Capability Analysis on Malicious Software

Master's Thesis

by

Felix Bilstein
2629449

in fulfillment of requirements for degree
Master of Science (M.Sc.)

submitted to
University Bonn
Institute of Computer Science IV
Work group Communication Systems

in degree course
Computer Science (M.Sc.)

First Supervisor: Prof. Dr. Peter Martini
University of Bonn

Second Supervisor: Prof. Dr. Michael Meier
University of Bonn

Advisors: Daniel Plohmann, Luca Corbatto
Fraunhofer FKIE

Bonn, November 8, 2021

Abstract

Information security is a key issue for government, business and society. In an ever-changing threat environment from organized crime, foreign intelligence agencies and other highly specialized attackers, malware is a commonly used tool to compromise systems. Malware researchers are confronted with a large amount of malware attacking thousands of systems every day. To process such large amounts of data, a number of automated processes are necessary.

In the context of malware analysis there are two classic sub-areas: static and dynamic analysis. There are various automation strategies and technical advances for both sub-sectors. In this work we address the capability analysis of malware by applying static and dynamic analysis approaches to a set of malware samples.

We analyze more than 2000 memory dumps and extract malware capabilities of the samples. The capabilities are connected to the Malware Behavior Catalog (MBC) and MITRE ATT&CK framework. We evaluate Malpedia, a publicly available and curated malware repository, and contribute data on distribution and frequency of malicious capabilities and behaviors. In addition, we analyze the data using dynamic analysis. We built a framework for the orchestrated execution of malware, allowing the automated execution and evaluation of hundreds of malware samples in parallel. The amounts of functions of the samples, which are reached during the execution in our sandbox, are evaluated to make statements about the potential runtime coverage. Moreover, we compare the results from both approaches to find out which approach tends to be able to potentially cover more capabilities.

Contents

List of Figures iv

List of Tables v

1 Introduction 1

2 Foundation Concepts and Terms 4

2.1 Malware . . . 4

2.2 Malware Analysis . . . 6

2.2.1 Static Analysis . . . 6

2.2.2 Dynamic Analysis . . . 8

2.2.3 Packing . . . 13

2.2.4 Taxonomy . . . 14

2.3 Code Analysis . . . 15

2.3.1 Reverse Engineering . . . 15

2.3.2 Fundamentals of the x86 Architecture . . . 16

2.3.3 Binary Analysis Primitives and x86 Disassembly . . . 20

3 Related work 22

3.1 The relevance and challenges of packed malware . . . 22

3.2 Overcoming packed malware - unpacking and sandboxing . . . 23

3.3 Representative data sources for statistical evaluation of automated malware analysis approaches . . . 26

4 Static Capability Analysis 28

4.1 Motivation . . . 28

4.2 Methodology . . . 29

4.3 Implementation Details . . . 31

4.3.1 Introducing Capa . . . 33

4.3.2 Extending Capa . . . 36
4.4 Evaluation Procedure . . . 37
4.4.1 Data Set . . . 37
4.4.2 Methodology . . . 38
4.5 Results . . . 39
4.5.1 Frequency of Capa Rules . . . 39
4.5.2 Coverage of the malware repository . . . 42
4.5.3 Connecting capabilities to the MITRE ATT&CK framework . . . 44
4.5.4 Linking the results to the Malware Behavior Catalog . . . 51
4.5.5 Discussion . . . 58
4.6 Summary . . . 59

5 Runtime Coverage Analysis 60
5.1 Motivation . . . 60
5.2 Methodology . . . 61
5.3 Implementation Details . . . 64
5.3.1 Introduction to Frida . . . 65
5.3.2 Malware execution orchestration . . . 69
5.3.3 Code execution in the guest . . . 71
5.3.4 Limitations of our Implementation . . . 71
5.4 Evaluation and Results . . . 72
5.4.1 Data set . . . 73
5.4.2 Methodology . . . 73
5.4.3 Results . . . 74
5.4.4 Discussion . . . 84
5.5 Summary . . . 86

6 Summary 87
6.1 Conclusion . . . 87
6.2 Future work . . . 88

References 90

7 Appendix 100
7.1 Additional Malpedia Statistics . . . 100
7.2 Additional Capa Statistics . . . 100

List of Figures

1 Malware growth per year by AV-Test [Insa]. The number of samples per year categorized by potentially unwanted applications (PUA) and malware. 5
2 Function invocation, with and without interception. [HB99] 10
3 Packing illustration by *Biondi et al.* [Bio+18] 13
4 Illustration of the software compilation steps adapted from *Tanenbaum et al.* [TB14] . 16
5 The compilation and reverse engineering process constructed as a graph showing both procedures and their opposite direction. [LD03] 17
6 Possible construction of a computer design by following the principles of the Von-Neumann Architecture (VNA). [Hof13] 18
7 General Purpose CPUs and their bitness in x86 architecture [Cor21] 19

8 Illustrated coverage evaluation procedure, showing the data flow of the input data and the resulting output reports and statistics. 32
9 Capa disassembly features shown in disassembly. [BR20] 34

10 Architecture of Frida, based on the diagram from the documentation [Fri21d] but with focus on Windows. 67
11 Design of Frida's Interceptor, its hooking engine. 68
12 The setup of our scripts and virtual machines. 70
13 Exploration of the analyzed data, comparing the amount of executable binaries and DLLs. 75
14 The runtime data we analyzed to compare how much of the malware was covered and how much samples crashed during their execution time. 78
15 Procedure for the function address intersection between dumped and unpacked samples. The illustration is chosen schematically, the median of the actual overlap is approximately 99 % as shown in Table 11. 80

List of Tables

1 Software versions used in this setup. 37
2 Top 30 Capa rules ordered by counted hits (absolute, sample and family). 40
3 Five-number summary with additional mean and standard derivation for the amount of coverage by Capa of SMDA disassembly reports. 42
4 Enterprise matrix of the counting every technique per sample part 1. 45
5 Atta&ck 9.0 enterprise matrix, counting every technique per sample part 2. 46
6 MBC matrix, counting every technique per sample part 1. 53
7 MBC matrix, counting every technique per sample part 2. 54
8 MBC matrix, counting every technique per sample part 3. 56
9 Five-number summary with additional mean and standard derivation showing the reduction of potentially analyzed functions of the unpacked malware samples in Malpedia and the associated coverage of functions. 76
10 Five-number summary with additional mean and standard derivation showing the coverage of all samples in Malpedia that have a dumped and unpacked version. . . . 83
11 Ratio of all Windows samples where unpacked and dumped files are available. 100
12 Top 20 capa rules ordered by counted hits (family, sample and absolute). 101

1 Introduction

Information systems are constantly under attack. Different threat actors vary from cyber crime groups to nation state attackers. Compromises using malware, short for malicious software, are a widely used intrusion vector to compromise information systems and their respective data. Malicious software is a common tool for targeting computer systems and networks in complex attacks.

Malware researchers face millions of malware samples in circulation. Payloads are usually distributed in packed form, protected against immediate inspection using static analysis methods. Authors use packing as a technique to evade and bypass anti-virus protections by increasing the difficulty for static detection.

Although different malware families may share common packers and therefore unpacking routines, malware detection and especially classification is a complex task. Therefore, automation is an important key for the successful and effective assessment on a large scale. While detection aims to rate software as malicious or not, classification describes the process of clustering malicious software into families [GF11].

To handle large amounts of data, analysts use automated malware analysis systems and approaches like sandboxes and capability analysis. The results are useful for a quick overview and can help analysts to get a first idea of a malware's functionality. Modern sandboxing solutions are able to trace malware and detect its functionality during execution. While some approaches classify samples into harmless or potentially malicious categories, other sandboxes monitor suspicious samples over their execution time and track their behavior providing a more detailed analysis. These reports often include information about network usage, processes and threads as well as file system usage. Analysts can decide further actions on the basis of additionally gained information.

On the other hand, indications of capabilities can be extracted by static analysis tools like Capa. Capa heuristics detect the possible control-flow, function imports and library usage as well as high-level behavior. This potentially provides more complete coverage of a malware's code base as the ability of sandbox approaches is limited by the runtime behavior of malware during its execution duration. In

order to better understand how well this approach works, we assess and test Capa's abilities extensively as being a new malware analysis tool.

In this work we consider the two major approaches of malware analysis, static and malware analysis. Static analysis is the analysis of malicious software without executing it while dynamic analysis is the analysis of malware at runtime. We evaluate the ability of capability analysis for both approaches by comparing the potential coverage of sandboxes, a typical dynamic analysis approaches, against static capability detection with the novel tool Capa.

First, we evaluate the spectrum of potential capabilities used by modern malware by analyzing the malware corpus Malpedia. The respective data set contains over 2000 malware families and more than 5000 samples. We analyze the results of the capability extraction process statistically and interpret the outcome. Additionally, we evaluate a set of malware samples using a dynamic analysis approach and detect which capabilities are reached during exection. Therefore we construct a malware execution orchestration framework which consists of an agent and an isolated sandbox. We use a binary instrumentation framework which allows the agent to launch and instrument malware samples for dynamic analysis. To evaluate the potential coverage of dynamic malware analysis we hook a certain amount of functions.

We analyze more than 900 unpacked samples and the corresponding memory dumps for an evaluation between both approaches. In conclusion we assess the approaches and both of our implementations to make statements about the advantages and disadvantages of the approaches. Additionally, we address the difficulties of the methods of analysis.

We contribute the following techniques and results:

- We measure how many capabilities are in a malware corpus present, how it is distributed, how much code in terms of functions and basic blocks of the samples is covered when using static analysis methods on memory dumps.
- As novelty, we apply two industry-relevant taxonomies, the Malware Behavior Catalog and MITRE ATT&CK, by using static analysis techniques to a large and publicly available malware corpus.
- We instrument more than 2000 unpacked malware samples in order to determine the relationship between the functions and the frequency of their invocations.
- We evaluate the potential of static and dynamic analysis by comparing statically analyzed memory dumps and their coverage with unpacked malware and the amount of functions reached during execution.

The remainder of this master thesis is structured as follows:

Section 2 outlines basic notions and foundations this work is based on. We shortly present typical malware analysis methods and their limitations. In addition we briefly summarize the idea of reverse engineering software and how possible challenges affect our approach. In Section 3 we present the current state of research. Beginning with techniques and publications which are dealing with the challenges of packed malware, we present related publications about modern approaches of unpacking and dynamic analysis tools like sandboxes. In Section 4 we describe our static analysis approach. We connect our findings to the MBC and MITRE ATT&CK frameworks. In Section 5 we present our dynamic analysis approach. We present the malware execution orchestration framework and its design, the implementation of our binary instrumentation approach and discuss the results. In Section 6 we provide a short summary and conclusion to the reader.

2 Foundation Concepts and Terms

In this section we define and describe several technological terms and ideas which serve as a basis for our work. We show what *malware* is in general and how to deal with malware from the analyst's perspective. Additionally, we discuss some of the most common problems of malware analysis and specially how to circumvent *packing*, a technical strategy to avoid quick detection. Then we present various taxonomy systems, their measurement and how we use them in our work.

2.1 Malware

Malware (short for malicious software) is software that is designed to break typical protection goals of computer systems [Nora]. In the early days of the computer industry, malware was developed by their authors to show development skills or to prank their victims [Reg20]. Nowadays, malware authors are typically driven by more serious goals, including financial gain or espionage operations [Gee+14].

Malware is a growing field in the cyber crime sector. More than 450.000 new malware samples and potentially unwanted applications (PUA) are registered per day by the AV-TEST Institute [Insb]. Figure 1 illustrates the enormous growth of malware. The malware in circulation that is found by researchers is continually growing over the years. For more than 13 years the amount of malware and potentially unwanted applications in the wild is growing every year. Although the amount of malicious samples in circulation is considerably growing, malware researchers estimate the actual amount of individual malware families is orders of magnitude smaller. The samples are obfuscated and protected against analysts by using packers.

Packing is a significant problem for efficient malware analysis as the amount of data rises strongly. In 2010, *Reynaud* analyzed almost 100 000 samples for his PhD thesis of which more than 95% of the malware he analyzed is packed [Rey10]. *Calvet et al.* investigated thousands of representative samples by utilizing their framework WaveAtlas in 2015 [Cal+15]. They discovered that 98% of the analyzed samples were packed. Therefore packing can be understood as a relevant problem for malware analysis.

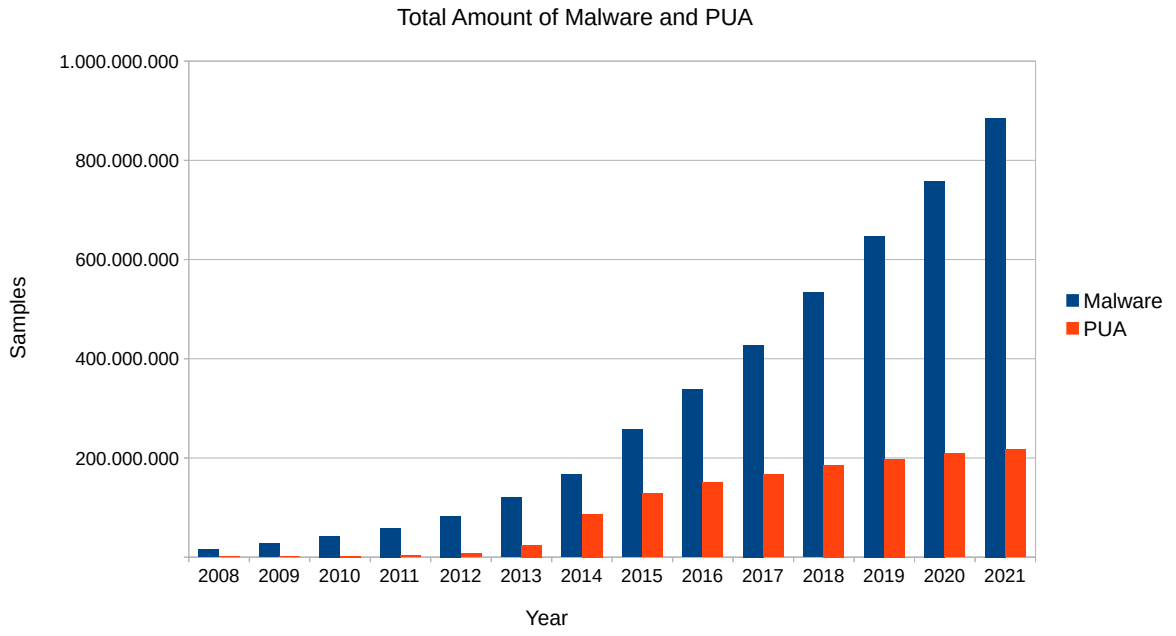

Figure 1: *Malware growth per year by AV-Test [Insa]. The number of samples per year categorized by potentially unwanted applications (PUA) and malware.*

Malware Categories There exist various sub types of malware such as worms, viruses, Trojan horses, ransomware and many more. The term malware itself is used as an umbrella term for all kinds of harmful software [Lat21].

A Computer worm is malicious program that is able to replicate itself to infect other systems such as computers, routers, printers, IoT-devices (internet of things) or various other entities [Bar10]. Worms are typically designed to exploit software bugs automatically (i.e. without user interaction) and begin with the infection of new targets from the compromised entity. Computer viruses are another category of self-replicating malware [Lat21]. Unlike worms, computer viruses need user-interaction for the infection of new targets. Typical infection vectors are e-mails and file sharing.

A Trojan horse or Trojan is malicious software which is covered as legitime software. Trojans are designed to compromise the system of a potential victim. Users are tricked into the execution of an unknown file or clicking on illegitimate links. This way Trojans infect their victims. They are not able to replicate themselves unlike computer viruses or worms. There exist various sub types of Trojan horses such as Remote Access Trojans, Banking Trojans, Infostealer Trojans, Ransomware Trojans and many more. Remote Access Trojans (RATs) are used by different threat actors to establish and keep access to compromised systems such as mobile devices or computers. Banking Trojans are used by

cyber crime operators to acquire banking credentials of their victims. Established malware families are e.g. *Citadel*, *Dridex* or *Emotet* [Inc]. Infostealer Trojans are used to obtain further information from compromised systems. A ransomware trojan is used to blackmail the victim by encrypting files stored on the compromised system. [Norb]

Ransomware is malicious software that threats private users as well as large enterprises [McA]. The global economic harm amounts to $11.5 billion by 2019 [ZC19]. The malware actors blackmail their victims by encrypting their information on the compromised system. Victims are forced to pay a ransom to get the decryption key for their files. Some actors even pressure their victims to pay the ransom by threatening them to leak the stolen information or destroying them on the victims infrastructure. Typical infection vectors are e-mail phishing campaigns and the exploitation of software vulnerabilities [Inv20].

2.2 Malware Analysis

Malware Analysis is the process of dissecting malicious software by understanding the internal structure, its capabilities, how to identify it and how to remove or defeat it [SH12]. It is a complex topic that requires time and expert knowledge. In this section we present typical techniques for malware analysis. There are basically two malware analysis approaches: *static analysis* and *dynamic analysis*. Another category worth to note is called *hybrid analysis*, a mix between those two methods [Bak20]. All approaches are important during the process of reverse engineering of malware. We briefly describe all methods in this section although due to complexity of the topic we refer to technical literature for further information.

2.2.1 Static Analysis

Static analysis is a technique describing malware analysis by applying tools on a sample to determine its functionality [SH12]. The (binary) file is not executed during static analysis [Mil14]. There are several techniques and tools to check if a malware is malicious or how to determine its behavior. The most important techniques are [SH12]:

- Antivirus Scanning
- Hashing
- String determination
- PE file analysis

- Analysis of imported and exported functions
- Disassembly Analysis

Static malware analysis is usually the first step for reverse engineering samples. All listed techniques offer different information based on their functionality although each technique has limitations.[SH12]

Antivirus Scanning is a possible first step to obtain further information about given samples by taking advantage of earlier analysis by third-party. It is possible to use scanning engines from multiple vendors to get an overview of a possible threat. Antivirus software vendors implement different signatures and detection engines leading to varied evaluations or estimations of samples [SH12]. A popular web service allowing to scan and identify a suspicious file is VirusTotal [Vir]. VirusTotal offers a web interface and an API for the creation of analysis reports of samples based over 60 Antivirus Scanning Engines. Malware analysts can use the automatically created reports to obtain information about the suspicious file.

Cryptographic Hashing is used by malware analysts to create a unique fingerprint of a sample. A cryptographic hash function is a one-way function mapping a string of arbitrary size to a fixed size one [Buc08]. The values of cryptographic hash functions are strong identification features for malware samples. Therefore, malware analysts use cryptographic hashes of malware to communicate clear and precise about the respective samples. Those Hashes are shared in reports and knowledge bases so other analysts can reproduce results or find reports of suspicious samples. [Lig+11]

String determination is a step in static malware analysis for the enumeration of strings in binary files. Strings are a sequence of characters in a program [SH12]. When developers implement software, every text message displayed to the user, file name, URLs and comparable artifacts are merged into the binary. Detected strings can uncover functionality to the analyst leading to further information about the suspicious file. Other strings may also reveal the malware family a given sample belongs to as some strings are very characteristic for the respective malware family. [SH12]

PE file analysis is used by malware analysts to detect irregular or conspicuous files. Executable files targeting the Microsoft Windows platform are binary files following a format defined by Microsoft in the respective PE/COFF (Portable Executable/Common Object File Format) specification [Lig+11] [Mic13]. Malware authors tend to craft special PE files when covering malicious code

to evade antivirus signatures. Some common techniques that can be spotted are invalid timestamps, sections of arbitrary length or suspicious entry point addresses [Lig+11].

Imported and exported functions can reveal potential functionality and used libraries. Software developers in general use third-party functions provided by Windows API calls and other dependencies such as DLLs (Dynamically Linked Libraries). These functions are listed as imported function names in the import table of a PE file. They can provide information about possible capabilities of an executable. The exported functions on the other hand are listed in DLLs. DLLs are modules which are loadable by other executable binaries containing code from libraries. They offer exported functions which are listed in an export table in the respective PE file. Malware analysts can inspect the export table to estimate the capabilities of a suspicious DLL. [SH12]

Disassembly Analysis is an advanced technique to examine the functionality and capabilities of software [SH12]. The analyst disassembles the code of a binary file to inspect the assembly instructions of a given sample. Reconstructed assembly instructions can be abstracted into larger control-flow graphs and function templates. Disassembly analysis requires expert knowledge and a deep understanding of the underlying technology such as, i.e. x86 or ARM architecture. We present the necessary details for disassembly analysis and code analysis in Section 2.3.

2.2.2 Dynamic Analysis

Dynamic analysis is the second important approach for malware analysis. In contrast to static analysis, where the target sample is not being executed, dynamic analysis methods include the execution of the respective sample [SH12]. Besides low-level techniques such as debugging there exist more high level approaches such as process monitoring tools, network sniffing programs, Windows registry diffing tools and file system monitoring software. [Mil14] Modern, automated solutions like sandboxes are used by malware analysts in parallel to accelerate the analysis process.

Process monitoring tools can inspect the currently active processes and threads of programs. Malware samples can launch processes, threads and potentially write into arbitrary processes, too. Analysts can use process monitors to detect such behavior without going into detail.

Network sniffers are programs that can monitor or inspect network traffic such as internet packets [Con03]. They are a common tool used by IT professionals to inspect the content of network traffic streams, transferred files or network communication in general [Web]. Packet analyzer can also be

used by malware analysts to inspect a sample's network traffic or to uncover hidden communication channels.

A popular software package for dynamic analysis tools for the Windows platform is distributed by Microsoft under the name *Sysinternals* [Mic21b]. The project offers more then 70 different tools for behavior inspection and analysis tools for the Windows platform.

Debugging In the field of software development, debuggers are tools to inspect or manipulate a target process to find information about the control flow and program logic. Typically they are used by developers during software development trying to fix complex bugs or faulty program logic. There exist various debugger models and techniques. Some involve the source code of the software and are available in modern Integrated Development Environments (IDEs), other work completely without source code. These kind of debuggers can be useful during software development if the source code is for example not available any more. Other debuggers have a hybrid strategy and can operate without source code but if it is provided by the user it can be associated to support the debugging process. A popular example for a hybrid debgging approach is the debugger GDB [Sta+10].

Although debuggers originate in the field of software development they are powerful tools for malware analysts. Debugging is a possible step after an analyst was not able to retrieve any further information about a sample using static analysis techniques or automated dynamic analysis approaches [SH12]. Debugging is very time-consuming and therefore not the first approach analysts start with. Analysts can use a debugger to inspect the memory, instructions and control flow of a sample during runtime which allows analysts to determine the inner structure and behavior of a malware sample. Debuggers are not only able to give the analyst a proper view of the internals of a target process, they are also able to alter the data or code inside the given process context. This enables analysts to run step by step through the instructions in the control flow or be able to set breakpoints which pause the execution if reached. [Sou12]

Debugging processes is generally a manual task although developers or analysts can typically script the respective tools for semi-automatic analysis. This is specially important for obfuscated or very complex malware.

Hooking Interception of Functions or *hooking* is a technique where developers or system researchers overwrite existing functionality of applications or operating system components [HB99]. If the source code of a component is available changes are trivial but the source code of third-party components, commercial software or malware is usually not available. Functions of such software components can

be intercepted and overwritten. The overwritten function can be called after execution of the hook function to maintain the original functionality. The new "hooking" procedure can contain additional code to extend the original functionality. This technique can be abused by malware as well as being used by developers to extend legacy code. [HB99]

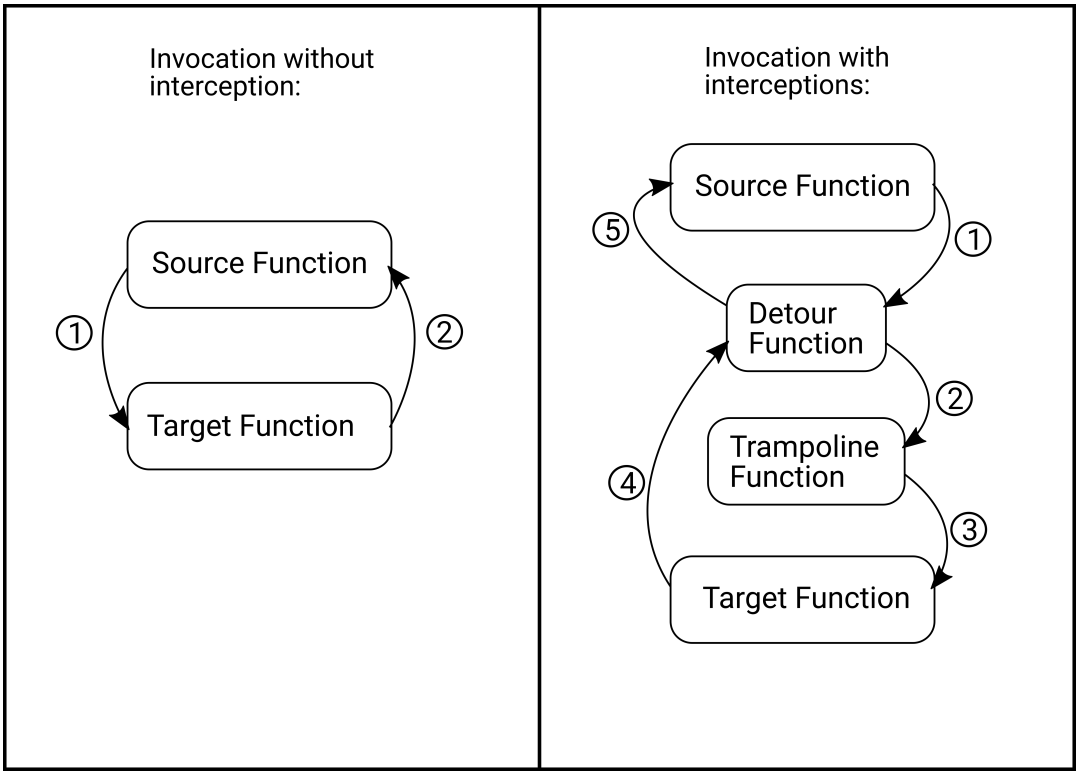

Figure 2: *Function invocation, with and without interception. [HB99]*

Hunt and Brubacher describe, besides their function interception framework *Detours*, the process how functions can be intercepted in detail [HB99]. Figure 2 illustrates the difference between a regular function call and a function invocation with interceptions by using their framework *Detours*. The left side describes calls without interception. A program has several functions, in this example we focus on a source and a target function. The source function (caller) calls the target function (callee) and when the callee is finished, the function returns back to the caller. The illustration on the right side describes function calls with interceptions by using Microsoft Detours. The source function is redirected due to modification of the control flow by the developer to a detour function instead of the target function. The detour function handles pre-processing where necessary and is able to call the trampoline or return to the source function from the target function after handling the post-processing. To ensure the original functionality the framework keeps the original instructions from the target function and a jump instruction to the target instruction. [HB99]

In the context of malware analysis, hooking is an invasive technique to control the sample's native code or API functions by the operating system. It can be used to log, monitor or inspect code fragments or notify when specific functions are called but also to change the functionality for example to direct execution to code for further analysis that is unreachable otherwise. Additionally, not only a sample's control flow can be customized, hooking may also reveal the data and parameters of functions during runtime. A common use case is the extraction of secrets from the process such as passwords, configurations or cryptographic keys. This allows analysts to take advantage of the technique by passing the data of target functions revealing valuable information for malware researchers and reverse engineers.

Therefore, hooking is a common technique used by handlers in sandboxing environments for the analysis of target processes. Hooking of functions in the Windows API can inform the sandbox application about events, used APIs, their data and the return values of the originally called functions. This information can be used by sandboxes to generate reports for malware analysts to present basic information about a sample. [Mil14]

Binary Instrumentation Binary Instrumentation is an advanced debugging technology for dynamic analysis. Dynamic Binary Instrumentation frameworks are able to analyze programs during runtime by introducing additional instructions and code into a target process or program. For this technique, the target code can be instrumentated or the original architecture of the code emulated to analyze its behavior [Mil14]. Binary instrumentation tools offer a broad range of potential use cases such as covert debugging and advanced instruction analysis. [cor]

Established tools are e.g. FRIDA [Fri] and Intel PIN [Lev]. FRIDA is a binary instrumentation framework based on JavaScript. It injects an interpreter in the target process which handles the interaction between its user and the target. Intel PIN is a framework which can be used by developers to build custom reverse engineering and binary analysis tools. It provides access to the target binary via API.

Sandboxes Sandboxes are environments for automated dynamic analysis. It is a security mechanism to run arbitrary, potentially malicious software in a safe environment so that malware cannot harm productive systems [SH12].

Malware analysis sandboxes are typically equipped with various monitoring, scanning and detection abilities. By using techniques like API hooking they are able to monitor changes to the file system, networking components, different processes or memory [Dam+17]. Every call to the API can be

monitored so that the analysis systems can provide a detailed report to the activity of a sample to the analysts. [Mil14]

Although sandboxes are a powerful tool the technology has multiple drawbacks. Malware can try to detect sandboxes and behave different than on an actual compromised host system. It is also possible that even if the malware behaves regularly, some functionality will not be observable due to absent network connectivity, nonexistent software or other events that are never triggered due to missing user activity. Besides these problems, it is also possible that sandboxes do not detect low-level interaction that circumvents monitoring through Windows API hooks. [SH12]

A very popular vendor is the Cuckoo Sandbox, an open-source automated malware analysis system [Cuc]. Cuckoo offers a modular, automated and customizable analysis environment. It is able to analyze different file types such as executables, office documents, websites and many others under virtualized instances of Windows, macOS, Linux and Android. Cuckoo can trace API calls and the behavior of the target sample. Additionally it is able to monitor even SSL/TLS encrypted network connections and their data as well as memory analysis of the underlying virtual machine by using YARA [Alv21] and Volatility [Fou20]. All findings are delivered to the user as high level information as report. [Cuc]

Besides open-source solutions there also exists various proprietary and commercial sandboxing vendors. Products like ANY.RUN [LLC21] and VMRay [VMR21] offer support for the automated analysis of malware. ANY.RUN focuses on interactive usability and URL checking in a sandboxed environment with different browser support. VMRay Analyzer offers automated analysis of samples in a virtual environment using a hypervisor approach. VirusTotal [Vir21] as a malware intelligence provider also offers different sandboxes among other tools and techniques. [Lig+11]

Hybrid Analysis Hybrid analysis is a mixed form of static and dynamic analysis. It enables analysts to combine static and dynamic strategies to find capabilities and detect behavior that is otherwise beyond the scope of traditional approaches and their limitations [RM10].

Hybrid analysis goes a step beyond dynamic analysis. Static analysis approaches can be used on top of the data which can be extracted using dynamic analysis techniques. There exist several examples of this category of hybrid analysis approaches. Solutions like Joe Sandbox Cloud (formerly JoeBox) [Joe21a] and Hybrid Analysis [Hyb21] offer products that use techniques of both dynamic and also static analysis tools.

2.2.3 Packing

Packing is a form of modification of binary files. Historically, binary files could be compressed as self-modifying archives to reduce the bandwith used for the delivery. Publicly available tools like UPX can compress binary files for various platforms and architectures [UPX]. Nowadays bandwith is usually available and compression of binaries before delivery not needed any more. [Arn17]

While packing can be used legitimately for binary compression, malware authors modify their payloads with packing techniques to camouflage their malware. Packing is an established technique used by malware authors to evade anti-malware detection systems. Without prior unpacking, static analysis techniques can not be applied to the sample. [Arn17]

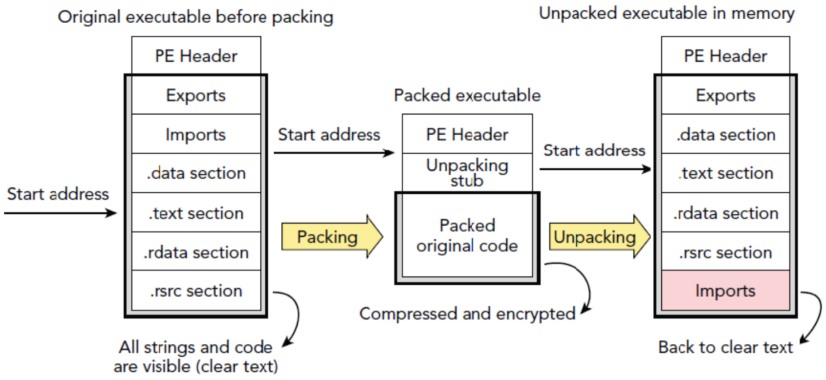

Figure 3: *Packing illustration by Biondi et al. [Bio+18]*

Figure 3 illustrates the procedure how executables can be packed. The content of a binary file can be packed into a new section of a PE file or into a stub. Therefore, strings and code are not visible anymore. Additionally to the data stub containing the original code and data sections, an unpacking stub is created and added to the packed executable. The new entry point or start address points to the unpacking stub which rebuilds the original data and code into memory. Unpacked samples contain the original clear text strings and code, consequently unpacked samples can be analyzed statically by malware researchers. [Bio+18]

Unpacking is a complex challenge for malware analysts. First, one has to recognize if a sample has actually been packed. There exist various strategies to determine if a sample has been packed: There are publicly available tools such as ClamAV and PEiD as well as more advanced techniques like syntactic signatures, entropy metrics, or machine learning. [Bio+18]

When an analyst realizes that a sample has been packed, the next challenge is to unpack the sample. Several techniques for unpacking are published by academic research and tools are available from the

practical context. Some approaches involve the discovery of the used packer for automated or semi-automated unpacking [Bio+18]. Other approaches try to establish a technique for generic unpacking such as RoAMer [JPP19]. RoAMer is a generic unpacker based on dynamic analysis, being able to dump the memory of processes and samples of the researcher's interest.

2.2.4 Taxonomy

For an effective exchange between malware researchers, a structured and standardized platform is necessary. The MITRE ATT&CK framework offers a taxonomy for offensive and defensive oriented information security specialists by providing a knowledge base of cyber adversary behavior [Str+18]. The framework contains attacker tactics and advisory techniques. They are standardized into different categories and groups of advisory tactics, characteristics and properties, simplifying the exchange between people.

The MITRE ATT&CK framework is loosely based on the cyber kill chain. The cyber or intrusion kill chain is a collection of different phases which describe the behavior of Advanced Persistent Threats (APTs) attacking targets of interest. *Hutchins et al.* from Lockheed Martin established the expression in the computer security industry as well as in academia. All phases depend on each other and are characterized by a stage-based attack. [HCA11] Although malware does not necessarily operate in the same way as the authors describe in their paper their are many similarities. But the cyber kill chain has no direct focus on malware and is not offering detailed techniques for the key points of the chain as the MITRE ATT&CK framework provides.

The MITRE ATT&CK frameworks offers multiple MITRE ATT&CK matrices for different use cases. The most popular matrix of the ATT&CK framework is the *MITRE ATT&CK Matrix for Enterprise*. It provides information about tactics and techniques for the platforms Windows, macOS and Linux. Moreover, it also supports common cloud, network and container environments and it offers a category called *PRE* containing preparatory tactics and techniques. Additionally, there are two other specialized matrices *MITRE ATT&CK Matrices for Mobile* which are covering iOS and Android. And there is the *MITRE ATT&CK for ICS Matrix* that contains selected tactics and techniques for use in the context of Industrial Control Systems (ICS). In this work we use MITRE ATT&CK enterprise framework, its tactics, techniques and its matrix in version 9 and focus on the Windows platform. The Windows Matrix of the enterprise matrix contains only fields of the full enterprise table that contain information about the Windows platform.

The Malware Behavior Catalog (MBC) is very similar to the attacker framework but is rather focused on malware instead of attackers [Bec20]. Therefore, it is structured in objective and behavior and

not in tactics or techniques. Objectives describe the intention of certain action of malware while behaviors represent the action a malicious sample performs. The framework evolved from the ATT&CK framework, adding several techniques or behaviors and reducing the large ATT&CK entities to which exclusively are useful in the context of malware. In this work we focus on the MBC framework in version 2.1.

Both frameworks use the Structured Threat Information Expression (STIX), a language for the exchange of cyber threat information [Sta21]. MBC and ATT&CK offer JSON schemas as machine-readable data as JSON documents in STIX 2.1.

2.3 Code Analysis

In this section we present the basics of code analysis and similar topical areas. Code analysis requires fundamental understanding of the underlying platform or architecture, e.g. ARM or x86 as well as basic understanding of reverse engineering and assembly language.

Reverse engineering is an important element for malware analysis. For our work, basic understanding of reverse engineering of software is necessary. In Section 2.3.1 we describe software reverse engineering and provide basic knowledge to the reader.

Furthermore, knowledge of the underlying architecture is necessary to understand the technical details of machine code and binary files. To apply code analysis, researchers need information about the instruction set. For our work, we provide knowledge about the abstracted functionality of x86 CPUs which is provided in Section 2.3.2.

Additionally, knowledge of assembly language is essential for code analysis of given binary files. Since we focus on x86 in this work we provide a short introduction of x86 (32-bit and 64-bit) assembly as well as the respective disassembly generation and binary analysis primitives in Section 2.3.3.

2.3.1 Reverse Engineering

Reverse engineering describes the process of the extraction of certain design artifacts or components of a subject of interest. *Chikofsky et al.* define reverse engineering as an analysis process to identify components of a system and their connections and the recreation of that system to another form or higher abstraction level [CC90]. In our context, reverse engineering of software is the process of reconstructing a higher level of abstraction from a given binary file containing machine code.

To understand the details of reverse engineering and its practical implementation one needs to take a step back and highlight the process of software development. Software which is developed in native languages such as C or C++ are written in high-level syntax. The source code of such software is compiled into binary files.

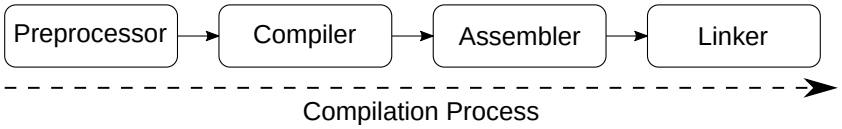

Figure 4: *Illustration of the software compilation steps adapted from Tanenbaum et al. [TB14]*

Figure 4 shows the compilation process. First, the preprocessor handles macros and source code imports. Then, the source code files produced by the preprocessor are compiled by the compiler. The compiler optimizes the input code and is able to produce assembly language files for the given platform or architecture. Compilers allow the implementation of software in a higher level and porting the code to any architecture the compiler is available for. Assemblers assemble the respective assembly language files to object files. These files are linked into executable binary files for a certain operating system and architecture.

Figure 5 illustrates the compilation process and the corresponding reversing engineering procedure defined by *Cullen et al* [LD03]. The compiler parses the source code and builds up a syntax tree which is used to generate a control flow graph. This graph is translated into assembly language which is assembled into machine code afterwards.

The compiled binary files cannot be translated back into source code as various information is lost during the compilation process. Instead, only the assembly language can be reconstructed to a certain level of quality by processing the machine code. Therefore, reverse engineers and malware analysts face disassembled binary files and need to understand the assembly instructions that are revealed by disassembling tools.

2.3.2 Fundamentals of the x86 Architecture

In this work we focus on x86 architecture as most modern personal computers are supposed to run on x86 central processing units (CPUs). Additionally we want to analyze especially malware targeting the Microsoft Windows platform which is very closely tied to the architecture. The architecture is conceptional build as a modified Von-Neumann Architecture (VNA). Von-Neumann described several points that are key features of a VNA [Neu93]:

- A processing unit, containing an arithmetic logic unit (ALU) and processor registers

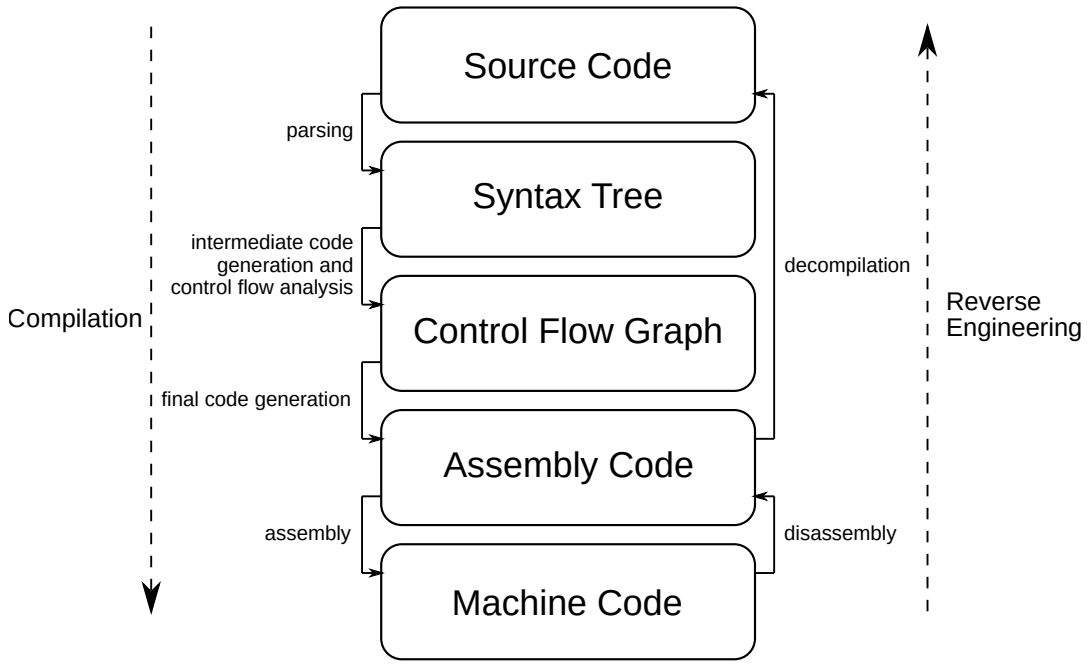

Figure 5: *The compilation and reverse engineering process constructed as a graph showing both procedures and their opposite direction. [LD03]*

- A control unit (CU)
- Memory storing data and code
- External storage
- I/O-mechanisms

Computers built via the VNA approach are universal usable computing machines. Therefore, the designed instruction set has to be powerful enough to cover all of their potential use cases. It is a multi-purpose machine which is not built for a specific task. Figure 6 illustrates the design of a computer based on the VNA. [Hof13]

The CPU is divided into two parts, the ALU which is the calculation unit of the CPU, and the CU, which contains the control logic of the CPU. Both are connected via a bus system to communicate with each other as well as with different peripheral devices. VNA instructions and data are shared in the same memory without strict separation. [Hof13]

Modern x86 architectures still share code and data in memory. This introduces a fundamental problem for the generation of disassembly and for reverse engineering of software in general. Since code and data cannot be distinguished by any obvious properties, tools need to parse syntactically correct

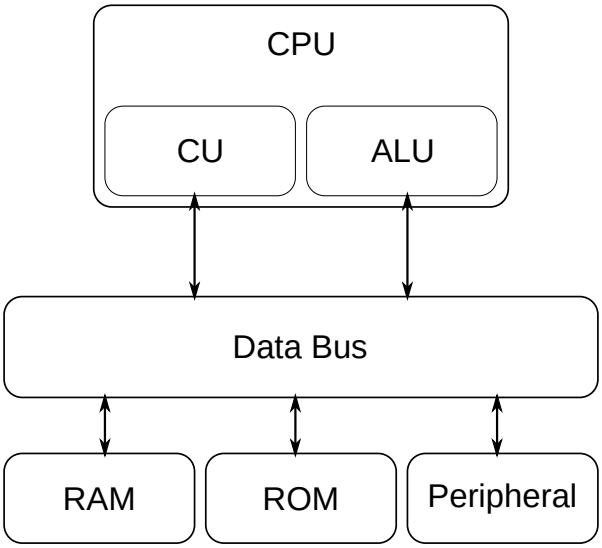

Figure 6: *Possible construction of a computer design by following the principles of the Von-Neumann Architecture (VNA). [Hof13]*

binary formats to produce accurate disassembly. Therefore we need to take this into consideration as our tools are applied to the memory sections of programs and processes where they face instructions as well as data.

Additionally, the x86 instruction set composed for its underlying architecture consists of a very large and complex set of instructions. Therefore, the x86 architecture computer is a typical representative of the Complex Instruction Set Computer (CISC) family. Their instruction set architecture (ISA) includes various arithmetic and logical operations, memory transfer operations and support for complex addressing variants. Because of the enormous amount of available instructions [1] almost every byte combination leads to syntactically valid assembly instructions increasing the complexity to detect disassembly errors or memory segments without actual code. We discuss the details of the x86 binary analysis and assembly language in Section 2.3.3. [Hof13]

The IA-32 architecture offers 16 basic program execution registers which are used for general programming [Cor21]. There are four different categories of registers:

- General-pupose registers (8 registers)
- Segment registers (6 registers)

[1] The Intel© 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes [Cor21] lists almost all x86 instructions for many modern CPUs. There exist several hundreds of instructions for x86, and every instruction set extension increases the instruction set since the ISA is backwards compatible in general.

- EFLAGS register (1 register)
- EIP register (1 register)

The EIP register is the program counter, containing a pointer to the next instruction. It can be used with a word width of 16 bit or 32 bit. If the respective software is executed in 64-bit mode, the EIP register is called RIP and contains 64 bit. [Cor21]

The EFLAGS register is another specialized register, offering status bits (flags) which can be checked using assembly instructions to modify the control flow of a program. In 64-bit mode, the register is 64 bit large and is called RFLAGS.

All six segment registers are used to store segment selectors. A segment selector is used to identify a special memory area (segment). They point to the code (CS), stack (SS) and data segment (the four registers DS, ES, FS and GS). In 64-bit mode, segmentation is not used and the registers can be used as additional registers for data access. [Cor21]

General-purpose registers can be used for storing data, operands or pointers. The registers are supposed to store data as operands for operations such as logical or arithmetical as well as operands for the calculation of addresses and memory pointers. The register ESP is a special register as it is used by programs as stack pointer, pointing into the stack segment although it is possible to use the ESP as regular general purpose register. [Cor21]

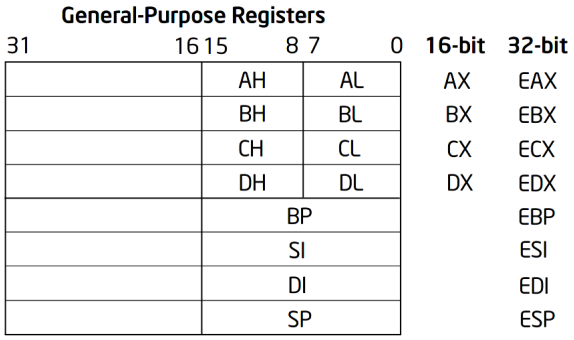

Figure 7: *General Purpose CPUs and their bitness in x86 architecture [Cor21]*

Figure 7 shows the structure of the general purpose registers. Lower words can be accessed via different register names, some offer 8 bit words while other support 16 bit words as minimal word size. In 64-bit mode, the registers are extended and can be called as RAX, RBX, RCX, RDX, RBP, RSP, RSI and RDI. Additionally, there are eight new registers introduced, the general-purpose registers R8, R9, R10, R11, R12, R13, R14 and R15. When a 32-bit operand instead of 64-bit is specified the registers the 32-bit register naming convention is used and the additional registers are called R8D to R15D. [Cor21]

2.3.3 Binary Analysis Primitives and x86 Disassembly

Assembly language is the least abstract but still reasonable way to communicate with a general purpose CPU from a developer's point of view. Assembly developers as well as reverse engineers need knowledge of the underlying hardware such as registers and instruction sets. Although assembly language requires a certain level of expert knowledge, the assembler and linker already abstract from addresses and translate the given assembly code into machine readable binary files and machine code.

There exist various assembly instructions for the x86 architecture. Describing all of them would be beyond of the scope of this work but we name and describe a few operation categories in this section. There are several types of assembly instructions:

- data movement
- arithmetic and logic
- comparison
- control flow

Data movement instructions are used to pass data between memory or registers to exchange data. The content of memory or registers can be changed using arithmetical or logical operations. Comparison instructions are used to compare values between registers or immediate values. The result of a comparison operation is stored in the EFLAGS register and can be obtained via special instructions. Common instructions are jump instructions which alter the control flow. Using comparison and jump instructions one can implement IF-ELSE-, SWITCH-CASE- or LOOP-statements known from high level languages such as C or C++. Another way to change the control flow are call instructions which call functions. Control flow instructions are important for disassemblers to reconstruct binary analysis primitives from binary files or machine code.

Disassemblers allow reverse engineers to extract the original functionality up to a certain point as assembly language output by analyzing the respective binary files. The resulting disassembly consist of a hierarchical structure of binary disassembly primitives. The primitives can be derived from the disassembly based on essential features or characteristics. We define four binary analysis primitives which are discussed in detail in the following paragraphs.

Instruction Assembly instructions can consist of an address, an operation with its operands in its mnemonic representation and the opcode bytes. They are the lowest possible disassembly primitive.

Basic Block (BB) Basic Blocks (BBs) consist of a sequence of instructions. A basic block has exactly one entry and one exit point. The begin is declared by the jump to the first instruction of it. Its end is determined by another instruction which is potentially altering the control flow. The instructions in a basic block are usually executed linearly without any interruption via control flow altering instructions. Some disassemblers consider call instructions not as basic block breaks while others do.

Function A function is a set of basic blocks which belongs semantically to the function. The disassembled function and its basic blocks ideally match the functionality of the function defined in the original source code although there are several cases where this is not entirely possible. Techniques like function inlining or compiler-optimized subfunctions are not necessarily reconstructible as stand-alone functions. Typically a function contains a start and end procedure which handle the creation and deletion of a stack frame. Furthermore, a function has a signature which is defined by its return type and a list of parameters and their respective types.

Control Flow Graph (CFG) The control flow graph consists of the basic blocks (nodes) with their control flow edges. A control flow graph can be constructed for two scenarios, local and global. Local control flow graphs contain the basic blocks and the related edges for a specific function. Global control flow graphs are constructed for the complete control flow of all available code.

3 Related work

There are various related projects and scientific publications in academia in connection with automated malware analysis and feature extraction. In this section we present related work our research is based on.

3.1 The relevance and challenges of packed malware

Modern malware is distributed to a large extent packed. The spread of packed malware is already growing rapidly since 2006. Therefore, malware unpacking was becoming more and more popular for the investigation of code that is hidden by packing routines.

Brosh et al. described malware packers and detection by demonstrating the effect of packers on anti-malware solutions by comparing different engines by multiple vendors in 2006 [BM06]. They analyzed publicly reported samples from the malware repository WildList [Int21]. The authors found out that 92% of the malware in the wild is packed with runtime packers.

In 2010 *Reynaud* analyzed a large collection of malware containing nearly 100000 samples [Rey10]. As a data basis the author used a honeypot approach for the collection of malware samples. For further analysis the malware samples were dynamically analyzed using multiple virtual machines in parallel and he found out that 95% of the analyzed samples are packed although the data was heavily biased due to the honeypot approach. 56% of the collected samples were from a single malware family.

Ugarte-Pedrero et al. state that the majority of samples are mutated or polymorphic variations [Uga+15]. Malware authors mutate their software to conceal its functionality and to evade anti-virus detection very frequently. The authors contribute a taxonomy of packers by associating them with a model of six different types of packers. For the evaluation of their study they use two different data sets. One data set is packed with off-the-self packers which are either publicly available software or the products of commercial vendors. Additionally there are customized packers which are not used by the general public as a second data set used by the author's for their study. The authors found out that the most common packer type is designed to distinguish between the payload

of the original program and the layers of protection of it. For their data set the authors used malware submitted by users to an online service over several years. To reduce the input set they chose one sample for each malware family per month of equally, based on their submission time distributed ground data.

Calvet et al. analyzed a representative set of malware samples selected via several criteria in 2015 [Cal+15]. The authors focused on successful and popular malware by searching for families that either were target by law enforcement or security community operations and by selecting families that have been frequently mentioned in publications. Their setup *WaveAtlas* is a cluster of 100 machines containing multiple virtual machines allowing the analysis of the samples with an analyzer the authors build upon Intel Pin [Lev]. They found out that 98% of all malware in their sample is packed and that the program logic and the protection logic are typically separated in malware protected by packers.

3.2 Overcoming packed malware - unpacking and sandboxing

Based on the studies listed, one can clearly see that most of the malware samples in circulation are obfuscated with some kind of packer. The topic is part of many publications trying to overcome the problems that packed malware causes in automated analysis procedures. As we want to analyze the capabilities of malware samples it is important to understand how packers affect malware analysis in general and what solutions industry and academia offer to get behind the facade of protected samples.

Royal et al. described the need for automated solutions for uncovering hidden code in malware and presented an approach of automated extraction of hidden code by unpacking malware via execution in 2006 [Roy+06]. They contribute *PolyUnpack*, a software for the automated identification and extraction of packed malware.

Martignoni et al. described a similar approach to uncover the hidden code of packed malware in 2007 [MCJ07]. They present *OmniUnpack*, a kernel-based application for generic unpacking of malware. OmniUnpack is able to monitor target programs and detect the different layers of encoding during runtime. The authors compare their tool with PolyUnpack and claim that their approach is able to deal with packers which make use of anti-debugging, anti-vm or anti-sandbox techniques.

Comparetti et al. [Com+10] presented a paper in 2010 describing general problems of sandbox solutions and presenting a new approach to analyze malware capabilities called *Reanimator*. Sandboxes are able to circumvent malware camouflage techniques like packing or obfuscation. But sandboxes tend to detect only a minor subset of possible actions since malware usually does not execute every potential functionality in a specific environment. Their approach tries to detect dormant functionality

during dynamic execution. This code is being isolated and compared against a malware repository to identify additional functionality that has not been detected in previous analysis steps.

Rieck et al. presented a framework for automated malware analysis with a focus on machine learning in 2011 [Rie+11]. The authors show a framework for clustering and classification of malware to detect behavior of already known as well as novel malware. The approach is described in multiple stages: First, the malware binaries are being monitored and API/system-calls with their respective arguments are being traced for later inspection. These data sets are embedded into a multi-dimensional vector space. Each feature is linked to a dimension so that in the next steps clustering and classification methods can be applied. Those steps can be iteratively repeated so that undefined features can be mapped to novel malware categories while previous detected behavior can be connected to already known classes or behavior.

Beaucamps et al. released a paper in 2012 dealing with abstraction-based malware analysis using a formal approach to summarize malicious behavior [BGM12]. Program traces are abstracted and rewritten to generic behaviors as high-level representation of a given sample. The behavior abstraction can be used to express certain features or capabilities of software and to analyse programs on a higher level.

Eskandari et al. published a paper in 2013 using a hybrid analysis approach [EKH13]. The authors contribute *HDM-Analyser*, an approach based on data mining and hybrid analysis techniques for malware analysis. They claim that their solution can provide better accuracy than static or dynamic analysis techniques.

Miller released a white paper in 2014 discussing hybrid analysis techniques [Mil14]. Hybrid analysis describes a combination of static and dynamic analysis. Malware is analyzed dynamically and the resulting data such as memory dumps or runtime import symbols are analyzed statically. The approach offers a trade-off between strict static or dynamic analysis. While static analysis is complex and time-consuming on modern malware which can be packed or obfuscated it can be applied to malware after unpacking and memory decryption. Although hybrid analysis approaches can be a powerful tool for automated malware analysis the problem of dormant code remains unchanged since it is not possible to reliably determine which code is ever executed.

Damodaran et al. compared three malware analysis techniques which are based on static, dynamic or hybrid analysis using Hidden Markov Models in 2017 [Dam+17]. The authors compare the results of all three analysis strategies in either leaning phases and the respective opposite approach in the detection phase and vice versa. By using this approach they try to understand the advantages and disadvantages of the three analysis techniques and if hybrid analysis has significant advantages over

the other two. The authors evaluate that dynamic analysis is the superior analysis technique in general although they did not focus on detection accuracy or completeness in their work. Therefore, the results may not be readily generalizable today.

Korczynski et al. describe in their paper how to capture the characteristics of malware by detecting code injection techniques and code-reuse attacks in 2017 [KY17]. Automated malware analysis is a complex task since malicious software is designed to complicate analysis with various obfuscation techniques. The authors provide an approach to detect typical malware strategies for dynamic code generation via control flow analysis. They present an implementation called *Tartarus* and compare their approach against artificial and actual malware.

Cheng et al. presented a new generic unpacking prototype in 2018 [Che+18]. They release the tool *BinUnpack* and show the performance improvement compared to other generic unpackers in their study. The authors state that modern malware typically remove the Import Address Table (IAT) of PE files and reconstruct it during runtime to increase the complexity of the protection schema. Their approach tracks the unpacking procedure of packed malware until the IAT is rebuilt in memory. The authors contribute an unpacker following a novel strategy called "rebuild-then-called" instead of the popular "written-then-executed" approach that typically unpacks layer after layer of encryption.

Jenke et al. published another approach for automated and malware unpacking in 2019 [JPP19]. They contribute *Roamer* (Robust Automatic Malware Unpacker), a generic unpacking tool using dynamic analysis for automated malware unpacking. Unpacking is an important step in the process of malware analysis. The authors provide a tool to unpack protected malware automatically by applying a generic approach to detect memory injections and unpacked memory sections during runtime. The project uses a virtual machine running Windows via the VirtualBox [Ora21] hypervisor and consists of two units. A controller running on the host system is controlling the hypervisor while the unpacking procedure is running inside the guest system.

Oosthoek et al. presented an overview about typical malware tactics, techniques and procedures (TTP) and how to measure them effectively in 2019 [OD19]. The authors linked automated malware analysis of Windows malware samples to the MITRE ATT&CK framework [MITb]. The framework consists of an overview over several tactics and techniques used by various threat actors and in attack scenarios. The authors provide a summary of many different techniques used by modern malware mapped to a comparable attacker taxonomy.

Küchler et al. presented a paper in 2021 describing the trade-off between time and performance in sandboxes [Kuc+21]. A sandbox can only monitor malicious processes over a reasonable amount of time. When malware samples have a long execution time, analysts need to balance how long they

want to monitor the execution. On one hand, a short lifetime of a sandbox might be unable to detect malicious behavior as it would not have taken place yet. On the other hand it takes resources if a sandbox monitors a process for too long. This gap becomes more impactful when performing analysis of large batches. The authors provide a study over a large malware collection evaluating how long a typical sandbox scenario should be operated per sample. Their results showed that most of the relevant behavior happens in the first two minutes during execution of a respective malware sample.

Cheng et al. published a new approach for automated unpacking and specially API reconstruction in 2021 [Che+21]. Based on their work on BinUnpack [Che+18], the authors contribute *API-Xray*, a hardware-assisted, automated import table reconstruction tool. The authors aim to automatically generate a new executable file following their approach based on the unpacked code, its analysis and the reconstructed import table.

3.3 Representative data sources for statistical evaluation of automated malware analysis approaches

The selection of malware samples for a representative sample is not easy and depends on the type of approach to the research. There exist several criteria for the selection of malware in the publications we discussed previously. But there are various approaches in scientific publications discussing multiple options to select a statistical representative data set although some approaches suffer from different biases or limitations than others.

Rossow et al. pointed out several weaknesses in papers and articles on malware execution in 2012 [Ros+12]. They discussed the current situation at that time and suggested solutions to both authors and reviewers in the community as even some papers published in highest-ranked security conferences have methodological weaknesses. The authors criticize missing descriptions of the experimental setup and problematic assumptions that have not been elaborated. For an improvement of the situation they suggest four major key points: correct datasets, transparency, realism and safety. Additionally they provide a guideline with several subpoints. They contribute a set of improvements for data sets used for the evaluation of malware analysis tools and techniques and the respective publications. Besides that the authors test popular data sets and check them for violations. The authors come to the conclusion that specially randomly collected malware from public services are heavily biased, e.g. a data set from Kaspersky has a string bias, 80% of the malware is associated with 10% of all available families. *Rossow et al.* claim that a malware repository should contain equally distributed samples following an equal number of samples per family. Therefore, malware data sets for evaluation of analysis tools should be curated to reduce the bias of the data.

Plohmann et al. presented *Malpedia*, a collaboration platform for malware researchers aiming to build a curated corpus of samples and families in 2017 [Plo+17]. Although the idea of collecting large amount of malware is not new, the corpus of the authors is the first open, publicly available repository of sorted, curated and labeled malware. One primary goal of the authors is to create a malware corpus that contains representative data continuously collected over time. Moreover, the authors keep an additional focus on static analysis and provide unpacked malware samples on a large basis. They provide a data set which can be used by researchers for testing malware analysis tools as test bench and reference data.

4 Static Capability Analysis

In this section we analyze how malware capabilities are detectable using static analysis techniques, whether one can explore them statically and to what extent this is possible. First, we introduce a motivation for our approach in general in Section 4.1. We discuss our central research issues referring to the capability analysis of malware and describe the current field of research. The Methodology of our approach is located in Section 4.2. In Section 4.3 we present the implementation of our approach and show the underlying technology and tools we used. We present the procedure of our evaluation and the data set we used in Section 4.4. Additionally we describe the used data set and our methodology. The evaluation and its results are located in Section 4.5. In Section 4.6 is a summary of this part of our work and its different subsections.

4.1 Motivation

Malware is an omnipresent and significant threat in computer systems and networks. The analysis of malware requires expert knowledge, experience and skills. Due to the sheer number of circulating malware samples, a certain level of automation is necessary to keep up with the growth of cyber crime business and the evolution of malware threats. Malware analysts aim to classify malicious capabilities and features to make statements about possible consequences by using static and dynamic analysis techniques.

One of the reasons for the elaborate analysis is the extraction of capabilities of malware to determine the functionality for the recognition of trends and possible threats. A known limitation, for example, is the issue of coverage to evaluate the statements of prior analysis. Previous work does not sufficiently address the issue of dynamic imports of functions and the resulting lack of coverage in static analysis scenarios. *Plohmann et al.* have measured a large corpus of malware and discovered that around 50% of the samples studied use dynamic imports or at least combine dynamic imports with static imports [Plo+17]. In this work we investigate how large malware data sets can be explored statically and automated for their capabilities by taking these imports into account in order to guarantee a sufficiently large coverage.

Oosthoek et al. studied as first researchers the capabilities of malware using a dynamic analysis approach by utilizing a sandbox and mapping their findings to a cyber threat intelligence taxonomy [OD19]. The authors classified the results into techniques and tactics of an industry-proven framework, the MITRE ATT&CK framework. In order to compare the results of our static approach we map our results to the same framework and evaluate the results, taking into account the different approaches that were used. Since we analyze all samples as memory dumps and statically, our approach implies superior coverage compared to dynamic approaches. But since there is no data to such a procedure publicly available we will evaluate this concept and contribute our findings to current academic research.

Research objectives We investigate the possibility of inferring possible malicious behavior of out data set through static exploration. In our work we want to determine the potential of static exploration within the framework of a quantitative study. We investigate what malware typically does, which behavior can be found and which is detectable using static analysis. In order to be able to evaluate the results conclusively we determine how much of the code is actually covered when exploring malicious binaries statically.

4.2 Methodology

By detecting malware capabilities analysts face a large amount of malware and several malware variants. Since there exist large quantities of malware, analysts depend on automation tools such as static analysis tools and specific dynamic analysis tools. Malware keeps close contact to the operating system interfaces. On Microsoft Windows, malware uses the Windows API to interact with the respective interface and to interact with low-level functionality of the operating system. Therefore, analysis of the API calls leads to a significant information gain about the functionality of malware.

The characteristic behavior by the Windows API call actions can be summarized to malware capabilities and can be analyzed in further steps. We analyze the API calls of all samples and determine potential capabilities by associating the results and the meaning of covered API calls. According to *Plohmann et al.*, one can assume that about 50% of the malware samples resolve dynamic imports at runtime [Plo+17]. Therefore we need to put API information into consideration when analyzing malware statically. Static analysis approaches typically need memory dumps or unpacked samples. But there are not always PE headers, sections or import tables available after the sample unpacked itself into memory. Therefore we need a tool that is able to interpret memory dumps but also recover APIs.

Beyond that, we want to produce comparable results. By generalizing certain practices and techniques of malware we base our work on established taxonomies. These taxonomies allow other researchers to compare their approaches against our work on an abstract level and are considered in the industry. Therefore we require a tool chain that allows us to implement these different requirements. The most important requirements we have for our setup are listed in the following enumeration:

1. The software candidate is able to interpret windows binary files
2. Publicly available, open source tool to guarantee expandability and transparent functionality
3. Ability to detect capabilities of malware by using static analysis techniques
4. The tool has support for memory dumps
5. Detection of the use of API functions to enable the widest possible coverage
6. Ability to map the detected capabilities to an industry-proven taxonomy framework

Extensive research has revealed only one software that meets our criteria and is suitable in large parts for our approach. Only *capa*, an open source software released by FireEye[1] in 2020 has the functionality that we need. Capa can detect capabilities in executable binary files and is able to map detected capabilities to the MITRE ATT&CK framework. However, Capa is very new and has not yet been used extensively and on large datasets in any published work.

Since we want to achieve a certain level of statistical relevance we need a sufficient selection of malware samples. In order to ensure a sufficiently large sample we need a large malware corpus. The malware repository must meet several criteria: adequate size, statistical relevance and the absence of any biases. Additionally, a malware repository candidate has to be publicly available to reproduce potential results and to provide transparency. For our work we focus on Malpedia, a malware repository curated by Daniel Plohmann containing more than 2000 malware families for all modern platforms and families that are broadly diversified over a field of different threat actors. Moreover the project offers unpacked samples as memory dumps where we can reconstruct API references through which we hope to achieve particularly broad coverage.

We develop a strategy to implement these key points in a concept. First, we analyze a representative set of malware samples using static analysis techniques. We determine the capabilities of the studied data set and analyze the sample's properties. Then we examine the completeness of our approach. Additionally, we compare the coverage and detected capabilities to the approach of dynamic analysis by

[1] FireEye has been renamed to Mandiant on 4. October 2021 [Inc21]. As some of the URLs are not available under the new company name we keep calling the company FireEye to remain consistent in our work and to avoid potential misunderstandings about our references.

Oosthoek et al. and their results. Moreover, we classify our observations into two different taxonomies, the MITRE ATT&CK framework and the Malware Behavior Catalog (MBC).

We compare our results against dynamically evaluated malware capabilities to get an idea of the coverage and precision of our approach. *Oosthoek et al.* conducted a broad study of malware behavior in 2019 using a dynamic analysis approach [OD19]. The authors analyzed Malpedia and described how to detect malware behavior dynamically and what typical capabilities can be observed. Additionally, the authors assign the capabilities they discovered to the MITRE ATT&CK framework. While the authors examined the malware landscape provided by Malpedia in a dynamic way we determine the capabilities and features via static analysis using capa. We generate a large amount of potential capabilities of malware and additionally measure statistical data of a representative malware repository. Similar to the authors we aggregate our findings and associate them to the MITRE ATT&CK framework. Additionally, we connect the findings to the Malware Behavior Catalog (MBC) framework. The MBC framework is another, newer taxonomy that evolved from MITRE ATT&CK and describes malware objectives. It allows us to group and illustrate malware capabilities in a more precise way as it is specialized on malware capabilities and not threats.

4.3 Implementation Details

In this section we describe the implementation for our malware capability analysis approach. We implement a data generation and evaluation pipeline operating on a malware repository to capture the characteristics of the data set in full breadth. We need several components for the data generation which is necessary for our static malware capability analysis approach. The following levels are the core of our data generation procedure:

1. Generation of disassembly reports of the malware repository
2. Creation of Capa reports targeting the same malware repository
3. Generation of coverage reports
4. Evaluation on a family and sample level
5. Composition of the MBC and MITRE ATT&CK heat maps

First, we build disassembly reports for our malware repository. We chose Malpedia as malware repository and reference data set. Malpedia is comprehensive, publicly available and we can provide additional API call information for the memory dumps. Additionally, the data set is comparable against

the results by *Oosthoek et al.* as they also use Malpedia. Therefore, we can compare our data against their published results.

As next step we generate Capa reports for all malware samples of the malware corpus we want to inspect. The Capa reports contain the capabilities detected by Capa per sample. The reports are merged into a coverage evaluation for the respective sample as a third step. The comprehensiveness of the Capa reports can be evaluated by comparing the amount of functions or basic blocks in our disassembly reports with the output of Capa for the respective sample.

Then we evaluate the output on a family and sample level. It is possible to identify commonalities and differences between samples and families such as frequency of certain capabilities and other characteristics by comparing the Capa reports.

The data flow of our analysis approach is listed in Figure 8. There are two components that analyze the Malpedia corpus directly. The SMDA report generator creates disassembly information of all memory dumps in Malpedia. Additionally, we run a modified version of Capa over the selected samples of the corpus. We build statistical reports of the coverage for each sample individually as well as aggregated over all samples. These statistics are created by merging the information given by the two input generators.

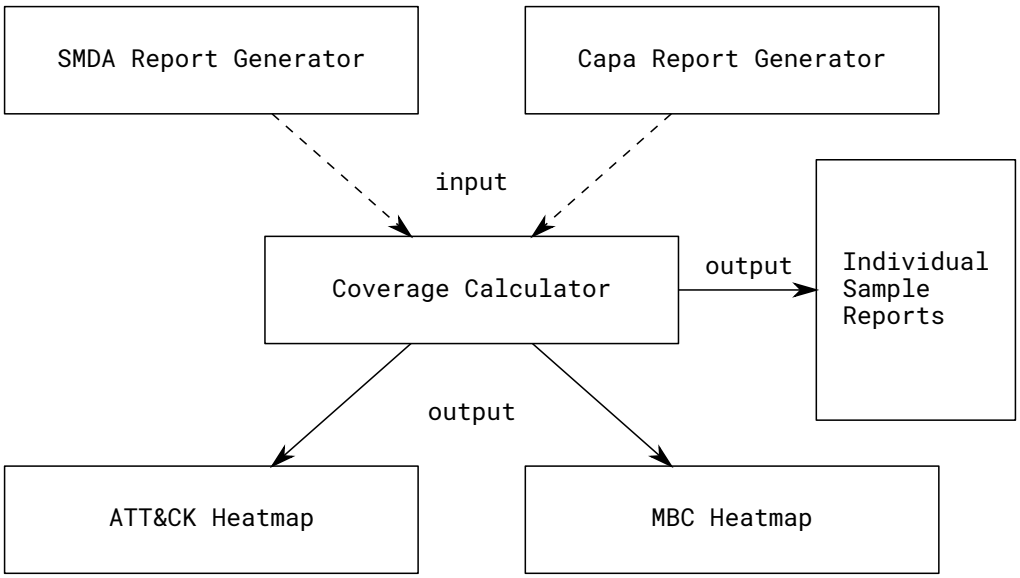

Figure 8: *Illustrated coverage evaluation procedure, showing the data flow of the input data and the resulting output reports and statistics.*

Capa is an important tool for our work and contributes significantly to our results. Its malware capability analysis approach is unique and very advanced in the current implementation. Accordingly, it is a valuable module of our approach. In Section 4.3.1 we introduce Capa and its functionality. Although it is very powerful tool it has some limitations which we discuss more closely. We fork Capa and extend the tool with additional functionality for our specific use case. The changes we made are discussed in Section 4.3.2.

4.3.1 Introducing Capa

Capa is an open-source malware analysis tool by the FLARE team [Fir20a]. The FireEye unit released a framework for automated static analysis of the capabilities of given binaries and malware. Effective malware analysts determine the capabilities and given functionality of analyzed software. To extract valuable knowledge of a sample, analysts need a certain level of experience and reverse engineering skills.

With capa, FireEye introduces a framework containing various rule sets to detect a samples' capabilities by analyzing API calls, functions, strings and constants. The authors describe their algorithmic approach of the automatic capability identification in a two-stage process: First, the code analysis engine extracts the low-level features of a given sample using file features and disassembly features. Then, Capa utilizes its rule repository containing a match context and a specific rule logic. Used on a large scale Capa can identify the capabilities by following the logic and match components. [BR20]

Features and Capabilities Malware capabilities in Capa are defined by a collection of so-called *features*. Capa has different feature categories: *file features* and *disassembly features*. There exist several file features which can be used for the creation and definition of capabilities:

- Extracted from raw file data/structure
- PE Header
- Strings
- Imported APIs
- Exported functions
- Section names

File features target the format of the binary file and need valid structures. The features can be derived from the file directly by parsing the header and fields of the format. Besides file features

there exist also disassembly features. Before features of this category can be detected, Capa needs to disassemble the sample to generate the belonging disassembly. They cover various disassembly entities such as API calls, assembly instructions and string and number references. Figure 9 illustrates the miscellaneous disassembly features.

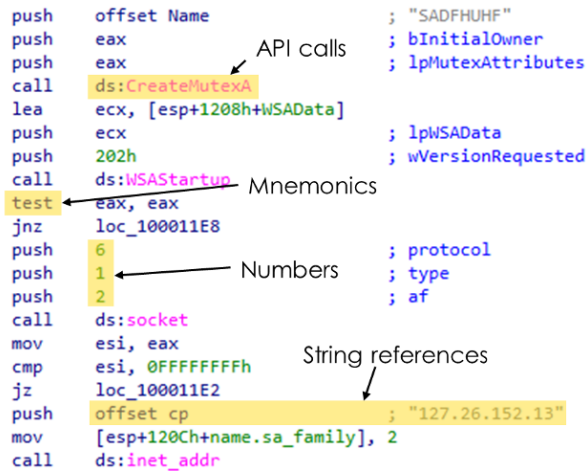

Figure 9: *Capa disassembly features shown in disassembly. [BR20]*

The different features can be compiled to a so-called *capability*. Capa detects capabilities when a certain set or configuration of features is detected in a sample. These configurations can be formulated as Capa rule. We describe the syntax and functionality of Capa rules in the next paragraph.

Capa Rules A set of features combined with a logical filter expression is used to build a Capa rule, an important component of Capa that allows researchers to search for a wide selection of malware capabilities. The developers of Capa supply a large rule repository for the detection of capabilities containing 579 rules in the version we use (Git commit `a207fbba`) [Fir20b]. Capa signatures are based on YAML syntax and are manually written by malware analysts or the open source community. They have two main categories, a head containing meta information and a body combining the features with logical expressions. Listing 1 shows an exemplary Capa rule for the detection of anti-VM instructions by targeting special assembly instructions.

```yaml
rule:
  meta:
    name: execute anti-VM instructions
    namespace: anti-analysis/anti-vm/vm-detection
    author: moritz.raabe@mandiant.com
    scope: basic block
    att&ck:
      - Defense Evasion::Virtualization/Sandbox Evasion::System Checks
        ↪ [T1497.001]
    mbc:
      - Anti-Behavioral Analysis::Virtual Machine Detection::Instruction Testing
        ↪ [B0009.029]
    examples:
      - Practical Malware Analysis Lab 17-03.exe_:0x401A80
  features:
    - or:
      - mnemonic: sidt
      - mnemonic: sgdt
      - mnemonic: sldt
      - mnemonic: smsw
      - mnemonic: str
      - mnemonic: in
      # many misleading hits in runtime code, see #194
      # - mnemonic: cpuid
      - mnemonic: vpcext
```

Listing 1: *Exemplary Capa rule for the detection of anti-VM instructions in a binary file.*

The meta header contains a name of the specific rule, an associated name space and a field for the author of the rule. Additionally, Capa rules contain a field called scope, which selects a validity area that either is targeting basic blocks, functions or the entire file. Furthermore there are fields available for examples of the Capa rule and properties linking to the MITRE ATT&CK framework or the Malware Behavior Catalog.

The features section contains logic filters which can be interpreted as a tree structure, where the expression is composed as tree and the concrete values as leafs. Possible values can be assembly instructions as shown in Listing 1 as well as a set of *Capa features* such as file or disassembly features.

Limitations Capa operates on correctly formatted binary files with proper import tables, sections, valid disassembly and all other assumptions tools typically make. Accordingly, Capa has potential limitations when it comes to malware capability detection. Capa requires correctly aligned binary files, therefore it is not trivial to analyze memory dumps statically without modifications. But since malware comes usually packed, it is important for our work to be able to inspect also memory dumps of malware. In the following Section 4.3.2 we discuss how to implement the key features for our work by extending capa. We use our implementation to analyze a large malware corpus of unpacked malware for creating data that allow us to overview the malware landscape focusing on their capability.

4.3.2 Extending Capa

We need to analyze unpacked malware and memory dumps for our work. To be able to use Capa we extend the tool by increasing the functionality. We add a new processable file type to the input parsing procedure.

Internally, Capa has an interface that allows the usage of multiple disassembly frameworks such as SMDA [Plo21b] and vivisect [Viv21] but also supports the extension by third-party to add new engines. We intercept the SMDA interface and add a method to process malware dumps. The SMDA disassembler is already specialized to process memory dumps, therefore the disassembled assembly instructions are already precise and have a low error rate. But since all imports are lost in memory dumps in general, we need to add this information so that Capa can determine which API functions of the Windows API were used or loaded. ApiScout [Plo21a] is a tool that is able to recover API functions from memory dumps by scanning for their signatures.

This procedure enables one to produce disassembly with additional information about Windows API usage. Due to the API information Capa can detect wether a rule hits the specific basic block or function or not, enabling the creation of reasonable and precise results. For the implementation of our approach we used several libraries and projects. In Table 1 we present the exact versions of our dependencies.

Table 1: *Software versions used in this setup.*

Artifact	Technology	Version
Python	Programming Language	`3.8.10`
Capa	Fork of Capa	`c547519e (tag: v1.6.3, 29.04.2021)` ()
Capa Rules	Capa Rules Repo	`a207fbba` (20.06.2021)
SMDA	disassembly library	`1.5.13`

We forked the version 1.6.3 of Capa (Git commit: `c547519e`) and use the related capa-rules repository (Git commit: `a207fbba`). The version contains 579 rules. We use smda 1.5.13 in this setup and Python 3.8.10.

4.4 Evaluation Procedure

In this section we present the procedure for our evaluation. To evaluate the results we used several resources and data we show in Section 4.4.1. We describe the data sets and the advantages of it for our approach. The methodology we follow is described in Section 4.4.2.

4.4.1 Data Set

To evaluate the resulting software against the data we use a defined data set to create reproducible results. In this section we give an overview about the data set we use for the evaluation.

Malpedia is a community-based malware repository curated by Daniel Plohmann [PE21d]. The malware library and repository offers information about threat actors and groups, popular malware families and various statistical information about malware. *Plohmann et al.* [Plo+17] describe in their paper the selection criteria, degree of sorting and concept of Malpedia as a curated malware corpus. After the initial buildup, the project was handed over to the community which is allowed to use to project and involved in the further development.

Malpedia contains over 2000 unique malware families. All identified samples are clustered to the respective malware family. The project provides a wide range of information about most different malware families targeting various platforms or architectures, threat categories and actors. For every malware family in the repository, at least one sample is available. Popular malware families include a wide selection of different samples which were collected over time.

Besides offering samples and families, the project also provides miscellaneous meta information and signatures. Information about APTs but also organized cyber crime groups are available in the project as well as collected references of malware analysis reports or mentions of the respective malware family or threat actor. Additionally, signatures for the popular pattern matching language YARA are available. Over 50% of the available malware families are covered by YARA signatures [PE21c].

For our work, we use Malpedia in version `82cd8c53` (git commit) from the git repository. This version is from 25. June 2021. In this snapshot are 1584 malware families containing targeting the Microsoft Windows platform. These families contain all in all 4035 unique samples based on their hash in the file name in Malpedia.

We focus on the dumped samples as we want to ensure to get the highest possible coverage. Most samples are at least dumped if there is not an unpacked version available and packed samples cannot be analyzed using our static analysis approach. In Malpedia exist 3087 unique samples based on their hash in the filename that are dumped. Of this data we successfully analyzed 2931 malware samples of 1004 malware families from Malpedia.

4.4.2 Methodology

In this section we describe the method we apply in our evaluation. For our work we analyze the capabilities of malware to gain an impression of typical malware capabilities and elaborate a representative overview. As we want to build a comprehensive report showing the capabilities of a statistically relevant malware repository we implement various concepts. Our approach is derived as follows:

First, we elaborate an overview of malware capabilities in a representative manner in general. We count all capabilities we can detect and aggregate them for the different properties and categories of our data set. This allows us to get a first impression of the variety of malware capabilities and allows us to make first assessments.

Then we further clean up the data by removing any distorting outliers. Moreover we analyze the coverage of our approach by comparing the size of the data set with the coverage through the use of the static capability analysis tool capa. This procedure allows us to estimate the coverage rate of potential static capability analysis approaches and points out possible strengths as well as potential weaknesses.

Afterwards we merge our findings together with our data and link the malware capabilities to categories described by taxonomy frameworks like MITRE ATT&CK and the MBC framework. This results in a picture from which the following can be estimated. We can make assumptions about

the capability coverage of malware via static analysis in general and we see at the same time to which groups they tend to belong to. It also gives a first impression of whether the data collected and generated are at all meaningful or not. Additionally we can judge if the respective capabilities are malicious, how harmful or dangerous the concrete capability is and to which category one could classify them. In a final step we create illustrations of the linked respective samples in the form of heat maps and the counted malware capabilities with the associated taxonomies.

4.5 Results

In this section we present the results of our detailed analysis. We analyze several characteristics and properties of the generated data. All statistical evaluations we performed were composed with a database of 2942 pairs of SMDA and Capa reports. 132 of the reports were empty and we removed them accordingly for a first sighting of the data we use. Therefore the table only contains data we pre-selected to get a statistical result without bias. After the adjustment of the input data we continue our evaluation with 2810 reports.

For a first impression we evaluate the frequency of all malware features detected by Capa and its rules applied to our data set. We discuss the most common malware capabilities we detect and place them in the overall context.

4.5.1 Frequency of Capa Rules

To find out what capabilities are typical implemented in malware, we determined what capabilities are discovered in each sample. We aggregate the data over all analyzed samples to count which feature occur how often. Note that the word *feature* in the context of our statistical evaluation does not necessarily mean *Capa feature* as it is used in the context of the software capa. We have 420 different Capa rules that detected hits in the data by applying 579 Capa rules to the data. Therefore, 72,54% of all Capa rules were hit during our analysis. Table 2 shows the top 30 Capa rules that are most common in malware families from Malpedia.

Table 2: *Top 30 Capa rules ordered by counted hits (absolute, sample and family).*

Capa rule	hits					
	family		sample		absolute	
contain loop	1	892	1	2759	1	630733
delay execution	2	754	2	2250	2	37743
write file	3	744	5	2160	11	9348
calculate modulo 256 via x86 assembly	4	742	3	2184	3	27864
terminate process	5	732	8	2067	15	7360
create or open file	6	715	6	2118	5	13997
contain a resource (.rsrc) section	7	697	15	1593	53	1593
parse PE header	8	684	4	2182	8	11907
read file	9	661	9	1896	17	6407
encode data using XOR	10	656	7	2073	4	27560
link function at runtime	11	641	11	1801	6	13965
create thread	12	609	10	1808	14	8065
query environment variable	13	605	13	1743	22	4625
create process	14	603	14	1647	18	5568
get common file path	15	584	12	1786	12	8983
accept command line arguments	16	573	16	1576	42	2232
delete file	17	527	17	1554	19	5262
create or open registry key	18	518	18	1533	7	12315
allocate memory	19	517	20	1493	9	10957
PEB access	20	486	21	1466	16	6683
get thread local storage value	21	486	28	1240	27	3505
set thread local storage value	22	482	29	1235	35	2855
parse PE exports	23	480	19	1520	32	3001
send data	24	477	25	1306	24	3949
get file size	25	476	22	1457	26	3730
receive data	26	462	27	1248	23	4060
query or enumerate registry value	27	440	24	1334	21	4971
encrypt data using RC4 PRGA	28	438	23	1349	36	2639
allocate thread local storage	29	437	35	1126	45	2065
link many functions at runtime	30	430	38	1040	51	1670

On the left side, the table contains 30 different malware capabilities defined as a rule for capa. The right side shows all hits split in three different categories: family, sample and absolute hit count. All entries in the table contain the rank of the Capa rule within their group and the absolute frequency of hits inside the comparison group.

In the family column, a Capa rule which matched at least one sample inside the malware family was scanned by Capa increases the count per one respective the family is counted as hit. Therefore, all samples in this category are grouped into their malware family. The Capa rules in this table are ordered by the rank of the family counts of the hits by capa.

The sample column on the other hand does not distinguish between malware families, all samples are counted individually. If a Capa signature matches a sample, the sample is counted as hit. The rank of the samples are not numerically ordered but are aligned in the table to the Capa rule.

Additionally, we have a column where we count all hits. Every hit of a rule is counted, no matter if the sample is already recorded. Therefore, multiple hits per sample are possible. Just as in the sample column, the rank is not ordered numerically but aligned to the order of the family column. As Capa rules can operate on different scopes, either basic blocks, functions or on a file level, it is conceivable that the numbers of all absolute hits tend to be biased. In a sample are always more basic blocks available than functions, therefore the potential search space is significantly larger.

The 30 most common rules that identify capabilities within malware families provide clues to the prevalence of certain behaviors in malware in general. Table 2 shows a wide ranges of features. Some capabilities we detected are more common than others, they are strongly unequally distributed.

Additionally, all capabilities have in common that they are not necessarily malicious by their definition. It depends very much on the context if a capability is evil or not. While an ability such as *read file*, *write file*, *send date* and *receive data* can be used to exfiltrate data from a computer it does not provide any information about the actor. The behavior mentioned can belong to highly specialized malware used by APTs or organized cyber crime groups as well as it is part of almost every legitimate onboard tool such as a web browser, mail client or even a text editor.

Therefore, it is necessary to examine the data we obtained in this step in relation to different malware capability categories. We gain further information in several further steps: First, we check how the Capa rule "contain loop" potentially skews the results. We inspect how the rule affects the results, analyze the coverage of the malware repository through Capa and put the results in perspective. Then we match the data of the Capa rule statistics to the MITRE ATT&CK framework and to the Malware Behavior Catalog. For a clear illustration we use a heat map for both approaches.

4.5.2 Coverage of the malware repository

For a fully comprehensive analysis of the coverage of our malware repository data set using Capa we need to find out how the coverage is distributed. To make statements about the completeness in relation to the coverage of malware capabilities of the repository we need to examine the distribution of the capabilities, most particularly with regard to the Capa rule *contain loop*. The Capa rule *contain loop* was counted over 630 000 times in the data set we examined. It is the most common rule regardless of categorization. The second most counted Capa rule, *delay execution*, appears in a magnitude of about 37 700 times only in the data set. According to this, the *contain loop* rule occurs more than 16 times as often.

We inspect how the most significant Capa rule *contain loop* affects the coverage results. Table 3 shows the proportion between the coverage of the respective basic block or function level.

Table 3: *Five-number summary with additional mean and standard derivation for the amount of coverage by Capa of SMDA disassembly reports.*

contains Capa rule *contain loop*	basic blocks		functions	
	yes	no	yes	no
mean	10.44 %	6.87 %	35.69 %	13.63 %
std	0.07	0.06	0.17	0.13
min	0 %	0 %	0 %	0 %
25%	6.12 %	2.76 %	24.65 %	4.27 %
50%	8.55 %	5.06 %	33.17 %	9.12 %
75%	12.74 %	8.80 %	44.83 %	19.46 %
max	50 %	50 %	100 %	100 %

We created a five-number summary with additional mean and standard deviation to allow a clear interpretation of the data. There are four different cases we consider, we provide a five-number summary for each. We group the data according to two criteria, basic block and function level. For both groups we build a five-number summary each, once with the Capa rule *contain loop*, the second without the problematic rule. The meaning of the statistical evaluation is interpreted in the upcoming paragraphs.

First of all one can state that the coverage per sample is lower than the coverage per function in the data in Table 3. Since a function consists of many basic blocks, the amount of basic blocks hit by Capa tends to be smaller than the absolute number of functions hit as the search space is unequally larger.

Plohmann et al. describe in their paper published in 2017 the statistical data of Malpedia [Plo+17]. The malware corpus consisted of 607 families and 1792 samples at this time. The authors describe the control flow graph statistics in a cursory code evaluation among other things. As a data set they filtered out several malware families by certain criteria leaving and building the average of the amount of functions, basic blocks, instructions and function calls in the remaining 382 families. The median of the basic blocks per family in the underlying data is approximately 4262 and functions have a median of 438 per family. They found out that the ratio of functions and basic blocks is about 10 in the data studied. Therefore one can assume that the amount of basic blocks is in the order of magnitude of a factor of 10 larger than the count of functions. This makes it easier to classify the statistical results.

Against the background of this assessment, the available statistics can now be evaluated. In our data we see that the mean is near to the median in both groups and independent of the Capa rule *contain rule*. We see the lower quartile in the functions column is reduced drastically from about 25% to almost 4%. The basic blocks lower quartile is also greatly reduced from about 6% to approximately 3%. Thus one clearly sees that the Capa rule *contain loop* very strongly influenced the result.

The other fields in the basic block table are less volatile, the other quartiles and the minimum and maximum do not heavily change by separating the results. This is also underlined by the standard deviation which is nearly unchanged. On the other hand, the standard deviation is slightly reduced when the Capa rule is removed from the evaluation of the functions. In general, the standard deviation is for all columns on a low level which emphasizes the clear spread of the given data.

The other fields in the function table differ strongly from each other, the quartiles and the minimum and maximum do change heavily although the data converge slightly as quartiles rise. Besides the lower quartile we already discussed the remaining quartiles differ strongly: The median is reduced from about 33% to 9% by about one third, the upper quartile is still significantly reduced from approximately 45% to 19% by about half.

There are several reasons why the coverage is subject to a strong fluctuation. It is conceivable, for example, that the coverage is relatively low in comparison to the complete code as there may be only a few relevant code sections available which are detected via capa. On the other hand it is possible that there are too few Capa rules available to cover all capabilities, especially since the project is relatively new. Besides this possibility, the existing rules may not cover enough cases. If the rules are not comprehensive enough they might not cover rare cases or overlook certain relationships of features in the analyzed file.

To get a deeper impression of the generated data, we connect the data from our results to the taxonomy frameworks MITRE ATT&CK and the Malware Behavior Catalog. The MITRE ATT&CK

framework focuses on the description of operations of advisories attacking in a multi-stage operation arbitrary targets. The MBC framework is another model specialized in describing malware procedures, techniques, tactics and capabilities.

4.5.3 Connecting capabilities to the MITRE ATT&CK framework

The MITRE ATT&CK framework is designed as a large knowledge database of adversary tactics and techniques. The organization developed a matrix for different tactics that are loosely based on the cyber kill chain and connect the tactics to the different techniques and sub-technique. They call their approach MITRE ATT&CK Matrix for Enterprise. While the enterprise model is designed to cover the full attack cycle and all stages of sophisticated attacks, there exists also a version for mobile applications and a *PRE Matrix* for the coverage of preparation techniques. In our work we focus on malware targeting the Windows platform, therefore we focus on the MITRE ATT&CK Matrix for enterprises (version 9.0, released on 29. April 2021).

The MITRE ATT&CK framework matrix can be easily used as a basic structure for the representation of our results by building a heat map data visualization of the objectives. We have 46 different attack tags that were covered in our data. The MITRE ATT&CK framework version 9 has 185 techniques and 367 sub-techniques [MITc].

For our evaluation we map the sub-techniques to the corresponding technique. We analyze the discovered capabilities and their connection to the tactics and techniques in the MITRE ATT&CK framework. We evaluate the capabilities in the context of their respective MITRE ATT&CK techniques and tactics to identify different types of behavior more clearly.

Table 4 and Table 5 illustrate the frequency of certain techniques sorted by tactics via MITRE ATT&CK framework specification in a heat map data visualization. In the upcoming section we describe the overall data structure, what tactics typically do and do not appear in our data, which appear particularly frequently and which data particularly stand out. Furthermore, some techniques will not be detected because there are no rules for them as they have either not been created yet or the respective technique or tactic cannot be reasonable mapped by a Capa rule. Additionally some techniques are assigned to several tactics, that means that a technique is not necessarily matching exactly one tactic.

Table 4: *Enterprise matrix of the counting every technique per sample part 1.*

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion I	Defense Evasion II	Credential Access
Drive-by Compromise	Command and Scripting Interpreter	Account Manipulation	Abuse Elevation Control Mechanism	Abuse Elevation Control Mechanism	Modify Registry	Brute Force
Exploit Public-Facing Application	Exploitation for Client Execution	BITS Jobs	Access Token Manipulation	Access Token Manipulation	Obfuscated Files or Information	Credentials from Password Stores
External Remote Services	Inter-Process Communication	Boot or Logon Autostart Execution	Boot or Logon Autostart Execution	BITS Jobs	Pre-OS Boot	Exploitation for Credential Access
Hardware Additions	Native API	Boot or Logon Initialization Scripts	Boot or Logon Initialization Scripts	Deobfuscate/Decode Files or Information	Process Injection	Forced Authentication
Phishing	Scheduled Task/Job	Browser Extensions	Create or Modify System Process	Direct Volume Access	Rogue Domain Controller	Forge Web Credentials
Replication Through Removable Media	Shared Modules	Compromise Client Software Binary	Domain Policy Modification	Domain Policy Modification	Rootkit	Input Capture
Supply Chain Compromise	Software Deployment Tools	Create Account	Escape to Host	Execution Guardrails	Signed Binary Proxy Execution	Man-in-the-Middle
Trusted Relationship	System Services	Create or Modify System Process	Event Triggered Execution	Exploitation for Defense Evasion	Signed Script Proxy Execution	Modify Authentication Process
Valid Accounts	User Execution	Event Triggered Execution	Exploitation for Privilege Escalation	File and Directory Permissions Modification	Subvert Trust Controls	Network Sniffing
	Windows Management Instrumentation	External Remote Services	Hijack Execution Flow	Hide Artifacts	Template Injection	OS Credential Dumping
		Hijack Execution Flow	Process Injection	Hijack Execution Flow	Traffic Signaling	Steal or Forge Kerberos Tickets
		Modify Authentication Process	Scheduled Task/Job	Impair Defenses	Trusted Developer Utilities Proxy Execution	Steal Web Session Cookie
		Office Application Startup	Valid Accounts	Indicator Removal on Host	Use Alternate Authentication Material	Two-Factor Authentication Interception
		Pre-OS Boot		Indirect Command Execution	Valid Accounts	Unsecured Credentials
		Scheduled Task/Job		Masquerading	Virtualization/Sandbox Evasion	
		Server Software Component		Modify Authentication Process	XSL Script Processing	
		Traffic Signaling				
		Valid Accounts				

Legend

= 2 (min)
= 68 (1. Q)
= 267 (median)
= 754 (3. Q)
= 2544 (max)

Table 5: *Atta&ck 9.0 enterprise matrix, counting every technique per sample part 2.*

Discovery I	Discovery II	Lateral Movement	Collection	Command and Control	Exfiltration	Impact
Account Discovery	Query Registry	Exploitation of Remote Services	Archive Collected Data	Application Layer Protocol	Automated Exfiltration	Account Access Removal
Application Window Discovery	Remote System Discovery	Internal Spearphishing	Audio Capture	Communication Through Removable Media	Data Transfer Size Limits	Data Destruction
Browser Bookmark Discovery	Software Discovery	Lateral Tool Transfer	Automated Collection	Data Encoding	Exfiltration Over Alternative Protocol	Data Encrypted for Impact
Domain Trust Discovery	System Information Discovery	Remote Service Session Hijacking	Clipboard Data	Data Obfuscation	Exfiltration Over C2 Channel	Data Manipulation
File and Directory Discovery	System Location Discovery	Remote Services	Data from Information Repositories	Dynamic Resolution	Exfiltration Over Other Network Medium	Defacement
Network Service Scanning	System Network Configuration Discovery	Replication Through Removable Media	Data from Local System	Encrypted Channel	Exfiltration Over Physical Medium	Disk Wipe
Network Share Discover	System Network Connections Discovery	Software Deployment Tools	Data from Network Shared Drive	Fallback Channels	Exfiltration Over Web Service	Endpoint Denial of Service
Network Sniffing	System Owner/User Discovery	Taint Shared Content	Data from Removable Media	Ingress Tool Transfer	Scheduled Transfer	Firmware Corruption
Password Policy Discovery	System Service Discovery	Use Alternate Authentication Material	Data Staged	Multi-Stage Channels		Inhibit System Recovery
Peripheral Device Discovery	System Time Discovery		Email Collection	Non-Application Layer Protocol		Network Denial of Service
Permission Groups Discovery	Virtualization/Sandbox Evasion		Input Capture	Non-Standard Port		Resource Hijacking
Process Discovery			Man in the Browser	Protocol Tunneling		Service Stop
			Man-in-the-Middle	Proxy		System Shutdown/Reboot
			Screen Capture	Remote Access Software		
			Video Capture	Traffic Signaling		
				Web Service		

Legend

= 2 (min)
= 68 (1. Q)
= 267 (median)
= 754 (3. Q)
= 2544 (max)

Both tables share the same legend which illustrates the amount of samples covered by each ATT&CK technique through the coloring of the cells in a spectrum from yellow to red. The white cells contain techniques that were not discovered in any sample. The colors shown are distributed proportionally according to the number of samples hit. The minimal covered ATT&CK technique was only hit by two samples and is colored in the brightest yellow while the most often detected technique is colored dark red. Between the extremes, all colors of the color spectrum are used depending on the number of hits, not only those depicted in the legend.

Table 4 shows the first of two parts of our evaluation of the detected malware capabilities classified into the MITRE ATT&CK framework's enterprise matrix. There are several tactics in the table described, each tactic is completed by a collection of associated techniques. In this context, the term *tactics* describes the type of attack and the category. All belonging techniques to each tactic are listed as a column although some columns are split into parts for a better readability. As every technique is counted in our previous evaluation we illustrate the frequency of a given technique by means of color in our heat map to underline the visual effect and clarify the context.

In Table 4 we see the first part of the MITRE ATT&CK framework tactics. We discuss several tactics to evaluate their impact on the data as well in terms of the actual relevance in the context of malware capabilities and potential limitations of the techniques and tactics regarding to malware analysis in general.

The first tactic, *Initial Access*, is not covered by any Capa rules that are part of the results from our data. Initial access describes a tactic to compromise individual users or exploitation of available services. Although we cannot be certain about all results due to the nature of our approach as well as potential limitation of the data set it is plausible that this tactic is not covered by our data. The detection of these techniques might involve server log files or various other, additional information that is typically not available in the disassembly of memory dumps and therefore maybe not covered by Capa.

Execution is a summary of various techniques that execute artifacts on a target host. We detected some techniques that are common in the samples we analyzed in our data set such as *Command and Scripting Interpreter* and *Shared Modules*. Both are among the most common techniques used by malware in Malpedia.

The techniques of the *Persistence* tactic are common for tools that try to persist the current access. They are common in attacks and can be found in our data although they do not dominate our findings. All of the detected findings in this tactic are also listed in *Privilege Escalation* as for example the

respective techniques like *Boot or Logon Autostart Execution* or *Create or Modify System Process* can be used for both.

The *Privilege Escalation* category holds miscellaneous techniques that are commonly used by advisory to potentially escalate the current privileges. Common techniques are summarized in this tactic such as *Process Injection* or manipulation of the task scheduling engine by adding new tasks, events or elements to the auto start.

Defense Evasion is split into two parts for a better overview although they belong to the same category in the MITRE ATT&CK framework. The tactic contains various techniques that are common to prevent any kind of detection. Most frequent techniques that we could detect in our data set are sandbox or virtualization evasion and the obfuscation of files or information techniques.

Credential Access, is a collection of techniques that gain access to credentials via different attack vectors such as brute force attacks, network sniffing e.g. man in the middle attacks, credential dumping from the operating system and many more. Although many malware families such as Clipsa [Rub19] also known as Sysraw Stealer [PE21e] from Malpedia have functionality to use a lot of the techniques in this category, only the techniques *Credentials from Password Stores* and *Input Capture* were discovered in our data set.

We continue with our evaluation of malware capabilities and the MITRE ATT&CK framework by discussing the other heat map. Table 5 shows the remaining techniques and tactics of the framework and illustrates the frequency of the different capabilities we detected in our evaluation. The tactics *Lateral Movement* and *Exfiltration* are not discovered in our data set. This result is not to be expected as both categories contain techniques that are known within the Malpedia malware database. Some malware families such as ComRat [Fao20] are known for e.g. the replication via removable media which is assigned to the tactic *Lateral Movement* or *Exfiltration Over C2 Channel* which is a technique of *Exfiltration*. We present possible false negatives or limitations of our approach and the used technologies in the upcoming paragraphs after describing the rest of the table.

Discovery is split into two different columns although they are the same tactic in the MITRE ATT&CK framework. The tactic consist of methods that are used by adversaries to collect knowledge about the targets like computer systems or networks. Techniques such as *File and Directory Discovery* and *System Information Discovery* are one of the most often techniques used by the malware we analyzed. In general the tactic is widely covered by hits of different samples.

Collection is a tactic that refers to collecting data by threat actors. Common techniques that we detected in our data are collected data from clipboard, input devices, screen, archives or data from

information repositories. While we detected many capabilities of this kind of tactic in our data set using our approach there are still techniques that are not found using Capa. Attacks like *Man in the Browser* or *Man-in-the-Middle* are used by online banking trojans in many cases. Malware families like UrlZone, Vawtrak [Pro16], Citadel [Seg12] and many more use these techniques frequently and Malpedia has a large data set of multiple samples of these families [PE21f] [PE21a]. Therefore, Capa should have detected the those capabilities for the respective samples. There might be no Capa rules available for the associated MITRE ATT&CK technique or the rules do not cover the implementation of the malware samples.

Command and Control is a tactic category containing various elements that are typically used to communicate with a compromised system and control it via commands. The technique *Ingress Tool Transfer* was found in our data set and is used by some samples. This is a technique where advisories transfer tools to compromised hosts e.g. using board tools such as ftp [Cor20]. Techniques like *Remote Access Software*, *Traffic Signaling Web Service* or *Data Obfuscation* and *Data Encoding* were not found in our data although it is very likely that malware that is available in our input data implemented this technique.

The last tactic, *Impact*, contains multiple techniques that are were also detected. Besides the covered techniques like *System Shutdown/Reboot*, *Service Stop*, *Disk Wipe*, *Resource Hijacking* and many other there are again some techniques that are unexpectedly not covered like *Data Manipulation* or *Data Encrypted for Impact*. Both techniques are widely used in malware and could be associated to ransomware. There are plenty of ransomware samples available in Malpedia.

As one has seen there are many malware capabilities covered by techniques and the respective tactics. Some of the techniques are assigned ambiguously, like *Process Injection*. They appear in different tactics categories and cannot always be completely broken down at this point as various information may be missing since they are not available through the abstraction level. On the other hand, a lot of techniques are not covered by Capa. It is possible that the rules are not complete enough or techniques of the MITRE ATT&CK framework are not connected to the Capa rules yet. On the other hand, Further there are many techniques and tactics of the framework which do not fit completely out use case. Many technologies and malware characteristics created by threat actors and adversaries are not detectable in file or disassembly based Capa features such as disassembly entities or file format features.

Comparison with the publication by *Oosthoek et al.* As we are interested in a comparison between static and dynamic analysis approaches and the resulting coverage we compare our results against the dynamic approach by *Oosthoek et al.* although the comparison is limited due to the different

data and methodology. Compared to the previous research by *Oosthoek et al.* we have several differences in the evaluation which must be taken into account when considering the data. Most importantly, the authors used a dynamic analysis approach by utilizing the sandbox Joe Sandbox Cloud [Joe21b]. Additionally, we used Capa and its rule engine for matching hits to the MITRE ATT&CK framework while the authors used a custom implementation. They used a reference file for the matching between the sandbox signatures and

The authors did not state which version of the MITRE ATT&CK enterprise framework they used although their data suggests that version 3 was used as the techniques for the tactic *Initial Access* miss the technique *External Remote Services (T1133)* which was added in version 4 [MITa]. In contrast we use version 9.0 of the MITRE ATT&CK enterprise framework [MITc]. Although the frameworks are comparable, the MITRE ATT&CK enterprise framework continually kept growing. Therefore we have a larger space of capabilities potentially matching the respective ATT&CK techniques. The version of the framework the authors used has 11 different tactics, containing only the following points [OD19]: 62 attack techniques have been covered out of 223 techniques in version 3.0. As only recent versions of MITRE ATT&CK introduced sub-techniques the version by the authors does not contain them. The version we used contains 185 techniques containing 367 sub-techniques. Therefore, the illustration as well as the coverage of all ATT&ck elements deviate due to the different sizes of the search spaces.

Besides different versions of the frameworks and their great differ we also we can also see a difference in the data selection. The authors analyzed samples from 951 malware families from Malpedia. Of these families the most up to date samples were used in order to detect network usage of active malware families during runtime. We sucessfully analyzed 1004 malware families with 2931 malware samples.

The tactics *Initial Access*, *Lateral Movement* and *Exfiltration* are not covered at all by our approach. In the data by *Oosthoek et al.* there are some hits but the category is relatively low covered. The tactic *Command and Control* has only one covered technique in our data, *Ingress Tool Transfer*. Compared with the data of the authors we have significantly less techniques detected in our data.

The tactics *Execution* and *Discovery* are both medium covered. On the other hand, the tactics *Persistence* and *Privileges Escalation* have certain similarities and some techniques are not equally covered. The coverage of the tactic *Defense Evasion* suggests that the evaluated data of *Oosthoek et al.* include unpacking routines and other obfuscation schemes which are not part of our data as we performed our analysis on memory dumps. Therefore, it is possible that the distinction between the actual action of the malware and the packing stuff is not possible afterwards. This reason also makes comparability more difficult.

The MITRE ATT&CK framework has some limitations for our use case. As the tactics are based on adversaries and threat actors that target any organization, but malware capabilities do not match their strategy in general. Most of the tactics and techniques do not fit the approach malware follows or what objectives they typically have. Although the MITRE ATT&CK framework does not perfectly match it is a very useful tool for putting the data we generated in a more specific context. As a result, we use an additional framework for the visualization of important capabilities that is specially designed for malware behavior categorization, the Malware Behavior Catalog (MBC) [MBC21a].

4.5.4 Linking the results to the Malware Behavior Catalog

The Malware Behavior Catalog (MBC) is a standardized malware behavior catalog containing malware specific objectives and behaviors [MBC21a]. Although the framework has many similarities with the MITRE ATT&CK framework there is no direct formal relationship between them [Bec20]. The MBC extends the MITRE ATT&CK framework by adding several malware specific techniques and tactics to the framework and the project directs the focus on malware by removing more than 100 techniques that do not specify malware behavior or capabilities from MITRE ATT&CK. The MITRE ATT&CK framework defined tactics and techniques to focus on adversary behavior but MBC defines objectives and behaviors. Objectives can be described as the reason why malware perform any action. Behaviors on the other hand are the concrete action it does. [Bec20] Besides regular behaviors there exist also so-called *micro-behaviors* which are also associated with *micro objectives* but are less specific. Typically these behaviors do not necessarily imply malicious use, also legitimate software uses a lot of functionality that is connected to the objectives. [MBC21b]

For a proper visualization we decided again to split all categories and their entries into different figures to ensure a good overview. The different illustrations Table 6, Table 7 and Table 8 show the matches of our results associated with the respective objectives and behaviors specified in the MBC as a heat map visualization. Just like the description of the previous connection to the MITRE ATT&CK framework, we describe the association of our results and the MBC table by table. All heat maps are constructed as tables that contain the MBC objectives in the table header and the corresponding behaviors are listed in the columns in the table body. Again, comparable to the MITRE ATT&CK heat maps, all tables share the same legend. The legend illustrates the amount of samples covered by each MBC behavior through the coloring of the cells in a spectrum from yellow to red. The white cells contain behaviors that were not discovered at all. The colors shown in the legend are distributed proportionally according to the number of samples hit. The minimal covered MBC behavior was only hit by one sample and is colored in the brightest yellow while the most often detected technique is

colored dark red. Between the extremes, all colors of the color spectrum are used depending on the number of hits, not only those depicted in the legend.

Table 6 shows the first of three parts of our evaluation of the data set we generated and the connection to the MBC. The objective *Anti behavioral analysis* is a collection of various behaviors that can be observed by malware that tries to prevent being analyzed with tools such as debuggers or virtual machine [MBC21b]. Malware typically tries to hide from analysts by using evasion techniques and this was confirmed by our observations. We found out that a lot of malware uses *Debugger Detection* and *Dynamic Analysis Evasion* techniques to detect dynamic analysis by malware analysts. Other behaviors that can be observed are e.g. the runtime detection of sandboxing tools, emulators and virtual machine hypervisors.

Besides dynamic analysis, malware also tends to evade static analysis as the *Anti static analysis* objective shows. We were able to trace the behaviors *Disassembler Evasion* and *Obfuscated Files or Information*. Both are common techniques used by malware to prevent malware analysts from analyzing the respective samples [MBC21b]. The MBC objective *Collection* contains behaviors that gather (sensitive) information that may be exfiltrated in further steps [MBC21b]. It collects the same behaviors as the objective *Credential access* although it more specifically describes behaviors trying to obtain credentials for accounts to compromise target systems. We detected *Keylogging* and *Screen Capture* behaviors. Additionally, we found *Command & Control Communication* behavior (and the respective objective of the same name) in our data set.

The both micro objectives *Communication micro abjective* and *Crypthography micro objective* are almost completely used, only *SMDA Communication* and *WinINet* were not observed. *Communication micro objective* describes miscellaneous types of network communication used by malware [MBC21b]. This kind of behavior is very common as malware typically loads other components and sends or receives various data from networks among others. Most malware in our data set communicates via HTTP and DNS, communication via sockets are also very common. Although network communication on a lower level typically involves sockets it is possible that the Capa rules we used for the detection of HTTP or DNS capabilities no socket features and therefore we have less socket communication than e.g. HTTP. The *Cryptography micro objective* on the other hand is a collection of malware behaviors that deal with software-based encryption. Behaviors of the associated objective make use of cryptographic libraries, hash functions or (pseudo-) random number generation sequences. Additionally, they generate or use encryption keys and encrypt or decrypt data. Interesting is the fact that the encryption of data is way more common than the decryption behavior. It is conceivable that the higher rate of encryption behavior is related to ransomware.

Table 6: *MBC matrix, counting every technique per sample part 1.*

Anti behavioral analysis	Anti static analysis	Collection	Command and control	Communication micro objective	Credential access	Cryptography micro objective
Capture Evasion	Call Graph Generation Evasion	Cryptocurrency	C2 Communication	DNS Communication	Cryptocurrency	Crypto Library
Debugger Detection	Disassembler Evasion	Hooking	Domain Name Generation	FTP Communication	Hooking	Cryptographic Hash
Debugger Evasion	Executable Code Obfuscation	Input Capture	Remote File Copy	HTTP Communication	Input Capture	Decrypt Data
Dynamic Analysis Evasion	Executable Code Optimization	Keylogging		ICMP Communication	Keylogging	Encrypt Data
Emulator Detection	Executable Code Virtualization	Screen Capture		Interprocess Communication	Screen Capture	Encryption Key
Emulator Evasion	Obfuscated Files or Information			SMTP Communication		Generate Pseudo-random Sequence
Executable Code Virtualization	Software Packing			Socket Communication		
Execution Guardrails				WinINet		
Hooking						
Memory Dump Evasion						
Sandbox Detection						
Software Packing						
Virtual Machine Detection						
						Legend
						= 1 (min)
						= 72 (1. Q)
						= 594 (median)
						= 1493 (3. Q)
						= 2250 (max)

Table 7: *MBC matrix, counting every technique per sample part 2.*

Data micro objective	Defense evasion	Discovery	Execution	Exfiltration	File system micro objective	Hardware micro objective
Check String	Alternative Installation Location	Analysis Tool Discovery	Command and Scripting Interpreter	Archive Collected Data	Alter File Extension	Install Driver
Checksum	Bootkit	Application Window Discovery	Conditional Execution	Automated Exfiltration	Copy File	Load Driver
Compress Data	Bypass Data Execution Prevention	File and Directory Discovery	Exploitation for Client Execution		Create Directory	Simulate Hardware
Compression Library	Component Firmware	SMTP Connection Discovery	Install Additional Program		Create File	
Decode Data	Covert Location	Self Discovery	Prevent Concurrent Execution		Delete Directory	
Decompress Data	Disable or Evade Security Tools	System Information Discovery	Remote Commands		Delete File	
Encode Data	Execution Guardrails	Taskbar Discovery	Send Email		Get File Attributes	
Modulo	Hidden Files and Directories		Send Poisoned Text Message		Read File	
Non-Cryptographic Hash	Hooking		User Interaction		Read Virtual Disk	
Use Constant	Indicator Blocking				Set File Attributes	
	Install Insecure or Malicious Configuration				Writes File	
	Modify Registry					
	Obfuscated Files or Information					**Legend**
	Polymorphic Code					= 1 (min)
	Process Injection					= 72 (1. Q)
	Rootkit					= 594 (median)
	Self Deletion					= 1493 (3. Q)
	Software Packing					= 2250 (max)

Table 7 illustrates the second part of our data. The *Data micro objective* is a collection of data manipulation behaviors [MBC21b]. We detected malware which modifies data by encoding, compression or decompression and the use of respective libraries as well as building hashes and checksums among other things. Due to the nature of this objective most findings are very generic and not necessarily malicious or malware specific, especially since this objective is one of the micro objectives. They tend to be more general by behaviors that are not very abstract and therefore generic in software as they are too low level to describe broader, malicious behavior.

Defense evasion is an objective which was hardly hit or not hit at all by the Capa rules we applied to our data set. The most significant detection is the *Obfuscated Files or Information* behavior [MBC21b]. Some of the behaviors as part of objectives we have seen in the last table are also listed here, such as *Process Injection* and *Hooking*, which were not detected in our data set at all. Behaviors like *Modify Registry* or *Software Packing* among others are common in samples and families in Malepdia but could not be detected by the respective Capa rules in our data set.

Discovery on the other hand is widely covered, although on a moderate level. The objective describes malware that uses certain behaviors trying to gain information and knowledge about the target system or network [MBC21b]. *Execution* and *Exfiltration* are not covered by our results except for the behavior *Install Additional Program* from the former objective. As with the other objectives there are several behaviors not detected although they are part of the malware in Malpedia. Behaviors such as *Send Email* and others could not be found in the data although families like Emotet [PE21b] which are in the Malpedia database and spread e.g. via Email [NZ20].

The *File system micro objective* is another micro objective, it is a collection of miscellaneous behaviors for file manipulation [MBC21b]. We detected various behaviors for file or directory creation and deletion as well as copy, read and write operation to files. In addition, the behaviors for getting and setting file attributes were also found in our data set. Again, the category of behavior and the objective itself are not necessarily malicious and therefore only have a low informative value. Additionally we detected the behaviors *Install Driver* and *Simulate Hardware* of the *Hardware micro objective*. The objective contains micro-behaviors that are related to hardware.

Table 8: *MBC matrix, counting every technique per sample part 3.*

Impact	Lateral movement	Memory micro objective	Operating system micro objective	Persistence	Privilege escalation	Process micro objective
Clipboard Modification	Malicious Network Driver	Allocate Memory	Console	Bootkit	Hooking	Allocate Thread Local Storage
Component Firmware	Remote File Copy	Change Memory Protection	Environment Variable	Component Firmware	Kernel Modules and Extensions	Check Mutex
Compromise Data Integrity	Send Email	Free Memory	Registry	Hidden Files and Directories	Modify Existing Service	Create Mutex
Data Destruction	Send Poisoned Text Message	Heap Spray	Wallpaper	Hooking	Process Injection	Create Process
Data Encrypted for Impact	Supply Chain Compromise	Overflow Buffer		Install Insecure or Malicious Configuration		Create Thread
Denial of Service		Stack Pivot		Kernel Modules and Extensions		Resume Thread
Destroy Hardware				Malicious Network Driver		Set Thread Local Storage Value
Disk Content Wipe				Modify Existing Service		Suspend Thread
Exploit Kit Behavior				Modify Registry		Synchronization
Exploitation for Client Execution				Registry Run Keys / Startup Folder		Terminate Process
Generate Fraudulent Advertising Revenue				Remote Access		Terminate Thread
Manipulate Network Traffic				Remote File Copy		
Modify Hardware				Shutdown Event		**Legend**
Remote Access						= 1 (min)
Resource Hijacking						= 72 (1. Q)
Spamming						= 594 (median)
						= 1493 (3. Q)
						= 2250 (max)

Table 8 contains the last heat map showing the connection of our results to the MBC framework. The objective *Impact* is a collection of behaviors of malware which aim to manipulate, interrupt or destroy systems or data [MBC21b]. We detected the behaviors *Clipboard Modification*, *Disk Content Wipe*, *Modify Hardware* and *Remote Access*. Several other behaviors were not covered by our data although the objective itself unites many behaviors that potentially are part of the capabilities of the malware in Malpedia. Behaviors like *Data Destruction*, *Spamming* or *Exploit Kit Behavior* are very common in most families we already briefly addresses in the former paragraphs.

Lateral movement is an objective that describes the behavior of malware trying to propagate on a system or to move for the compromise of other machines . The malware can move actively and passively, e.g. by actively manipulating files on the current system or by passively being distributed via email or various other possible attack vectors [MBC21b]. We detected no behaviors of this category in our data set although we already know that the behavior *Send Email* which is also part of other objectives.

The *Memory micro objective* is a summary of memory manipulation behaviors [MBC21b]. Typical behaviors are e.g. the allocation, return or change of the privileges of memory but also manipulation techniques like heap spraying. In the detected data we found only the behavior of memory allocation. Atypical is the absence of the detection of the *Free Memory* behavior as it is a common opponent of the *Allocate Memory* behavior we detected. Unlike clearly malicious activities like exploitation strategies, the allocation of memory is very common and therefore not relevant.

Another objective, the *Operating system micro objective*, contains behaviors linked to operating systems [MBC21b]. It contains in version 2.1 of the MBC framework four behaviors and all are in our data set. The capabilities we evaluated modify the console, environment variables, the registry and the wall paper. The most common behavior, registry modification, targets generic registry usage such as read and write keys of the registry as opposed to the *Modify Registry* in the *Persistence* objective which describes behavior that persists by manipulating the registry like adding malicious software to the system's auto-start.

The objective *Privilege escalation* was not covered at all by our evaluation of the malware capabilities. We could not detect any sample using behaviors that are part of the objective which collects behaviors trying to obtain privileges of higher levels [MBC21b].

Additionally, the *Process micro objective* describes various behaviors that are related to processes [MBC21b]. Besides *Synchronization*, all of the behaviors related to mutexes, creation and termination of threads and processes are covered. The most frequent behavior is *Terminate Process*, which is not particularly surprising due to the nature of software in general. The creation of threads

and processes occurs also frequently. Moreover, the characteristics of this micro objective are not very specific and are therefore not particularly strong.

In this section we have presented the results of our evaluation. The reader will find a comprehensive classification of the results in the next section. We discuss the evaluation and give an overview about the results in Section 4.5.5.

4.5.5 Discussion

In this section we discuss the evaluation of the malware capabilities we detected by our approach. We have looked at many different points in this evaluation. We discovered that most of the malware capabilities we analyzed with our approach and the software Capa were only to a very low level covered. In general, the coverage of functions and basic blocks by Capa and the rule set we used is low as shown in Figure 3. The coverage of the third quartile is about almost 20% for functions and almost 9% for basic blocks. A static analysis of memory dumps should already be able to detect a large part of the data. Therefore one can assume that dynamic analysis approaches theoretically can only record a fraction of this data at runtime although there are no studies that consider this scenario. In addition, potential sources of error such as disassembly errors and memory dumping in general complicate the clean collection of data.

For a detailed consideration we have created a mapping between the analyzed samples in our data and the MITRE ATT&CK framework. Additionally we briefly compared the data against the approach by *Oosthoek et al.* which is the only work we know of that analyzed a publicly available malware repository and mapped the data to a industry-proven taxonomy at the same time. Although the data differs in many points we see a slight trend in the data. The mostly covered techniques in their data are also approximately well covered in our evaluation. The same applies to the less covered techniques. Additionally they probably cover more techniques than we do in general as they analyzed packed malware. The unpacking procedure is therefore part of their analysis while we do explicitly exclude the unpacking methods as we use dumped malware.

On the other hand we mapped our data also to the MBC framework which is a good fit due to the specialization on malware behavior and objectives. The MBC framework tends to be a good candidate as a potential taxonomy for malware capabilities. While it covers many behaviors it still lacks of precision for certain techniques. For example the objectives *Collection* and *Credential Access* contain the same behaviors. The behavior *Command and Control* on the other hand has only two behaviors, missing several points the ATT&CK framework has included and that are potentially useful as malware behaviors, e.g. the technique *Remote Access Software*.

Additionally, it is possible that Capa needs improvement being a very new tool. While it is a superior tool for capability detection in the context of static analysis, the rules have some mismatches in the taxonomies. Fore example, *Process Injection* is not covered in MBC, but in the ATT&CK framework we collected several hits on samples. There is a certain gap between the taxonomy and coverage of techniques of the three different approaches.

In addition, we run the risk that the data selection can distort the result. Due to the rapid further development in this subarea, tools, frameworks and also the data change quickly. Accordingly, the comparisons between the static and dynamic approach of two different versions of MITRE ATT&CK cannot be made with the best possible precision. Nevertheless, the data give us an initial assessment of coverage, namely that we can basically achieve good coverage with our approach. The coverage of the MBC framework further emphasizes this aspect. In order to further evaluate the coverage and thus the informative value of our investigations, we additionally test our approach dynamically in Section 5.

4.6 Summary

In this section we provide a short summary to the reader. In Section 4.1 we begin with a short motivation. We show how we fit into the current state of research and briefly present our approach. Additionally we present a short categorization of the research objectives which we want to take a closer look at in this work.

In Section 4.2 we discuss how we can describe this idea in more detail. We create a red thread through this chapter and form a first approach for implementation. The research questions are triggered and described in detail. In Section 4.3 we present the implementation of our approach. We describe Capa and its rules, the toolchain we implemented to derive data and statistics for a closer look at the capabilities of malware.

In addition we discuss the limitations of it and how to expand Capa accordingly to meet our requirements. Section 4.4 contains a briefly description of the evaluation procedure. We discuss the data set and the corresponding key data as well as the methodology we follow in our evaluation. In Section 4.5 are our results of the evaluation as well as a discussion of them. We show the frequency of Capa rules applied to a large malware repository. Additionally, we describe the coverage of the data set and correct a possible bias by removing outliers. Then we relate the data to the two taxonomies MITRE ATT&CK framework and MBC.

5 Runtime Coverage Analysis

For a more comprehensive picture, in addition to the static analysis of malware capabilities, we also want to analyze runtime capabilities observable during the runtime of malware. In the context of this quantitative analysis, we face different challenges than in the static analysis. The motivation of our approach in our field of research is introduces in Section 5.1. Its implementation is described in Section 5.3. Then we present our results and the evaluation of our approach for dynamic analysis in Section 5.4. We close this section with an outlook and a short summary in Section 5.5.

5.1 Motivation

Malware is a significant threat in modern and complex information society. As already shown in the previous sections, malware is a fast growing component of the threat landscape and automated analysis approaches are an important tool to counteract these threats.

Reliable malware unpacking is complex and still considered unsolved as shown in Section 3. There are several problems with current state-of-the-art solutions. Many of them are not publicly available, many of them are prone to errors and most of them are not able to create executable, unpacked versions of packed malware samples. Therefore, static analysis has to deal with unpacking and memory dumping to overcome packed malware. Most of the time, missing runtime information can decrease the precision of the results such as missing API imports after dumping the memory of the target. The lack of static analysis can be overcome by using memory dumps and appropriate cover of them. If analysts want to solve this issue, they usually need to recover API imports, dump or unpack the malware and analyze it partially manually which requires a lot of time, expert knowledge and experience.

As opposed to static analysis approaches, there exist dynamic malware analysis techniques. Automated dynamic analysis approaches suggest that most of the action of a malware sample is revealed during runtime and analysts can rely on the assessment of tools like sandboxes to a certain extent. But how much analysts must rely on the sandbox is not clear. There is no knowledge base or available information in other publications about how much they actually cover of the malicious behavior in real-world scenarios or how much code is actually reached.

Sandboxes are popular with malware analysts as they have many advantages. They detect a lot of API calls and input/output devices, network or file system interaction after a sample is launched without the need of expertise or deep understanding of the underlying platforms. Modern sandbox solutions like *Joe Sandbox Cloud* [Joe21b] even offer partially static exploration of inspected samples. But the samples remain black boxes to the analyst when they are executed in a sandbox. Sandboxes do execute the samples of interest like a black box and malware analysts do not learn about the target's internals. Therefore, there are major difficulties dynamic analysis approaches need to face. As it is not possible to reliably determine the end of protection schemes, the malware's action and the unpacking handling procedures get mixed together. This potentially reduces the informative value.

Research objectives In our work we want to investigate the different aspects of analysis coverage, e.g. detection of malware capabilities. Besides the detection of malware capabilities using a static analysis approach we also want to see how much of the capabilities is actually used or executed during runtime. This information can help to assess and explore the completeness of both types of analysis. The runtime code coverage of dynamic malware analysis needs to be explored and evaluated as there is no concrete data published. Ideal setup conditions would mean that one is able to detect self modifying code, trace a malicious sample across all processes and threads and reliably detect the point of time where the last unpacking stage of a malware end and the actual behavior or action begins. Through the data obtained from such a procedure, the coverage of the different types of analysis can be measured and statistically evaluated.

5.2 Methodology

We aim to investigate how much runtime code is actually covered on a sample when it is executed. For our runtime coverage analysis we need an environment that allows us the isolated execution of malware. The term *coverage*, in the context of dynamic analysis, describes the coverage of capabilities of an inspected sample. Therefore, the function has to be executed during the dynamic analysis. Dynamic malware analysis is typically performed in virtual machines which are controlled by a hypervisor. Inside the virtual machine can be applied various techniques for the monitoring of a system as well as the processes controlled by the malware. *Leon et al.* formulate four categories in their paper which are the common techniques used for dynamic malware analysis which are hooking, emulation, hypervisor and bare-metal based methods [Leo+21]. We want to apply a commonly used dynamic analysis technique to the malware data set we analyzed in Section 4 to compare the coverage and briefly discuss the results. Therefore, we need to find all functions in the potential targets and be able to detect if they are reached during runtime. This can be resolved by using a disassembly engine

that reveals the target functions. But since Malpedia contains not only executable binary files but also DLLs we need to find a way to initialize the execution of both binary file types. In addition we also try to evaluate a large bulk of malware and need therefore a certain level of automation. And lastly, we need a virtual environment that allows us the safe execution of a large and heterogeneous malware pool. Therefore we define the following criteria that our toolchain has to meet for our approach and to meet the non-technical requirements beyond this.

Since we used memory dumps taken during the execution of the samples our static analysis approach in Section 4 can potentially cover wide areas of the target samples. Additionally one can assume that most of the unpacking logic is not present in the memory dumps. We want to analyze if the coverage of malware capabilities by using dynamic analysis approaches differs to our automated static analysis approach and if there is a significant difference in the amount of coverage provided by the approach.

1. Choice of a method for the supervision of execution
2. The dynamic analysis approach needs to allow fast prototyping
3. For the processing of the samples we need a recursive disassembler
4. We need a malware orchestration tool or a scriptable hypervisor
5. For our evaluation we need a curated, open data set to prevent strongly biased data

First, we need to select a dynamic analysis approach that meets our criteria. *Leon et al.* [Leo+21] describe four popular approaches that can be applied for supervision of code execution for dynamic malware analysis:

Hooking describes a technique where tools overwrite certain functions or API calls to monitor or change the original behavior [Leo+21]. In the context of dynamic malware analysis hooking can be used to monitor the malware and detect its capabilities. The approach is very efficient as it only introduces a few assembly instructions per hook and typically a memory module in the target process. Although the procedure is very fast it is easily detectable by malware since the control flow and the original code section is altered. Additionally there is always the risk of missing important API calls due to missing hooks and malware may interact directly with low level APIs or system calls without using regular API functions.

Emulation is another technique for the detection of malware behavior during runtime [Leo+21]. The technique is based on system call tracing to detect interaction with the operating system and the resulting behavior. A typical weakness of this approach is the performance; system call tracing tends to be very expensive and slows down the analysis system. Additionally, system calls are very low level

and a high level API call can involve multiple of them. The large amount of data that is generated during analysis might obscure the most important system call sequences.

Hypervisor-based approaches are considered as a state-of-the-art technique for dynamic malware analysis. They require a hypervisor that is either written from scratch such as the approach by *Leon et al.* [Leo+21] or a off-the-shelf solution like VirtualBox [Ora21] or XEN [Xen21]. They are more complex than hooking based approaches but generally faster than emulation techniques.

Bare-metal solutions are not feasible in our context as they are extremely complex and essentially unmaintainable. Virtualization introduces the possibility of fast and easy creation of snapshots, virtual networks and disks and do not require additional real hardware except for running the hypervisor. Specialized solutions like BareBox by *Kirat et al.* [KVK11] claim to overcome this issue by allowing fast reconstruction of the analysis system but introduce other problems, e.g. an analysis component must be installed on the guest OS [Leo+21].

In our work we use a hooking approach as it is the ideal solution to create a fast prototype although the approach is a trade-of between implementation time and best possible extension. Due to the restricted time frame in the context of this thesis we need a quick prototype and cannot aim for greatest possible completeness of all malware techniques. Hooking allows us to overwrite potential functions or addresses of interest to detect malware capabilities and to measure the coverage of the execution during runtime. The requirement is not to achieve maximum coverage but rather to inspect the data set to thereby enable a basic estimation of coverage to be made. These basic principles are intended to provide an initial data-based, well-founded statement on the static and dynamic approach.

Besides the hooking approach we also need to address a malware analysis scenario that allows the batch processing of large amounts of samples. Additionally, as we need to determine which functions shall be hooked, we need a recursive disassembler that does not rely on meta information such as file headers or debugging symbols as they are usually not available in malware samples. To keep the malware inside a foreclosed, isolated environment we use a hypervisor. Inside the virtualization layer we run an agent that allows us to hook and trace all functions we want to address. To process a large amount of samples automatically, we script the hypervisor using its API. As a data set we process we use a curated malware repository to prevent strong biases and to ensure that the malware landscape is widely covered by the data. We use Malpedia as the data is representative and we can compare our results against the static analysis approach we discussed in Section 4.

5.3 Implementation Details

In this section we present the details of our implementation. A relatively complete dynamic analysis solution should be able to cover most of the malicious behavior of a malware sample. The ideal solution should cover all malicious activity after determining the correct point in time when the unpacking procedures have finished. In addition, all actions over all treads and processes should be covered, even those that go beyond their own process. Self-modifying code, what is usually a complex challenge for analysis tools, needs also to be analyzed for most thorough precision and coverage. Since these are very large and complex challenges we can implement these goals in a real world scenario only to a limited extend. Typically, different platforms and approaches introduce trade-offs when analyzing large malware sets as all have different weaknesses and none of the approaches captures all data perfectly. To implement our dynamic analysis tool we first need to choose a hooking framework:

Microsoft Detours is a project for monitoring API calls on Windows [Mic]. It allows developers and reverse engineers to inspect API and function calls during runtime. The framework offers low-level interaction with processes via a C++ interface.

Frida is a memory injection and binary instrumentation framework [Fri]. It offers a scriptable interface in multiple languages such as Python, JavaScript and many others. Users can access functions, instructions and addresses of a target process using JavaScript. Frida injects a shared object into the target process that is able to execute predefined actions by the user, e.g. tracing, hooking or altering functions. The design of Frida allows users to interact with low-level components of closed-source binaries on a high level using modern and flexible scripting languages like JavaScript.

Intel PIN is a binary instrumentation framework for a broad set of different platforms [Lev]. The software offers a C/C++ API and has two different operation modes for binary instrumentation [DBC13]. One of the two modes offers a framework for the automated recreation of selected binaries called JIT Mode. The original binary file is not executed but a copy is created, containing equal instructions and the same behavior. Probe Mode is a classic hooking approach comparable to Detours or Frida. The target instructions are patched and the original code is relocated. After the hooking logic is executed the original code is processed and the hook sequence returns to the rest of the original instructions and continues.

To tackle a complex problem, we need a flexible tool that allows us to adapt to the challenges during implementation. Frida offers the required flexibility closest to the mentioned. Therefore, we use Frida as hooking framework for the implementation of our prototype. Frida allows dynamic code instrumentation and offers a large framework for API tracing and function interception. Its many

features allows rapid changes without large modifications of the remaining toolchain. On paper, this makes Frida the most suitable option for implementation.

To find the corresponding functions we need a recursive disassembler that is able to analyze and process binary files without intact meta data or debugging information. SMDA is a recursive disassembler that is able to operate on unpacked and dumped executable malware as well as memory dumps. Therefore, the disassembler is very flexible and through its specialization in malware analysis particularly suitable. Additionally, we use SMDA for our implementation to enable compatibility to the results in our static analysis approach in Section 4, which enables the comparability of the results.

To isolate the malware during execution and to maintain fast recoverable setups we need a hypervisor for a virtual malware analysis environment. VirtualBox is an easy-to-use, scriptable hypervisor that allows the creation of snapshots, virtual networking devices. As a reliable and well-established hypervisor it is a good suitable candidate for our implementation.

To fulfill the criteria we defined in Section 5.2 we build a prototype that allows us to measure the coverage of malware capabilities during execution.

Cuckoo Sandbox is an open-source malware analysis system [Cuc]. It offers automatic analysis of suspicious files for the operating systems Windows, Linux, macOS and Android. Cuckoo is not limited to binary files and also covers the analysis of office documents, malicious websites, emails and many more. The sandbox is able to dump network traffic, even decrypted TLS traffic. Cuckoo can be configured and extended with plugins offered by the community. As we are basically interested in code analysis only we do not use Cuckoo but adopt the concept as we script the hypervisor VirtualBox to perform similar in a sandbox.

We construct our toolchain on top of the tools Frida and VirtualBox. VirtualBox allows us to create a sandbox in an isolated environment with additional virtual networks and support for snapshots. We use Frida to create an agent that runs inside the guest OS allowing us to trace API calls and functions of interest. For the orchestration of multiple virtual machines and their communication with the host we use custom scripts. Additionally we hardened the VirtualBox machine we used for our analysis approach by following best practices to prevent detection by malware [Dan17].

5.3.1 Introduction to Frida

The binary instrumentation framework Frida allows hooking and instrumentation of target binaries. It supports high level programming languages like Python and JavaScript which allows us fast prototyping.

Therefore it is a good candidate for our hooking toolchain. We build our implementation by using the Python as well as the JavaScript bindings. Frida is used in version `15.0.18` and Frida tools in `10.2.1`.

Frida is a large framework, its core is implemented in C and Vala [The21] to support multiple architectures and operating systems. Vala generates portable C code that can be compiled to different platforms. Frida is typically used for the reverse engineering and binary instrumentation of mobile platforms. But since Frida is a multi-platform binary instrumentation framework it suppoorts multiple architectures such as ARM and x86 as well as various platforms. Frida supports Linux, macOS and Windows but also the most popular mobile platforms Android and iOS. A binary instrumentation framework typically requires a very low level implementation that cannot be platform-independent by design. Frida tries to overcome this issue with an implementation that is very abstract on the one hand and very low level on the other hand. Vala templates for most interfaces define an API that can be used by the low level implementations, which are mostly written in C. This separation makes the framework very modular. We especially use the following components from Frida which are all available in one github account for the project [Fri21a]:

1. frida-core: frida main component
2. frida-agent: Shared library injected into the target process, using gumjs and frida-gum, written in Vala and C
3. frida-gum: the core utility for native hooking, targeting the different architectures and platforms, written in C and Assembly
4. frida-python: python bindings for the usage of Frida in python scripts
5. frida-server: server software deployed to the guest for the communication with our host system

Frida-core is the core utility of Frida that offers interfaces for various bindings. A target process that shall be instrumented needs the frida-agent library. Frida offers multiple options to load the library into the target process depending on the use case. The concrete implementation of the binary instrumentation can be found in *frida-gum*. For each architecture and platform supported by Frida are low level implementations in C available.

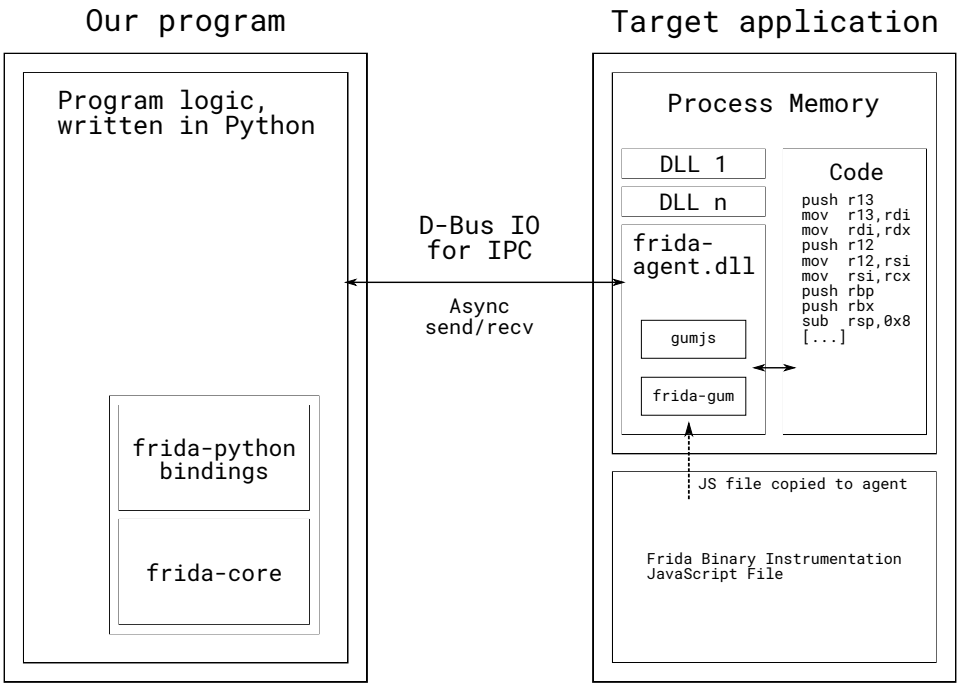

Figure 10: *Architecture of Frida, based on the diagram from the documentation [Fri21d] but with focus on Windows.*

A brief overview of the Frida framework and its design is illustrated in Figure 10. The diagram is focused on Windows as we see DLLs and the exemplary process design with code and shared objects during runtime inside a target process that shall be instrumented with Frida. We mainly use the python bindings for Frida which are implemented in the component called *frida-python*. Binary instrumentation of Frida is implemented in JavaScript. This JavaScript is loaded via the one of the many bindings that are available into the frida-agent shared object during runtime. The JavaScript file is executed in the context of the target process and allows binary instrumentation. This is possible as the frida-agent object contains a JavaScript engine. The communication between the Frida code in the target application and the tool users create with the Frida bindings is handled via asynchronous, json-based messages send over D-Bus [fre21], a middleware for interprocess communication.

Additionally, as a utility software, the authors of Frida created *Frida-server*. Frida-server is a program that can act as a proxy between the program created by the user that uses the frida bindings and the target application that is being instrumented. Moreover, frida-server allows the communication with the binary instrumentation scripts over network as the server. Frida bindings can connect to frida-server which enables the users of Frida to separate the program for binary instrumentation and the control logic even more.

There are two main components of Frida that are most relevent for our work. Frida offers two main modules for inspection of binaries during runtime. The first one of the Instrumentation implementations is called Stalker, which is allows code tracing inside processes [Fri21e]. Stalker allows the trace of instructions, basic blocks or functions ans is therefore a good candidate for our approach. Unfortunately the implementation is very resource exhausting and in our tests it slows down the target process too much. This seems to be caused by the implementation through call back functions which introduce more overhead than a simple hooking approach [Fri21b]. The other approach, *Interceptor* [Fri21c], allows the user to add hooks on addresses in executable sections that can trigger any desired actions. We use the interceptor to write hooks to each function to monitor if a function is executed during runtime or not. We see the functionality of Frida's Interceptor in Figure 11.

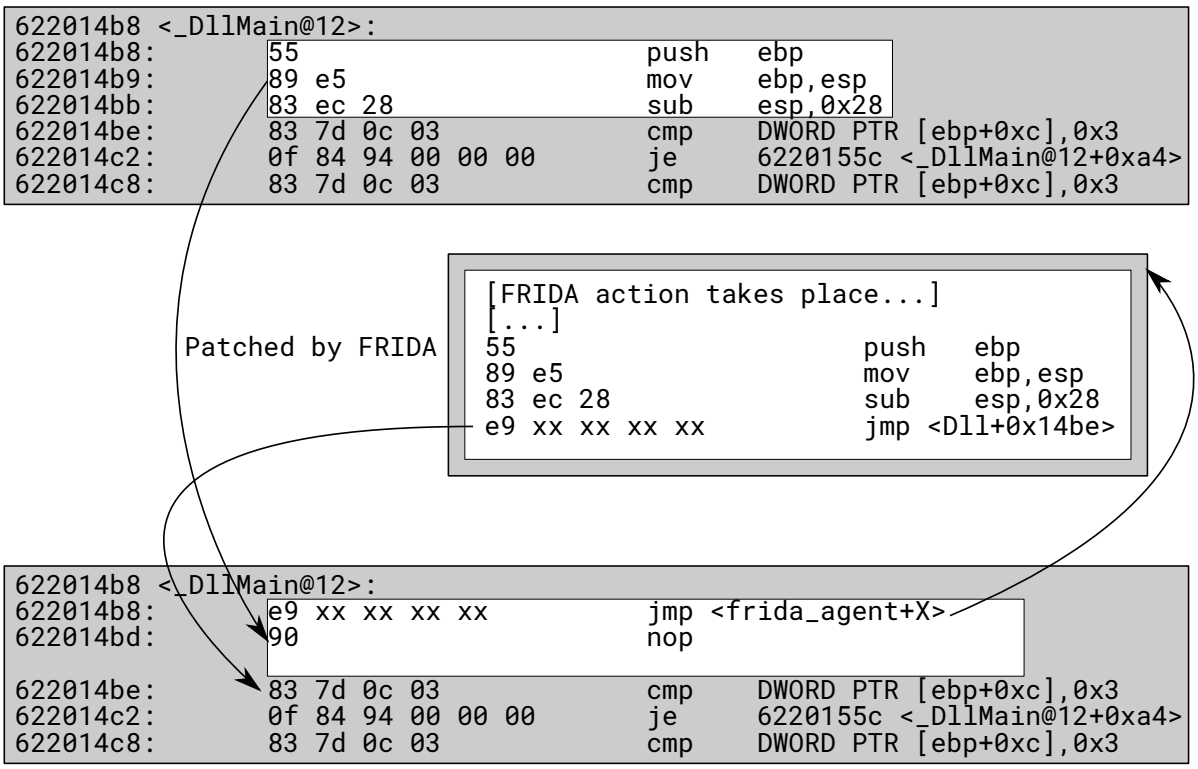

Figure 11: *Design of Frida's Interceptor, its hooking engine.*

The interceptor allows us to easily count the number of calls of a function of interest. In our case we simply hook all functions to estimate the coverage of the program. The identification on the other hand is complex. Since we use Capa in Section 4 which uses the recursive disassembler SMDA we keep using this disassembly engine. But as SMDA is focused on static analysis, it searches very greedy for potential functions. If one function is not correctly discovered it can result in a hook on an address that

is not the start of a function or even worse, not even the start of an instruction. In the latter case, a hook would be placed wrongly across valid instruction borders, which may lead to unrecoverable errors leading to a crash during execution. Therefore we configure SMDA to only use a very conservative function entry point discovery criterium, requiring a well-known function prologue and more than one incoming code reference.

Additionally, we can only hook functions that are at least five bytes large as FRIDA overwrites five bytes at the target address for its hooking implementation. An example for the concrete hooking implementation when using FRIDA is illustrated in Figure 11. First, the original instructions at `622014b8` are overwritten with a 5 byte jump instruction to the hooking handler inside the `frida-agent.dll` which is loaded by Frida into the target process. When the logic is done and the instructions supplied by the user of Frida are finished, the original code that was at the address before the jump was written is being executed. As a side node, Frida detects automatically if the instructions we overwrite are not exactly five bytes large and fixes the instructions by adding nops. This is important to keep the instruction schema at this position intact.

5.3.2 Malware execution orchestration

Additionally to the hooking of target functions we also need a setup for the bulked execution of large malware data sets in an isolated environment. We use VirtualBox [Ora21] as a hypervisor for our virtual instances and script it using its API.

Figure 12 illustrates a high level view of our implementation. To create a system for bulk malware analysis we use a set of three daemons that are running during the entire analysis session. The hypervisor (VirtualBox), a webserver and our python script *pandemic*. *Pandemic* shall enable the virtual machines to launch malware samples and communicate with the guest. The webserver is necessary as we need to supply the malware samples for each run to the guest and it is not possible to use the VirtualBox guest additions as we want to keep the virtual machine as stealthy as possible towards the malware that is being run inside. How a Windows 7 machine is installed with minimal virtualization detection from the guest OS can be found at this blog post [Dan17]. The virtual machine we use is not connected to the internet and has only one network adapter for communication with the host system.

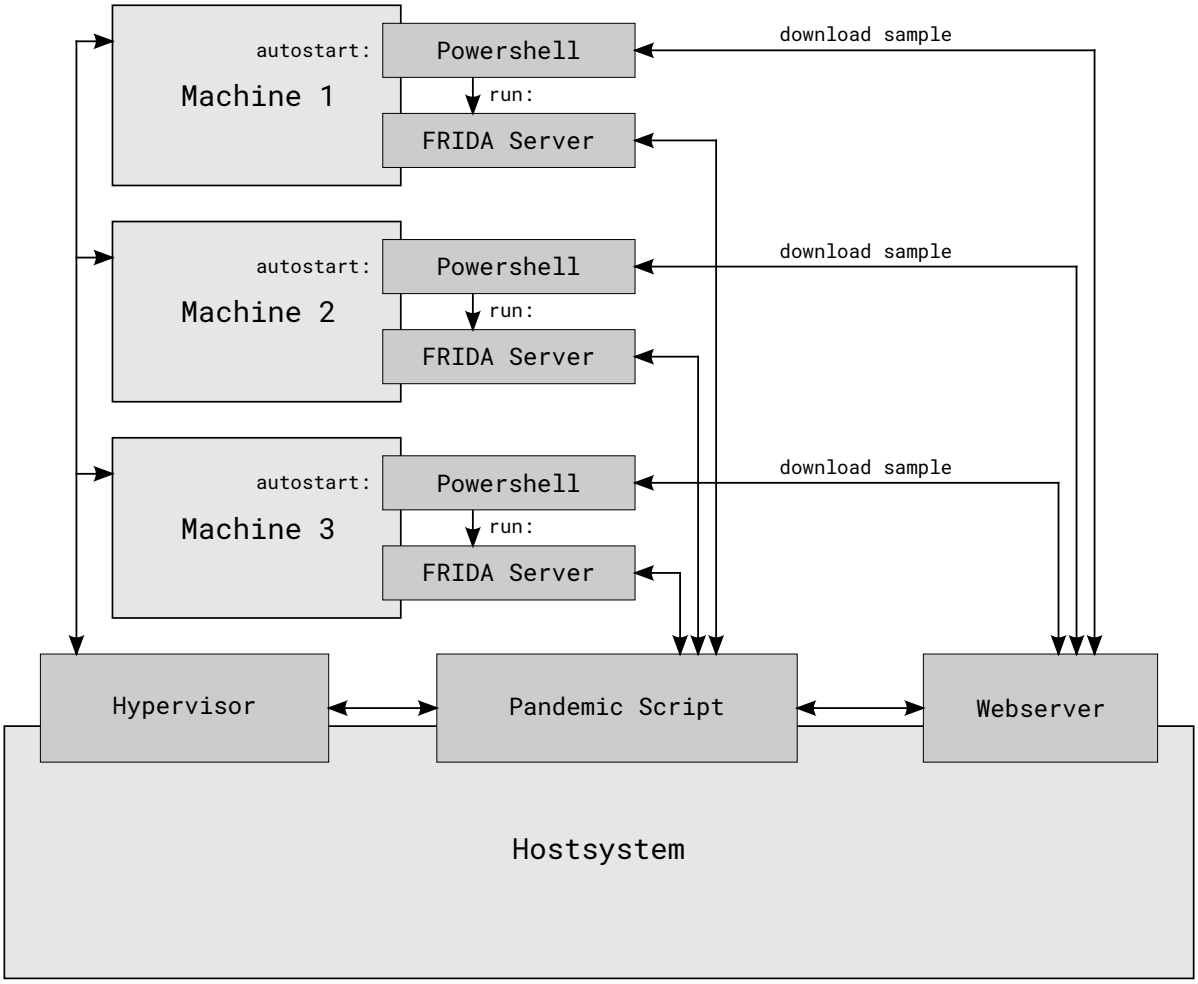

Figure 12: *The setup of our scripts and virtual machines.*

The virtual machine is modified so that it automatically runs a powershell script during start up. Inside this powershell script we make the virtual machine connect itself to our webserver and download a malware sample. Then it starts Frida-Server and terminates. Frida-Server is listening on the network interface of the virtual machine afterwards. *Pandemic* connects to Frida-Server and launches the execution of the sample we previously downloaded. This allows monitoring of the sample from the beginning without missing any instructions. If the program terminated or we run into a timeout, *Pandemic* stops the virtual machine and resets it to a snapshot before the execution of any malware. We let took a timeout of 120 seconds as *Kuchler et al.* suggest in their publication since they come to the conclusion, that most actions of a malware takes place in the first two minutes [Kuc+21]. For every sample a report is generated by *Pandemic* which can be processed by our toolchain in further

processing steps. This procedure is repeated until the batch has been fully processed. In our setup we used 3 threads where pandemic controls three different copies of the same image of a guest VM.

5.3.3 Code execution in the guest

The code execution in the guest is handled by the Frida-Server. But since Malpedia does not only contain executables but also DLLs we try to run the DLLs by following another approach. *Plohmann et al.* found out that there is a significant amount (26,9 % of the families) of malware available and distributed in the form of DLLs [Plo+17]. Windows has a builtin software that allows the execution of any exported function of a dynamically loadable library (DLL). This program is called `rundll.exe` [Mic21a].

This program basically maps the dll into its current process space and tries to execute the selected function which was passed as argument by the caller. Since the DllMain, an entry point of every DLL, is executed automatically when the DLL is mapped into a process we do not necessarily need to pass a name to `rundll32.exe`. Instead we call the mechanism with a dummy function name and just let the program load the DLL which begin the execution of the sample. Although we do not reach DLLs where the logic is inside another function than the DllMain, we still are able to execute the majority of them.

To add the hooking functions to the DLL we need to find the point in time where the DLL is mapped into the memory of `rundll32.exe` but before the DllMain is executed. We hook the function `ntdll!LdrLoadDll` which is part of the `ntdll.dll`. This function is called every time a DLL is loaded into the memory but before the DllMain is executed. Then we can check if the DLL we want to apply our hooks to is already available in the memory. By using this approach it is possible to execute the code of DLLs although we do not execute all exported functions that might be triggered by third-party modules, programs or events.

5.3.4 Limitations of our Implementation

Although we try to achieve comprehensive coverage of the samples in our approach there are several shortcomings and limitations. First, we run into the classic limitations of dynamic malware analysis which are not directly related to the tools we use. Since the virtual machine we use is configured without network and internet access it is possible that we do not see some behaviors that are related to such a setup. Additionally we run the malware samples without any arguments. If they require some, e.g. because they are loaded by other malware or loader stubs, we do not know that for every sample as they can differ.

Then there are limitations which are directly related to our implementation. Frida offers some basic functionality to catch new threads and processes on some platforms. This method is called child gating and spawn gating. While child gating [Fri18] is able to capture new threads, spawn gating shall address new processes. Child gating is documented and seem to work in our implementation. Spawn gating on the other hand is not documented for Windows but we can inspect the source code [Fri20]. Frida seems to detect only new processes if they called `CreateProcessInternalW` from `kernelbase.dll` or `kernel32.dll`, an undocumented, internal Windows function that is not exposed as importable API function. This limits our approach to stay in the initially launched process by Frida-Server, e.g. memory injections into other processes are not detectable [1].

Due to this limitations we only execute already unpacked malware samples as they are also available in Malpedia and are a sufficiently large group to derive information from it. The precise monitoring of packed malware is disproportionately more complex and requires more extensive tools.

On top of the problems of our implementation we ran into difficulties when it comes to dynamic hooking during runtime. The creation of new hooks during runtime led to very unstable and unpredictable behavior. There are several issues when it comes to dynamic hooking and changes to the code. The JavaScript code snippets which are loaded into `frida-agent` do not support hot reloading such that modifications of the source can be applied easily to the running process. Additionally there is no interface available to pause a running process or thread. The communication with the running JavaScript code in the target process is another problem. Since the communication is asynchronous we ran into several race conditions where e.g. the target process was already dead and we could not monitor any actions.

Since we were unable to implement complex hooks to deal with unpacking and self-modifying code we decided to keep the analysis evaluation reduced to the unpacked malware samples of Malpedia. There are sufficient malware samples and families available in Malpedia to obtain a statistically relevant result.

5.4 Evaluation and Results

In this section we present the results of our approach and the associated evaluation. In Section 5.4.1 we present the data set we used. We provide further information about the data and discuss the most

[1] The definition of **all** necessary hooks in Windows API functions is not trivial as many of them are not documented. Malware and their packers can bypass them or rebuild the application code of many API functions making an import superfluous. ReactOS [Rea21b], an operating system based on the Windows subsystem aiming to be binary compatible, has a reconstruction of the respective process implementation [Rea21a] in Windows although this is far out of scope of this work and only relevant for the interested reader.

important key data. In Section 5.4.2 we present the evaluation approach and what aspects we want to inspect in detail in our interpretation. The methodology is followed by Section 5.4.3 containing all results and statistics we build upon our approach. We discuss our results in Section 5.4.4.

5.4.1 Data set

To evaluate the resulting software against the data we use a defined data set to create reproducible results. In this section we give an overview about the data set we use for the evaluation. For our coverage evaluation we use Malpedia which has been described comprehensively in Section 4.4.1. It is also necessary to use Malpedia as we want to compare our results to our static capability detection we discussed in Section 4.

Malpedia offers basically two types of executable files: packed and unpacked malware samples. Due to our implementation we are not able to detect the exact point of time where the malware samples are unpacked in memory. Therefore we execute and evaluate only unpacked malware samples from Malpedia. Additionally we connect the memory dumps to the unpacked samples if they are available for the respective sample. Table 11 shows that the functions in both versions for identical samples are very similar. This enables us to compare the results from our dynamic analysis against the data from our static analysis approach.

5.4.2 Methodology

We use various tools and technologies to evaluate our data. To provide a reproducible approach to the reader we document every property of our setup. Additionally, we provide all data and the respective tools we used for the generation in the appendix of this work. In the context of the evaluation we compare the *coverage* of the samples. By coverage, we mean the amount of traced functions using the hooked approach as only functions that are executed during runtime could be revealed to any sandbox solution, which could detect the capabilities of the inspected malware sample. On the other hand, when we use the phrase *coverage* in the context of static analysis, the function or capability needs to be detected by our approach discussed in Section 4, which is essentially given from the discovery of the function by SMDA and assignment to a capability by Capa.

Due to our limitations in our approach we focus on unpacked malware only. For our evaluation we begin with an exploratory data description of the data we processed. We sanitize the input data such as addresses we want to hook by applying a disassembly confidence filter to improve the function detection preventing too many crashes of the samples during execution. Then we execute the unpacked malware samples using our processing pipeline and asses how much of the hooked

functions are actually reached during execution. We handle DLLs and executable files differently in our implementation and due to the nature of their differences. Therefore we do separate them in some statistics.

Additionally to the data we generate by inspecting how much of the malware is executed in our sandbox environment we compare the coverage of our dynamic analysis approach with the data we generated in Section 4. We search for samples that are available as executable, unpacked files and as memory dumps in Malpedia. Then we compare their addresses and see how many capabilities probably have been executed. Although there is a certain mismatch in the data we can recognize a trend.

5.4.3 Results

In this section we evaluate our approach and discuss the results. First, we evaluate the data set we analyze by exploring the characteristics of the corpus.

Figure 13 shows the distribution of the data we analyze. We analyzed 2181 unpacked files targeting the Microsoft Windows platform from Malpedia. Of these unpacked files were 1319 executable binaries and 720 DLLs for Windows which are 2039 potentially executable binary files we try to analyze for our coverage analysis. The missing 142 files do not have an intact PE header which is a minimal criterion for being able to execute the files. The analyzed files were executable files (`.exe`) or dynamically loadable libraries (`.dll`).

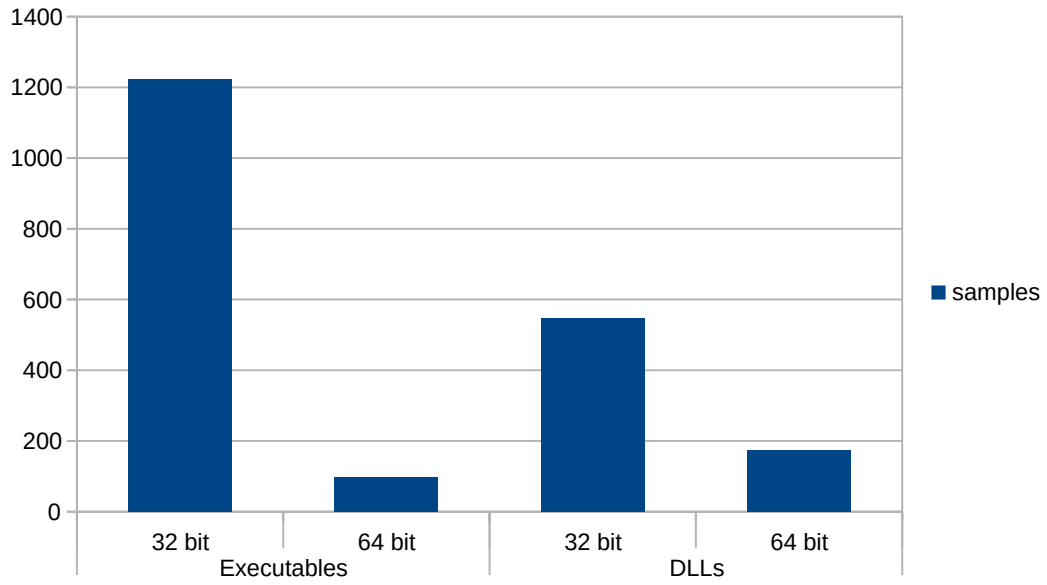

Figure 13: *Exploration of the analyzed data, comparing the amount of executable binaries and DLLs.*

There are 1319 unpacked, executable binary files and 720 dynamically loadable libraries that have been unpacked. Of those executable binaries, 1222 target are built for 32-bit systems and 97 are created for 64-bit systems. 93 % of all executable binary files are built as 32-bit executables compared to 7 % 64-bit binaries. The picture is somewhat different for malware that is distributed as DLL. Of overall 768 unpacked DLLs are 547 or 76 % built for 32-bit and 173 or 24 % for 64-bit systems.

Potential dynamic analysis coverage of the samples To obtain a more precise impression of our dynamic analysis implementation and to a certain extent generally, we examine how many functions that we monitored were reached during execution. By counting the functions in general, determining how many of them can be hooked and how many of them are reached during execution we try to obtain more information about the potential coverage of samples during dynamic analysis approaches. For our setup we use the approach shown in Section 5.3.2. We execute and monitor a batch of malware samples and record the executed functions. All samples are either finished within two minutes or their virtual machine is reset.

Table 9 shows the coverage data for executable binary files and DLLs. All columns in this table are distributions we described in five-number summaries. We represent in this table the order of magnitude of the number of functions, their frequency and distribution. In addition, we evaluate how many functions we discovered are interesting targets for our analysis approach. We monitor these

functions using our implementation and batch process all 2039 unpacked samples. The table header is split into two different parts. First, we show the amount of functions which are generally available. Then we reduce them to those we want to hook. And then we show the coverage for all samples C_a, for executable binaries C_{EXE} and for DLLs C_{DLL}.

Table 9: *Five-number summary with additional mean and standard derivation showing the reduction of potentially analyzed functions of the unpacked malware samples in Malpedia and the associated coverage of functions.*

	F_a	$F_{>5}$	F_i	$F_{c=1}$	C_a (2039)	C_{EXE} (1319)	C_{DLL} (720)
mean	924,17	909,01	605,91	289,94	15,74 %	18.98 %	9,79 %
std	1825,26	1796,29	1042,48	498,40	0,21	0,23	0,15
min	0	0	0	0	0 %	0 %	0 %
25%	108	107	83,5	34	0 %	0,64 %	0 %
50%	396	389	313	141	8,12 %	11,20 %	4,34 %
75%	808	798	600,5	294	21,43 %	30,09 %	14,56 %
max	19408	19372	11784	4862	100 %	100 %	100 %

When evaluating the coverage it is important to obtain the correct functions we want to monitor. We use the recursive disassembler SMDA to determine functions in the unpacked samples. Due to the design of SMDA and its focus on malware dumps it searches very greedy for functions. In addition, our approach requires that a function that can be hooked is at least five bytes large. During our test runs, we discovered that the functions SMDA discovers are error prone during runtime. While the disassembler detects a lot of functions which can be useful for static analysis this behavior leads to lateral effects if one actually hooks them for binary instrumentation. As one incorrect address can eb enough to potentially crash the program. Therefore we remove function addresses which do not fulfill our requirements.

The first data column in Table 9, F_a, shows the distribution of all functions that were detected by SMDA. There are some constraints due to the toolchain which we apply, consisting of Frida and SMDA. Frida's hooking functionality is implemented by patching the instructions at the target address and adding a jump instruction. Therefore we require that the functions we inspect during runtime are not smaller than five bytes and filter the rest out. Column $F_{>5}$ contains only functions larger than five. The amount of function does almost not change. Our tests show that the functions detected by SMDA still contain incorrect functions. Therefore we remove all functions which are at least called one time.

These data can be found in column F_i. While the samples with most functions lost almost 50 % of their functions we have a reduction in the third quartile and below of around 20 %.

Unfortunately we still need to reduce the amount of functions we aim to monitor due to crashes. SMDA has a score for each detected function which reflects the confidence. We enforce the maximum score to reduce the probability of runtime crashed due to wrong hooking addresses. Column $F_{c=1}$ contains only functions where SMDA's confidence score is equal to one. While we lose between 65 % and 75 % of all functions compared to F_a, this method allows us to execute the samples and monitor the coverage of samples by hooking the remaining functions. In principle, it can be assumed that the result will not be distorted too much, since the procedure is comparable to sampling.

The distribution of function coverage of all samples is shown in column C_a. All coverage columns are in relation to the actual hooked functions in $F_{c=1}$ as for the others we cannot detect if the functions were hit during execution. In the mean 16 % of the functions we hooked were hit. The median is approximately 8 %. Only from the third quartile onwards does the coverage increase more strongly to about 21,5 %. Between the maximum and the third quartile are the most covered samples. Therefore more than 25 % of the samples we analyzed executed over 20 % of the hooked functions inside our sandbox.

The data of the executables and DLLs are similar. We covered executables better, the distribution in column C_{EXE} shows that we cover even over 30 % of the functions in the top 25 % of the samples. The DLLs we analyzed are less covered. Column C_{DLL} shows that we only reach close to 14,5 % of the coverage in the higher 25 %. In general one sees that the data of all columns that contain the coverage metrics are clearly skewed, all three distributions are unevenly distributed. This shows that coverage is not uniform at any point in time, but that large parts of the executable functions are not reached at runtime.

In order to evaluate not only the number of executed functions but also the runtime in general, all error codes and crashes of the unpacked samples were evaluated. Figure 14 illustrates the different error types that were faced during execution. Most of the samples terminated without error. Almost 60 % of the samples could be finished properly and about 20 % of the samples were terminated after a timeout of two minutes. Both are considered as successful and complete execution in the context of our study. Even those that were terminated by a timeout of two minutes are considered sufficiently investigated, as the majority of malware actions tended to be reached after two minutes as *Küchler et al.* state [Kuc+21].

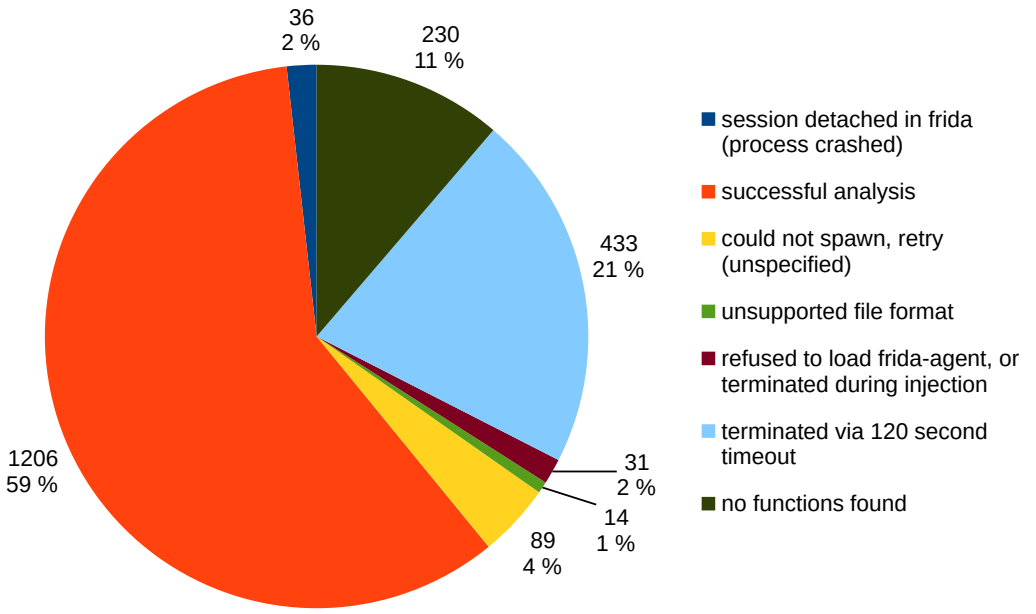

Figure 14: *The runtime data we analyzed to compare how much of the malware was covered and how much samples crashed during their execution time.*

Besides the two optimal states, there are also error handling states we address. The largest error affects 11 % of the samples and is not a runtime error, although it is a direct consequence of our approach. Since we remove all functions from the samples without SMDA confidence score equal to one we drastically reduce our search space. Therefore, we do analyze a smaller group of candidates.

The other errors are triggered during runtime, to be precise when executed by *Frida-Server*. There can be various causes for these types of errors which cannot always be clarified. Some samples could not be started respectively the processes could not be spawned (4 %). There is no clear indicator why the samples could not be executed. Other samples could not be executed with our toolchain as Frida was not able to inject the `frida-agent.dll`. Around 2 % of the samples could not be executed successfully due to this reason. 36 samples or 2 % of the data set could not be monitored as the process of the executed sample or the Frida session crashed. These three error categories are very heterogeneous and a distinction between them is not even always possible. We have found in our experiments that timing plays a role; some of the processes cannot be spawned correctly until Frida set the environment up as the samples crashed before. To nevertheless enable an analysis we launched a sample up to five times to face possible race conditions. All samples that still did not finish the start up in our analysis chain ended up in the category *could not spawn*. 14 samples could not be launched

due to the error *unsupported file format* although we performed a previous, superficial check of the file format by searching for the PE header. This error might also be indicated a crash in the early bootstrapping phase of the sample.

Although these error types are not very meaningful, one can still derive a trend from it. Sources of error cannot be identified easily in the context of dynamic analysis and the sources of error are also very diverse. The fact that we are unable to trace all samples, even though that the samples being launched directly through *Frida-Server*, emphasizes that our approach is incomplete. In addition, we also do not know what kind of actions or behaviors are reached by the samples during execution. Tracing functions without further information about them only shows the potentially reachable functions and principal coverage.

Evaluating the coverage between static and dynamic analysis As our implementation has several shortcomings that we described in Section 5.3.4, it is not possible to create a (reasonable) statistical evaluation for the packed malware samples in Malpedia. By choosing Frida, it was not possible to solve the challenges posed by packed malware. Since we cannot reliably determine the point of unpacking directly before the execution of the sample we test our implementation only against unpacked files.

Unpacked files in this context are malware samples of which one can assume that most or all packing techniques are removed from the sample. Additionally, unpacked files are executable and we expect that the malware sample behaves the same way as a packed one, except for the specific unpacking mechanisms during the begin of the execution. For a reasonable comparison between the coverage of called functions at runtime and the static analysis of memory dumps, a narrowing of the data is required.

To determine what functions are reached we analyze samples for which there are memory dumps and unpacked versions available in Malpedia. We take both versions of the same malware sample to obtain more information about them. Since we analyzed the capabilities of the memory dumps in Malpedia statically, we can compare the unpacked versions of the respective sample with them.

But due to the fact that the data sets are very different in nature, the data must first be filtered in order to ensure a certain comparability. This allows us to compare our static analysis approach against our dynamic analysis approach. We can abstract the results of this comparison to a certain extend and assess how both analytical techniques differ in terms of potential coverage and completeness. In order to relate the data, we only consider common address sets and additionally set the same restrictions for both compared samples on the functions.

First, we examine how many samples are in Malpedia for which both versions are available. There exist 968 samples where we can compare at least one unpacked version and one memory dump of the malware. Table 11 shows, that the functions in both versions are largely identical, since both are derived from the same malware sample. The median of the combination of unpacked and dumped samples is about 99 %.

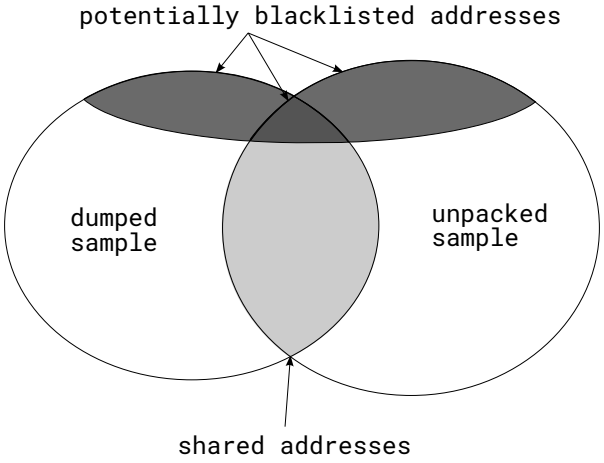

Figure 15: *Procedure for the function address intersection between dumped and unpacked samples. The illustration is chosen schematically, the median of the actual overlap is approximately 99 % as shown in Table 11.*

Figure 15 illustrates the filter procedures which are necessary for the following comparative evaluation. As the data sets are very different, we define some data set configurations which are used in the following evaluation. These configurations of data sets are defined for the comparison between the static approach using *Capa*, and the dynamic approach using *Frida*.

In general, the following applies to our evaluation: A function is counted as covered, if it is either analyzed via Capa and a Capa-Rule hit the respective function or the function has been hooked with Frida and been reached during runtime. If a function is hit by Capa, it is considered as that a malware capability was detected. When a function is hooked and it is reached during the execution of a sample in the context of dynamic exploration of the samples, the function is considered as covered because e.g. a sandbox could detect the function as it was called during runtime. When analyzing the resulting data, it is important to keep these different methodologies in mind.

In order to reasonably compare the coverage metrics of the different approaches we need to define several configurations of sets with which we relate the hits of the respective approach. The following

data sets have been filtered according to certain criteria and are explained in more detail after this enumeration.

1. $F_{s,a}$ - All functions
2. $F_{s,c=1}$ - All functions with confidence score $= 1$
3. $F_{s,a} \cap C_{BL}$ - All functions without Capa-Rule *contain loop*
4. $F_{s,c=1} \cap C_{BL}$ - All functions with confidence score $= 1$ and without Capa-Rule *contain loop*
5. $F_{s,c=1} \cap C_{Hits}$ - All functions with confidence score $= 1$, only Capa hits
6. $F_{s,c=1} \cap C_{Hits} \cap C_{BL}$ - All functions with confidence score $= 1$ and only Capa hits, but without Capa-Rule *contain loop*
7. $F_{c=1}$ - Only functions that target unpacked samples, derived from all samples for that a dumped version also exist.

Since our static approach has less limitations than the dynamic approach we need to adapt the number of addresses we analyze in order to maintain the ratios. In our dynamic approach we reduced the number of functions by enforcing a confidence score equals to one for all functions discovered by SMDA. Therefore, we need one data set for comparison that only contains functions with a confidence score of one. Additionally, we can construct a counterpart data set that includes all potential functions to emphasize the largest possible pool of function addresses for the respective samples to thereby reveal blind spots that have arisen due to the reduction of the data set by applying the confidence score.

Data sets of category $F_{s,*}$ contain only functions that appear in both sets, the set of functions from the unpacked and from the dumped sample version, the **shared** addresses remain in the data set. These data sets are used to compare the performance of both approaches on a common set of functions, containing functions both approaches have the option to either detect them or monitor them during execution. With one exception, which we will discuss later, this applies to all datasets.

Besides joining all addresses to a shared address space of the unpacked and dumped representations of the respective samples, we also distinguish between **all functions** and the functions with a confidence score of one that are available in the intersection of the functions from both representatives. The first filter, $F_{*,a}$, contains **all** functions; there is no confidence score constraint. In contrast to that, $F_{*,c=1}$ contains only functions where the SMDA confidence score is equal to one.

Additionally, there are specialized data sets for more concrete comparison. We reduce the target data set by intersecting the data with filtering data sets to sharpen our results for specific comparisons.

The filter data set C_{BL} contains only functions that are not only detected by the Capa-Rule *contain loop*. We know that this Capa-Rule skews statistical results and therefore we can eliminate these outliers by intersecting data with sets that do not contain them. The data sets $F_{s,a} \cap C_{BL}$ and $F_{s,c=1} \cap C_{BL}$ contain all functions that are not part of the function blacklist. That means, all functions but those that are only covered by the rule *contains loop* are part of this data.

The both data sets $Frida : F_{s,c=1} \cap C_{Hits}$ and $Frida : F_{s,c=1} \cap C_{Hits} \cap C_{BL}$ are special, because both datasets contain, in addition to the known functions, exclusively functions that are also covered by Capa. The former ignores the blacklist, the latter removes all functions that only contain the rule *contains loop*. Functions which are blacklisted can be considered separately this way. These data sets allow a detailed evaluation although the data nevertheless show differences. We can determine how many of the functions that contain malware capability features are actually reached during runtime.

The last data set, $Frida : F_{c=1}$ contains functions that must fulfill the criterium of a confidence score equal to one but it only contains unpacked samples which are nevertheless part of the intersection. More concrete, that means 651 or respective 317 samples are evaluated against all unpacked samples and their respective functions only, **without** intersecting them between dumps and unpacked versions. This value is considered because it shows the population of hits on all dynamically examined samples and their functions. Because the functions are not merged, the complete coverage we detected via hooking are mapped to the data.

Table 10 shows an evaluation of the static and dynamic approach. The column *data set* contains the combination of *function hits by approach* and the data set to which these hits refer to. We divide the coverage evaluation between the static and dynamic approach and additionally separate the results between executable binary files (`EXE`) and dynamically loadable libraries (`DLL`). All results are broken down as a five-number summary.

First, we discuss the evaluated data from our static analysis approach. For the evaluation of the coverage using Capa for static malware capability extraction we have two different data sets. On the one hand, we evaluated Capa for all functions that are available in unpacked and dumped samples ($F_{s,a}$). For both, DLLs and executable files, we see that the coverage is similar to the statistics that cover all memory dumps of the malware presented in Table 3. The same applies to the data set with a confidence score equal to one, $F_{s,c=1}$. The median of the function coverage is about 34 % to 38 % which is similar to a median of 33.17 % from Table 3. It can be seen, moreover, that the degree of coverage shrinks to the same extent when the blacklist is applied which excludes the outlying Capa-Rule.

If we compare this against the coverage of hooked functions during runtime, it can be seen that we reach significantly less coverage. Out of all functions from the shared addresses and a confidence

Table 10: *Five-number summary with additional mean and standard derivation showing the coverage of all samples in Malpedia that have a dumped and unpacked version.*

	data set	count	mean	min	25%	50%	75%	max
EXE	$Capa : F_{s,a}$	651	0,38	0 %	25,85 %	34,41 %	46,03 %	100 %
	$Capa : F_{s,c=1}$	651	0,40	0 %	29,50 %	35,94 %	49,19 %	100 %
	$Capa : F_{s,a} \cap C_{BL}$	651	0,14	0 %	3,90 %	8,57 %	19,89 %	100 %
	$Capa : F_{s,c=1} \cap C_{BL}$	651	0,13	0 %	4,05 %	7,65 %	18,18 %	100 %
	$Frida : F_{c=1}$	651	0,22	0 %	5,74 %	15,44 %	32,90 %	100 %
	$Frida : F_{s,a}$	651	0,07	0 %	0,88 %	4,40 %	10,02 %	100 %
	$Frida : F_{s,c=1}$	651	0,18	0 %	2,44 %	11,67 %	28,91 %	100 %
	$Frida : F_{s,c=1} \cap C_{Hits}$	651	0,20	0 %	2,98 %	13,04 %	30,70 %	100 %
	$Frida : F_{s,c=1} \cap C_{Hits} \cap C_{BL}$	651	0,19	0 %	0,00 %	9,72 %	30,73 %	100 %
DLL	$Capa : F_{s,a}$	317	0,38	0 %	26,53 %	35,67 %	47,55 %	100 %
	$Capa : F_{s,c=1}$	317	0,43	0 %	30,14 %	38,85 %	51,18 %	100 %
	$Capa : F_{s,a} \cap C_{BL}$	317	0,16	0 %	5,73 %	11,43 %	22,22 %	100 %
	$Capa : F_{s,c=1} \cap C_{BL}$	317	0,16	0 %	5,17 %	10,87 %	22,22 %	100 %
	$Frida : F_{c=1}$	317	0,12	0 %	1,35 %	5,28 %	15,41 %	100 %
	$Frida : F_{s,a}$	317	0,03	0 %	0,00 %	1,01 %	3,58 %	100 %
	$Frida : F_{s,c=1}$	317	0,09	0 %	0,00 %	2,65 %	10,92 %	100 %
	$Frida : F_{s,c=1} \cap C_{Hits}$	317	0,10	0 %	0,00 %	2,67 %	12,21 %	100 %
	$Frida : F_{s,c=1} \cap C_{Hits} \cap C_{BL}$	317	0,09	0 %	0,00 %	0,00 %	12,50 %	100 %

score equal to one ($Frida : F_{s,c=1}$), the data set in which all functions where potentially hooked, only about 18 % of the executables and 9 % of the DLLs where called and traced based on the mean. As the median is about two thirds for executables and about a third for DLLs we see that the distributions are strongly skewed and that only a few samples are widely covered.

The data set $Frida : F_{s,a}$ cannot be used to examine how much of the hooked functions were executed but it puts the totality of all functions in context with those reached at runtime. Accordingly, the data may indicate that our coverage tends to be incomplete. There is a large gap between the hooked functions and all functions with which we can confirm the data quantifying the factor of function reduction we evaluated in Table 9. Because of the constraints we needed to be able to execute the samples in the first place, we reduced the potential coverage ratio from a mean of 18 % and 9 % to only 3 % and 7 %.

The data set $Frida : F_{c=1}$ shows the coverage of the functions via hooking by Frida for the respective executable files or DLLs. It remains the same as in Table 9, only the unpacked samples are covered and not the shared functions from both variants of the sample, but we filter the samples so that we only analyze 651 executables and 317 DLLs. In addition, we we require a confidence score of all functions

equal to one since we did not hook functions without it. This distribution shows the cover without any dilution of the further function addresses from the memory dumps. Therefore, this evaluates only the actually tested samples and addresses. It is not a good distribution to be compared against any $F_{s,*}$ data set, but it shows how much was hit at maximum without reduction.

Additionally, we compare the amount of coverage between both by evaluating how many functions of the functions detected by Capa were traced by Frida during runtime. $Frida : F_{s,c=1} \cap C_{Hits}$ and respectively $Frida : F_{s,c=1} \cap C_{Hits}$ describe the intersection between both, the data of covered functions by Capa and hooked functions from Frida. The columns show a distribution of the entered, hooked addresses in relation to the functions that were hit via Capa taking the basic set of match-able functions into account. The basic set is defined by all potentially match-able functions. From our evaluation in Section 4.5.2 we know that the functions we take into account might be biased. In order to avoid a biased evaluation, in addition to the data mentioned above, an evaluation taking into account a blacklist was generated. These results can be found in column $Frida : F_{s,c=1} \cap C_{Hits} \cap C_{BL}$ for all executable binary files and in $Frida : F_{s,c=1} \cap C_{Hits} \cap C_{BL}$ for the DLLs.

The data in Table 10 shows that our static approach covered more functions and therefore more of the samples in general than our dynamic approach. Table 10 shows that our dynamic analysis approach was able to cover of the data which was hit by Capa about 20 % as mean for executables and approximately 10 % for DLLs. From our previous analysis in Table 9 we do know that the DLLs are less comprehensively hit. It is very likely that we need to analyze them more in detail to cover more functions, for example by executing every exported function and not only the `DllMain` function.

In addition to the comparison between the approaches and the amount of detected functions we also see that the coverage receptively the execution of the hooked functions of the shared data set of samples is comparable to the one containing only unpacked malware. This suggests a strong correlation between the processed dumped and unpacked samples and their functions which is backed by our evaluation of the dumped and unpacked samples in Table 7.1.

5.4.4 Discussion

In this section we present the results of our evaluation and discuss the data and the potential impact to the topic. Sandboxes are very popular among malware researchers but about the actual coverage of samples being executed in sandboxes is not much known. The data situation is very thin about the research status of current sandbox solutions.

We evaluated the general coverage of malware in a sandbox in a small test scenario in Table 9. In this dynamic malware analysis evaluation, the coverage is determined by the amount of functions that

were reached during the execution in our isolated environment in relation to the overall functions that we applied function hooks on. Thereby it is not absolutely evident how much of the reachable functions is lost due to the setup and how much is generally not executed during runtime.

Most of the samples we monitored with our dynamic analysis approach are not covered very well. Considering the median, the functions of executable binary files are covered by about 11,20 %. DLLs are even worse covered as the median of the coverage metrics is approximately 4,34 %. The lower half is even less covered which tends to reduce the significance of the study. Considering the 25 % of the best covered samples one can see that 30 % of the hooked functions were executed during our analysis if the sample is an executable binary and that we approximately executed 14,56 % of the functions of a monitored DLL (see Table 9, 3. Quartile).

The DLL files are even less covered as we only execute the `DllMain` only, a function that is called every time the respective DLL is loaded. Any other function that might be exported is not covered by our approach. Besides these limitations we also remove a lot of functions. It is conceivable that not all of the functions we have removed will cause a crash, as long as they are at least bigger than 5 bytes. Due to the frameworks and libraries we built our approach on, we are limited to the features and the precision they provide.

In order to successfully compare both approaches we have defined a data set consisting of samples from Malpedia, for which an unpacked and dumped version is available. We discovered that the unpacked samples in Malpedia and the memory dumps of the samples are very similar and contain largely the same functions. Therefore, we connect the data for every sample and compare between the coverage we were able to achieve. In conclusion, the coverage that Capa provides is superior compared to out dynamic analysis approach. Although the better coverage is strongly reduced when outliers are removed, from about 39 % to about 14 % in the mean. On the one hand, we have analyzed static and dynamic analysis approaches. This allows at least the comparison of the two results and represents an initial assessment of the exploration of data and procedures. On the other, it is not possible to judge the coverage by our approach, as we do not have any comparative data. Accordingly, the results can be generalized only to a limited extent.

For further assessments, one should use a more comprehensive strategy for observing functions at runtime. This may include also different runtime environments providing network access, simulating user interaction. Due to the setup we do not have any internet access, we use a hypervisor that could be potentially detected and we have no information what parameters or configurations some of the samples might need. Additionally, packers introduce further challenges. The unpacking stubs do

collide with the actual behaviors and capabilities of malware. Therefore, more advanced solution need to overcome this problem with more sophisticated dynamic malware analysis approaches.

5.5 Summary

In this section we summarize the content of this chapter which deals with the runtime coverage analysis. Section 5.1 begins with a short motivation of our approach and the outline of the current state of research with its potential shortcomings. We present our concept and the research objectives. In Section 5.2 we examine how potential solutions for a code supervision approach can be designed. Additionally, we discuss potential limitations of different code supervision approaches. In Section 5.3 we present our design and implementation of a malware orchestration framework. We discuss different frameworks for hooking and binary instrumentation. In Section 5.4 we provide an evaluation of our implementation and a statistical evaluation of the results. Additionally, we compare the results against our static analysis approach.

6 Summary

In this Section we summarize our work and provide an outlook on potential future work. Section 6.1 provides a conclusion to the reader. We briefly discuss the state of research and the contributions we made in static and dynamic analysis. In Section 6.2 we discuss potential shortcomings of our approach and what possible fields for future work are.

6.1 Conclusion

In this section we provide a summary of this work. We describe the topic and the belonging thematic framework in Section 1. In Section 2 we provide an introduction to the terms and foundations of our topic. We present basics from the field of malicious software, malware analysis and its sub-types and some information of low level approaches like hooking and binary instrumentation. Additionally, we give an outline about packers and malware taxonomies. We end the section by providing basic information about code analysis and its sub-disciplines. In Section 3 we present the current state of research. We give an overview about related work that covers packed malware and the corresponding challenges. As the packing of malware complicates further analysis steps we provide an overview about publications what introduced new techniques to overcome packed malware.

In addition to the related work, we also discuss data sets which can be used for statistical representative evaluations of malware analysis frameworks and tools. In Section 4 we introduce out approach for malware capability extraction by using static malware analysis approaches. We introduce a short motivation to the reader, summarize shortcomings of current approaches and formulate our research objectives. We extend Capa to enable the tool to operate on memory dumps. Then we evaluate the analyzed malware samples statistically and map the results to the taxonomy framework MITRE ATT&CK and Malware Behavior Catalog. In Section 5 we discuss our approach for runtime coverage analysis. We introduce a short introduction to the field and illustrate the fact that malware researchers tend to use sandboxes although there are few publications on the subject and basically no information about the coverage of samples in general. Additionally we present our approach of a malware execution orchestration framework for the automated instrumentation and evaluation of

large malware sample data sets. We present our results and potential shortcomings. In addition we compare our static and dynamic approach to be able to give a rough estimate about the coverage metrics of both approaches and which one tends to cover more.

In this work we discuss two approaches. We develop a framework for static malware analysis using a novel technique for malware capability extraction by using the tool Capa. Additionally, we implement a dynamic malware analysis toolchain that allows us to instrument large bulks of malware samples. We compare the advantages and disadvantages of both approaches and compare their results. We show that static analysis on memory dumps is able to detect large amounts of capabilities. Additionally, we see that the coverage can be skewed depending on the rules that define certain capabilities. As a results of the comparison of our static and dynamic approach we see that the potential coverage of static analysis on memory dumps tends to be larger. Although we see that automated capability analysis via static malware analysis can be successfully applied to malware it is not yet readily clear whether the approach is superior to dynamic analysis approaches in general.

6.2 Future work

In this section we discuss potential fields of research for further improvements of our approach and possible further research areas. Malware unpacking is a complex problem to malware researchers. For the inspection of packed malware, one can either use memory dumps of the process after the unpacking logic is finished or it is possible to analyze the behavior dynamically, e.g. by using a sandbox. In our approach we used a hooking technique. However, *Leon et al.* illustrate that API hooking has weaknesses in general and hypervisor might be the more powerful solution when it comes to completeness of behavior monitoring [Leo+21]. While hooking is invasive and needs to modify existing code, modified hypervisors are able to overcome this limitation as they are able to monitor every process from the outside of the guest OS.

This also solves the problems we ran into when analyzing malware with Frida; as the hooking engine needs to overwrite an API, one need to know the API in advance. Additionally, self-modifying code can result in unpredictable behavior if the wrong instructions of the process are modified. Another challenge is the detection of the correct unpacking status as it is not trivial to see if the malware is finally unpacked or not.

Cheng et al. provide another way out of the dilemma: The authors state that their novel approach allows reliable unpacking of malware samples into executable binary files [Che+21]. We have already proven that our approach works on unpacked malware binary. Therefore it would be trivial to adapt

their method in our toolchain as we would be able to process packed binary files in order to execute them after unpacking. The data set would not be reduced either, and another shortcoming, the separation of unpacking code and malware code, would be solved.

All in all, we need more data about malware; we need to find out how many code is typically reached, how do malware analysis environments affect the execution and how sophisticated are malware authors really when it comes to sandbox detection. At present there is no sufficient information about the quality of sandbox reports since we do not know how much of the code is actually reached inside the sandbox.

References

[Alv21] Alvarez, Victor Manuel: *YARA - The pattern matching swiss knife for malware researchers.* Last Accessed: 2021-10-22. 2021. URL: https://virustotal.github.io/yara/.

[Arn17] Arntz, Pieter. Last Accessed: 2021-08-01. 2017. URL: https://blog.malwarebytes.com/cybercrime/malware/2017/03/explained-packer-crypter-and-protector/.

[Bak20] Baker, Kurt: *Malware Analysis.* Last Accessed:2021-07-06. 2020. URL: https://www.crowdstrike.com/cybersecurity-101/malware/malware-analysis/.

[Bar10] Barwise, Mike: *What is an internet worm?* Last Accessed: 2021-07-06. 2010. URL: http://www.bbc.co.uk/webwise/guides/internet-worms.

[Bec20] Beck, Desiree: "Standardized reporting with the malware behavior catalog". In: (2020). URL: https://vb2020.vblocalhost.com/uploads/VB2020-Beck.pdf (visited on 08/08/2021).

[BGM12] Beaucamps, Philippe, Gnaedig, Isabelle, and Marion, Jean-Yves: "Abstraction-Based Malware Analysis Using Rewriting and Model Checking". en. In: *Computer Security – ESORICS 2012.* Ed. by David Hutchison et al. Vol. 7459. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 806–823. URL: http://link.springer.com/10.1007/978-3-642-33167-1 (visited on 02/02/2021).

[Bio+18] Biondi, Fabrizio et al.: "Effective, Efficient, and Robust Packing Detection and Classification". In: *Computers and Security* (2018), pp. 1–15. URL: https://hal.inria.fr/hal-01967597.

[BM06] Brosch, Tom and Morgenstern, Maik: *Runtime Packers: The Hidden Problem?* 2006. URL: https://www.av-test.org/fileadmin/pdf/publications/blackhat_2006_avtest_presentation_runtime_packers-the_hidden_problem.pdf.

[BR20] Ballenthin, W. and Raabe, M.: *Threat Research - capa: Automatically Identify Malware Capabilities.* Last Accessed: 2021-07-22. 2020. URL: https://www.fireeye.com/blog/threat-research/2020/07/capa-automatically-identify-malware-capabilities.html.

[Buc08] Buchmann, Johannes: *Einführung in die Kryptographie*. 4. edition. Springer-Verlag, 2008.

[Cal+15] Calvet, Joan et al.: "WaveAtlas: surfing through the landscape of current malware packers". In: *Virus Bulletin*. Prague, Czech Republic, 2015. URL: https://hal.archives-ouvertes.fr/hal-01198594.

[CC90] Chikofsky, E.J. and Cross, J.H.: "Reverse engineering and design recovery: a taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17.

[Che+18] Cheng, Binlin et al.: "Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost". en. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, 2018, pp. 395–411. URL: https://dl.acm.org/doi/10.1145/3243734.3243771 (visited on 10/24/2021).

[Che+21] Cheng, Binlin et al.: "Obfuscation-Resilient Executable Payload Extraction From Packed Malware". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021, pp. 3451–3468. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/cheng-binlin.

[Com+10] Comparetti, Paolo Milani et al.: "Identifying Dormant Functionality in Malware Programs". en. In: *2010 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, 2010, pp. 61–76. URL: http://ieeexplore.ieee.org/document/5504706/ (visited on 03/06/2021).

[Con03] Connolly, Kevin J.: *Law of Internet Security and Privacy*. Aspen Publishers, 2003.

[cor] coresecurity: *Dynamic Binary Instrumentation Frameworks: I know youre there spying on me*. Last Accessed: 2021-08-02. URL: https://www.coresecurity.com/core-labs/publications/dynamic-binary-instrumentation-frameworks.

[Cor20] Corporation, MITRE: *Ingress Tool Transfer*. Last Accessed: 2021-10-10. 2020. URL: https://attack.mitre.org/techniques/T1105/.

[Cor21] Corporation, Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Order Number: 325462-075US. 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[Cuc] Cuckoo: *Cuckoo Sandbox*. Last Accessed: 2021-06-22. URL: https://cuckoosandbox.org.

[Dam+17] Damodaran, Anusha et al.: "A comparison of static, dynamic, and hybrid analysis for malware detection". In: *Journal of Computer Virology and Hacking Techniques* 13 (2017).

[Dan17] Daniel Plohmann: *Knowledge Fragment: Hardening Win7 x64 on VirtualBox for Malware Analysis*. Last Accessed: 2021-10-31. 2017. URL: https://danielplohmann.github.io/blog/2017/02/05/kf-hardening-win7.html.

[DBC13] Devor, Tevi, Berkowits, Sion, and Corporation, Intel: *Pin: Intel's Dynamic Binary Instrumentation Engine - Pin Tutorial*. Last Accessed: 2021-08-01. 2013. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf.

[EKH13] Eskandari, Mojtaba, Khorshidpour, Zeinab, and Hashemi, Sattar: "HDM-Analyser: A Hybrid Analysis Approach Based on Data Mining Techniques for Malware Detection". In: *J. Comput. Virol.* 9.2 (2013), pp. 77–93. URL: https://doi.org/10.1007/s11416-013-0181-8.

[Fao20] Faou, Matthieu: *From Agent.BTZ to ComRAT v4: A ten-year journey*. Last Accessed: 2021-10-31. 2020. URL: https://www.welivesecurity.com/2020/05/26/agentbtz-comratv4-ten-year-journey/.

[Fir20a] FireEye: *Capa*. Last Accessed: 2021-06-22. 2020. URL: https://github.com/fireeye/capa.

[Fir20b] FireEye: *Capa Rules*. Last Accessed: 2021-06-22. 2020. URL: https://github.com/fireeye/capa-rules.

[Fou20] Foundation, Volatility: *The Volatility Foundation - Open Source Memory Forensics*. Last Accessed: 2021-10-22. 2020. URL: https://www.volatilityfoundation.org/.

[fre21] freedesktop.org: *What is D-Bus?* Last Accessed: 2021-10-31. 2021. URL: https://www.freedesktop.org/wiki/Software/dbus/.

[Fri] Frida: *Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*. Last Accessed: 2021-06-22. URL: https://frida.re/.

[Fri18] Frida: *Frida 10.8 Released*. Last Accessed: 2021-10-31. 2018. URL: https://frida.re/news/2018/04/28/frida-10-8-released/.

[Fri20] Frida. Last Accessed: 2021-10-31. 2020. URL: https://github.com/frida/frida-core/blob/84efaca65b0028fb0b7074e812ed64bb1b5fb5a5/lib/payload/spawn-monitor.vala.

[Fri21a] Frida. Last Accessed: 2021-10-31. 2021. URL: https://github.com/frida.

[Fri21b] Frida. Last Accessed: 2021-10-31. 2021. URL: https://github.com/frida/frida-gum/blob/master/gum/gumstalker.c.

[Fri21c] Frida. Last Accessed: 2021-10-31. 2021. URL: https://github.com/frida/frida-gum/blob/master/gum/guminterceptor.c.

[Fri21d] Frida: *Hacking*. Last Accessed: 2021-10-31. 2021. URL: https://frida.re/docs/hacking/.

[Fri21e] Frida: *Stalker*. Last Accessed: 2021-10-31. 2021. URL: https://frida.re/docs/stalker/.

[Gee+14] Geers, Kenneth et al.: "WORLD WAR C: Understanding Nation-State Motives Behind Today's Advanced Cyber Attacks". In: (2014). URL: https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/fireeye-wwc-report.pdf.

[GF11] Gennari, Jeff and French, David: "Defining malware families based on analyst insights". In: *2011 IEEE International Conference on Technologies for Homeland Security (HST)*. Waltham, MA, USA: IEEE, 2011, pp. 396–401. URL: http://ieeexplore.ieee.org/document/6107902/ (visited on 05/12/2021).

[HB99] Hunt, Galen and Brubacher, Doug: "Detours: Binary Interception of Win32 Functions". In: *Windows NT 3rd Symposium (Windows NT 3rd Symposium)*. Seattle, WA: USENIX Association, 1999. URL: https://www.usenix.org/conference/windows-nt-3rd-symposium/detours-binary-interception-win32-functions.

[HCA11] Hutchins, Eric, Cloppert, Michael, and Amin, Rohan: "Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains". In: *Leading Issues in Information Warfare & Security Research* 1 (2011).

[Hof13] Hoffmann, Dirk W.: *Grundlagen der technischen Informatik*. 3. Edition. Carl Hanser Verlag, 2013.

[Hyb21] Hybrid Analysis: *Free Automated Malware Analysis Service - powered by Falcon Sandbox*. Last Accessed: 2021-10-22. 2021. URL: https://www.hybrid-analysis.com/.

[Inc] Inc., Trend Micro: *A Brief History of Notable Online Banking Trojans*. Last Accessed: 2021-07-06. URL: https://www.trendmicro.com/vinfo/it/security/news/cybercrime-and-digital-threats/online-banking-trojan-brief-history-of-notable-online-banking-trojans.

[Inc21] Inc., FireEye: *FireEye Announces Plans to Relaunch as Mandiant, Inc. at Cyber Defense Summit 2021 and Trade on Nasdaq as MNDT*. Last Accessed: 2021-10-31. 2021. URL: https://www.fireeye.com/company/press-releases/2021/fireeye-announces-mandiant-relaunch-at-cds-2021-trade-as-mndt.html.

[Insa] Institute, AV-TEST: *AV-ATLAS - Malware & PUA*. Last Accessed: 2021-10-25. URL: https://portal.av-atlas.org/malware/statistics.

[Insb] Institute, AV-TEST: *Malware Statistics & Trends Report*. Last Accessed: 2021-07-06. URL: https://www.av-test.org/en/statistics/malware/.

[Int21] International, WildList: *The WildList Organization International*. Last Accessed: 2021-10-24. 2021. URL: http://www.wildlist.org/.

[Inv20] Investigation FBI, Federal Bureau of: "Internet Crime Report 2020". In: (2020). URL: https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf.

[Joe21a] Joe Security LLC: *Deep Malware Analysis - Joe Sandbox*. Last Accessed: 2021-10-22. 2021. URL: https://www.joesecurity.org/.

[Joe21b] Joe Security LLC: *Joe Sandbox Cloud Community Edition*. Last Accessed: 2021-10-22. 2021. URL: https://www.joesandbox.com/.

[JPP19] Jenke, Thorsten, Plohmann, Daniel, and Padilla, Elmar: "RoAMer: The Robust Automated Malware Unpacker". en. In: (2019). 14th International Conference on Malicious and Unwanted Software (MALWARE), Nantucket, MA, USA, 2019, pp. 67-74., p. 8. URL: https://github.com/fkie-cad/RoAMer.

[Kuc+21] Kuchler, Alexander et al.: "Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes". en. In: (2021), p. 18.

[KVK11] Kirat, Dhilung, Vigna, Giovanni, and Kruegel, Christopher: "BareBox: Efficient Malware Analysis on Bare-Metal". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC '11. Orlando, Florida, USA: Association for Computing Machinery, 2011, pp. 403–412. URL: https://doi.org/10.1145/2076732.2076790.

[KY17] Korczynski, David and Yin, Heng: "Capturing Malware Propagations with Code Injections and Code-Reuse Attacks". en. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, 2017, pp. 1691–1708. URL: https://dl.acm.org/doi/10.1145/3133956.3134099 (visited on 03/06/2021).

[Lat21] Latto, Nica: *Worm vs. Virus: What's the Difference and Does It Matter?* Last Accessed: 2021-07-06. 2021. URL: https://www.avast.com/c-worm-vs-virus.

[LD03] Linn, Cullen and Debray, Saumya: "Obfuscation of Executable Code to Improve Resistance to Static Disassembly". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. Washington D.C., USA: Association for Computing Machinery, 2003, pp. 290–299. URL: https://doi.org/10.1145/948109.948149.

[Leo+21] Leon, Roee S. et al.: "Hypervisor-assisted dynamic malware analysis". en. In: *Cybersecurity* 4.1 (2021), p. 19. URL: https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00083-9 (visited on 10/27/2021).

[Lev] Levi, Osnat: *Pin - A Dynamic Binary Instrumentation Tool*. Last Accessed: 2021-08-01. URL: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[Lig+11] Ligh, Michael et al.: *Malware Analyst's Cookbook*. Wiley, 2011.

[LLC21] LLC, ANY.RUN: *Interactive Online Malware Sandbox*. Last Accessed: 2021-10-22. 2021. URL: https://any.run/.

[MBC21a] MBCProject: *Malware Behavior Catalog*. Last Accessed: 2021-10-20. 2021. URL: https://github.com/MBCProject.

[MBC21b] MBCProject: *Malware Behavior Catalog v2.1*. Last Accessed: 2021-10-20. 2021. URL: https://github.com/MBCProject/mbc-markdown/tree/v2.1.

[McA] McAfee: *What Is Ransomware?* Last Accessed: 2021-07-06. URL: https://www.mcafee.com/enterprise/de-de/security-awareness/ransomware.html.

[MCJ07] Martignoni, Lorenzo, Christodorescu, Mihai, and Jha, Somesh: "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware". In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pp. 431–441.

[Mic] Microsoft: *Microsoft Research Detours Package*. Last Accessed: 2021-06-22. URL: https://github.com/microsoft/Detours.

[Mic13] Microsoft: "Microsoft Portable Executable and Common Object File Format Specification". In: (2013). Revision 8.3.

[Mic21a] Microsoft: *rundll32 | Microsoft Docs*. Last Accessed: 2021-08-01. 2021. URL: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/rundll32.

[Mic21b] Microsoft: "Windows Sysinternals". In: (2021). Last Accessed: 2021-07-06. URL: https://docs.microsoft.com/en-us/sysinternals/.

[Mil14] Miller, Jan: "Hybrid Analysis - NextGen Technology for Advanced Malware". en. In: (2014), p. 14.

[MITa] MITRE: *MITRE ATT&CK*. Last Accessed: 2021-10-26. URL: https://attack.mitre.org/versions/v3/.

[MITb] MITRE: *MITRE ATT&CK framework*. Last Accessed: 2021-06-22. URL: https://attack.mitre.org/.

[MITc] MITRE: *Techniques - Enterprise*. Last Accessed: 2021-10-26. URL: https://attack.mitre.org/versions/v9/techniques/enterprise/.

[Neu93] Neumann, J. von: "First draft of a report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75.

[Nora] NortonLifeLock: *Malware*. Last Accessed: 2021-07-06. URL: https://us.norton.com/internetsecurity-malware.html.

[Norb] NortonLifeLock: *What is a Trojan? Is it a virus or is it malware?* Last Accessed: 2021-07-06. URL: https://us.norton.com/internetsecurity-malware-what-is-a-trojan.html.

[NZ20] NZ, CERT: *Emotet Malware being spread via email*. Last Accessed: 2021-10-20. 2020. URL: https://www.cert.govt.nz/it-specialists/advisories/emotet-malware-being-spread-via-email/.

[OD19] Oosthoek, Kris and Doerr, Christian: "SoK: ATT&CK Techniques and Trends in Windows Malware". en. In: *Security and Privacy in Communication Networks*. Ed. by Songqing Chen et al. Vol. 304. Series Title: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Cham: Springer International Publishing, 2019, pp. 406–425. URL: http://link.springer.com/10.1007/978-3-030-37228-6 (visited on 03/06/2021).

[Ora21] Oracle: *Oracle VM VirtualBox*. Last Accessed: 2021-10-24. 2021. URL: https://www.virtualbox.org/.

[PE21a] Plohmann, Daniel and Enders, Steffen: *Citadel (Malware Family)*. Last Accessed: 2021-10-10. 2021. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.citadel.

[PE21b] Plohmann, Daniel and Enders, Steffen: *Emotet (Malware Family)*. Last Accessed: 2021-10-20. 2021. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.emotet.

[PE21c] Plohmann, Daniel and Enders, Steffen: *General Statistics*. Last Accessed: 2021-10-07. 2021. URL: https://malpedia.caad.fkie.fraunhofer.de/stats/general.

[PE21d] Plohmann, Daniel and Enders, Steffen: *Malpedia (Fraunhofer FKIE)*. Last Accessed: 2021-07-27. 2021. URL: https://malpedia.caad.fkie.fraunhofer.de/.

[PE21e] Plohmann, Daniel and Enders, Steffen: *Sysraw Stealer (Malware Family)*. Last Accessed: 2021-10-30. 2021. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.sysraw_stealer.

[PE21f] Plohmann, Daniel and Enders, Steffen: *UrlZone (Malware Family)*. Last Accessed: 2021-10-10. 2021. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.urlzone.

[Plo+17] Plohmann, Daniel et al.: "ia: A Collaborative Effort to Inventorize the Malware Landscape". In: *The Journal on Cybercrime & Digital Investigations* 3.1 (2017). URL: https://journal.cecyf.fr/ojs/index.php/cybin/article/view/17.

[Plo21a] Plohmann, Daniel: *ApiScout*. Last Accessed: 2021-10-22. 2021. URL: https://github.com/danielplohmann/apiscout.

[Plo21b] Plohmann, Daniel: *SMDA*. Last Accessed: 2021-10-22. 2021. URL: https://github.com/danielplohmann/smda.

[Pro16] Proofpoint: *Vawtrak and UrlZone Banking Trojans Target Japan*. Last Accessed: 2021-10-10. 2016. URL: https://www.proofpoint.com/us/threat-insight/post/Vawtrak-UrlZone-Banking-Trojans-Target-Japan.

[Rea21a] ReactOS Team and Contributors: *proc.c File Reference*. Last Accessed: 2021-10-31. 2021. URL: https://doxygen.reactos.org/d9/dd7/dll_2win32_2kernel32_2client_2proc_8c.html#a13a0f94b43874ed5a678909bc39cc1ab.

[Rea21b] ReactOS Team and Contributors: *Welcome to ReactOS*. Last Accessed: 2021-10-31. 2021. URL: https://reactos.org/.

[Reg20] Regan, Joseph: *A Brief History of Computer Viruses*. Last Accessed: 2021-07-06. 2020. URL: https://www.avg.com/en/signal/history-of-viruses.

[Rey10] Reynaud, Daniel: "Analyse de codes auto-modifiants pour la sécurité logicielle". 2010INPL049N. PhD thesis. 2010. URL: http://www.theses.fr/2010INPL049N/document.

[Rie+11] Rieck, Konrad et al.: "Automatic analysis of malware behavior using machine learning". en. In: *Journal of Computer Security* 19.4 (2011), pp. 639–668. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/JCS-2010-0410 (visited on 03/08/2021).

[RM10] Roundy, Kevin A. and Miller, Barton P.: "Hybrid Analysis and Control of Malware". In: *Recent Advances in Intrusion Detection*. Ed. by Somesh Jha, Robin Sommer, and Christian Kreibich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 317–338.

[Ros+12] Rossow, Christian et al.: “Prudent Practices for Designing Malware Experiments: Status Quo and Outlook”. en. In: *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, 2012, pp. 65–79. URL: http://ieeexplore.ieee.org/document/6234405/ (visited on 10/25/2021).

[Roy+06] Royal, Paul et al.: “PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 2006, pp. 289–300.

[Rub19] Rubín, Jan: *Clipsa – Multipurpose password stealer*. Last Accessed: 2021-10-31. 2019. URL: https://decoded.avast.io/janrubin/clipsa-multipurpose-password-stealer/.

[Seg12] Segura, Jérôme: *Citadel: a cyber-criminal's ultimate weapon?* Last Accessed: 2021-10-10. 2012. URL: https://blog.malwarebytes.com/threat-analysis/2012/11/citadel-a-cyber-criminals-ultimate-weapon/.

[SH12] Sikorski, Michael and Honig, Andrew: *Practical Malware Analysis*. No Starch Press, 2012.

[Sou12] Soulami, Tarik: *Inside Windows Debugging*. Last Accessed: 2021-07-27. 2012. URL: https://www.microsoftpressstore.com/articles/printerfriendly/2201303.

[Sta+10] Stallman, Richard et al.: *Debugging with GDB - The GNU Source-Level Debugger*. 9. Edition. Free Software Foundation, 2010. URL: https://www.eecs.umich.edu/courses/eecs373/readings/Debugger.pdf.

[Sta21] Standard, OASIS: “STIX Version 2.1”. In: (2021). URL: https://docs.oasis-open.org/cti/stix/v2.1/os/stix-v2.1-os.pdf.

[Str+18] Strom, Blake E. et al.: “MITRE ATT&CK™: Design and Philosophy”. en. In: Project No.:01ADM105-PI (2018), p. 37. URL: https://www.mitre.org/sites/default/files/publications/pr-18-0944-11-mitre-attack-design-and-philosophy.pdf (visited on 08/08/2021).

[TB14] Tanenbaum, Andrew S. and Bos, Herbert: *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[The21] The GNOME Project: *Vala - Compiler Using the GObject Type System*. Last Accessed: 2021-10-31. 2021. URL: https://wiki.gnome.org/Projects/Vala/About.

[Uga+15] Ugarte-Pedrero, Xabier et al.: “SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 659–673.

[UPX] UPX: *UPX: the Ultimate Packer for eXecutables*. Last Accessed: 2021-08-01. URL: https://upx.github.io/.

[Vir] VirusTotal: *Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community*. Last Accessed: 2021-06-22. URL: https://www.virustotal.com/.

[Vir21] Virustotal: *Virustotal - Home*. Last Accessed: 2021-10-22. 2021. URL: https://www.virustotal.com/.

[Viv21] Vivisect: *Vivisect / Vdb / Vtrace*. Last Accessed: 2021-10-22. 2021. URL: https://github.com/vivisect/vivisect.

[VMR21] VMRay: *Cyber Security Malware Analysis & Detection Sandbox*. Last Accessed: 2021-10-22. 2021. URL: https://www.vmray.com/.

[Web] Webroot: *What is a packet sniffer?* Last Accessed: 2021-07-06. URL: https://www.webroot.com/us/en/resources/glossary/what-is-a-packet-sniffer.

[Xen21] Xen Project: *Home - Xen Project*. Last Accessed: 2021-10-24. 2021. URL: https://xenproject.org/.

[ZC19] Zimba, Aaron and Chishimba, Mumbi: "On the Economic Impact of Crypto-ransomware Attacks: The State of the Art on Enterprise Systems". In: *European Journal for Security Research* 4 (2019).

7 Appendix

In this section are additional data and statistics located. The attached data supplement the results of this work but are not core results. In addition to the listed data we supply a large repository of all statistics in the form of `csv` and `ods` tables.

7.1 Additional Malpedia Statistics

We listed additional information about Malpedia and side results that are not directly related to the topic of this work although they supplement our statistics.

Table 11: *Ratio of all Windows samples where unpacked and dumped files are available.*

	$Ratio_{ALL}$	$Ratio_{EXE}$	$Ratio_{DLL}$
count	968	651	317
mean	0,89	0,89	0,91
std	0,26	0,27	0,24
min	0,00 %	0,00 %	0,00 %
25%	96,15 %	95,67 %	96,91 %
50%	99,64 %	99,62 %	99,73 %
75%	100,00 %	100,00 %	100,00 %
max	100,00 %	100,00 %	100,00 %

7.2 Additional Capa Statistics

In this section we provide additional Capa statistics. We break down the frequencies of the Capa rules found by hits per family,hits per sample and absolute hits.

Table 12: *Top 20 capa rules ordered by counted hits (family, sample and absolute).*

family		sample		absolute	
capa rule	hits	capa rule	hits	capa rule	hits
contain loop	892	contain loop	2759	contain loop	630733
delay execution	754	delay execution	2250	delay execution	37743
write file	744	calculate modulo 256 via x86 assembly	2184	calculate modulo 256 via x86 assembly	27864
calculate modulo 256 via x86 assembly	742	parse PE header	2182	encode data using XOR	27560
terminate process	732	write file	2160	create or open file	13997
create or open file	715	create or open file	2118	link function at runtime	13965
contain a resource (.rsrc) section	697	encode data using XOR	2073	create or open registry key	12315
parse PE header	684	terminate process	2067	parse PE header	11907
read file	661	read file	1896	allocate memory	10957
encode data using XOR	656	create thread	1808	contain obfuscated stackstrings	9642
link function at runtime	641	link function at runtime	1801	write file	9348
create thread	609	get common file path	1786	get common file path	8983
query environment variable	605	query environment variable	1743	execute anti-VM instructions	8734
create process	603	create process	1647	create thread	8065
get common file path	584	contain a resource (.rsrc) section	1593	terminate process	7360
accept command line arguments	573	accept command line arguments	1576	PEB access	6683
delete file	527	delete file	1554	read file	6407
create or open registry key	518	create or open registry key	1533	create process	5568
allocate memory	517	parse PE exports	1520	delete file	5262
PEB access	486	allocate memory	1493	allocate RW memory	5234

Statement of Authorship

I hereby confirm that the work presented in this master 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 8, 2021

Felix Bilstein