne	1173.65	2.62%
13	ret	894.20	2.00%
14	sub	623.76	1.39%
15	inc	508.16	1.14%
16	dec	475.10	1.06%
17	and	465.03	1.04%
18	movzx	387.64	0.87%
19	or	354.57	0.79%
20	shr	174.94	0.39%
21	jb	161.68	0.36%
22	shl	148.63	0.33%
23	nop	133.80	0.30%
24	jl	120.71	0.27%
25	jle	118.26	0.26%

Table 3: *Statistics about frequency of instruction occurrence, averaged over all malware samples.*

Table 3 shows how often instructions are used regarding data or control-flow management (“mov”, “push”, “call”, “pop”, “cmp”, “lea”, “je”, etc.) and how often instructions are beeing used for arithmetic or logical operations (“add”, “xor”, “sub”, “and”, “or”, “shr”, “shl”, etc.).

The top ten instructions make up almost 80% of all instructions in an average sample. This leads to the assumption that the ranking system should focus on instruction sequences that include less common mnemonics to introduce specificy while being not too exclusive to cover important blocks of the malware sample in the generated YARA rule. While operations like “xor”, “or” and “and” can be important for YARA rules, there might be more exotic mnemonics which are able to identify some families as long as they are used in each of their samples, for example mnemonics provided by the advanced vector extensions (AVX).

6.5 Ngram statistics

The following statistical data show the average of all families from Malpedia focusing on the ratio of the ngram occurrence within the malware families after normalization per family. They are created for each ngram size seperately, except the last row which contains all different ngram sizes combined. First,

the result for every family itself is normalized by dividing all ngram occurrence values by the number of ngrams observed in total. Then, an average value is created by summarizing all of these values for each field (total 475) from all families and dividing these fields by the total of observed families. The maximum co-occurrence observed were ngrams that occurred in 475 families.

Table 4 shows that a bigger ngram size yields a higher probability to hit an ngram that is unique to a family, which is usable for creation of a YARA. While longer ngrams offer a higher probability to increase the hits of the YARA rules leaveraging this effect, they provide less fexibility than combined short ngrams in the context of the YARA signature.

ngram-size	occurrence				
	1	2	3	4	5
n = 4	0.718	0.051	0.022	0.013	0.010
n = 5	0.753	0.047	0.019	0.011	0.008
n = 6	0.777	0.044	0.017	0.010	0.008
n = 7	0.796	0.042	0.016	0.009	0.007
n = 4-7	0.759	0.046	0.018	0.010	0.008

Table 4: *Statistics about the relative co-occurrence of ngrams across families, normalized on family level.*

Table 4 shows that the large amount of ngrams of a sample is unique within a family (occurrence is one). The ratio significantly drops down after one occurrence and this effect lasts to the highest occurrence level, interrupted only in rare cases. It also shows that the probability of randomly picking a family-unique ngram in an average sample is between 70% and 80%, although these values might differ in reality since we processed 372 of the 636 provided families which only contain a single family leading to a potential bias in these numbers. Even if these data differs in real cases, the statistical data provided in Table 4 show that our approach of focusing on instruction ngrams only occuring within a family seems reasonable.

As one easily sees the trend of the increasing ngram size shows that the greater an ngram value is set, the higher is the probability that it is unique within the malware family and not occuring in the other families from Malpedia. This leads to the conclusion that it is a good idea to handle all ngrams inside the program and let a ranking system make an informed decision if an ngram is a good candidate for the YARA signature. This offers a good flexibility and is especially a way to generate as short as possible rules instead of for example using a fixed ngram size.

In order to rule out that these statistics are distorted, we also looked at the relative number of unique ngrams per family and again provide the results in a five-number summary (cp. Table 5).

ngram-size	min	25%	50%	75%	max	mean	total
n = 4	0.035	0.565	0.752	0.893	1.0	0.720	455.912
n = 5	0.039	0.606	0.787	0.926	1.0	0.754	477.932
n = 6	0.041	0.641	0.823	0.945	1.0	0.778	493.203
n = 7	0.041	0.669	0.838	0.957	1.0	0.797	504.660
n = 4-7	0.039	0.621	0.796	0.928	1.0	0.760	482.723

Table 5: *Statistics about instruction ratio of all provided malware samples (normalized).*

Table 5 shows monotonically increasing values across all quantiles for a growing ngram-size. This is plausible since the higher the ngram-size is increased, the lesser ngrams are created in total, leading to a higher ratio. While smaller-sized ngrams are more precise, larger ngrams can detect larger code

sequences which might be more meaningful. This affects particularly smaller ngrams, since they can provide only a very short excerpt of a sequence or function which might be misleading. Therefore we decided to use all ngrams in a range of four to seven instructions for our approach to cover advantages of long and short ngrams.

The most important fact is that every family we analyzed has a certain amount of unique ngrams that do not appear in any other families of the provided data set. Even though we only have 3.5% for the minimum case with the ngram size four there is still a decent amount of ngram candidates to generate YARA rules covering edge cases. The lowest quantile already has over 50% unique ngrams which is a quite interesting fact. This shows that there is actually a bias towards higher uniqueness which benefits the approach. The developed approach is very promising and there is large set of potential ngram candidates. The samples with 100% unique ngrams are likely the result of imprecise disassembly reports containing only a very few and rare instructions, architecture mismatches or disassembly errors.

6.6 YARA rule statistics

This sections contains various statistics about the generated data and YARA signatures from the provided disassembly reports. There are three different tests, two of them present the results against the learning data set and the last test is against a current Malpedia snapshot to evaluate if the generated YARA rules are able to cover samples which were not learned previously.

The sample size from the data set of 1828 samples is reduced by a few due to mismatch between the provided data and the Malpedia data set. As Malpedia is an active project, changes may affect the represented families from the original snapshot. This is expressed through some family names having potentially changed or samples being inventorized to another family, e.g. after identification from a previously unidentified cluster. The three different test data sets “samples”, “families”, and “Malpedia” describe the different data on which the resulting YARA signatures are applied.

The data set “samples” is a test regarding all provided disassembly reports against the same samples which were provided in Malpedia. The test scenario covering “families” is very similar, with the difference that all samples of a given family have to be covered with matches in order to evaluate as a true positive - thus being a way stricter criterion that may incur false negatives if rules do not generalize well.

The test scenario against the Malpedia corpus is a complex test which distinguishes between covered and uncovered families regarding to Malpedia, containing 1105 families and different platforms (e.g. IoT, Android) in contrary to 636 families as in the provided and previously learned data set. Instead against the Malpedia snapshot (commit “5e9ac34d”, date “Thu Jun 7 12:49:02 2018”) the YARA rules are evaluated against the current (commit “806ffe82”, date “Fri Oct 26 02:41:04 2018”) Malpedia corpus to evaluate their precision and specially their false positive rate.

Table 6 shows the determination of scoring within the Malpedia test case (“TP” corresponds true positive, “FP” to false positive, “TN” to true negative and “FN” to false negative). A major difference towards the previous tests is the distinction between packed and dumped/unpacked samples as well as between covered and uncovered samples. Packed samples are typically not covered by the generated YARA rules since they rely on instructions that are usually hidden by a packer. For the “samples” and “families” test, only the covered family / dumped row represents the result scoring.

Uncovered families are generally considered as “noise” in this data set, as there were no rules created for these families in the first place. Therefore, created YARA signatures are mainly evaluated against covered samples while checking if YARA rules for covered families are detecting samples of uncovered ones resulting in false positives.

covertness	sample status	hit category family hit	other hit	no hit
covered family	packed	TP	FP	TN
	unpacked	TP	FP	FN
	dumped	TP	FP	FN
uncovered family	packed		FP	TN
	unpacked		FP	TN
	dumped		FP	TN

Table 6: *Evaluation schema for the YARA rules against a recent snapshot of the Malpedia data set.*

By testing the YARA rules against the current Malpedia data set it is possible to give an estimation about covering samples of previously unlearned samples for detecting future malware. Table 7 provides the calculated metrics and measurements previously described in Section 6.3 for a YARA rule output generated without any ranking system:

test data	false pos.	false neg.	true pos.	precision	recall	f-score
samples	0	415	1368	1.0	0.767	0.868
families	0	134	492	1.0	0.786	0.880
Malpedia	73	1279	2422	0.971	0.654	0.782

Table 7: *YARA rule built without ranking system.*

Table 7 shows that even without any ranking (i.e. by just randomly picking ngrams) it is already possible to cover a majority of samples without any false positives for the learned samples. The generated YARA rules cover as much as 79% of the families derived from the data being learned on without any false positives. However, the results are still improvable as shown in Table 8.

Table 8 provides the calculated statistical data for the YARA rules generated with the basic ranking system.

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 8: *YARA rule built using the simple ranking system.*

Table 8 shows that the simple ranking system allows derivation of rules that clearly outperform the randomly composed rules (cp. Table 7). The F1 score and the recall of the tested data is lower, only the precision is minimally higher. The test mode operating on the family data shows that there is an increase from 492 to 534 families which are perfectly covered although there is one false positive detection. The YARA signature generated for the “win.gravity_rat” family covers its own and the “win.microcin” family. Table 9 shows the statistical evaluation for a YARA rule output generated with the prototype module of the ranking system.

test data	false pos.	false neg.	true pos.	precision	recall	f-score
samples	15	235	1546	0.990	0.868	0.925
families	5	105	521	0.990	0.832	0.905
Malpedia	149	998	2782	0.949	0.736	0.829

Table 9: *YARA rule built using the prototype module of the ranking system.*

The YARA rules generated by the ranking system module prototype are not performing better than the provided data as one can easily see in Table 8, containing the data of the simple ranking system and Table 9, describing the statistic data of the ranking system prototype. They generate more false positives, which is quite obvious since they adjust the score honoring a predefined set of instructions instead of focusing on most frequent instruction sequences within a malware family exclusively. But they did not cover a meaningful larger amount of unlearned malware samples from Malpedia.

Although these data suggest that the ranking system focuses on the wrong instructions, the data set of Malpedia might still be to small for this ranking approach. Since only 264 of 636 families of the provided data corpus are clustering more than one sample, the filter can not find a code base within a family for the other families. The other 372 have only one sample, which might be a problem. Specially the fact that many samples are very small like shown in Table 2 can be misleading for our approach in general.

One of our main goals was to generate YARA signatures which cover interesting or rare instruction sequences that are precise and offer stable detection of a family. The YARA rule in Figure 15 shows an exemplary rule candidate which was generated using the ranking system »prototype« module. In comparison to a YARA rule generated by an analyst offered in Malpedia for the malware family “win.gameover_dga” it offers the same detection while not triggering false positives on the complete Malpedia database being as good as a manually written rule. Even when the YARA rules generated by the ranking system prototype module are performing slightly worse behind the benchmark result of the simple ranking system module mentioned in Table 8 the rules can compete with manually generated YARA rules. The rule shown in Figure 15 is exemplary for the precise signatures covering logic or arithmetic instructions trying to cover complex instruction sequences within a family.

```
rule win_gameover_dga
{
    meta:
        Author = "Felix Bilstein - fxb at cocacoding dot com"
        Date = "2018-10-29"
        Description = "autogenerated rule brought to you by yara-signator
            "
        Tool = "yara-signator 0.1a"
        Reference = "Malpedia data set"

    strings:
        $ = { 324df88b55f480c901324e18335618 }
        // score = 7000
        // n = 5
        // bytes: #324df8#8b55f4#80c901#324e18#335618
            //  #324df8 [xor, cl, byte ptr [ebp - 8]]
            //  #8b55f4 [mov, edx, dword ptr [ebp - 0xc]]
            //  #80c901 [or, cl, 1]
            //  #324e18 [xor, cl, byte ptr [esi + 0x18]]
            //  #335618 [xor, edx, dword ptr [esi + 0x18]]

        [...]

        $ = { 0c62eb0424010c60 }
        // score = 6500
        // n = 4
        // bytes: #0c62#eb04#2401#0c60
            //  #0c62 [or, al, 0x62]
            //  #eb04 [jmp, 0x1e24ce3]
            //  #2401 [and, al, 1]
            //  #0c60 [or, al, 0x60]

        $ = { 33ff8b750824fe0c02 }
        // score = 6500
        // n = 4
        // bytes: #33ff#8b7508#24fe#0c02
            //  #33ff [xor, edi, edi]
            //  #8b7508 [mov, esi, dword ptr [ebp + 8]]
            //  #24fe [and, al, 0xfe]
            //  #0c02 [or, al, 2]

    condition:
        7 of them
}
```

Figure 15: *Example for a YARA rule generated with the ranking system prototype module covering the gameover_dga family.*

6.7 Discussion

The resulting YARA rules and their quality as well as the performance of the implementation is the subject of this section, also targeting various limitations of the approach and ranking system implementations in their current prototype state.

As the evaluation statistics show, many YARA rules were initially created on a very high level. Even though that in the first attempt were already 492 YARA rules generated without any false positive detection as shown in Table 7 the basic ranking system as described in Section 5.3 could generate 534 YARA signatures fully covering the families while causing just one additional false positive detection as presented in Table 8.

All created YARA rules have a very high precision although the basic ranking system could create the YARA signatures with the best coverage and the highest F1 score with 92% which describes the precision and sensitivity of the result. Only the recall for the sample data set as shown in Table 8 is even higher than for the families data set which clearly shows that the remaining YARA signatures could probably be improved since only some of the samples within a family could not be detected.

As shown in Section 6.4 there are a lot of samples which have only an instruction size around 50,000 instructions per sample. While these data is easy to handle, some samples containing almost one million instructions or even more are challenging for the developed framework. This is the major reason why the filtering system is existential for our approach. With a more powerful setup there might be enough memory to process all families without limiting the processed amount of ngrams to create even better YARA rules.

There are several aspects of our evaluation approach which may lead to different assumptions than real-world cases would take advantage of. There are plenty facts which lead to a challenging evaluation of the given data, i.e. due to renaming and non-matching sample clustering. A practical example is the "win.mokes" family, which is a multi-platform malware. A YARA rule covering the "win.mokes" family and the "osx.mokes" leads technically to false positives since they are identified with different family names while a YARA signature covering a malware family over multiple platforms is actually a very good result.

Although the framework creates good YARA rules it is comparatively slow. Currently, a complete process takes up to 27 hours, highly depending on the load of the work station. The reading of all SMDA files takes around eight minutes, combined with the linearization process nearly 15 minutes. The creation of the ngram database, containing over 300,000,000 elements takes around 12 hours, even longer when not using the bulk insertion mode due to searching the inserted data before inserting the new ngrams. The creation of the unique-ngram database (cp. Chapter 5, "unique_samples", Figure 11) takes around two hours, building indexes takes another 30 minutes. The final creation of the YARA rules takes 12 to 16 hours, highly depending on configurable limits in the filtering step and by composition of the ranking system. Ranking systems which need to analyze the instructions by searching through the lookup-table collections are certainly more expensive than the basic ranking system.

The YARA rules generated by the framework are still of a high quality as Section 6.6 described. Although MongoDB was not perfectly optimized it was still a good choice for the implementation of this prototype, being an excellent tool for agile development.

7 Summary

Following the initial approach, concept, implementation and the evaluation, in this Chapter each topic is briefly described and summarized.

Starting with an introduction in Chapter 1 and a short overview of the most important themes in the Chapter 2, the initial question is described and defined as well as several information essential for this topic. Moreover, as described in Chapter 3 covering related work, there are various projects covering the automatic generation of YARA signatures, but only a few like yarGen (cp. Section 3.1.1) and YaBin (cp. Section 3.1.2) are fully automatic generators while most of the YARA rule generation approaches are semi-automatic tools, helping the researcher to generate YARA signatures.

These approaches still require a significant amount of experience as well as expert knowledge as opposed to a fully-automatic approach we developed. The main idea was to develop a framework, covering our processing chain as a sequence of modularly designed tools which are potentially interchangeable for high flexibility as described in Chapter 4. Each element of the processing chain is described conceptually and in detail in Section 4.2, covering the task that every element fulfills.

Chapter 5 describes the matching between the conceptual design pattern and the actual implementation. While “parsing”, “linearization” and the “ngram creation” step are done by »smda-reader«, the inter-process communication between all components and the “filtering” as well as the “ranking system” module are implemented in »yara-signator«, the core component of the framework. The module »java2yara« is a component offering interfaces for YARA rule generation by passing ngrams to it, managed by »yara-signator« in the “YARA rule composer” step and used to generate YARA signatures based on the remaining ngram candidates derived from the ranking system. The resulting YARA rules can be validated by »yara-signator« for a quality assessment although this last step is not mandatory for YARA rule generation.

The evaluation, described in Chapter 6, summarizes the metrics and statistical measurements of the generated YARA signatures by the developed framework. The discussion of the quality of the generated YARA rules in the evaluation contains several aspects of the framework, containing a description of the provided data set and the derived YARA rules of them, as well as performance data and a subsumption of them.

7.1 Conclusion

The major topic of this thesis involved the automatic generation of YARA signatures which cover precisely samples and families with a false positive detection rate as low as possible. Most of the generated YARA rules were able to cover the corresponding family without detecting others. Fore the training data set, the framework could generate YARA signatures that reach nearly 100% precision as presented in Table 8, leading to a very satisfying F1 score between 92% and 93.4% which shows the ratio between their precision and their sensitivity.

Another key point was the question if it would be possible to detect even samples which were not learned previously. Following our approach, the generated YARA rules were able to classify more than 2700 samples of Malpedia correctly. This is a good result relative to the provided data set which covered effectively 1828 samples as described in Section 6.2 and the framework was able to generate signatures that cover most of the samples in Malpedia. This shows very plainly that this initial approach as outlined in this thesis has the potential to grow and to be extended to a more mature solution for automatic YARA rule generation.

Although the generated YARA rules do not cover every sample perfectly, the framework was still able to reach a high level of quality. Since there exist only few publicly available and high quality YARA rule repositories as described in Section 3.2, the created signatures are a valuable addition to the community.

7.2 Future work

The automatically generated YARA rules using our approach are able to cover a majority of the Malpedia data set and most of the learned samples and families. Even though the YARA signatures possess a high quality, there are several options which have the potential to increase the functionality, speed, or precision of the framework. In this section these options are presented and discussed, showing advantages of used technology for the implementation of our approach as well as further challenges, providing the basis for future work.

In this work only samples for Windows are covered. While the approach searches for instructions in general and not for a specific instruction set based on the x86 architecture, it would be interesting if the framework could create YARA rules for e.g. Android or IoT malware.

As well as focusing on other operating systems and architectures, using different disassemblers together might lead to better results. For example comparing results of the IDA Pro disassembler and SMDA might reduce the amount of wrongly disassembled junk instructions. Junk instructions are disassembled when data is being interpreted as code [Har17], leading to a certain number of junk instructions which are processed as part of the set of ngram candidates. Reducing junk instructions might lead to better YARA rules which can be used to detect more precisely.

This is a special problem for the ranking system which was implemented in this work. There are several sequences of junk instructions which contain a lot of less frequent mnemonics the ranking system prototype is favoring, such as “and” operations when disassembling null-bytes as shown in Figure 16.

```
00000000  0000              add [eax],al
00000002  0000              add [eax],al
00000004  0000              add [eax],al
```

Figure 16: *Disassembled data leading to junk instructions, interpreted as code [Har17].*

The current ranking system increases the score as mentioned in Section 5.3 leading to a very high score for some potential junk instructions. This creates YARA rules for some families, especially those with only a few samples in the provided data (since they get a high base score), that are based on these junk instructions. These YARA rules lead to a relatively high number of false positive detection since many samples contain a sequence of null-bytes, which are for example code caves created by the compiler as described in Section 4.2.4. The ranking system might be improved by avoiding junk instructions to ensure that only actual instructions are used for YARA signature generation.

Some strings in the generated YARA rules contain absolute addresses. YARA offers the possibility to mask certain bytes and by the use of wildcards. An improvement of our appproach might include the use of wildcard characters for absolute addresses. A similar technique is used in YARA-FN which is described in Section 3.1.4. YARA-FN is able to use wildcard characters in YARA rules to prevent absolute addresses and jump instructions which is a technique that has the potential to improve the developed framework.

The current implementation is pretty slow and can take up to 27 hours as described in 6.7. The approach is not optimized for runtime performance and clustering and database optimization might lead to a significant improvement since most of the time, the application is waiting on database queries. Clustering data on different machines could lead to parallel reading and in some cases might even parallel writing, reducing (for example when writing temporal collections) the waiting time for disk IO. MongoDB offers support for this kind of database optimization technique although this is out of the scope for this thesis since the restricted working period [Inc18d].

Data partitioning based on the family and using stored procedures might increase even more, although MongoDB doesn't support stored procedures [1] [Inc18e] or partitioning [Gat10] which would require a switch of the database management system.

Stored procedures are native functions offered by the database management server. The advantage of stored procedures is the execution natively on the database server and not on application level on the client. Without the overhead between server and client it is possible to operate faster, especially on large databases where a client's memory can become a limiting factor very quickly [Gro18]. Therefore, a filtering system or some processing steps of a ranking system might be faster when executed directly on the database server.

Another option using partitioning for database optimization is motivated due to the fact that reading queries are taking most of the time when generating new YARA rules from already existing collections. Queries take between 10 seconds and several minutes for each family, which could be significantly reduced if the DBMS (database management system) wouldn't search the complete database but just for example a partition containing the matching family, reducing the search space by a factor of over 600 depending on the number of families. This could reduce the generation time from 12 to 16 hours to an estimated 1-2 minutes, that could make it applicable for near real-time YARA generation. Even if the partitioning was only half as fast as estimated it would still increase the speed of the framework enormously. However, the optimization of the framework would be appropriate for future work. MongoDB was still a good choice for this prototype, since it made the fast development possible in the first place.

The ranking system is currently creating YARA rules for a predefined and by the user configured settings such as boosting and punishing scores. If previously mentioned database optimizations are applicable, then a recursive YARA rule improvement step becomes viable. Several YARA rules are still detecting wrong samples leading to false positive hits. With a faster YARA signature generation engine there are miscellaneous potential improvement possibilities such as generating in a second step only those YARA rules again which do not cover the complete family or wrong samples.

The current implementation of the ranking system produces YARA signatures with the fixed condition "7 of them" (cp. Figure 15). A recursive approach could examine the best ratio for YARA string occurrence in a sample as well as more complex conditions in general. YARA offers a set of logical operators for complex conditions which could be used for better classification. This might be useful for covering families with a significant change in their common code base in different versions. Scanning a family for different common code clusters in their samples might detect indications for several major releases, whereby each cluster of a family could be inspected individually contrary to our approach. Detecting common code clusters instead of searching only shared code within all samples of a family might increase

[1] Meaning stored procedures known from relational database systems like PostgreSQL [Gro18]. MongoDB supports "stored JavaScript", which is comparable but slower and explicitly not recommended by the handbook. https://docs.mongodb.com/manual/tutorial/store-javascript-function-on-server/

the accuracy of the generated signatures. An improved ranking system could cover this aspect by creating more complex conditions using logical operators and concatenating YARA strings that detect potential common code clusters shared by samples within each family.

8 Bibliography

[ACC+18] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. Ligh, J. Levy, and A. Walters. Rekall memory forensic framework. https://github.com/google/rekall, 2018. [Online; accessed 13 July 2018].

[ACL+18] M. Auty, A. Case, M. Ligh, J. Levy, A. Walters, and N. Petroni. Volatility. https://github.com/volatilityfoundation/volatility, 2018. [Online; accessed 13 July 2018].

[Alv18] S. Alvarez. radare2. https://rada.re/r/, 2018. [Online; accessed 03 November 2018].

[ash13] ashkbiz. Yoda's protector. https://sourceforge.net/projects/yodap/, 2013. [Online; accessed 12 July 2018].

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

[Bei06] S. Beitzel. On understanding and classifying web queries, 2006.

[CLR17] J. Clapper, M. Lettre, and M. Rogers. Joint statement for the record to the senate armed services committee foreign cyber threats to the united states. 2017. [Online; accessed 31 October 2018].

[Cor03] Microsoft Corporation. Defining malware: Faq. https://technet.microsoft.com/en-us/library/dd632948.aspx, 2003. [Online; accessed 11 July 2018].

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

[Dur14] S. Durbin. Cybercrime: The next entrepreneurial growth business? https://www.wired.com/insights/2014/10/cybercrime-growth-business/, 2014. [Online; accessed 03 November 2018].

[ERC+16] 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.

[Fou18] Free Software Foundation. objdump. https://sourceware.org/binutils/docs/binutils/objdump.html, 2018. [Online; accessed 03 November 2018].

[Gat10] M. Gattol. Mongodb jira - feature request server-2097. https://jira.mongodb.org/browse/SERVER-2097/, 2010. [Online; accessed 01 November 2018].

[Gro18] PostgreSQL Global Development Group. Postgresql - a powerful, open source object-relational database system. https://www.postgresql.org/docs/8.0/static/plpgsql.html, 2018. [Online; accessed 01 November 2018].

[GTBo18] C. Guarnieri, A. Tanasi, J. Bremer, and other. Cuckoo sandbox. https://cuckoosandbox.org/, 2018. [Online; accessed 13 July 2018].

[Har17] N. Harbour. Recognizing and avoiding disassembled junk. https://www.fireeye.com/blog/threat-research/2017/12/recognizing-and-avoiding-disassembled-junk.html, 2017. [Online; accessed 03 November 2018].

[HR18] Hex-Rays. Ida: About. https://www.hex-rays.com/products/ida/index.shtml, 2018. [Online; accessed 03 November 2018].

[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] Google Inc. gson - a java serialization/deserialization library to convert java objects into json and back. https://github.com/google/gson/, 2018. [Online; accessed 03 November 2018].

[Inc18c] MongoDB Inc. Mongodb. https://www.mongodb.com/, 2018. [Online; accessed 03 November 2018].

[Inc18d] MongoDB Inc. Mongodb handbook - sharding. https://docs.mongodb.com/manual/sharding/, 2018. [Online; accessed 01 November 2018].

[Inc18e] MongoDB Inc. Mongodb handbook - store a javascript function on the server. https://docs.mongodb.com/manual/tutorial/store-javascript-function-on-server/, 2018. [Online; accessed 01 November 2018].

[Inc18f] StarForce Technologies Inc. Aspack software - application for compression, packing and protection of software. http://www.aspack.com, 2018. [Online; accessed 12 July 2018].

[Lab17] McAfee Labs. Mcafee labs threat report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf, 2017.

[McA18] McAfee. Mcafee - securing tomorrow blog. https://securingtomorrow.mcafee.com/technical-how-to/malware-packers-use-tricks-avoid-analysis-detection/, 2018. [Online; accessed 29 September 2018].

[Mjjm18] J. Martin, j0sm1, jovimon, and mmorenog. Yara rules. https://github.com/Yara-Rules/rules, 2018. [Online; accessed 15 October 2018].

[OM17] M. Oberhumer and L. Molnar. Upx - the ultimate packer for executables. https://github.com/upx/upx, 2017. [Online; accessed 12 July 2018].

[PCEP18] D. Plohmann, M. Clauss, S. Enders, and E. Padilla. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *The Journal on Cybercrime & Digital Investigations*, 3(1), 2018.

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

[Plo18a] D. Plohmann. The big zeus family similarity rundown v2. https://pnx.tf/slides/zeus_similarity_showdown.html, 2018. [Online; accessed 03 November 2018].

[Plo18b] D. Plohmann. Malpedia - general statistics. https://malpedia.caad.fkie.fraunhofer.de/stats/general, 2018. [Online; accessed 03 November 2018].

[Plo18c] D. Plohmann. Smda. https://github.com/danielplohmann/smda, 2018. [Online; accessed 03 November 2018].

[Ran17] S. Ranger. Us intelligence: 30 countries building cyber attack capabilities. https://www.zdnet.com/article/us-intelligence-30-countries-building-cyber-attack-capabilities/, 2017. [Online; accessed 31 October 2018].

[Rot18a] F. Roth. Yara rules from signature base. https://github.com/Neo23x0/signature-base/tree/master/yara, 2018. [Online; accessed 15 October 2018].

[Rot18b] 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].

[Sup18] SupportIntelligence. Icewater. http://icewater.io/#life-as-art, 2018. [Online; accessed 15 October 2018].

[Tea18] Apache Maven Project Team. Apache maven. https://maven.apache.org/, 2018. [Online; accessed 03 November 2018].

[Vis14] Visgean. Zeus github repository. https://github.com/Visgean/Zeus, 2014. [Online; accessed 03 November 2018].

[Wor18] M. Worth. Open-source-yara-rules. https://github.com/mikesxrs/Open-Source-YARA-rules, 2018. [Online; accessed 15 October 2018].

[WS18] R. Wessen and SupportIntelligence. Project icewater. https://github.com/SupportIntelligence/Icewater, 2018. [Online; accessed 15 October 2018].

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

9 Appendix

The attached disc contains:

- A digital copy of this Bachelor thesis.
- The complete source code, unit tests and build scripts for the framework, containing »smda-reader«, »yara-signator«, »java2yara«.
- The provided disassembly reports, compressed.
- A set of generated YARA rules.
- Generated statistics of the evaluated YARA signatures and the provided disassembly reports.

For building and executing the software stack, a similar setup as described in Section 6.1 is required as well as access to a copy of the Malpedia [PE18] repository.