Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

MASTER THESIS

Signature-Based Detection of Behavioural Malware Features with Windows API Calls

Simon Jansen

Erlangen, June 24, 2020

Examiner: Prof. Dr.-Ing. Felix Freiling
Advisor: Dr. Christian Gorecki

This version of the master thesis does not comply with the version originally submitted to the Friedrich-Alexander University Erlangen-Nuremberg in fulfillment of the requirements for the degree of Master of Science.

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, June 24, 2020

Simon Jansen

Abstract

The rising complexity of IT infrastructures in general offers a large surface for cyber attacks and adversaries to understand how to take great advantage of this. Malicious software plays thereby a key role in compromising a victim's infrastructure as it is used to carry out the attack and gain persistence. As a consequence, effectively protecting against cyber attacks is directly correlated with the reliable detection of malicious software. However, it is not only the complexity of IT infrastructures that is rising, but also that of the malicious software.

In this work, we introduce the signature-based detection approach *dynmx* which helps domain experts to efficiently analyse malicious software based on its behavioural characteristics. Therefore, a concise and easy-to-learn domain specific language is proposed in order to allow domain experts to specify known behavioural malware features in a precise and consistent manner. This, in turn, enables domain experts to collaboratively share these specified behaviour signatures. Since recent research has shown that the API call usage is a valuable resource for analysing malicious software, it is exactly this abstraction layer that we make use of for the definition of malware features. The source of information used for our proposed detection approach are function logs – chronologically ordered sequences of API functions called by a certain process which is monitored by a sandbox. For the detection of such signatures in function logs, we propose a detection algorithm based on the solution to the longest common subsequence (LCS) problem. For the sake of simplicity, the detection approach introduced in this work can be classified as equivalent to the well-adopted YARA framework but for behavioural instead of static characteristics.

All of the theoretic concepts presented in this work were implemented as a fully-working prototype application. This prototype application formed the basis to evaluate our approach in three dimensions – the detection quality, the capabilities of the signature DSL and the detection performance. In conclusion of this work, it is shown that the *dynmx* detection approach has the potential to improve the malware analysis process in general as it allows domain experts to precisely and efficiently detect known malicious behaviour with the help of signatures.

Zusammenfassung

Die steigende Komplexität von IT-Infrastrukturen bietet eine große Angriffsoberfläche für Cyberattacken – und Angreifer verstehen daraus großen Nutzen für sich zu ziehen. Schadsoftware bildet eine der Kerninstrumente bei dem Angriff auf IT-Infrastrukturen, da sie für sowohl die eigentliche Angriffsausführung als auch für die dauerhafte Kompromittierung der Infrastruktur zu Anwendung kommt. Folglich steht der Schutz vor derartigen Angriffen im direkten Zusammenhang mit der zuverlässigen Erkennung ebendieser Schadsoftware. Jedoch steigt nicht nur die Komplexität der IT-Infrastrukturen, sondern auch die Komplexität der Schadsoftware als solches.

In der vorliegenden Arbeit wird der signaturbasierte Detektionsansatz *dynmx* eingeführt, der Domänenexperten die effiziente Analyse von Schadsoftware auf Basis von Charakteristiken des Laufzeitverhaltens ermöglicht. Die prägnante und leicht zu erlernende domänenspezifische Sprache hilft Domänenexperten bei der präzisen und konsistenten Spezifikation von bekannten Merkmalen schadhaften Verhaltens. Dies wiederum ermöglicht Domänenexperten untereinander den leichten Austausch dieses spezifizierten Schadsoftwareverhaltens. Da aktuelle Forschungen in diesem Bereich gezeigt haben, dass die von Schadsoftware verwendeten API-Funktionsaufrufe eine wertvolle Ressource für die Analyse darstellen, wird ebendiese Abstraktionsschicht für die Spezifikation von schadhaftem Laufzeitverhalten genutzt. Die Detektion erfolgt auf Basis von so genannten Function Logs, welche die flache chronologische Abfolge von API-Funktionsaufrufen eines mithilfe einer Sandbox untersuchten Prozesses darstellen. Um derartige Signaturen in Function Logs erkennen zu können, wird ein auf der Lösung des Longest Common Subsequence (LCS) Problem basierender Detektionsalgorithmus vorgestellt. Der Einfachheit halber, kann der in dieser Arbeit vorgestellte Detektionsansatz als Äquivalent des weitverbreiteten Werkzeugs YARA eingeordnet werden, jedoch für die Erkennung von schadhaftem Laufzeitverhalten anstelle von statischen Merkmalen der Schadsoftware.

Die in dieser Arbeit vorgestellten theoretischen Konzepte wurden als voll-funktionsfähige prototypische Applikation implementiert. Diese Applikation bildete die Grundlage für die Evaluation des Ansatzes in drei verschiedenen Dimensionen – die Detektionsqualität, den Funktionsumfang der domänenspezifischen Sprache und die Performance der Detektion. Das Fazit dieser Arbeit zeigt, dass der *dynmx* Detektionsansatz das Potenzial zur generellen Verbesserung des Schadsoftwareanalyseprozesses hat, indem Domänenexperten die präzise und effiziente Detektion von bekanntem schadhaftem Laufzeitverhalten mithilfe von Signaturen ermöglicht wird.

Contents

List of Figures

List of Tables

1 INTRODUCTION

In light of the rapidly growing risk and importance of cyber attacks targeting central IT infrastructures, both the private sector as well as state institutions and authorities in the public sector are facing an increasing threat level. Since the value creation of a majority of today's economy requires the efficient and uninterrupted usage of IT infrastructure, attacks on it can threaten the very existence of private companies and therefore the overall economy. For example, digital industries in the economy of the United Kingdom had a 32% faster growth than the rest of the economy between 2010 and 2014 [47]. In particular, small and medium-size enterprises (SMEs) which make up 99.8% of European enterprises are seeing a rising threat level and are not being well-prepared for such attacks against their IT infrastructure [73]. Furthermore, the cost impact of handling and investigating a successful attack is somewhat higher for SMEs in comparison to large enterprises since they lack sufficient knowledge and funding to raise adequate IT security measures and external knowledge to professionally handle such IT security incidents [73]. As a consequence, SMEs in particular are a vulnerable and valuable target for adversaries since they hold a considerable share of intellectual property. In short, SMEs *"are a source of innovation within our European economy"* [73, p. 31]. However, it is not only SMEs that are exposed – large enterprises also face a severe risk of being attacked but can typically handle security incidents better in the overall view. In general, as of 2017, at least 80% of European companies were hit by at least one IT security incident according to [73] at an overall increasing incidence of cybersecurity attacks in general. Even the global COVID-19 pandemic could not prevent adversaries from carrying out perfidious attacks against critical healthcare infrastructure, in particular hospitals [34, 74, 178]. Not least, this reveals the determination and scrupulousness with which adversaries proceed in their campaigns.

IT infrastructure in general is typically very complex and therefore provides a vast surface for attacks. This technical attack surface, together with human factors exploited for social engineering techniques is the main source of malicious software (malware) – the most common threat and core instrument for carrying out the actual attack and gaining persistence in the victim's IT infrastructure. Consequently, in order to protect against this kind of attack, malicious software needs to be detected efficiently and reliably. This is a challenging task if we take into account that malicious software in general became more and more complex over the last three decades [17]. However, the detection

efficiency is thereby indispensable in order to cope with the sheer volume of new malware samples arising every single day. In 2019 alone, Kaspersky identified 46,156 ransomware modifications and in total 24.6 million unique malicious objects [54]. According to the AV-Test Institute, as of June 2020, they register over 360,000 unique malicious software samples every single day [8] and overall identified 137.5 million unique samples in 2018 [7]. This amount of new malware is by no means made up of new malware developments according to their actual code basis but instead different binary representations of samples belonging to the same malware family sharing the same code basis to some extent. By leveraging cryptography and obfuscation techniques (also referred to as packing [146]), malware authors are enabled to produce a vast amount of unique malware samples that have a completely different binary representation but share the same code basis [122]. The high volume of malware samples with a wide variety of binary representations helps malware authors to evade detections and thus circumvent typical security measures like the usage of an anti-virus software.

Malware sandboxes (referred to as sandboxes in the following) or more general dynamic analysis methodologies allow domain experts to examine malicious software with less effort compared to the traditional static reverse engineering approach. Since the malware is automatically executed in a controlled environment, it unpacks itself during runtime and reveals certain parts of its malicious behaviour. This behaviour is closely monitored in the sandbox environment and made available to the domain expert in processed form, typically in form of an analysis report. Depending on the sandbox used, this analysis report can be comprehensive and gives a good overview of the malware's characteristics and functionality. As the effort to gain knowledge about unknown malware is reduced with the usage of sandboxes in the analysis process, they help domain experts to cope with the complexity and high volume of newly arising malware. In addition, domain experts are able to quickly derive actionable indicators needed in order to secure the IT infrastructure and detect the existence of attacks. Consequently, sandboxes have improved the overall malware analysis process by providing valuable behavioural information of the analysed malware at relatively low effort by relying on automation.

1.1 Motivation

The way malicious functionality is detected and derived based on the monitored behaviour by a sandbox is typically not published by the vendors. Instead, the actual detection capability of a certain sandbox represents an important unique selling point. Of course, this is not applicable to open source sandboxes like Cuckoo [153] since the source code implementing the detection is accessible but does apply to commercial sandbox products. Furthermore, from the perspective of a domain expert, the detection of behaviour implemented by sandboxes is not flexible enough and limited in its extendability. However, it is exactly this flexibility and extendability that are desirable for domain experts in order to keep pace with the complexity and variety of malware in general. Consequently, if at all possible, former unknown malicious behaviour manually identified and analysed by domain experts can only be incorporated into the behavioural detection capability of a sandbox to a limited extent. Moreover, malware authors are typically creative in finding many different ways to implement certain malicious behaviour in order to evade detection. In light of this evolution, it would be audacious to claim that sandboxes aim for completeness in detecting malicious behaviour. In practise, this leads to repetitive analysis tasks since the automation provided by the sandbox can not be leveraged in order to detect these newly identified behavioural characteristics. Rather, domain experts manually assess the raw unprocessed information provided by the sandbox in order to manually detect certain malicious behaviour. As this raw information provided by sandboxes can become very extensive and confusing, this approach seems not to be efficient and effective but rather susceptible to errors and cumbersome.

The focus of sandboxes generally lies in the broadest possible detection of malicious behaviour. This can be derived directly from the core objective of sandboxes – to detect a wide variety of malware or malicious behaviour in general. Hence, the detection capabilities implemented in sandboxes

typically strive for generic malicious behaviour. Generic, in this context, refers to behaviour that is shared between a large subset of malware samples and thus can not be used for the characterisation of certain malware types or families. The flexible adaptation of these generic behaviours to more specialised forms is desirable in practise in order to allow domain experts to define characteristics of certain malware types or families. This refers especially to the identification of similar malware samples that belong to a given family which is of high relevance in the analysis process. For instance, the effective prevention and mitigation of successful compromises depends primarily on the identification of similar malware samples that are incorporated in the analysis process in order to gain complete analysis results. In terms of a successful compromise, these analysis results, which should be as complete as possible, help analysts determine the extent of the compromise in the incident response process. As for the prevention, complete analysis results contribute to the best possible detection and defence since relevant indicators can be blocked in perimeter systems of the infrastructure. This identification is mainly achieved on the basis of static characteristics rather than the actual malicious behaviour in today's practise. Overall, the broad detection of generic malicious behaviour is beneficial for the triage and initial assessment of malware but the flexible definition of specialisations of this generic behaviour is desirable in order to improve the identification of related malware samples.

With reference to defining static characteristics of malware or more general binary data in the form of a signature or rule, YARA [1] has become a well-adopted industry standard. For instance, the identification of similar malware samples relies to a large extent on YARA rules. The majority of security products that are used for the detection of malware including sandboxes provide integrations for the incorporation of YARA rules into the detection process. Hence, YARA rules are agnostic in terms of the security product since an own independent framework that implements the signature matching process is provided. As a result, domain experts extensively use and collaboratively share YARA rules for a wide variety of use cases. For example, to detect and classify malware samples or malicious network traffic. The key advantage of using signatures is obvious since they assure robust and reproducible results. Moreover, signatures define characteristics that were manually identified by domain experts at considerable effort in a precise and consistent manner. However, since YARA has its focus on static characteristics, the detection lacks of precision and often produces false positive detections in practise. As previously mentioned, the binary representation of malware changes frequently and often dramatically due to the use of packing mechanisms [11, 43]. This, in turn, is challenging for domain experts as they need to find robust characteristics that exist in all of the malware samples that should be detected with a certain YARA rule. Therefore, domain experts proceed to define YARA rules for unpacked versions of a certain malware or match YARA rules directly in memory images of sandbox executions where malware often resides at least partly unpacked. These unpacked versions are more stable in their binary representation. But even the unpacked versions can be obfuscated beyond the packing mechanisms [143, 144, 146, 182]. In addition, unpacking a malware sample can be a complex and time consuming manual task. In particular, if several packing layers are combined by the malware authors.

As for the reusable specification of behavioural characteristics of malware, no practically relevant and sandbox-agnostic equivalent to YARA rules exists. Behavioural characteristics identified in a rather complex manual examination are typically described in prose by domain experts in the form of blog entries or white papers. Of course, this information is very valuable for other domain experts since they can rely on analysis work that was already done within the community. But the information provided by a white paper or blog post is typically not directly actionable for the malware analysis process and not as precise as a signature. In the past, a lot of effort was invested in the development of possibilities to precisely define and share so called Indicators of Compromise (IoCs). As the name suggests, these indicators can be leveraged in order to detect a successful compromise in an IT infrastructure. For example, STIX™ [108] and OpenIOC [42] are structured languages used to precisely specify IoCs and the MISP project [90] aims to provide a platform for sharing IoCs. But from the viewpoint of a domain expert, IoCs only reflect one of the results of the malware analysis process. They define the characteristics of malware which can be detected in infrastructures rather than the actual behaviour during the runtime. But it is precisely this runtime behaviour that is of relevance for domain experts in order to study and understand the functionality

of a certain malware sample. Consequently, languages used to define IoCs do not qualify for the definition of malware behaviour as it is needed by domain experts.

To implement malicious behaviour, malware heavily relies on the usage of the application programming interface (API) provided by the underlying operating system, especially on the Microsoft Windows platform. Consequently, the usage of API function calls is a precious source of information for the analysis of a malware's functionality [44, 58, 123]. Moreover, this source of information is robust since malware authors have to meet the individual requirements and specification of the API like every other developer leveraging API functions. Furthermore, most of the available obfuscation techniques focus on the assembly or binary layer of a malware sample rather than the API call level. As a result, this leads to characteristic API call sequences that are used to implement certain functionality while being immune to most of the available obfuscation techniques. Plohmann, Padilla and Enders suggest that this API call sequences used by malware can likely be more robust than the actual code basis since *"they are the semantic skeleton that describes the overall capabilities of a given malware family"* [123, p. 10].

The field of research proposed in this thesis can be outlined as such: combining the previously described gap in terms of the missing possibility to define signatures for behavioural characteristics of malware with the significance of the API usage for the malware analysis process. Since sequences of API function calls can be used to detect certain malicious behaviour [58], the introduction of a signature-based detection approach for the same seems to be a logical step towards the improvement of malware detection and analysis in general. The definition of signatures for recurring behaviour has the potential to lift the efficiency of the malware analysis process since the repetitive manual assessment of raw unprocessed information provided by the sandbox is automated. Hence, we form the research question addressed in the present thesis as follows: from the viewpoint of a domain expert, to what extent can the signature-based detection of malicious runtime behaviour based on API function calls which is derived from the dynamic analysis of malware in sandboxes contribute to the improvement of the malware analysis process?

1.2 Objective

The objectives of the present thesis are directly derived from the research question defined in the previous section outlining the motivation. Therefore, the main objective of our work is the introduction of a domain specific language (DSL) which can be used by domain experts in order to define behavioural characteristics of malware in a precise, consistent and sandbox-agnostic manner. Moreover, the DSL should be practise-oriented as well as easy to learn for domain experts. By introducing such DSL, we expand the signature-based and security product agnostic detection of malware from static characteristics to behavioural ones. In addition, we lay the foundation for domain experts to share analysis results of behavioural malware characteristics collaboratively.

Analogous to YARA, we propose a suitable detection algorithm in order to automatically identify known dynamic malware behaviour for unknown samples based on the signatures defined by domain experts. This detection algorithm is thoroughly designed independent from the sandbox that performed the actual dynamic analysis. As for the basis of the signature detection, we leverage the raw unprocessed information provided by the sandbox that is otherwise manually assessed by domain experts. Although the detection approach proposed in this thesis is sandbox-agnostic in general, we use the raw information provided by the VMRay Analyzer sandbox [166] in particular as basis for the development of the detection approach. Foremost, we chose the VMRay Analyzer sandbox as source of information since it provides high-quality behavioural information, especially in terms of the API call usage of analysed executables. This API call usage is represented by so called function logs in terms of the VMRay Analyzer sandbox. In general, function logs are understood as the chronological sequence of complete API function calls including the function call arguments as well as the return value as they were monitored by the sandbox in the course of the dynamic analysis process.

Since not only API call sequences but also the usage of operating system resources is relevant to the analysis process, we leverage the research on access activity models proposed by Lanzi et al. in [63] and adapt the results to our needs. Hence, the objective is to deduce the usage of operating system resources from the function log in order to allow domain experts the detection of malicious behaviour. Furthermore, the access activity model derived from the function log can help the domain expert to better understand the malware's behaviour independent from the detection of certain signatures. For our adaptation of the access activity model we take file, Windows Registry and network resources into account.

In order to sustain our sandbox-agnostic approach, the harmonised storage of function logs originating from different sandboxes seems to be valuable. Therefore, we introduce a generic function log format which abstracts from the specifics of the function log's origin. The generic function log format will be specifically designed for our use case and thus does not claim to be complete.

In order to prove the practical feasibility of our proposed detection approach, we will implement the concepts proposed in this thesis as a fully working prototype. Moreover, a prototype signature library will be provided together with this work. The signatures that will be defined in the course of this thesis, specify malicious behaviour of real-world malware samples and substantiates the practical relevance and practicability of our work. As for the signature library, we take generic as well as specific malware features of selected malware families into account.

The last but not least important objective of the present thesis lies in the evaluation of our proposed prototype implementation. The most essential part of the evaluation is the assessment of the detection precision of our proposed signatures. In addition, we measure the detection performance as well as work out advantages of our proposed generic function log format in comparison to the function log formats provided by the VMRay Analyzer sandbox. Moreover, our signature DSL is evaluated in terms of the comprehensiveness and practicability in comparison to the solution provided by the VMRay Analyzer sandbox. Consequently, the evaluation provides a multi-dimensional assessment of our approach in terms of the detection quality, the comprehensiveness of the signature DSL and the advantages of the generic function log format.

1.3 Results

Overall, we make the following contributions with the present thesis:

1. a practise-oriented, sandbox-agnostic and concise signature DSL based on the general data representation syntax Yet Another Markup Language (YAML) [12],
2. a detection algorithm based on the well-known longest common subsequence (LCS) algorithm used to efficiently detect behavioural signatures in function logs,
3. an adapted and extended form of the access activity model proposed by Lanzi et al. [63],
4. the definition of a generic function log format leveraging the general data representation syntax JavaScript object notation (JSON),
5. a fully working proof of concept implementation of the aforementioned contributions,
6. a prototype signature library consisting of 13 signatures (9 signatures detecting generic malware features of the categories malicious code injection and persistence mechanisms as well as 4 signatures detecting specific malware features of the malware families DarkComet, Emotet, Formbook and Remcos) and
7. a comprehensive evaluation of our proof of concept implementation together with the proposed signature library in terms of the detection quality based on a large malicious data set consisting of 31,898 function logs.

By making the aforementioned contributions we fulfil all of the objectives described in the previous section. Thus, the contributions of this thesis can be seen as a comprehensive full-stack solution to

the identified gap of a missing equivalent for YARA rules in the domain of behavioural characteristics. We introduce the name *dynmx* [daɪˈnæmɪks] for this full-stack solution.

As a further results of the thesis, we identified the following limitations of our detection approach.

- The detection approach is designed process-centric and can not be used to detect certain malicious behaviour implemented with the use multiple processes working together. Furthermore, malicious behaviour spread over different threads can only be partly detected.
- The detection quality stands and falls with the quality of the function logs that we use as a basis for the detection.
- The malware features that should be detected and are defined in the form of signatures need to be specific on the API call or resource level. Unspecific sequences directly lead to false positive detections.
- Certain, complex function logs containing a large number of processes are challenging for the detection algorithm and, as a result, can lead to long detection runtimes.
- In general, our detection approach shares the exact same limitations of dynamic analysi stated in Section 2.2.

1.4 Related Work

In the course of the present thesis, we identified several related research and publications. In this section we will classify the identified research and differentiate our research from this related work.

The detection of malware based on malicious behaviour in general and system and API calls in particular was addressed in the majority of the reviewed research. For instance, Canzanese, Mancoridis and Kam propose a detection system used to identify malicious processes based on system call traces in [20]. They leverage several machine learning algorithms in order to detect unknown malware samples and thereby identify malicious processes on compromised hosts. The system call traces used in this research are limited to the system call function name solely and omit arguments as well as return values of the called functions. Furthermore, the system call traces are obtained using the hooking technique. Both Nikolopoulus and Polenakis [106] and Anderson et al. [5] present a graph-based detection approach which differentiates malicious from benign software. While Anderson et al. focus on dynamically collected in-order list of executed instruction on the assembly level as basis for the detection, Nikolopoulus and Polenakis construct system call dependency graphs based on system call traces and perform the detection on a higher level than individual instructions. As shown by Confora et al. in [19], malware detection based on API call sequences is not restricted to the Windows platform but can also be applied to malware that focuses on mobile devices. In this particular case, the authors detect Android malware by using machine learning methodologies. Since the actual level of detail of the collected system calls is not clearly stated, we can only deduce based on the publication that only the function call names were collected. Their approach mainly aims to distinguish malicious from benign Android applications. A comprehensive overview of research that leverages system calls for the detection of malware is provided by Canali et al. in [18]. The authors verify the detection results of a large set of distinctive detection approaches (behavioural models in their terms) and propose that the accuracy of certain detection approaches is very poor. However, the best performing detection approaches are based on more generic system call abstractions that incorporate the system call arguments into the detection process. All of the research described beforehand has in common that it proposes fully automated detection approaches which have the main objective to distinguish malicious from benign software. In contrast to this related research, our proposed approach does not aim to be fully automated but intentionally relies on the domain expert as critical subject of the overall detection process. What is detected by our proposed approach is solely defined by the domain expert and is not automatically identified using machine learning or other classification algorithms. Moreover, we aim to improve

the efficiency of the malware analysis process from the viewpoint of a domain expert rather than finding improvements for anti-virus solutions.

A further prominent research domain is found in the classification of malware based on API function calls. Recent research published by Kolosnjaji et al. [60] as well as Gupta, Sharma and Kaur [44] propose fully automated systems to classify malware by leveraging behavioural characteristics, in particular system and API call sequences. While Gupta et al. use fuzzy hashing for the classification, Kolosnjaji et al. make use of deep learning methodologies (neural networks) in order to classify unknown malware samples into families. Both of the approaches have in common that they rely solely on the API function call name by omitting parameters and return values. Since our proposed detection approach can also be used to classify malware, the objectives of the related research in the classification domain are somehow similar to our research. However, the same distinction is applicable. Our classification approach is not fully automated but relies on the expertise of domain experts. As malware is quite stable on the API call level, the signature definition can be handled manually, in particular, if we take into account that we allow domain experts to share signatures collaboratively. Furthermore, in distinction to the related research in the field of malware classification, we take complete API calls including all parameters and the return value into account.

The detection of malware can not only happen based on API call sequences alone but also on higher-level information derived from these sequences. For instance, Lanzi et al. [63] and Lu et al. [72] introduce the detection of malware based on the usage of resources deduced from system call sequences. By tracing resource handles of processes and subsequent activities based on these handles, their proposed systems are enabled to distinguish malicious from benign behaviour. We adopt the basic idea of extracting the usage of resources from function logs. We extend this approach in several ways. On the one hand, we consider API calls in addition to system calls. Hence, our resource extraction aims at producing more comprehensive results since function calls that happen only in the userland are considered. Similar to Lu et al., we also extract network resources and therefore extend the approach of Lanzi et al. Moreover, the access operations defined in our approach are more fine-granular than initially proposed by Lanzi et al. In terms of the overall detection approach, our detection is not solely based on these extracted resources but leverages this information in addition to API call sequences.

In terms of API call sequence matching, related research if found in [23, 58]. Both of the publications use the multiple sequence matching (MSA) algorithm in order to find common API call sequences among API calls traces of related malware samples. The MSA algorithm can be understood as a more advanced version of the LCS algorithm used for the detection algorithm proposed in this thesis since the MSA algorithm is able to find similarities between multiple sequences instead of finding the longest common subsequence. In addition, the publication of Ki, Kim and Kim in [58] introduce the concept of *"critical API call sequence pattern"* [58, p. 5] for known malicious activities. This concept is equal to our definition of generic malware features proposed in this thesis. Unfortunately, the authors do not introduce a suitable language to define theses patterns. We fill this gap by introducing such DSL. Again, the proposed system is fully automated.

After we have given a broad overview of related publications in terms of the automated detection and classification of malware, we will now focus on research closest to our proposed detection approach. The Malware Instruction Set (MIST) introduced by Trinius et al. in [158] aims at converting system call traces originating from CWSandbox [175] into a binary sandbox-agnostic representation called MIST. MIST is therefore a similar approach to the generic function log format outlined in this thesis. While the binary format of MIST achieves promising results in terms of the storage and processing efficiency, it seems to be impractical for our particular use case where domain experts are involved in the overall detection process. In terms of the traceability of detection results, the MIST format is inefficient as it is not directly understandable by humans. In contrast to our proposed detection approach which is based on the actual API calls, MIST uses a more generic and abstracted form of API calls as basis. These abstracted API function calls are reminiscent of the generic function log format used by the VMRay Analyzer sandbox (please refer to Section 2.6.3 for more information on the generic function log format). Furthermore, the motivation of the MIST format is to enhance the efficiency of data mining and machine learning techniques which is

contrary to our motivation.

The publication on malicious behaviour pattern proposed by Dornhackl et al. in [28] is similar to the detection process proposed in this thesis. By introducing the concept of *"malicious behavior schemas"* [28, p. 1] the authors aim to deduce high-level goals of malware based on a multi-tier taxonomy in combination with a scoring system. The "malicious behavior schemas" rely on API call sequences (task patterns in terms of the publication) that are manually defined by domain experts. Unfortunately, the authors do not present the exact way in which these task patterns are defined practically in their publication. Furthermore, the main objective of the proposed system seems to be the classification of malware in context of the taxonomy rather than providing an extendable framework with that domain experts are enabled to define their own signatures.

A further system close to our detection approach is the Knowledge-Assisted Visual Malware Analysis System (KAMAS) introduced by Wagner et al. in [172, 173]. The system proposes a visual approach to malware analysis based on API call sequences and is related to the publication of Dornhackl et al. described beforehand. This somehow gives indications on answering the question how the task patterns are practically defined. Our assumption that KAMAS is focused on classifying malware rather than enabling domain experts to define their own rule set is sustained in the publication of Wagner et al. Although the authors mention the sharing of rules in the abstract, they do not present a practical way of sharing the defined rules which are stored in a database within the publication. Since our proposed DSL is defined in simple text-based signature files, sharing this is unproblematic. The publication of Wagner et al. is rather a design study focusing on the usability of the proposed system instead of on the actual detection capabilities. Therefore, the publication lacks an evaluation in terms of the actual detection quality. This gap is also not closed by the related publication of Dornhackl et al. as the evaluation only takes two malware samples into account and, unfortunately, does not state how the detection results were verified. This evaluation gap for the signature-based detection of malicious behaviour is closed by our thesis.

In conclusion, the VMRay Threat Identifier (VTI) engine which is part of the VMRay Analyzer sandbox offers the opportunity to incorporate so called custom VTI rules that can be defined by domain experts into the detection process. Although custom VTI rules are specific to the VMRay Analyzer sandbox and thus not sandbox-agnostic, this feature follows the same objective as our proposed detection approach. Therefore, we compared custom VTI rules with our proposed DSL in our evaluation chapter (Section 5.3) in detail. For detailed information on the VTI engine itself, please refer to Section 2.6.3 in the background chapter of this thesis.

1.5 Outline

After we have motivated our research, formed the research question addressed in the present thesis, outlined the objectives, briefly presented the results and differentiated our research domain from related work in this chapter, the remainder of the thesis is structured as follows.

In the following **Chapter 2** we introduce the basic terminology of malware analysis, signature-based detection and domain specific languages in general. Moreover, we outline fundamentals of the Windows operating system required to understand the concept of our detection approach. A detailed description of the basics of the VMRay Analyzer sandbox helps readers unfamiliar with the sandbox to understand the source of information that build the basis for our detection. Experienced readers that are familiar with the dynamic malware analysis domain in general as well as with the VMRay Analyzer sandbox in general can skip this chapter. However, for inexperienced readers it is recommended to consider this chapter as theoretic introduction needed to understand the remainder of this thesis.

Chapter 3 describes the key design concepts of our detection approach named *dynmx*. On the basis of an abstract overview of the overall detection process, the individual steps of this process are described in detail. Beginning from the function log parsing into a common data model and thus harmonised representation, the derivation of the access activity model is presented. The common

data model also builds the basis for the generic function log format. Afterwards the signature DSL is outlined in order to give the reader an intention of the features provided by the DSL. In addition the signature parsing process as well as the internal representation of signatures as directed acyclic graph is explained. The parsed function log and signature are the main inputs for the detection algorithm which is based on an adapted form of the LCS algorithm. The key concepts and mode of operation of this detection algorithm is described in the last section of this chapter.

Chapter 4 briefly outlines the prototype implementation of the detection approach. An overview of the code organisation as well as the implementation of the detection algorithm is the focus of the first part of this chapter. The second part presents the signature library developed in the course of this thesis.

This protoype implementation together with the signature library is evaluated thoroughly in **Chapter 5**. The evaluation is done in three dimensions: the assessment of the detection quality in terms of generic and specific malware features, the flexibility and accuracy offered by our signature DSL compared to VTI rules and the performance of our detection approach. The evaluation concludes with the discussion of our results.

The present thesis concludes with **Chapter 6** which summarises our research results and answers the research question asked in the introduction. An overview of the identified future work is found in **Chapter 7**.

2 BACKGROUND

The following sections describe the theoretical fundamentals of the signature-based detection approach of dynamic malware features presented in this thesis. The first part of the chapter defines the terminology of signature-based malware detection and dynamic malware analysis. Since the presented detection approach relies solely on application programming interface (API) calls, the second part of this chapter deals with key components of the API as well as operating system resources provided by the Microsoft Windows operating system. The source of information on that the detection process relies is subject of the third part of this chapter. General basics of the virtual machine introspection (VMI) technology used by the VMray Analyzer sandbox in order to extract API call information as well as specific features of the VMray Analyzer sandbox are described. The chapter concludes with a brief overview of key concepts of domain specific languages (DSL).

2.1 Malware

What probably started in 1988 with an academic experiment by the Cornell graduate Robert Tappan Morris unleashing the Morris worm on the Internet (see [31] and [111]) has evolved into the root cause of a wide variety of significant threats on today's Internet over the last three decades as proposed in [110], [173], [149] and [63]. *Malware* – the portmanteau of ***mal****icious soft****ware*** – is used by petty criminals, organised criminal entities as well as nation state threat actors with different motives, intentions and objectives. Besides the different motivations for developing malware, the form the malware is delivered to their victims varies also. While the standard Windows executable format (portable executable format, PE) is one of the popular formats for malware, macro based malware delivered as embedded scripts in office documents experienced a renaissance in the last few years [59, 124, 148]. Further formats malware can be delivered in are binary shell code and scripts such as JavaScript or Powershell. A typically more advanced malware format is firmware such as the malware "NotPetya" altering the boot loader of an infected system [147] or Stuxnet manipulating Siemens industrial programmable logic controllers (PLC) used by the Iran to enrich

uranium [127]. Therefore, the terms “malicious code”, “malicious script” and “malicious executable” are considered synonyms of malware in terms of this thesis.

Several definitions for the term malware can be found in publications of standardization institutes, of the scientific community or in reports presented by anti virus vendors. The National Institute of Standards and Technology (NIST) defines malware in their Special Publication 800-83 as follows.

> *“Malware, also known as malicious code, refers to a program that is covertly inserted into another program with the intent to destroy data, run destructive or intrusive programs, or otherwise compromise the confidentiality, integrity, or availability of the victim’s data, applications, or operating system.”* [149, p. 2]

As proposed by Or-Meir et al. in [110] the existing definitions of the term malware are very similar and cover at least the following two features (cited from [110, p. 88:5]):

- *“maliciousness or harmfulness”* and
- *“the ability to perform actions without the user’s consent or knowledge”*.

Further, existing definitions are formulated from the attackers’ perspective by relying on the fact that their intentions are known. Or-Meir et al. propose that the term malware is not properly defined by these existing definitions because the author’s intentions behind unknown code can not be determined. Consequently, Or-Meir et al. define malware from the system administrators’ perspective to emphasize the malicious aspects rather than the attackers’ intention.

> *“Malware is code running on a computerized system whose presence or behavior the system administrators are unaware of; were the system administrators aware of the code and its behavior, they would not permit it to run.*
>
> *Malware compromises the confidentiality, integrity or the availability of the system by exploiting existing vulnerabilities in a system or by creating new ones.”* [110, p. 88:6]

According to the authors, it can be assumed that code running on a particular system should be known and authorized by the system’s administrator. For the sake of completeness, it should be noted that this definition is only applicable under the assumption that the attacker is not the system administrator himself. The second part of the definition focuses on malicious behaviour compromising the CIA triad whereby CIA is derived from the initial letters of the three basic security principles confidentiality, integrity and availability [110]. These three basic security principles are assumed by users when working with computerized systems. In order to compromise the CIA triad a vulnerability is exploited allowing the attacker to use system resources for malicious activities. Based on the authors’ suggestion a vulnerability is defined as an error, bug or mistake in the attacked system. At this point the suggested definition needs to be extended to suit today’s attacks since the definition focuses solely on the system but not on the human factor. In today’s attack scenarios, especially when considering Advanced Persistent Threats (APTs) taking advantage of social engineering techniques, the human factor plays a central role in the exploitation of computer systems. An example of such a social engineering technique is the spear phishing delivery scenario. In this scenario the attacker sends a malicious resource like an attachment or a link to a victim using direct communication like an email. This communication needs to be crafted by the attacker so that it appears legitimate to the victim [130]. For this attack technique no vulnerability has to be present in the system itself since the user’s trust in the communication is exploited.

After defining the term malware, the taxonomy of malware is of interest. Based on the functionality, malware can be categorised into different types as supposed in [146] and [110]. It is noted that most of today’s malware can be assigned into multiple of the following historically determined categories. The categories are derived from [146] and [110]:

Backdoor Malware that installs and persists itself on an infected system to create a communication channel to the attacker. The attacker can connect to the infected system to control it without the user’s consent.

Bot A Bot (derived from ro*bot*) provides similar functionality like a backdoor by allowing the

attacker to access the infected system with the difference that bots receive their instructions from a central command and control (C2 or C&C) infrastructure operated by the attacker. The conglomerate consisting of several bots and the C2 infrastructure is referred to as *botnet*.

Cryptominer Malware that uses large amounts of the victim's computing resources to mine cryptocurrency for the attacker. Cryptominers are taking advantage of the anonymity provided by cryptocurrencies.

Downloader Malware that only loads and installs further malicious code. Downloader are often used in early phases of a multi stage infection process when attackers first gain access.

Information-stealing Malware Malware with the purpose to retrieve mostly confidential information from the infected system. The retrieved information is usually sent to the attacker. Examples falling into this category are keyloggers and sniffers.

Launcher (Loader) As the category name already suggests this type of malware launches malicious code on the infected system. In contrast to downloader, launcher use nontraditional techniques and exploit vulnerabilities to achieve better stealthiness or broader permissions. Further, launcher contain the malware that they are meant to execute.

Ransomware Malware that encrypts the user's personal files with strong and well known hybrid encryption methodologies using the combination of symmetric an asymmetric encryption algorithms [142]. Since the decryption key is unknown to the user, the data is effectively lost unless the user pays a ransom in order to get the decryption key. Similar to cryptominers, modern ransomware also takes advantage of the anonymity provided by cryptocurrencies.

Remote Access Trojan Malware masquerading as benign software pretending to be harmless and useful. For masquerading, Remote Access Trojans usually take advantage of social engineering techniques.

Rootkit Malware that is typically used to hide the existence of other malicious code running on an infected system causing the malware to be hard to detect from a user's perspective. Based on the privilege ring of the operating system the Rootkit is running in, this type of malware can be further distinguished into kernel (ring 0) and userland (ring 3) rootkits [137, 155].

Scareware While typically not harming the infected system, Scareware uses social engineering techniques to make the user believe that the system was infected by malware. Therefore Scareware usually masquerades as anti-virus software that has to be purchased in order to remove the non-existing malware from the system.

Spam-sending Malware Malware that uses resources from infected systems to send spam messages. Sometimes, this type of malware crawls address books on an infected system in order to send spam messages.

Spyware Malware used to collect (personal) information from infected systems without the consent of the user. Usually, the user's actions on the infected system are tracked and used to derive the personal information.

Worm Malware that is able to duplicate itself over the network in order to infect further systems. Without any countermeasures or rate limits implemented in the malware, worms can achieve exponentially infection rates.

Virus In distinction to a worm, this type of malware replicates on the infected system by injecting its malicious code into other typically benign files. The term virus is often used in colloquial speech as generalization for and therefore equally to the term malware.

Or-Meir et al. extend this traditional taxonomy in [110] by classifying malware additionally based on its malicious behaviour as well as by the privilege that the malware runs at in the infected operating system. Since the traditional taxonomy provided above is sufficient for understanding the remainder of this thesis, the extended taxonomy suggested by Or-Meir et al. is only referenced for further reading but will not be explained in detail.

In the last part of this section the terms *malware sample* and *malware family* are defined. In terms of this thesis a **malware sample** is considered as a single malicious file typically containing malicious code in unveiled or obfuscated form. With reference to the definition of Plohmann et al. in [122, p. 4], a **malware family** is defined as *"group of all malware samples that from a developer's point of view belong to the same project, i.e. code base"*. By implication, a malware sample can also be comprehended as single representative of a malware family [122].

2.2 Dynamic Malware Analysis

Basically, the term *malware analysis* is defined as the process of examining the functionality provided by a typically unknown and not human-readable malware sample. The person performing the malware analysis is referred to as *malware analyst* and *domain expert* in terms of this thesis. For attribution and classification purposes the origin of the malware sample can also be of interest in the course of the examination process. Based on [146] and [110], the main goals of malware analysis in the area of incident response are

- to determine the possible capabilities of the unknown malware sample,
- to improve the detection based on information from the malware analysis to identify infected systems as well as preventing the malware sample and similar variants from executing and
- to determine, contain and eradicate the impact of the malware infection.

Analysing malware for research purposes merely focuses on the improvement and evaluation of detection approaches as well as on the classification of malware samples based on specific characteristics (e.g. [60, 106]).

Malware analysis techniques can be assigned to two fundamentally different but complementary approaches – *static* and *dynamic malware analysis*. This classification was introduced by Ball independently of malware analysis in [9]. The distinctive feature between the static and dynamic approach is the question whether the malware sample is executed during the analysis. While static malware analysis techniques are characterized by dissecting the malware sample without executing it, the dynamic analysis approach takes advantage of executing and observing the malware sample in a controlled environment [110, 146]. Sikorski and Honig suggest in [146] to further distinguish between basic and advanced variants of these approaches. For the sake of completeness and to classify the dynamic analysis approach, these variants are described briefly. The descriptions below are based on [130, 146].

Basic Static Analysis The basic static analysis focuses mainly on gathering meta data like the cryptographic file hash or file properties from the malware sample. The analysis of the PE header of a Windows executable can also contribute valuable information in the static analysis, e.g. the calculation of the import hash [10]. Simple unstructured content data analysis like extracting strings contained in the sample are also of part of the basic static analysis. In distinction to the advanced static analysis the instructions of the malware sample, i.e. structured content data, are not dissected. Basic static analysis techniques can be automated and used by personnel without much domain knowledge.

Advanced Static Analysis In the advanced static analysis the malware sample is manually reverse-engineered by a malware analyst. Therefore, the sample is examined using a disassembler like IDA Pro[1]. The malware analyst needs to reconstruct the functionality on machine code level. Decompilers can assist the malware analyst by translating common machine code patterns into a high-level programming language.

Basic Dynamic Analysis The malware is executed in a controlled and monitored runtime environment while observing its behaviour in the basic dynamic analysis without compromising

[1] See `https://www.hex-rays.com/products/ida/index.shtml` (accessed on 16.01.2020) for further information on IDA Pro.

the security of the analysis platform. The malware is handled like a black box, thus the internal state of the malware during the execution is out of scope. In order to reduce the risk of damage while executing the malware the runtime environment has to be safe. *Malware sandboxes* provide safe environments by automating the dynamic analysis process. Therefore, the sandbox manages the set up process for the analysis environment, loads and executes the malware sample, automatically observes and analyses the behaviour and finally resets the analysis environment to a pre-defined state.

Advanced Dynamic Analysis Similar to the advanced static analysis the advanced dynamic analysis focuses on the internal state of the examined malware sample while executing it in a proper analysis environment. Typically the malware sample is manually analysed using a debugger like OllyDbg[2]. With the help of the debugger the malware analyst can step through the execution path of the malware sample while observing internal system states of CPU registers and memory regions.

Dynamic malware analysis provides – especially with the usage of sandboxes in the basic dynamic analysis approach – valuable runtime information (telemetry) of the investigated malware sample at relatively low effort. Hence, from the viewpoint of a malware analyst dynamic malware analysis is an efficient way of examining a given malware sample and getting helpful hints for a deeper dissection of the sample. Due to the automation implemented in a modern malware sandbox, the analysis process becomes standardised and repeatable allowing the usage by security analysts without deep knowledge in malware analysis [146]. Further, dynamic malware analysis is immune to common evasion techniques driving the complexity of the static representation of the malware sample, e.g. packers, obfuscation techniques or encryption layers [110]. Since the first an foremost aim of malware is to run on the victim's system the code has to be present in a form that is suitable for the targeted system architecture. Consequently, the malware has to unpack, deobfuscate and/or decrypt at least the part of the machine code that should be executed next during runtime.

But these benefits of dynamic malware analysis are only valid under certain assumptions that should be taken into account in the design and implementation of an analysis environment [110, 146]:

1. The analysis environment running the malware has to be safe in terms of avoiding unintentional harm to the system and network as well as avoiding manipulation of the data provided by the analysis environment. Therefore, the analysis environment must provide security measures isolating the executed malware sample in order to prevent it from taking over of control of the environment.
2. The malware sample must not be able to detect that it is executed in a monitored sandbox environment.
3. The analysis environment must collect and process as much runtime information as possible about actions taken by the malware sample.
4. The analysis environment has to meet the malware sample's expectations of the runtime environment. If the expectations are not met the malware might not expose its behaviour. Ideally, in an enterprise the analysis environment reflects the exact client configuration in order to be able to make valuable statements about the vulnerability of the enterprise regarding the malware sample under investigation.
5. The network access of the malware sample must be restricted by the analysis environment to prevent harm to the network.
6. The analysis environment has to provide a coherent and concise report allowing the malware analyst to efficiently classify the malware sample and to derive appropriate defensive measures.

Consequently, based on these six assumptions the dynamic analysis approach faces the following limitations [110].

1. Dynamic malware analysis can only cover executed code. Thus, branches in the execution path of the malware sample might not get analysed if certain conditions in the execution

[2] See `http://www.ollydbg.de/` (accessed on 17.01.2020) for further information on OllyDbg.

path of the malware are not met by the analysis environment. This is comparable to the branch coverage in software testing. If a test case does not meet the conditions of a certain conditional statement, the branch is not executed and not tested.

2. The scalability of dynamic analysis is limited since the malware sample has to be executed over an appropriate period of time in order to gather valuable telemetry. Further, the dynamic analysis environment typically requires computational overhead that can slow down the execution of the malware sample as well as the postprocessing.
3. The operating system and/or the hardware required by the malware has to be met or emulated adequately. For example, the Mirai variant targeting home routers in 2016 [53] relied, amongst others, on the PowerPC and MIPS (Microprocessor without interlocked pipeline stages) architecture.

In practice, a hybrid approach taking advantage of both static and dynamic malware analysis techniques is typically used in order to outweigh the mentioned limitations of the dynamic malware analysis.

2.3 Signature-Based Malware Detection

One of the main outcomes of the malware analysis process are *signatures* or also called *rules* [110, 130, 139, 146]. In terms of this thesis, a signature is a description of certain characteristics of examined malware samples in a pre-defined syntax. Signatures can be applied to typically unknown malware samples using detection algorithms. This process is referred to as *signature based malware detection*. The aim of the signature based malware detection is to simplify and streamline the detection of already known malware characteristics in a repeatable manner.

In contrast to a signature, an *indicator of compromise* (IOC) does not have to follow a pre-defined and common syntax. An IOC is *"a description of technical characteristics that identify a known threat, an attacker's methodology, or other evidence of compromise"* [130, p. 65]. This description can be in structured or unstructured form. Hence, IOCs can be expressed as a signature and the concrete detection of a signature can lead to IOCs.

In the context of malware detection *host-* and *network-based signatures* exist [146]. This classification needs to be extended by a third type called *malware-based signatures* to provide a consistent and sufficient background for the detection approach described in this thesis.

Host-based signatures focus on detecting malicious code on a possibly compromised system based artifacts resident on the system. This type of signature typically describes changes on a compromised system caused by the malware including newly created or altered files, changes in the system's configuration or characteristic patterns in the system's memory.

Network-based signatures are used to detect malware infections by focusing on the network traffic. Typically, modern malware communicates to C2 infrastructure over the network. Network-based signatures describe characteristics of this network communication that can be used in perimeter systems like intrusion detection (IDS) or intrusion prevention systems (IPS) to detect known malware. A well-known representative of network-based signatures are Snort IDS rules [130, 157].

Malware-based signatures describe characteristics of the malware itself. For example, antivirus software heavily relies on malware-based signatures in order to detect malicious code on a system. Usually, malware-based signatures focus on static aspects of malware samples. The de-facto industry standard for defining and sharing malware-based signatures is YARA [4, 183]. The signature languages used by antivirus vendors are typically proprietary and therefore not public. An exception is the open source antivirus software ClamAV that can also be used to define custom malware-based signatures [24, 67].

Since the detection approach described in this thesis relies on malware-based signatures, these

will be examined in more detail in the following. As mentioned before, YARA is the open source industry standard for defining and sharing malware-based signatures. Due to open source detection libraries, YARA is supported in many security products that have detection capabilities [1, 130]. In the terms of the YARA project signatures are referred to as rules [1, 130, 146]. YARA rules are used to describe exclusively static characteristics *"helping malware researchers to identify and classify malware samples"* [1]. Rules are defined in text-form by combining strings and logical expressions defining how the rule has to be evaluated by the YARA detection engine. Listing 2.1 presents a sample YARA rule taken from the open source collection "Yara-Rules" [75].

```
rule Upackv032BetaDwing
{
    meta:
        author="malware-lu"

    strings:
        $a0 = { BE 88 01 ?? ?? AD 50 ?? ?? AD 91 F3 A5 }
        $a1 = { BE 88 01 ?? ?? AD 50 ?? AD 91 ?? F3 A5 }

    condition:
        $a0 or $a1
}
```

Listing 2.1: Sample YARA rule to detect a certain packer (copied from [76])

As seen in the sample YARA rule, patterns can not only be defined as strings but also as binary series. Regular expressions, wildcards, case sensitivity and more language features allow the malware analyst to describe complex static characteristics of malware samples. Typically, the static characteristics are defined independently from the file format as unstructured strings or byte series. With the usage of YARA rules malware analysts are able to find similar malware samples belonging to the same family allowing them to classify malware samples. In an incident response scenario, malware analysts examine identified malware samples, describe the static characteristics in a proper YARA rule in order to hunt for similar malware samples in well-known malware repositories. One example for a well-known, established and comprehensive malware repository is VirusTotal allowing a malware analyst to hunt for malware based on self-defined YARA rules [164].

Derived from the advantages of the dynamic malware analysis described in Section 2.2, the usage of malware-based signatures as described above has limitations. First of all, this detection approach relies solely on static characteristics of the malware sample. Since packers, encryption layers and further obfuscation methodologies commonly used by malware authors can alter the static appearance of malware samples fundamentally, detection may fail to occur. Hence, the challenge for malware analysts is on the one hand to identify robust binary series and strings that characterise the examined malware sample sufficiently while on the other hand not choosing a too general definition. If the malware analyst chooses a too general definition this can lead to false positive detections. Of course, the signature based malware detection in general can only be used to identify known characteristics for that a signature is existent.

2.4 Microsoft Windows Fundamentals

A basic knowledge of Microsoft Windows fundamentals is needed in order to understand the detection approach developed in this thesis. Hence, this section covers basic terminology and concepts of the Windows API as well as how Microsoft Windows handles resources.

2.4.1 Microsoft Windows API

The Microsoft Windows application programming interface called *WinAPI* specifies common programming interfaces used by applications in order to interact with and to use core functionalities

provided by the Windows operating system family. The WinAPI is implemented in the unprivileged user-mode (ring 3) [137] and consists of various API versions like the *Win32 API* (interface to the 32-bit versions of the Windows operating system) and the *Win64 API* (interface to 64-bit versions of the Windows operating system). Following the suggestion of Russinovich et al. in [137], the term WinAPI covers both the 32-bit and 64-bit versions of the Windows API in terms of this thesis.

The WinAPI provides a comprehensive set of functionality to developers allowing them to create Windows applications without the need of incorporating third-party libraries [146, 155]. Hence, malware authors extensively make use of the WinAPI since they can assume that most of the functionality needed for their malicious activities is already present on the attacked system. Functionality provided by the WinAPI used by malware includes file system operations, manipulation of the Windows registry[3], process and thread management, interprocess communication and of course network operations [146]. The functionality provided by the WinAPI can be roughly classified in the following categories [137]:

- Base Services
- Common Services
- User Interface Services
- Graphics and Multimedia Services
- Messaging and Collaboration
- Networking
- Web Services

From a technical view, the WinAPI is provided as several *dynamically linked libraries* (DLLs) that export functions together with header files defining the declaration of the functions. These functions can be imported and used by developers in their applications. DLLs are binary files in the PE file format that can not be executed directly but include executable code. The usage of WinAPI functions is documented in the Microsoft Dev Center [86].

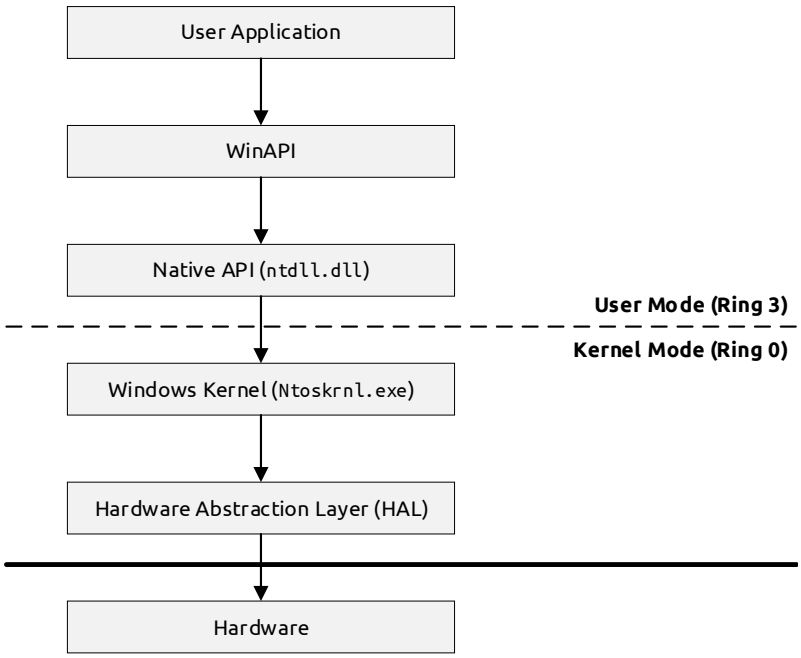

Figure 2.1: Microsoft Windows API abstraction layers (following [137, 146]

[3]The Windows registry is the central configuration database in the Windows operating system holding settings and options that influence the behaviour of the operating system and applications [137, 146].

Assuming the interaction with the Windows operating system is divided into different abstraction layers as shown in Figure 2.1 the WinAPI is the highest layer of abstraction, i.e. the WinAPI is the closest layer to the user application [146]. In the abstraction layer right below the WinAPI resides the so called *Native API*. The Native API is an internal low-level interface whose sole purpose is to provide the WinAPI with controlled access to routines running in the Windows kernel [134]. This separation is needed since most of the relevant data structures are just available in the kernel mode preventing the access by user mode applications. Therefore, user mode applications need a well-defined interface to call kernel routines in order to provide certain functionality like writing a file. Applications that only call functions provided by the Native API but no functions provided by the WinAPI are referred to as *Native Applications* [135, 146].

The Native API is provided by the DLL `ntdll.dll` and is mostly undocumented. As estimated by Russinovich in [135] only 10% of the 250 functions exported by the Native API were documented in the Windows NT Device Driver Kit in 2006. As of 2012, the `ntdll.dll` exports over 400 functions [137]. Unofficial and incomplete documentation of the Native API exists in [104, 107]. Functions provided by the Native API are present in the *System Service Dispatch Table* (*SSDT*) alongside with kernel routines and can be examined using memory forensics [68] to get a list of functions available in the Native API. Another way would be to examine the exports of *ntdll.dll* with a PE file parser. It should be noted that these methodologies only lead to function names but not to the complete declaration of the functions. The usage of Native API functions is popular among malware authors since there is some functionality that is only provided by the Native API and is therefore not available to developers using the WinAPI. Further, the Native API can be used to circumvent security products that only monitor calls to functions provided by the WinAPI [146].

Based on the abstraction layer of the Windows API different terms are used for the provided functions. The following list describes the terminology suggested by Russinovich et al. in [137] that is also used in this thesis.

WinAPI functions Officially documented functions provided by the WinAPI that can be called by user mode applications.

Native system services (system calls) Mostly undocumented, internal functions provided by the Native API. System calls are callable from user mode.

Kernel support functions (kernel routines) Mostly undocumented, internal functions used in the kernel of Microsoft Windows. Kernel routines are only callable from the kernel mode. Kernel routines are typically used by device drivers running the Windows kernel.

WinAPI Functions

The Listing 2.2 shows a typical WinAPI function declaration of the `CreateFileA` function [80] describing the return type, the function name and the function arguments with their type.

```
1 HANDLE CreateFileA(
2   LPCSTR                lpFileName,
3   DWORD                 dwDesiredAccess,
4   DWORD                 dwShareMode,
5   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
6   DWORD                 dwCreationDisposition,
7   DWORD                 dwFlagsAndAttributes,
8   HANDLE                hTemplateFile
9 );
```

Listing 2.2: Declaration of the `CreateFileA` function documented in [80]

As seen in the listing, the function argument names have a prefix describing the expected data type. This notation is referred to as the *Hungarian notation*[4] which is generally used for WinAPI identifiers [140, 146]. Common types and their prefixes are listed in Table 2.1. The table is based on [146].

[4]The name of the convention originates from the inventor Charles Simonyi who was born in Hungary [140].

Type	Prefix	Description
WORD	w	16-bit unsigned integer
DWORD (double WORD)	dw	32-bit unsigned integer
BOOL	b	Boolean (TRUE or FALSE)
Handle	h	Reference to an object where an object is defined as a runtime-instance of a pre-defined object type [137].
Long Pointer	lp	Pointer to another type. Referring to the example provided in Listing 2.2 the argument lpFileName provides a pointer to a character string and lpSecurityAttributes provides a pointer to the type SECURITY_ATTRIBUTES.
Callback	No specific prefix	Pointer to a function that will be called from the WinAPI function.

Table 2.1: Common WinAPI types and their prefixes

WinAPI function names can have several suffixes that provide information about supported character encodings and whether the function is overloaded. Table 2.2 provides an overview of the available suffixes and their meaning. The table is based on information provided in [137, 141]. Following the description in Table 2.2 the WinAPI function CreateFileA only supports ANSI character encoding. For Unicode support the function CreateFileW has to be used. The suffixes can also be combined like in the function name GetVersionExA.

Suffix	Long Form	Description
A	ANSI	Supports only ANSI character encoding
W	Wide	Supports Unicode character encoding
Ex	Extended	Overloaded version of an existing function providing typically more parameters to control the function execution more precisely

Table 2.2: Common WinAPI function name suffixes

From an implementation point of view, the subset of WinAPI functions that provide access to kernel routines can be seen as wrappers for their Native API system call equivalents. For example, the WinAPI function CreateFileA is a wrapper for the system call NtCreateFile [155]. This abstraction was introduced by Microsoft in order to be able to change internal routines of the operating system without affecting the interface used by applications.

System Calls

As described before, system calls provide access to functionality residing in the kernel of the operating system over a defined low-level interface. Therefore, a system call typically executes the following process to service kernel level functionality requested by user applications [68, 155]:

1. Save the user mode context.
2. Pass parameters to the kernel following a predefined calling convention.
3. Change the execution privilege to kernel mode.
4. Service requested kernel functionality by calling the suitable kernel routine. The kernel routine terminates by restoring the saved user mode context and changing the execution privilege back to user mode.
5. Provide the execution results back to the user application.

In the Windows Native API, system calls are implemented as stubs (*system service dispatch stub*). Stub means in this context, that the system call itself does not provide the requested kernel functionality but initiates the change to the kernel mode in order to call the needed kernel routines. Therefore, a specific software interrupt is initiated by the system call using the CPU instruction `SYSENTER` or `SYSCALL` [134, 137]. Figure 2.2 illustrates the process of writing to a file based on the different abstraction layers of the Windows API. The system service dispatcher (`KiSystemService` routine) handles the software interrupt by copying the system call arguments to the kernel-mode stack and then executing the requested kernel routine.

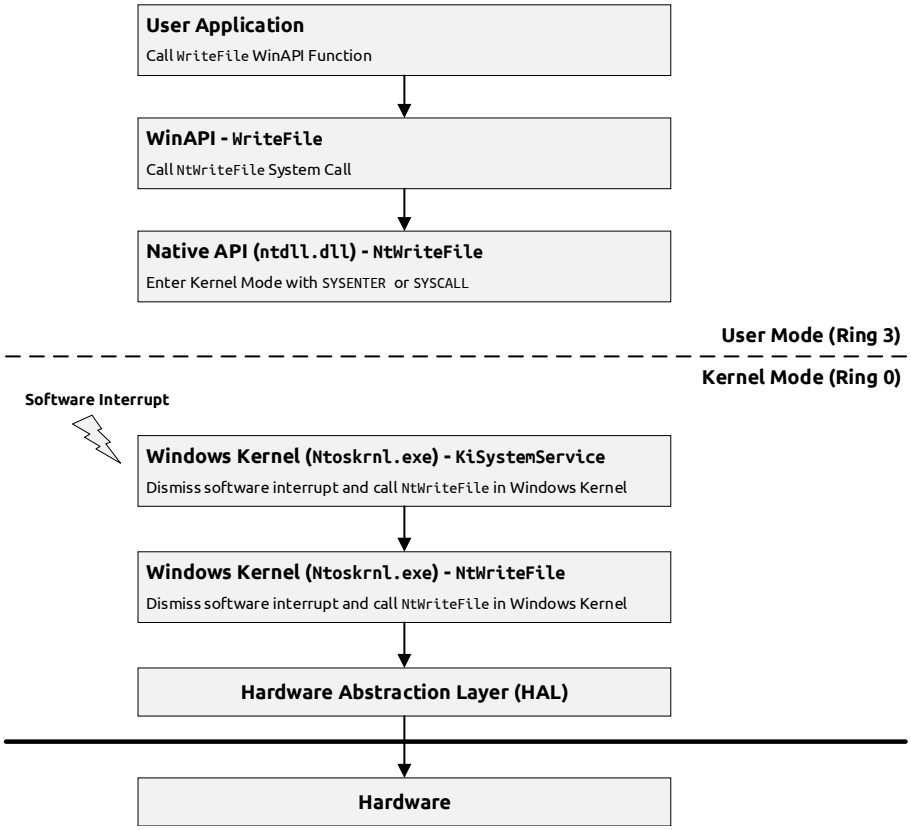

Figure 2.2: Process of writing a file with the Windows API (following [137, 155])

Since the `NtWriteFile` system call is officially documented for the driver development the declaration is publicly accessible in [79]. Listing 2.3 shows the declaration.

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtWriteFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,
  PULONG           Key
);
```

Listing 2.3: Declaration of the `NtWriteFile` system call documented in [79]

As seen in the listing, the declaration is not following the Hungarian notation because the argument names are not prefixed with a type identifier. But system call function names are prefixed with a

category identifier. Relevant for system calls are the prefixes `Nt` and `Zw`. Typically, `Nt` and `Zw` versions of the same system call exist (e.g. `NtCreateFile` and `ZwCreateFile`) since system calls can also be used by kernel mode drivers. So, these drivers indicate that the arguments are originating from a trusted kernel mode source and therefore don't need to be validated by calling the `Zw` version. The `Nt` version of system calls are only supposed to be called by user mode applications using the Native API [49, 137]. In addition to the prefixes `Nt` and `Zw`, further common prefixes exist for system calls as well as kernel routines. A comprehensive list of common prefixes is denoted in [137, p. 66-67].

2.4.2 Resources

One of the main tasks of modern operating systems is to centrally manage access to resources. In terms of this thesis, a *resource* is any kind of limited system component that can be acquired exclusively by some running process. A *process* in turn is an executed instance of a *program* which is a statically defined instruction sequence [137, 155]. Tanenbaum defines a resource simply as *"anything that must be acquired, used, and released over the course of time"* [155, p. 436]. This includes both physical (e.g. memory, CPU, hard disk) and virtual system components like files, database records or registry keys. Processes are associated with all relevant information needed for execution. Hence, a process can also be referred to as container for resources that are indispensable for the execution of a certain program [137].

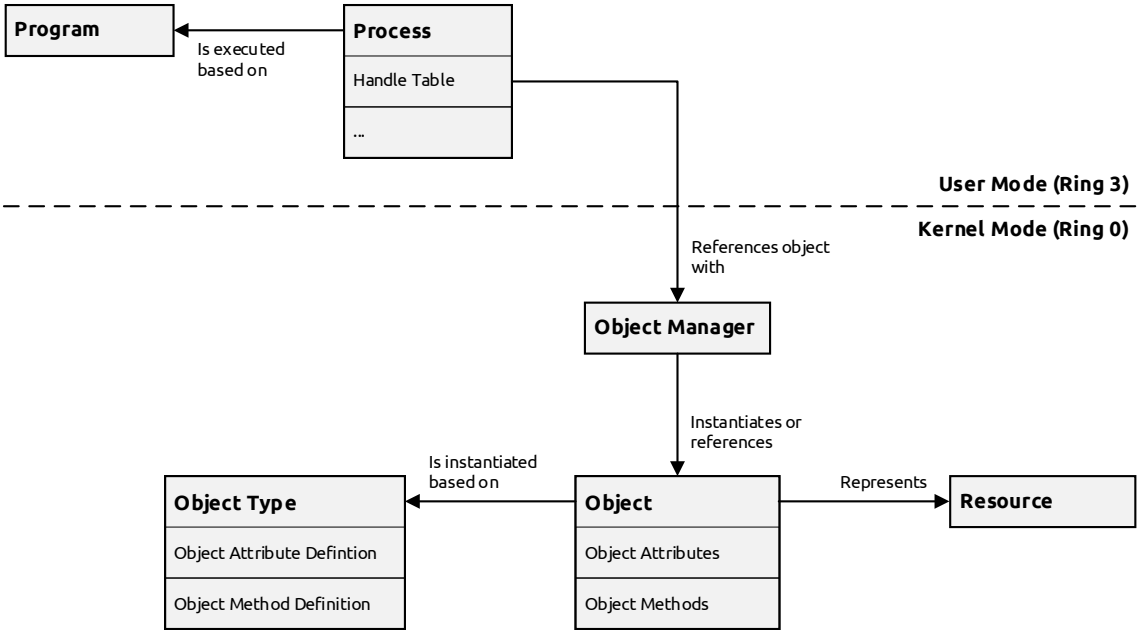

Figure 2.3: Objects in Microsoft Windows

Within the Microsoft Windows operating system, resources are typically represented by so called *kernel objects* or simply *objects*. For instance, a device object represents a physical hardware device and describes how this device is connected to the system [155]. As the name already suggests, objects are only present in the kernel memory address space. Objects are instantiated based on a static object type definition that comprises of

1. *Object attributes* describing the object's state and
2. *Object methods* defining the way an object is manipulated.

In contrast to ordinary data structures, the internal data structure of an object is hidden and can only be manipulated using the defined object methods. The Windows kernel component (object manager) is solely responsible for managing objects and therefore provides a convenient way for

interacting with resources [137]. Objects include, besides specific attributes, an object header providing generic meta information like the object's name, the directory in which the object lives and security information.

Every time an object is requested, i.e. created or opened, by a user-mode process a reference to that object called *handle* is returned. The concept of a handle can be fundamentally compared to the pointer concept with the restrictions that a handle does not represent the object's actual memory address and it can not be used within arithmetic operations [146]. The process that requested the object uses the handle in the subsequent execution flow in order to access the resource represented by the object. The handle concept provides a generic and consistent interface for user-mode processes to refer to kernel mode objects independent of the referenced object type. In other words, a handle to an event, a file or even a complete window or process has the same generic structure regardless of their type. Since the object manager is the central component for managing objects, only the object manager can create handles and locate the actual object based on existing handles. An exception to this rule are kernel routines which are able to access objects directly without needing to use handles [137].

Every process in the Windows operating system maintains its own process-specific *handle table* that contains every opened handle to objects used by the corresponding process. As soon as the process has finished manipulating the object, it closes the handle causing the corresponding entry in the handle table to be deleted. By using pointer indirection techniques in the implementation, the handle table can hold up to 16,777,216 handles [137]. Figure 2.4 illustrates the handle table organisation used in Microsoft Windows. As seen in the figure, Windows uses three-levels of pointer indirection. Hence, each entry in the top- and middle-level table is interpreted as pointer that references another middle-level or subhandle table. Only the entries in the subhandle table reference actual objects in the kernel.

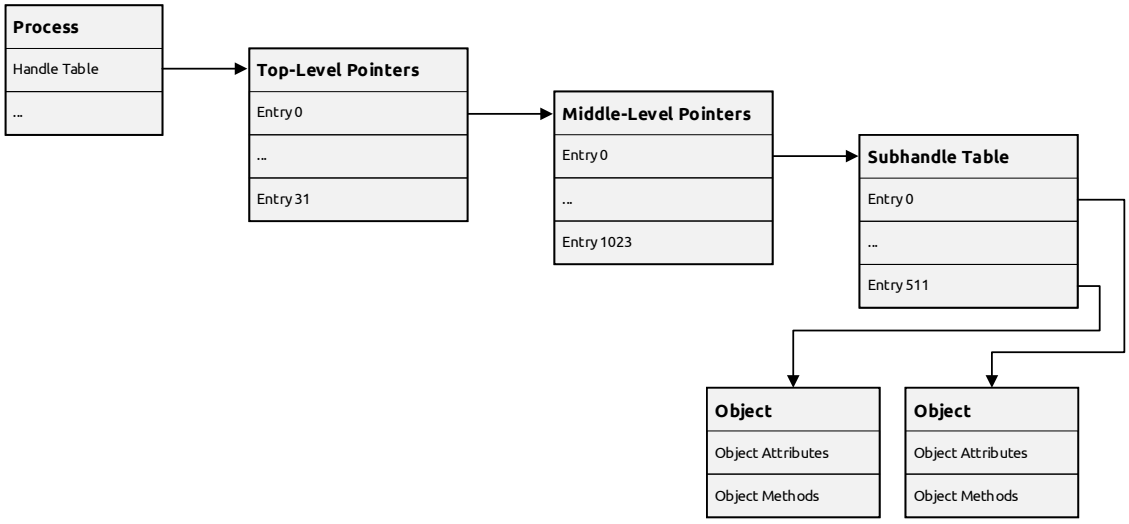

Figure 2.4: Microsoft Windows handle table architecture (based on [68, 134, 155])

Handles are not only used for referencing objects but also for enforcing access control. Since dereferencing a handle from user mode can only be accomplished with the help of the object manager, this kernel component is also responsible for protecting objects. Therefore, the object manager calls the security reference monitor (SRM) kernel component which decides based on a desired access level and a defined access mask whether the access to the object is granted.

2.5 Virtual Machine Introspection (VMI)

Virtual machine introspection (*VMI*) is an approach to inspect a virtual machine (VM) from the isolated outside in order to analyse and control the internal state of the VM [41, 66, 176]. Therefore, VMI relies on virtualisation technology which is widely used in the IT sector to run several, isolated operating systems within VMs on a shared underlying hardware. The VMI approach was introduced by Garfinkel and Rosenblum in [41] who presented an architecture for a host-based IDS leveraging VMI in 2003. Since then, the VMI approach has been applied in different domains of IT security such as intrusion detection [41], forensics [150, 181] and especially dynamic malware analysis [66, 115, 120, 156, 176].

The VMI functionality is typically implemented with the help of or directly in the *hypervisor* which is also referred to as *virtual machine monitor* (*VMM*) [155]. In terms of this thesis, the terms hypervisor and VMM are considered synonymous. The VMM is typically software that provides an abstraction layer to the underlying hardware by resembling and multiplexing its functionality to virtual machines. Consequently, multiple virtual machines can be operated isolated from each other on the same shared hardware resources. Virtual machines are also known as *guests* while the hardware system on that the VMM runs is referred to as *host*. Traditionally, VMMs can be classified in two distinctions based on the layer the VMM operates at [155].

Type 1 VMMs operate directly on the hardware of the host system. The VMM has to implement a thin operating system allowing to manage the hardware resources. Example implementations for type 1 VMMs are VMware vSphere Hypervisor[5], XEN[6] and Proxmox VE[7].

Type 2 VMMs presuppose an operating system on the host. The VMM is running as another application on the host operating system thus allowing it to make use of the already existing OS functionality. Example implementations of type 2 VMMs are Oracle VirtualBox[8] and VMware Workstation[9].

Figures 2.5 and 2.6 illustrate the location of the VMI component in the context of both VMM types. The figures are extended based on [155].

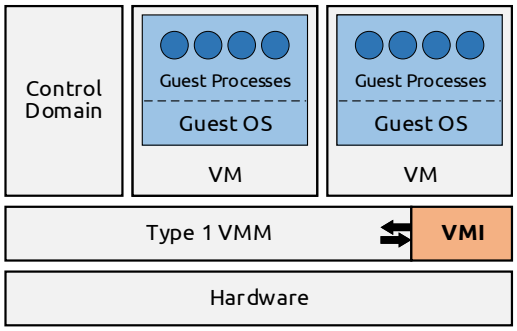

Figure 2.5: VMI component in a type 1 VMM

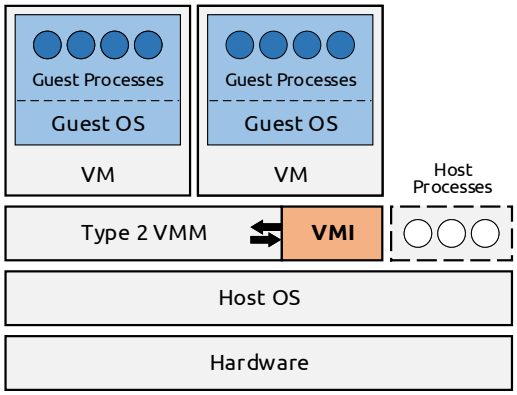

Figure 2.6: VMI component in a type 2 VMM

Since the VMM resembles all underlying hardware resources and multiplexes their functionality it basically provides the ability to examine the hardware state of a VM [41]. The hardware state is referred to as the sum of information stored by the system and its operating system at the lowest level, the hardware layer. Hence, as suggested in [118, 119] the hardware state consists for example out of the information stored in

[5] https://www.vmware.com/products/vsphere-hypervisor.html
[6] https://xenproject.org/
[7] https://www.proxmox.com/en/proxmox-ve
[8] https://www.virtualbox.org/
[9] https://www.vmware.com/products/workstation-pro.html

- CPU registers,
- volatile memory,
- persistent storage and
- virtual BIOS settings.

This hardware state information provided by the VMM is consulted in the VMI approach in order to deduce the internal state of a VM including the upper layers like the guest OS or the applications running in the guest OS. For most analysis techniques based on VMI the virtualised memory is of most interest since it represents the internal state of the VM entirely [66, 120, 176]. However, in order to retrieve meaningful information from the virtualised hardware and especially from the virtualised memory semantic knowledge about the guest's hardware architecture and typically also about the guest operating system is indispensable. For example, the VMI component needs to interpret the kernel memory space of a Windows guest properly in order to locate kernel data structures describing the currently running processes, their open handles to kernel objects or shared DLL functionality used by a running process. This problem is also known as the *semantic gap problem* introduced by Chen and Noble in [21]. For instance, CXPInspector bridges the semantic gap by presuming that the virtualised CPU has the x86 architecture and that the guest operating system is Windows 7 64-bit which allows CXPInspector to interpret the guest memory accordingly [176]. Based on the formal VMI model introduced by Pfoh, Schneider and Eckert in [118] this approach is referred to as out-of-band delivery pattern. A community-driven approach to bridge the semantic gap is represented by LibVMI – a C library designed to ease the usage of VMI by supporting multiple VMMs (Xen, KVM, QEMU) and guest operating systems (Windows and Linux) [113]. LibVMI abstracts from the different VMMs and guest OS and thus provides a uniform interface allowing reading and writing the guest memory, access to CPU registers as well as pausing and unpausing the guest.

One of the main objectives of VMI based dynamic malware analysis is focused on the examination of the control flow within the monitored VM [66, 120, 176]. Only by closely monitoring and analysing the control flow a substantiated statement about the malware's behaviour can be made. Especially calls to WinAPI functions and system calls are of interest as already motivated in Section 2.4.1. Different approaches where suggested in the scientific community to address the analysis of the control flow of a monitored VM leveraging VMI. While CXPInspector analyses transitions between memory regions in the guest memory to detect function calls [176], DRAKVUF takes mainly advantage of software breakpoints injected in the guest memory to examine the control flow [66]. In order to provide detailed information to single steps in the control flow like the name of the called function, the calling process, the provided arguments and the return value, both approaches need to obtain significant data structures of the guest operating system. Therefore, both approaches first need to locate the kernel in the guest memory. As the kernel is found in the guest memory, both approaches use publicly available debugging symbols to locate and traverse significant data structures as well as kernel routines. For instance, information about the running processes (doubly linked list containing `_EPROCESS` objects [68]) and their allocated memory regions stored in the virtual address descriptors (VADs) [68] can now be extracted from the guest's kernel memory space. In addition to DRAKVUF, the CXPInspector approach provides detailed information of WinAPI function calls by parsing the WinAPI function declarations from the header files which are part of Windows SDK. Please consider that this is a brief high-level presentation of the introspection methodologies used by the two approaches. For details, please refer to the publications in [66, 176].

That the VMI approach has been widely adopted in the IT security sector and especially in the dynamic malware analysis is primarily based on several beneficial features [176], for instance:

Transparency From the inside perspective of the monitored guest and thus from the perspective of the malware under investigation, the monitoring and analysis is imperceptible since it is performed outside of the guest. Placing the VMI component into the VMM allows it to run at a higher privilege level than the kernel of the guest OS.

Isolation Isolation prevents the malware under investigation from manipulating the analysis

platform. As described in Section 2.2, isolation is a compulsory feature that assures trustworthy analysis results.

Soundness Theoretically, the control flow of the examined malware can be analysed in any granularity and at any layer of abstraction since the hardware state is known in the VMM. However, practically, a trade-off between a sufficient level of detail and an appropriate execution performance has to be found since the VMI approach comes along with a degradation of performance.

Further benefits of the VMI approach are mainly inherited from the advantages of using virtualisation technology. For example, reverting the analysis environment can easily be accomplished by taking advantage of VM snapshots.

In contrast to the benefits, the VMI approach faces limitations with regard to the application in dynamic malware analysis. As discussed before, the approach comes along with a performance degradation which can lead to successful side-channel attacks, namely timing attacks, allowing the malware to detect the analysis platform [15, 27, 102, 174]. For instance, the VM needs to be suspended using the `VMEXIT` instruction several times during the analysis process in order to take control over to the VMM allowing the VMI component to examine the hardware state. These suspensions can be profiled from inside the monitored guest. A further limitation of the VMI approach is inherited by using virtualisation technology and is referred to as VM detection. Although this limitation has already been addressed in research sandbox vendors still invest effort to encounter VM as well as sandbox detection capabilities of malware. Regardless, it is questionable whether malware should prevent its execution in virtualised environments since they are heavily used in the industry [176].

2.6 VMRay Analyzer as Representative for a VMI Sandbox

Since the source information used in the detection approach introduced in this thesis originates from the *VMRay Analyzer* sandbox this section provides a brief introduction to the product in general as well as to the functionality relevant for understanding the detection approach. Since the presented detection approach is mostly agnostic toward the sandbox that provides the source information, VMRay Analyzer should only be considered as representative of a VMI sandbox. Although the presented methodologies and concepts used by VMRay Analyzer are not necessarily applicable to other VMI sandboxes, the analysis result and especially the ability to log API function calls are comparable among the available VMI sandboxes.

VMRay Analyzer is a commercial malware sandbox solution offered by the german company VMRay GmbH based in Bochum. VMRay GmbH was founded by the former PhD students of the Ruhr-University Bochum Dr. Carsten Willems and Dr. Ralf Hund in 2013 [167]. The sandbox solution can be deployed on premise or in the cloud [168]. Although the overall malware analysis process performed by the VMRay Analyzer sandbox generally consists of three consecutive steps (Figure 2.7), we will focus solely on the dynamic analysis step since this step provides the relevant information for the detection approach described in this thesis.

The technology used in the VMRay Analyzer sandbox for behavioural malware analysis evolved mainly based on the scientific prototype CXPInspector suggested in [176]. Thus, the sandbox solution leverages VMI in order to automatically analyse the behaviour of examined malware from the underlying VMM as described in Section 2.5. VMRay refers to their analysis approach taking advantage of the VMM in conjunction with VMI for dynamic malware analysis as the *"3rd generation threat analysis and detection"* [166, p. 6]. The former two generations (hooking and emulation) face severe limitations that are addressed by VMRay Analyzer representing the third generation, for instance [168]:

- Insufficient detection of evasion techniques,
- Missing of relevant information due to limited monitoring approach and

- Imprecise and noisy analysis results.

The general architecture of the VMRay Analyzer sandbox is basically classified as a Type 2 VMM. Hence, the VMI component (titled as "Sandbox Monitor" in the context of VMRay Analyzer [166]) is implemented in the VMM which in turn is executed as a host process in the host operating system as illustrated in Figure 2.6. Referring to the VMM, VMRay Analyzer relies on Kernel-based Virtual Machine (KVM) in combination with QEMU [165]. While KVM provides hardware based virtualisation taking advantage of the Intel VT and AMD-V technology, QEMU is responsible for emulating all other hardware resources that are not covered by hardware virtualisation like the network card [176]. Consequently, VMRay Analyzer needs to be run on Linux (Ubuntu) as the host operating system.

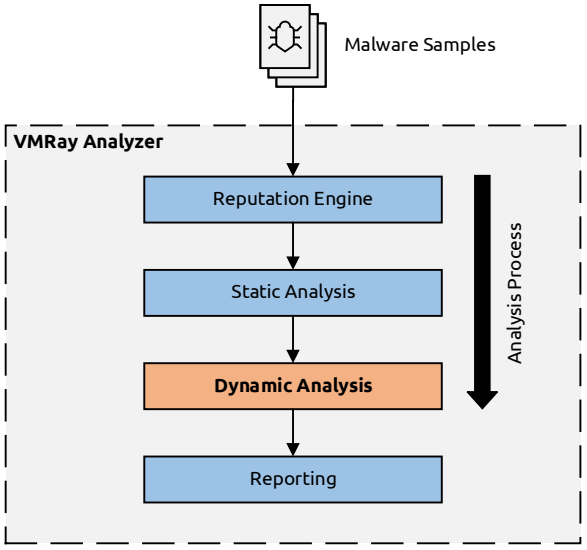

Figure 2.7: VMRay Analyzer sandbox analysis process (based on [169])

The analysis results are present as comprehensive report in the web-based VMRay Analyzer user interface[10]. The report generally consists of several sections providing information on behavioural observations, the VTI score (see Section 2.6.3), network connections, file operations, YARA rule matches, IoCs and the analysis environment. In addition to the report, an analysis archive is provided. This archive can be downloaded from the web-based report in order to process the included data with external tools. The analysis archive basically includes the raw data of the analysis like the network trace as packet capture (PCAP), the function log (see Section 2.6.2), process memory dumps and extracted files.

2.6.1 Control Flow Deduction

The following section is based on [176] unless otherwise stated. Since CXPInspector is the predecessor of VMRay Analyzer, it is assumed that the basic design and concept of the commercial VMRay Analyzer sandbox corresponds in general to the design of CXPInspector. This assumption can also be deduced by reading the technology whitepaper [166] and data sheet [169] of the VMRay Analyzer sandbox since the described technology correlates with the approach described in the CXPInspector publication.

As briefly introduced in Section 2.5, CXPInspector closely monitors transitions between memory regions in the guest memory in order to deduce the control flow. A memory region is thereby defined as *"a chunk of virtual memory that has been allocated as one single contiguous block"* [176,

[10] Please refer to `https://www.vmray.com/analyses/f7d2c4199f08/report/overview.html` for an example report.

p. 9] in terms of the CXPInspector approach. In order to identify memory regions in the guest memory, CXPInspector takes advantage of the *Two-Dimensional Paging* (*TDP*) or also called *Second Level Address Translation* (*SLAT*) feature[11]. Basically, TDP can be thought of as a hardware-assisted virtualised memory management unit (MMU) responsible for translating physical memory addresses of the guest to physical memory addresses of the host. Consequently, TDP ensures the isolation of guest machines on the memory layer, i.e., physical guest addresses are indeed virtual addresses translated by the TDP mechanism to real physical memory addresses residing in the host memory. This translation process is illustrated in Figure 2.8. For translating guest physical addresses to host physical addresses the VMM maintains TDP tables. The concept and architecture of TDP tables is mostly equivalent to traditional page tables used in the non-virtualised memory translation process. CXPInspector takes over the management of TDP tables from the VMM and is therefore able to identify the distinct physical memory of a monitored guest. Based on the guest physical memory. CXPInspector leverages the guest paging structures (GPS) to translate guest physical addresses to guest virtual addresses thus allowing CXPInspector to reconstruct guest virtual memory belonging to the guest's user mode processes.

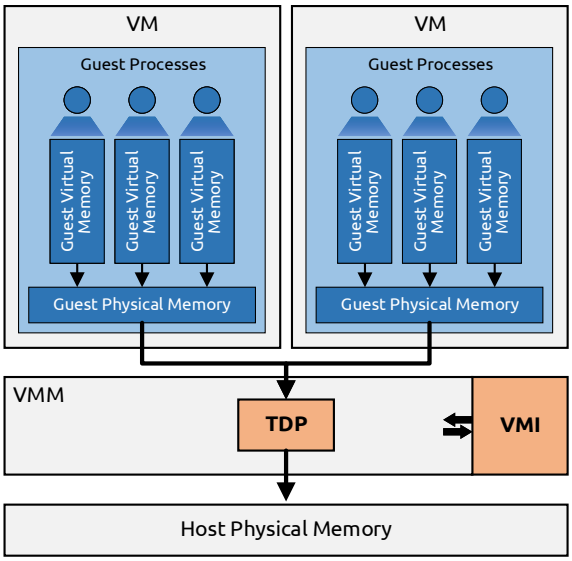

Figure 2.8: TDP translation process

After identifying the memory regions in the guest memory, CXPInspector divides the memory into two distinct partitions – the executable and the non-executable partition. These two partitions are not only virtual constructs inside of CXPInspector but are also represented in the guest memory since CXPInspector flags the actual memory regions according to their current execution status. The executable partition, also referred to as *currently eXecutable pages* (*CXP*), always reflects the memory region that is currently executed in the guest and is therefore flagged as executable. The currently executed memory region is derived from the instruction pointer (EIP register). By leveraging VMI and considering all relevant memory translation layers, CXPInspector is able to relate the CXP to a module of a running process in the guest. Every time this currently executed process module calls a function residing in another memory region outside of the CXP (for example, by using a system call) a transition from the CXP to the non-executable partition happens. Along with the transition, a `VMExit` trap is generated handing over the control to the VMM and thus to CXPInspector which is, in consequence, able to detect and log the function call. The `VMExit` trap is triggered since the target memory region of the function call should be executed but is flagged as non-executable. This causes an exception that must be handled by the VMM and therefore a `VMExit` trap occurs. Before handing

[11] While TDP or SLAT is the implementation agnostic term for the feature the specific implementation is referred to as Extended Page Table (EPT) for Intel based systems and Nested Page Table (NPT) respectively Rapid Virutalization Indexing (RVI) for AMD based systems.

over the control back to the VM using the `VMEnter` instruction, CXPInspector marks the memory region from which the call originated from as non-executable and the target memory region the function call points to as executable. Hence, the memory region the function call points to becomes the new CXP and can be executed since CXPInspector flagged this memory region as executable. Consequently, the next transition from the CXP to another region or even back to the memory region the function call originated from can be detected by CXPInspector using the exact same process. The sum of these memory region transitions represents the intercepted control flow within the guest. The control flow deduction used in the CXPInspector approach is illustrated in Figure 2.9. In the illustrated example, the guest process `a.exe` is currently executed in the guest and calls the WinAPI function `WriteFile`. The CXP scope is limited to the `a.exe` memory region. The `WriteFile` function is exported by `kernel32.dll` and therefore resides in the non-executable partition. By calling the `WriteFile` function a `VMExit` trap is raised since the memory region is not executable. Hence, the control is handed over to the VMM and in consequence to CXPInspector, which is now able to log the `WriteFile` WinAPI function call, changing the memory regions' executable permissions in the TDP tables and handing over the control back to the VM by using `VMEnter`.

The selected size of the CXP directly correlates with the granularity in which the control flow can be deduced. The larger the size of the CXP is selected, the more coarsely the monitored control flow of the guest becomes. For example, by selecting the whole virtual memory of a guest process (program executable, custom libraries and user mode OS libraries) as CXP, only system calls and thus transitions between user and kernel mode can be intercepted by CXPInspector. This CXP size selection would lead to an incomplete picture of the examined malware. In contrast, by selecting a small memory region, e.g. only the program executable of the process, as CXP all function calls to internal libraries as well as to operating system libraries can be monitored. While this behaviour is desirable for profiling applications, it is counter-productive for dynamic malware analysis. First of all, it can lead to noisy analysis results due to too many details and further comes along with a significant performance degradation caused by too many `VMExit` traps. In consequence, CXPInspector needs to carefully select an appropriate CXP size in order to provide relevant and meaningful control flow information. VMRay Analyzer refers to this adjustment of the CXP size as *"adaptive monitoring"* [166, p. 5].

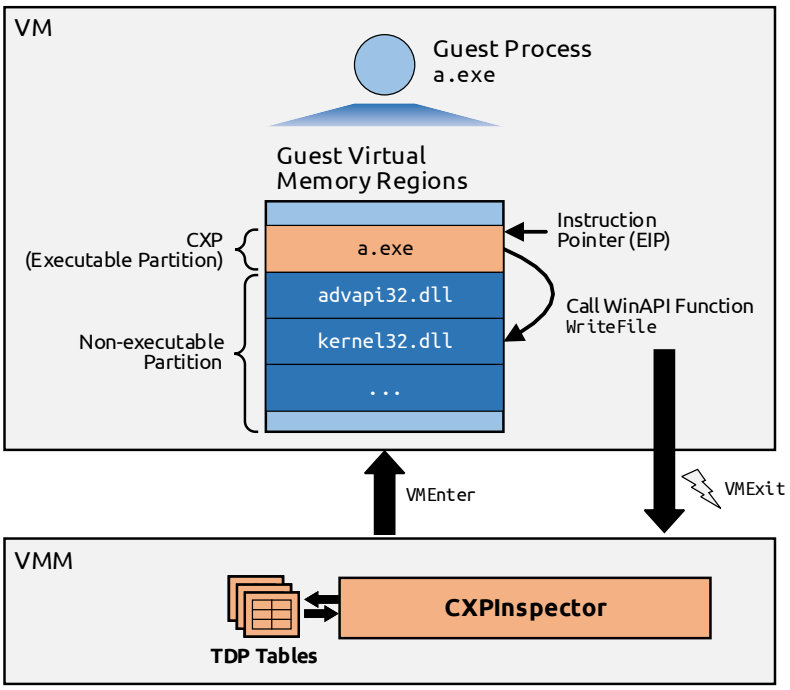

Figure 2.9: CXPInspector control flow deduction

2.6.2 Function Log

The result of the control flow deduction as described in the previous section is basically represented by the *function log* provided by the VMRay Analyzer sandbox. In terms of this thesis, the function log is defined as chronological sequence of functions called by a certain monitored process. Thus, the function calls are logged in relation to the process that caused the corresponding calls allowing a malware analyst to reconstruct the behaviour of a certain monitored process. It is noted, that the term function log, as used in this thesis, is always tied to the VMRay Analyzer product. However, function logs or also referred to as *function call traces* can also be generated by other sandbox solutions, e.g. by the PyRebox malware monitor module [116], by the Nitro approach introduced in [120], by the DECAF approach presented in [45], by DRAKVUF [66] or by Cuckoo [152]. Figure 2.10 illustrates the general structure of a text-based VMRay Analyzer function log. The indentation in the illustration indicates semantic relations between the different information categories or also referred to as entities. For example, allocated memory regions are related to the process information since the memory regions have been allocated by a certain process. The indentation can also be interpreted as hierarchy in that the information is semantically structured. Although these semantic relations are not necessarily explicitly represented in the function log, they result from the file structure.

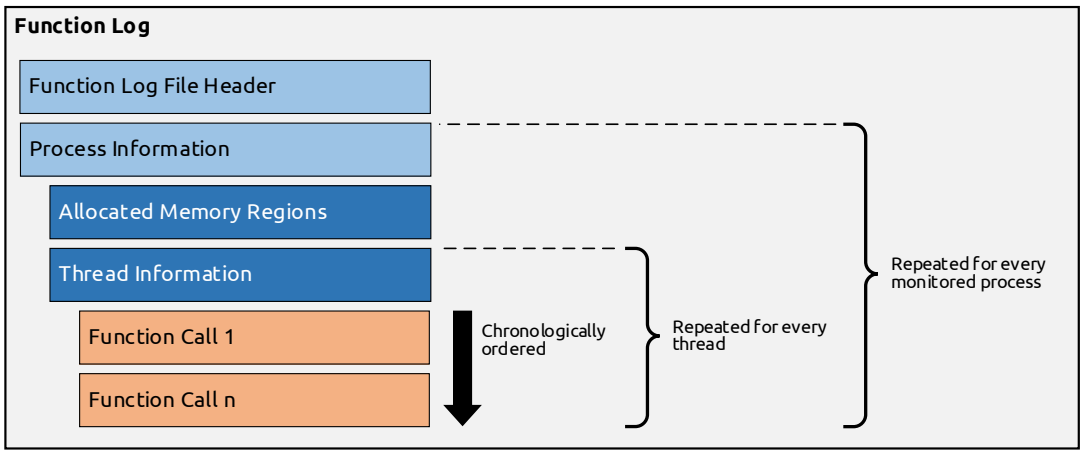

Figure 2.10: Semantic structure of a function log provided by VMRay Analyzer

Table 2.3 lists the detailed pieces of information contained in the function log categories illustrated in 2.10. Please consider that this table is not complete, but lists the important information relevant for the detection approach introduced in this thesis.

In addition to the semantic information provided by the function log, the file format is of interest, especially if the function log is processed automatically. The function log is provided in two file formats by the VMRay Analyzer sandbox. Appendix A.2 provides sample function logs in the text and XML file format. Further, more comprehensive function logs can be found in [170] and in the digital appendix on the enclosed flash drive[12].

Text format The function log is presented in an unstructured text-based format. Unstructured in this context means, that the function log is optimized for manual examination by a malware analyst rather than automated parsing since it does not follow a common markup language. Hence, the different information categories are basically distinguishable in the function log but the content is not directly machine readable. The semantic relations between the categories are not explicitly represented in the function log but can be deduced based on the file structure.

XML format This format uses the eXtensible Markup Language (XML) to structure the function

[12]The sample function logs are found in the directory `Digial_Appendix/2_Background/Sample_Function_Logs/VMRay/` on the enclosed flash drive.

log. This structure is directly machine readable and therefore predestined for automated parsing. However, this format is not suitable for manual analysis. The XML-based function log explicitly represents semantic relations between the information categories.

Category	Provided Information
Function log header	• Function log version • VMRay Analyzer version • VMRay Analyzer build date • Function log creation date
Process information	• Process ID • Executable path • Command that was used to start the exectuable • Memory address of the process
Thread information	• Thread ID
Allocated memory regions	• Start and end memory addresses of the memory region • Mapped files like DLLs • Entry points of mapped executable files
Function call information	• Timestamp • Function name • Function parameters and types • Return value and type

Table 2.3: Information provided by the VMRay Analyzer function Log

As shown in Listing 2.4, comprehensive information about a logged function call is provided in the function log. Not only the function name, parameter values and the return value are provided but in addition data types, parameter names and parameter directions. Since part of the information provided in the function log is not present in the stack frame when a function is called and therefore is not part of the guest memory, VMRay Analyzer needs to leverage external sources. For instance, the following information is present in the guest memory:

- Function name
- Parameter values
- Return value

The predecessor CXPInspector extracts the documented WinAPI function declarations from header files included in the Windows SDK [176]. Since VMRay Analyzer is closed-source, it can only be assumed that similar or the same sources for additional information on function calls are used. Nevertheless, it should be pointed out that the information provided in the function log is not solely based on the guest memory but is enriched by external sources.

```
1 [0050.439] GetSystemDirectoryA (in: lpBuffer=0x43e338, uSize↩
      =0x15 | out: lpBuffer="C:\\Windows\\system32") returned 0↩
      x13
2 [0050.439] GetProcAddress (hModule=0x762b0000, lpProcName="↩
      SaferIdentifyLevel") returned 0x762d2102
3 [0050.439] NtOpenKey (KeyHandle=0x12f128, DesiredAccess=0↩
      x20019, ObjectAttributes=0x12ee04*(Length=0x18, ↩
      RootDirectory=0x0, ObjectName="\\Registry\\Machine\\↩
      Software\\Policies\\Microsoft\\Windows\\Safer\\↩
```

```
LevelObjects\0", Attributes=0x40, SecurityDescriptor=0x0,↩
 SecurityQualityOfService=0x0))
```

Listing 2.4: Sample function calls logged in a text-based VMRay Analyzer function log

2.6.3 VMRay Threat Identifier (VTI) Engine

together with the function log, VMRay Analyzer provides the *VMRay Threat Identifier Score* (*VTI score*) as part of the analysis result (see Figure 2.11). The VTI score aims to express the severity and overall maliciousness of an analysed sample by an integer value in the range [0,100], where 0 means no detected malicious behaviour and 100 represents the highest severity possible indicating that the analysed sample is most probably malicious [160]. The VTI engine is the component responsible for calculating the VTI score by detecting *VTI rules* in the monitored and logged behaviour of an analysed malware sample. For each VTI rule a severity (also referred to as VTI rule score) is defined that contributes to the overall VTI score of the sample. The VTI engine distinguished between five severity scores in the range [1,5]. Table 2.4 presents the severity categorisation as introduced by VMRay Analyzer. The table is based on [168]. The severity scores can be defined for each malware sample class individually. The sample classes defined by VMRay Analyzer are [160]:

- Documents (PDF, MS Office)
- Scripts (JScript, VBScript)
- Browsers (URLs)
- Default (Samples that do not belong to the other classes, mainly PE samples)

The algorithm used to calculate the overall VTI score based on the individual scores of detected VTI rules is proprietary.

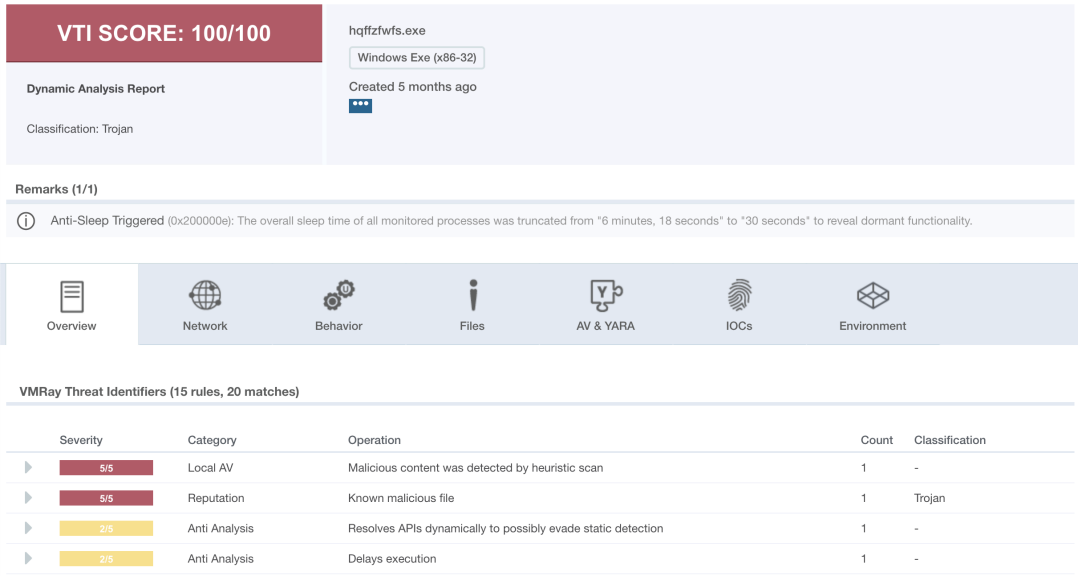

Figure 2.11: VTI score in the VMRay Analyzer analysis result

VTI rules define malicious behaviour based on API function calls originating from the monitored processes. VMRay introduced an abstraction layer in order to generalize specific API function calls providing similar functionality. Therefore, the concept of *generic function calls* (*gfncalls*) is used by VMRay Analyzer. The abstraction is achieved by mapping several specific API functions, e.g. the WinAPI function `CreateProcessA` and the native API system call *NtCreateProcess*, to

a single generic function call, e.g. `gfncall_create_process`[168]. Hence, VTI rules focus on actual functionality rather than specifics of OS APIs. In addition, the abstraction layer enables VMRay to define behaviour independent of the OS from which the API calls originate [160]. This generalisation has also been applied to OS resources like processes, files or registry keys as well as to the operations that can be used to manipulate resources. The abstracted form of the specific OS resources is referred to as *generic system object* (*gobject*) [168].

VTI rules are generally structured in a hierarchy consisting of three components which allow a proper rule classification:

Category The category defines the general domain to that the behaviour defined in the rule belongs to.

Operation The operation abstracts from specific techniques by describing a generic behaviour.

Technique A technique represents the actual implementation of a generic operation.

For example, specific techniques of malicious code injection can be classified as follows based on the presented hierarchy.

- **Category**: Injection
- **Operation**: Hooking of function tables in the kernel memory
- **Techniques**: Import Address Table (IAT) Hooking [146], SSDT Hooking [68]

The full set of available categories are listed in [168, p. 148-149]. By using the concept of finite state machines in the detection process, techniques defined in the VTI rule can be chained sequentially.

VMRay Analyzer is shipped with a curated pre-defined set of VTI rules. This pre-defined set can be extended by custom VTI rules written by malware analysts. VTI rules are defined as Python classes within the VMRay Analyzer framework. For the sake of minimizing the effort to write a custom VTI rule, VMRay provides a Python template in [168, p. 150] that defines the basic structure of the rule.

Severity Score	Description	Probability of Malicious Behaviour
1	Common behaviour mostly used by benign software but is also used by malware to prepare malicious behaviour (e.g. HTTP network connection to a remote host).	>0 % to <10 %
2	Uncommon behaviour mostly used by benign software but is also used by malware to prepare malicious behaviour (e.g. reading of sensitive web browser data).	>=10% to <50%
3	Behaviour that can be categorised as malicious but might also be used by benign software (e.g. monitoring of keyboard input).	>=50% to <75%
4	Behaviour that can be categorised as malicious but might also be used rarely by benign software (e.g. reading of credentials).	>=75% to <95%
5	Malicious behaviour that is nearly never used by benign software (e.g. injection of code into a running benign processes).	>=95% to 100%

Table 2.4: Severity score categorisation used by VMRay Analyzer

2.7 Domain-Specific Languages

The following section is based on [171] unless otherwise stated. In contrast to general purpose programming languages (GPLs) like C, C++ or Java, a *domain-specific language* (*DSL*) focuses on solving a specific kind of problems, the so called *domain*. For example, the simple query language (SQL) is a DSL representative that has been engineered with focus on relational databases (domain) and therefore provides the exact functionality needed to interact properly with this class of database. While a GPL is by definition Turing complete [159] and can therefore be used to solve any kind of problem (that can be solved by Turing machine), a DSL abstracts from this generality at the expense of loosing Turing completeness in most cases[13] but for the benefit of gaining specificity for effectively solving problems in a given domain. Language means in this context always a formal rather than a natural language. Formal languages are characterised by the fact that they are machine readable and consists of two components [46]:

1. the syntax defining the valid elements of the language itself (e.g. tags in HTML) and
2. the semantics which are not part of the language itself but add meaning each valid element of the language (e.g. the text between HTML tags).

Generally DSLs can be classified into two distinct types.

Internal (embedded) DSL An internal DSL is embedded in a general purpose language (host language) and is thus limited to the syntax provided by the host language.

External DSL External DSLs are not tied to a general purpose language and therefore need an specific *execution engine*. External DSLs define their own particular syntax independent of general purpose languages.

Referring to the execution engine, different approaches exist to run a DSL on a target platform. On the one hand, an *interpreter* can be used to parse and load the DSL properly in order to run actions defined by the DSL directly on the target platform. Another approach is the usage of a specific *compiler* (*generator*) that translates the DSL into GPL code that, in consequence, is executed on the target platform. As shown in Figure 2.12, the execution engine is the indispensable link between the DSL which is not directly executable and the target platform.

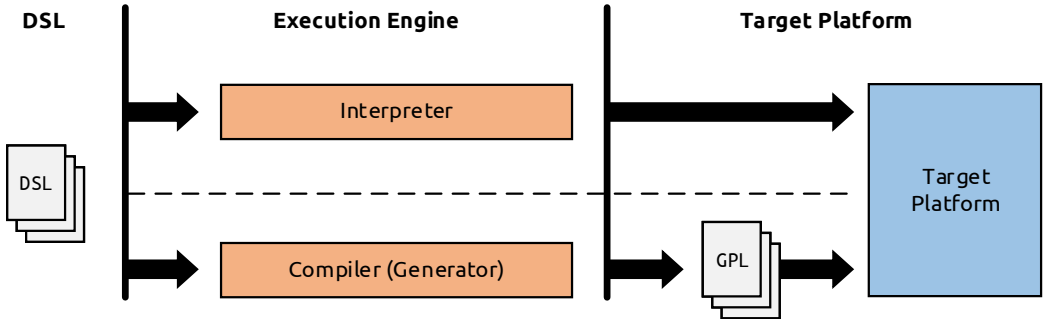

Figure 2.12: DSL execution engine

Instead of designing a complete new language including its syntax from scratch, DSLs can be encoded using a *general data representation syntax* [40, 46]. Well-known representatives for these general data representation syntaxes are INI, JavaScript Object Notation (JSON), XML and Yet Another Markup Language (YAML). DSLs based on general data representation syntaxes are classified as external DSLs [40] but with the limitation that the syntax has to follow the terms of the general data representation syntax. Typically, this approach is used for *configuration languages*, e.g. the configuration language for the Cuckoo sandbox uses the general representation syntax INI

[13] An exception in this context is PostScript which is by definition a DSL developed to describe pages but is Turing complete and can therefore theoretically used to solve more general problems [103].

[153]. Configuration languages are declarative which means, in terms of Voelter, that *"they provide linguistic abstractions for relevant domain concepts that allow processors to 'understand' the domain semantics without sophisticated analysis of the code."* [171, p. 70-71].

2.8 Summary

In this chapter, we introduced basic terms and definitions in order to establish a common understanding of the research domain. For instance, basic terminology of malware analysis in general and dynamic malware analysis together with signature-based detection in particular was defined. Since our detection approach is based on Windows API call sequences, we outlined certain fundamentals of the Windows operating system needed to understand the upcoming design concept. For instance, we explained basics of the Windows API and the handling of resources within the operating system. The third part of this chapter introduced the concept of VMI for malware sandboxes. To understand the source of information used in this thesis, we outlined the functionality implemented by the VMRay Analyzer sandbox and explained how function logs are created by the sandbox. The last part of this chapter outlines basic terminology of domain-specific languages.

The main part of this thesis begins with the upcoming chapter. We present the design concept of our proposed detection approach in detail and explain the decisions taken in the course of this design process. The chapter begins with a high-level overview of the detection process in general. This overview serves as orientation of the chapter's structure.

3 Design

After we have introduced basic terminology and general technology concepts on that this thesis relies, we present our main contribution which is the signature-based detection approach *dynmx* [daɪ'næmɪks] in the following. dynmx thereby comprises the definition of a DSL which allows a domain expert to define characteristic malware behaviour, the definition of a generic function log format as well as the DSL execution engine that implements the detection algorithm. Although we present this approach for detecting malware features based on Windows API calls, the general approach can also be applied to other operating systems that provide an API for applications.

In contrast to custom VTI rules (see Section 2.6.3) that generally follow the same objective, our approach uses the function log as source of information instead of the generic function log. Since the generic function log abstracts from the actual API call usage, malicious behaviour can be defined more precisely with our proposed approach. Moreover, our approach is designed sandbox-agnostic and is therefore not tied to a specific sandbox in general. The signature DSL proposed in this chapter does not require programming knowledge and is designed to be concise and easy-to-learn.

The structure of this chapter follows the overall dynmx detection process as described in Section 3.1. Beginning from the parsing and processing of the input function logs, our generic function log format is presented. This generic function log format is sandbox-agnostic and therefore emphasises the generality of the dynmx approach towards the sandbox that provides the function logs. The second part of this chapter deals with the derivation of an access activity model which provides an abstract representation of the resources used by a certain process. Afterwards, we focus on the definition of dynmx signatures. Therefore, we describe the concept and features of the DSL in detail. The presentation of the detection algorithm concepts concludes this chapter.

3.1 dynmx Detection Process Overview

The general idea behind the dynmx detection approach relies on the assumption that certain malicious behaviour, i.e. *malware features*, can be identified using characteristic API function call

sequences as proposed in [19, 106, 173] together with the usage of operating system resources [61, 63, 101]. Behavioural malware features in terms of this thesis are high-level actions deduced from API function call sequences and are categorised as follows.

Generic malware features are malware features that are not tied to a specific malware species or family and are thus used across a wide variety of malware types. These features are for example persistence mechanisms, malicious file operations, injection methods or network communication. Moreover, generic malware features must not be malicious by definition but can also include suspicious or even benign behaviour.

Specific malware features are malware features that are closely related to specific malware families. Hence, these kind of malware features can be used to characterise a certain malware family on the API call level. For example, the Emotet malware family creates a mutex on an infected system in a certain specific way.

Therefore, we present a signature-based and sandbox-agnostic approach for detecting known behavioural malware features using a DSL. The introduction of a DSL for defining the signatures ensures the extendibility by supporting domain experts at defining observed malicious behaviour. Rather than automatically clarifying the question whether a certain sample is benign or malicious the presented approach aims to help malware analysts to assess the functionality of a sample by detecting already known malicious or at least suspicious behaviour. In the end, the evaluation whether a sample is malicious still behoves the malware analyst. The proposed detection approach is divided into several, partly consecutive steps that are described in the remainder of this section.

Before the dynmx detection process itself is initiated, the malware sample needs to be analysed using a malware sandbox that traces and logs API function calls. Hence, dynmx does not provide any sandbox functionality in order to analyse the behaviour of a malware sample directly. Instead, dynmx detects characteristic malicious behaviour based on raw meta data gathered by established malware sandboxes and extends the detection capabilities of existing sandboxes. As mentioned in Section 2.6.2 on function logs, multiple sandbox solutions offer the functionality to log API function calls but in this thesis the VMRay Analyzer sandbox is presumed for providing the function logs which are used as the input data for the detection process.

The detection process itself consists out of four steps which are briefly described in the following. Each step is further described in an own section in the remainder of this chapter which is referenced in the corresponding brief description of the step in this section. The complete detection process is illustrated in Figure 3.1.

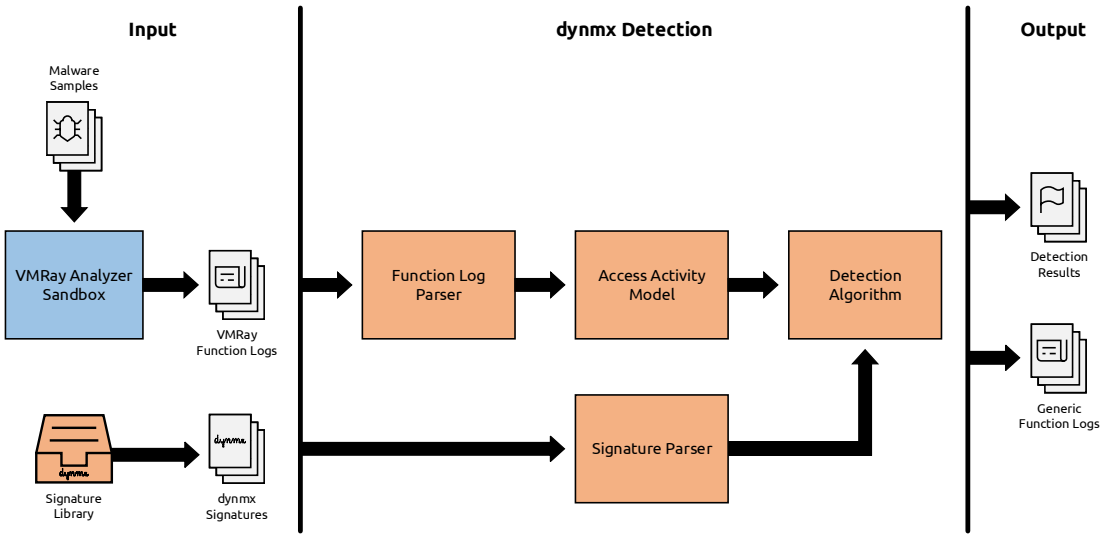

Figure 3.1: dynmx detection process overview

Function logs provided by the VMRay Analyzer are the starting point of the detection process. The file-based function logs are parsed at first hand (see Section 3.2 for the detailed description). During the parsing process, the relevant attributes from text-based API function call information are extracted and brought into a common, harmonised and structured data model (please refer to Section 3.2.3). This common data model is essential for the subsequent processing steps, especially for the detection algorithm, since it abstracts from the sandbox specific function log format. Therefore, the only component that needs to be designed sandbox-specific is the function log parser. The common data model generally follows the semantic structure as proposed in Figure 2.10. Hence, the data model maintains

- the relationship between API function calls and the processes they were called from and
- the chronological order in that the API function calls were called by the process.

After the raw function log data is parsed into the common data model proposed in this thesis (see Section 3.2.3), the derivation of the access activity model presented by Lanzi et al. in [63] can take place (see Section 3.4 for the detailed description). In summary, the access activity model represents the usage of file and Registry resources by a process. This resource usage is deduced based on the chronological order in that specific API function calls were called by a process. In addition to the resource itself, the model tracks how, i.e. with which access permission, the resource is used by a process. However, it is not the requested access level that is tracked by the model, but instead the actual access level the resource is used with by the process.

The resource usage together with characteristic API function call sequences representing malicious behaviour is defined by domain experts in dynmx signatures which are one of the main elements in the overall detection process. The signatures are used to manually define malicious behaviour in a sandbox-agnostic way leveraging a text-based DSL that is introduced in the course of this thesis (refer to Section 3.5 for a detailed description). Hence, the actual intelligence of the detection process is found in the signatures allowing the detection algorithm to be generic. As shown in Figure 3.1, dynmx signatures are the second input of the detection process. The domain experts manually defines known malicious behaviour in these signatures based on analysis results. Furthermore, the domain expert needs to specify what signatures should be detected in which set of function logs. The defined signatures are organised in an extendible signature library by domain experts. As the signature library represents the implementation of signatures leveraging our proposed DSL, it will be explained in the implementation chapter (see Section 4.2). In order to detect the malicious behaviour defined in the signature, it needs to be parsed and processed in the signature parser.

The parsed and processed signatures together with the parsed function log and the derived access activity model build the input of the main step of the detection process – the detection algorithm (see Section 3.6). Generally, the algorithm tries to detect the characteristic malicious behaviour defined in the signature in the function log and the access activity model deduced from the function log. We designed the algorithm process-centric which means that the processes are handled independently in the course of the detection. Therefore, malware leveraging inter-process communication (IPC) [155] to implement malicious behaviour cannot be detected using the dynmx detection approach.

First and foremost, the output of the detection process is the detection result indicating the malicious behaviour that has been detected in the function logs provided as input. Based on the desired level of detail the result just indicates whether the signature has been detected in the lowest level of detail. In the highest level of detail the exact sequence of API function calls as well as the detected resources are included in the detection result. To support the post-processing of the detection results in external systems the result can also be provided in a machine-readable manner. A further possible output of the dynmx detection process are generic function logs (refer to Section 3.3). Basically, a generic function log, in terms of this work, represents the common data model to that the function log is transformed in the function log parser. Generic function logs are a sandbox-agnostic and machine-readable form of the VMRay function log and thus supports the fact that dynmx is not designed specifically for the VMRay Analyzer sandbox.

3.2 Function Log Parser

As introduced in the previous section, the function log parser is responsible for parsing relevant attributes from the text-based function logs provided as input data (the *input function log* in terms of this thesis) in order to bring the parsed information into a common data model. This common data model generalises from the specific function log format by normalising atomic attribute values and structuring the extracted information. The output of the function log parser is an abstract but well-structured and machine-processable version of the input function log. In terms of this work, the output of the function log parser for a certain input function log is referred to as *function log object*. Figure 3.2 illustrates the parsing process.

The function log object is an abstract version of the input function log since only a subset of the information provided in the input function log is incorporated. The attributes parsed from the input function log are carefully selected with regard to the information needed in the subsequent detection process, i.e. the derivation of the access activity model and, in particular, the detection algorithm. With reference to Table 2.3 giving an overview of the information provided in the VMRay function log, the categories

- Function log header,
- Process information and
- Function call information

are considered in the function log parser and are thus part of the function log objects provided to the subsequent process. It is noted that we omitted thread information since the detection algorithm is process-centric. Hence, all API function calls are assigned to the higher-level process regardless of the thread. More precisely, we concatenate API calls of the threads belonging to the same process. The reason for concatenating the API calls instead of trying to reconstruct the API call sequence in a consistent manner as it occurred during runtime is that the timestamp provided in the function log is not precise enough. Consequently, a correlation based on this timestamp leads to collisions since several API calls in different threads can have the exact same timestamp. We note, that the concatenation of API calls belonging to different threads of the same process can lead to missing detections if a malware uses multiple threads to implement a certain malware feature.

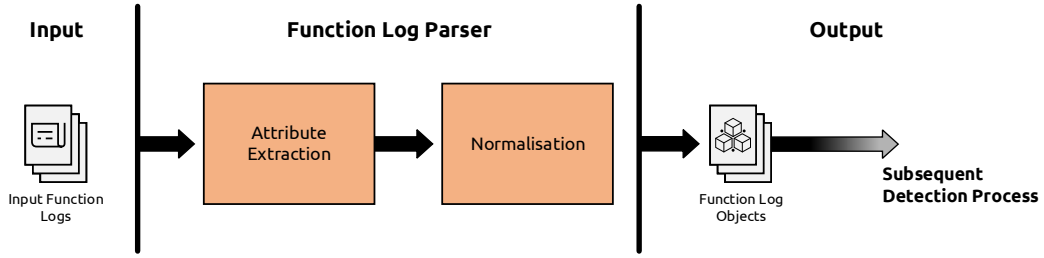

Figure 3.2: Function log parser overview

After the relevant attributes have been extracted from the input function log, we normalise the attribute values in terms of their format. This step ensures that the attribute values are available in a common format expected by the subsequent detection process. For example, certain strings are formatted consistent and number-based values are converted to a suitable data type. In addition to parsing and normalising atomic attributes and their values, the function log object explicitly maps semantic relationships present in the input function log. The basic semantic relationships that exist in the function log are as follows:

- One function log consists out of multiple processes (1:n relationship)
- One process calls multiple API functions (1:n relationship)
- One API function call has multiple parameters (1:n relationship)

- One API function call has one return value (1:1 relationship)

These basic relationships are further described in the common data model in the end of this section.

The design of the function log parser is based on the following requirements that we set out:

(R1) The function log parser needs to be extendible in order to allow the parsing of function logs originating from other sources than the VMRay Analyzer sandbox.

(R2) The function log parser has to maintain and explicitly map the semantic relationships present in the input function log.

(R3) The function log parser has to parse the relevant attributes from the input function logs without altering the semantics of the attribute values.

(R4) The function log parser has to normalise certain defined attribute values in order to suit the common data model.

(R5) The function log parser has to provide the parsed function log as structured object providing all relevant information for the subsequent detection process in a suitable format.

3.2.1 Plugin-Based System

Since the format of a function log is not standardised they typically differ fundamentally. Hence, the format of a certain function log type needs to be parsed individually in order to be able to extract the needed information. This aspect, in turn, suggests that the dynmx detection approach needs to be extendible in terms of the function log parser component and meets the requirement **(R1)**. Therefore, the function log parser is designed as an extendible plugin-based system. Each parser is thereby considered a plugin which can be dynamically incorporated into the detection process. Each parser plugin is responsible for parsing a certain function log format and providing a function log object according to the common data model. For example, there is a parser for the text-based VMRay Analyzer function log format and another parser for the XML-based VMRay function log format. If, in the future, Cuckoo sandbox reports containing API call information should also be processed using dynmx, a further plugin responsible for parsing these API call information needs to be added.

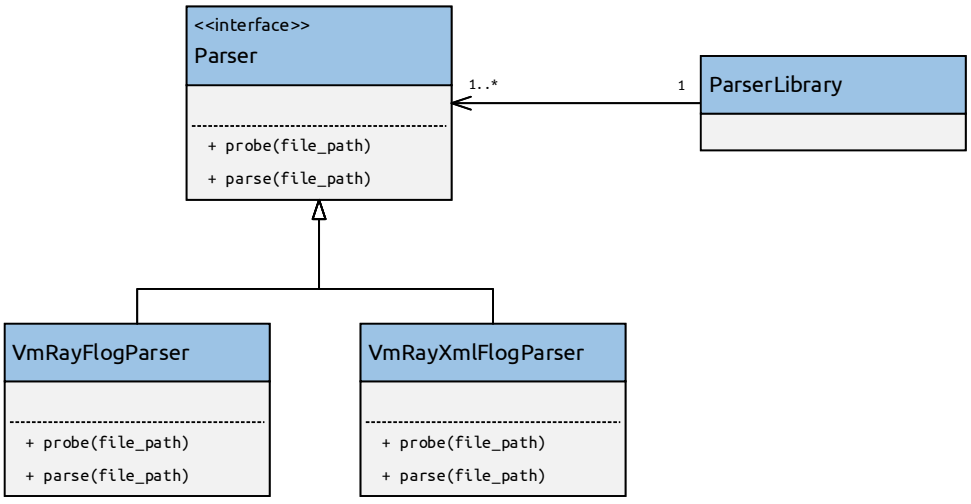

Figure 3.3: UML class diagram of the plugin-based function log parser design

The general design of the plugin-based system used for the function log parser is illustrated as UML class diagram in Figure 3.3. As seen in the class diagram there is an interface class called `Parser`. This class declares the two basic methods `probe` and `parse`. While `probe` is needed to identify a suitable parser for a certain function log format, `parse` is responsible for parsing the

needed information from the function log and providing the function log object. The parser library maintains a list of all available parsers. Consequently, it provides a central interface to the available parsers that is needed in the beginning of the detection process where the input function logs are read. For each input function log, we iterate over the list of available parsers maintained by the parser library in order to probe the input function log. As soon as a suitable parser is found indicated by a positive probing result, the parsing process immediately starts. Therefore, in the unlikely case of two different parsers providing a positive probing result for a certain input function log, the parser that comes first in the list is chosen.

```
1 # Flog Txt Version 1
2 # Analyzer Version: 1.8.0
3 # Analyzer Build Date: Nov  2 2015 15:01:39
4 # Log Creation Date: 09.12.2015 11:47
```

Listing 3.1: Header of the text-based VMRay Analyzer function log format

Generally, the suitable function log parser for a certain input function log is determined automatically based on characteristics of the function log format. For instance, the VMRay function log parser for the text-based function log format probes whether the first lines of the input function log begin with the header stated in Listing 3.1.

3.2.2 VMRay Function Log Parser

Since this thesis describes the dynmx detection approach based on the function logs provided by the VMRay Analyzer sandbox, we focus on the design principles of these parsers in the following. As introduced in the previous section, the VMRay function log parsers for the text-based and XML-based function log format are specialised parsers that implement the common `Parser` interface. While the probing process is already described in detail in the previous section, we will describe the actual parsing process in this section. The description is categorised according to the function log format, because the format and thus the procedure for parsing is fundamentally different.

Text-Based Function Log Format

As seen in the sample text-based function log in Appendix A.2, the function log is optimised for human-readability. Thus, parsing of the text-based function log is not trivial since pieces of information needed for providing a well-formed function log object are embedded in a semantic context rather than being explicitly marked up for machine-readability. In particular, semantic relationships between different information categories of the function log need to be deduced from the semantic content and handled by the parser. Since the pieces of information included in the function log are not marked up, we use low-level string operations as well as regular expressions to isolate and extract the needed attributes from the semantic context. The usage of these low-level operations allows us to be very flexible in parsing the exactly needed information and to handle format exceptions but makes the parser complex and drives the effort. Format exceptions in this context refer to individual entries of the function log that contain the same semantic information but differ in format. Because of many of these format exceptions which were identified during the development of the parser, we weighted the flexibility higher than the development effort. Consequently, we decided to pursue the low-level approach instead of leveraging a parser framework or lexical analysers. In the following, we describe the steps of the parsing process in detail.

Generally, the function log parser goes through the input function log in sequential order of the lines and parses the needed information line-by-line. Therefore, after successfully probing the input function log with the text-based function log parser, the header information residing in the beginning of the function log is parsed in the first step. Since the header information is formatted as key-value pairs separated by a colon (see Listing 3.1), parsing the needed attribute values is trivial. The header information is mainly parsed to gain meta-data of the function log and has no further relevance for the actual detection process.

According to the basic semantic structure of the text-based function log as illustrated in Figure 2.10, the next piece of information provided in the function log is the process information. The beginning of this section of the function log is identified by the line `Process:` (see line 6 in the sample text-based function log in Appendix A.2). The subsequent lines until the blank line contain process information such as the process ID and command line that was used for starting the process initially. Again, parsing of these attribute values is trivial since these are key value pairs separated by an equal sign (see line 7 to 16 in the sample function log).

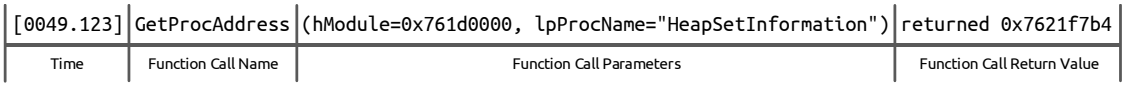

Figure 3.4: Parts of an API function call line of a text-based VMRay Analyzer function log

In order to find the API function calls made in the context of the identified process, the parser needs to find the corresponding threads of this process. Thread sections are generally introduced by lines containing the string `Thread:` (see line 38 in the sample function log). As already mentioned in Section 3.1, the detection approach is process-centric and thus the thread information is not parsed but only used to find the API function calls related to the thread. Characteristic for a function log line containing an API function call is the time in the beginning of the log line (see line 42 in the sample function log). By using a regular expression matching the time in the beginning of the line, the API function calls corresponding to the thread are identified. In order to extract the needed information from a function log line containing an API function call, the line is splitted as illustrated in Figure 3.4. As for the time, function call name and return value, the extraction of the atomic attribute values is trivial. The function call parameter string, on the other hand, requires more effort, since several format exceptions must be handled by the parser. We discuss these format exceptions in the following to show the complexity of the parser.

The example given in Figure 3.4 is trivial – the function call parameters are separated by a comma and have a simple key value structure. Exceptions of this format are shown in Listing 3.2. As shown in the first line of this Listing, the parameters can be categorised based on their direction modifier. While inbound parameters (identified by the `in` modifier) are handled as input parameters in the called function, outbound parameters (`out` modifier) are handled as return value by the calling function. The categorisation based on the direction occurs especially when parameters are called by reference. The parser separates the parameter string in this case into input and output parameters and consequently parses each category individually. The information whether a parameter is in- or outbound is kept and incorporated into the resulting function log object by the parser. The second line in the Listing 3.2 provides a further example for a format exception. In this case, the file name string of the file that is created by the function call is included as normalised string. The VMRay Analyzer sandbox automatically incorporates a normalised form of file paths into the text-based function log. The parser prioritises the normalised form of a file path in order to get a consistent file path format among all parsed API function call parameters. Please note, that a normalised form of a file path can also occur in the return value of the API function call entry. Handling these format exceptions carefully without altering the semantics of the attribute values contributes to the fulfilment of requirement **(R3)**.

```
1 [0049.117] RegOpenKeyExW (in: hKey=0x80000002, lpSubKey="↩
      Software\\Microsoft\\Windows Script Host\\Settings", ↩
      ulOptions=0x0, samDesired=0x20019, phkResult=0x12fb88 | ↩
      out: phkResult=0x12fb88*=0x7c) returned 0x0
2 [0050.438] CreateFileW (lpFileName="C:\\Users\\user\\Desktop↩
      \\test.vbs" (normalized:
      "c:\\users\\user\\desktop\\test.vbs"), dwDesiredAccess=0↩
      x80000000, dwShareMode=0x1, lpSecurityAttributes=0x0, ↩
      dwCreationDisposition=0x3, dwFlagsAndAttributes=0x8000000↩
      , hTemplateFile=0x0) returned 0x1d0
```

Listing 3.2: Format exceptions of the function call parameter string in text-based VMRay Analyzer function logs

Memory pointers are an additional format exception that needs to be handled by the function log parser. Memory pointers are identified by the star character after the memory address as illustrated in the example function call in Listing 3.3. In this example, the parameter `lpSystemTime` is a memory pointer to the data structure `SYSTEMTIME` [82, 83]. Generally, memory pointers are handled by the parser as arguments that, in turn, have parameters by themselves. In addition to the pointer parameters, the parser extracts the memory address. At this point, it has to be considered that pointer parameters can be memory pointers to other data structures. Consequently, function call parameters may contain a complex, nested structure that needs to be parsed thoroughly in order to provide all relevant information for the subsequent detection process and therefore sustaining requirement **(R5)**.

```
1 [0038.268] GetLocalTime (in: lpSystemTime=0x26ee40 | out: ↩
      lpSystemTime=0x26ee40*(wYear=0x7df, wMonth=0xc, wDayOfWeek=0↩
      x3, wDay=0x9, wHour=0xc, wMinute=0x2f, wSecond=0x2e, ↩
      wMilliseconds=0x2b8))
```

Listing 3.3: API function call containing a pointer

In order to be able to map semantic relationships deduced from the input function log, the function log parser keeps track of previously parsed information. For example, the function log parser needs to map API function calls to the previously parsed process information in order to maintain the semantic relationship between processes and API function calls belonging to this process. This parser behaviour is manifested in requirement **(R2)**.

The normalisation done by the function log parser mainly focuses on the parsed parameter values. API function call names as well as parameter names are kept since they are defined by the Windows API. Number-based parameter values initially parsed as strings from the input function log are converted to number types in order to allow arithmetic operations and the correct usage of relational operators in the detection process. Further, file paths are normalised by replacing double backslashes with a single backslash. Generally, parsed string values are stripped. Stripping in this context means that whitespaces and characters that are only added to the function log to improve the readability like quotes are removed from the beginning and end of the parsed string value in order to get clean atomic values. The normalisation taken into account is compliant to requirement **(R4)** and is thus only done to suit the common data model.

XML Function Log Format

In contrast to the text-based function log, the XML-based function log is optimised for machine-readability allowing the parser to be less complex. As attributes and their values are marked up with the generic data representation syntax XML, we can leverage standard XML parsing frameworks for the basic textual parsing of the input function log. Consequently, the main task of the function log parser for the XML-based format lies exclusively in the extraction of relevant attributes and their values rather than extracting them from an unstructured input string by using basic string operations and regular expressions.

In order to extract relevant information from the XML-based function log we need to identify the XML elements of interest within the XML tree. Based on the relevant function log categories defined in the introduction of Section 3.2, Table 3.1 shows the assignment of these categories to the relevant XML element tags available in the function log.

The basic process of parsing is analogous to the approach described in the previous section on parsing the text-based function log format. Hence, the function log header information is parsed first, afterwards the process information and finally the API function calls related to the process. However, identifying and extracting the relevant XML elements needed to extract the relevant information is accomplished by leveraging the XML Path Language (XPath) [179]. The resulting XML elements are structured as shown in the sample XML function log in Appendix A.2 (line 3). The `<monitor_process>` elements are so called empty-element tags and thus have no child elements.

Category	XML Element Tag
Function log header	<analysis>
Process information	<monitor_process>
Function call information	<fncall>

Table 3.1: Assignment of function log categories to XML element tags

Therefore, the process information is defined as attributes of the <monitor_process> element. Besides the attributes that characterise the process itself like the executable or the command line, the attribute process_id is of special interest for the subsequent parsing process (see Listing 3.4). The process_id attribute is used to correlate API function calls to the corresponding process from that the function call originated. This correlation is indeed the explicit mapping of the semantic relationship that exists between API function calls and processes that called the API functions. This explicit relationship was mentioned as one of the advantages of the XML-based function log format introduced before.

```
1 <monitor_process ts="40421" process_id="1" image_name="pgv.exe"↩
       filename="c:\\users\\user\\desktop\\pgv.exe" page_root↩
      ="0x16f600000" os_pid="0x6d0" [...] />
```

Listing 3.4: process_id attribute of the <monitor_process> XML element

In order to identify API function calls related to a certain process, the parser correlates the process_id attribute. For instance, with reference to Listing 3.4, a function call element related to the process pgv.exe has also the process_id attribute value 1. Such a related function call is shown in Listing 3.5. The corresponding XPath expression to locate the API function calls related to a certain process is fncall[@process_id='*<id>*'] where *<id>* is the attribute value of the process from that the function calls should be located.

The structure of a resulting <fncall> element representing an API function call that is related to the process pgv.exe is shown in Listing 3.5 (please refer to the XML function log sample in Appendix A.2, beginning from line 8, for further function call examples). In contrast to the <monitor_process> element, the <fncall> element is not flat-structured but has child elements. These child elements represent the function call arguments as well as the return value of the function call. The distinction based on the direction modifier is in opposition to the text-based function log not optional but mandatory. While the child elements of the <in> element are inbound parameters of the API function call, the child elements of <out> represent the outbound parameters. The return value of the function call is incorporated as outbound parameter with the constant name ret_val into this structure. Parsing the needed information from the XML elements is straightforward since the attribute values are atomic and well-formed.

```
1  <fncall ts="43187" fncall_id="79" process_id="1" thread_id="1" ↩
       name="RtlAllocateHeap" addr="0x777ee026" from="0x40604d">
2      <in>
3          <param name="HeapHandle" type="void_ptr" value="0↩
               x2b0000"/>
4          <param name="Flags" type="unsigned_32bit" value="0x0↩
               "/>
5          <param name="Size" type="void_ptr" value="0x94"/>
6      </in>
7      <out>
8          <param name="ret_val" type="void_ptr" value="0x2b3240↩
               "/>
9      </out>
10 </fncall>
```

Listing 3.5: Example <fncall> XML element representing an API function call related to the process pgv.exe

In the case of memory pointers, the structure of the <param> element becomes more complex since it contains child elements instead of being an empty-element tag. Generally a pointer is identified by the type `ptr` as shown in Listing 3.6. Parameters of the pointer, in this case a pointer to a local unique identifier (LUID) [78], are child elements of the <deref> element. Analogous to the pointer structure in the text-based function log format, the pointer structure in the XML function log format can also be nested since pointer parameters can, in turn, contain pointers.

For the sake of completeness, it should be noted that the XML-based function log not only provides the function log information in a machine-readable format but also additional information in comparison to the text-based function log format. For example, the XML-based function log provides type information for function call parameters as shown in Listing 3.5. However, the common data model is designed based on lowest common denominator of the two file formats. Therefore, this additional information is out of scope for the parsing process.

Most of the needed information is already provided as atomic well-formed values due to the XML format. Hence, the normalisation process is limited to a minimum. Only some special characters used in the XML markup like the quote character need to be escaped. According to the XML standard referenced in [180], an escaped character follows the format &<*abbr*>*;* where <*abbr*> is a defined abbreviation of the character like `quot` for the quote character. These escaped characters are replaced by their unescaped equivalent according to the XMl standard. Further normalisation steps are the parsing of integer values and the replacement of double backslashes in file paths to single ones.

```
<fncall ts="121605" fncall_id="4487" process_id="2" thread_id↩
    ="2" name="LookupPrivilegeValueA" addr="0x76d5404a" from↩
    ="0x4078a8">
    <in>
        [...]
    </in>
    <out>
        <param name="lpLuid"type="ptr"value="0x18fd88">
            <deref type="container">
                <member
                    name="LowPart"type="unsigned_32bit"value="0x5"/>
                <member name="HighPart"type="signed_32bit"value="0"/>
            </deref>
        </param>
        <param name="ret_val" type="bool" value="1"/>
    </out>
</fncall>
```

Listing 3.6: Example pointer representation in the XML function log format

3.2.3 Data Model

We map the data parsed according to the previously described approach to a common data model as introduced in the dynmx detection process overview in Section 3.1. Generally, the common data model is the blueprint for the function log object provided by the function log parser to the subsequent detection process. To be more precise, with reference to the general definition of an object in the domain of object oriented programming [6], the common data model defines the properties of the function log object but not its methods. The methods are out of scope in this section since we focus on the data model and the mapping of parsed function log information solely.

We suggest the common data model illustrated as logical UML entity class diagram in Figure 3.5 for the object oriented design of function logs. Our aim in defining a proper data model for our detection approach was to create a slim but sufficiently comprehensive model of the input function log that does not limit the subsequent detection. Of course, the data model also follows the

requirements that we set out for the function log parser defined in Section 3.2. First and foremost, the data model satisfies the requirement **(R5)** since the object is structured, contains the relevant information for the detection process (for instance, the processes and the API function calls with their arguments and return value) and thus provides the input function log in a suitable format. As seen in Figure 3.5, the data model explicitly defines the semantic relationships between the different components of the input function log as associations and hence sustains requirement **(R2)**. Further, the common data model is the basis for the requirements **(R3)** and **(R4)** as it defines the relevant attributes as well as the data type for each attribute which has to be considered in the normalisation process.

Generally, the designed data model represents the relevant entities or information categories of a function log introduced in Figure 2.10 (Section 2.6.2). Hence, as illustrated in Figure 3.5, the starting point of the data model is the `FunctionLog` entity. This entity basically represents the input function log with its general attributes like the function log name or the file path. By following the reference of the function log, the process information is identified as next relevant entity. Thereby, the function log *can* reference `Process` entities. A `FunctionLog` entity without any references to processes is referred to as *empty function log*. Empty function logs may occur in practice if, for example, the sample could not be executed in the sandbox environment.

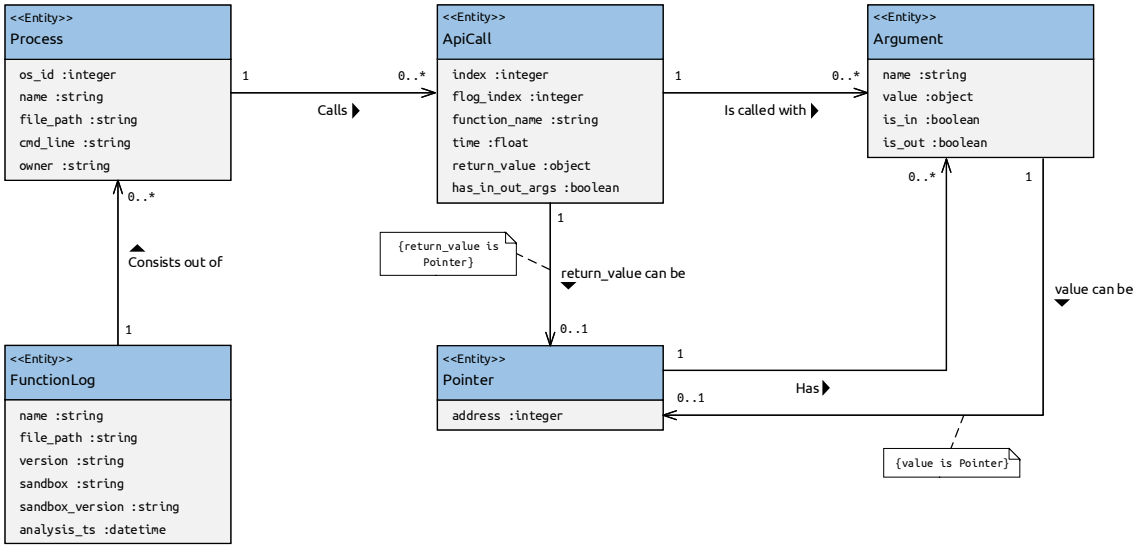

Figure 3.5: Common data model of a function log object

The `Process` entity provides general attributes of the process like the PID, the process name or the command line with that the process was executed. Analog to the function log, the process can also be empty, i.e. has no references to API function calls represented by the `ApiCall` entity. Since the API function calls are the main entity leveraged in the detection, we designed them as close to and as detailed as represented in the input function log in order to provide the most detailed and concise detection capability possible to the domain experts. The reference to the `Pointer` entity is only valid under the constraint that the return value of the function call is a pointer. Otherwise the reference is not existent and the return value has an atomic value. We apply the same design for the reference from the `Argument` entity to the `Pointer` entity. This reference is only valid under the constraint that the argument value is a pointer. Otherwise the argument value is atomic. The `Argument` entity generally represents an argument. As seen in the data model this can be the argument of an API function call as well as the argument of a pointer structure. At this pointer, it becomes obvious how nested pointers are represented according to the common data model. For instance, the argument value, for example of an API function call argument, is a pointer that has in turn an argument which has a pointer as argument value.

Some of the attributes used in the common data model need further explanation since they are not

explicitly stated in the input function log but deduced from it. For instance, the index attributes `index` and `flog_index` of the `ApiCall` entity. While the `index` attribute represents the position of the function call in the process (begins with zero for each process and is incremented for each parsed function call), the `flog_index` represents the actual line number of the function call in the input function log. The `index` attribute is needed in the detection process in order to reconstruct the sequence in that the function calls are ordered in the input function log. The `flog_index` attribute has no further relevance for the subsequent detection process but has been incorporated to explicitly list the line numbers of the input function log that caused a detection. This helps the domain expert to understand the detection caused by a certain signature and consequently increases confidence in the detection since the cause is transparent. A further deduced attribute is the `has_in_out_args` attribute of the `ApiCall` entity. This attribute reflects whether the arguments of the function call have decorators and are thus separated into in- and outbound parameters as discussed in Section 3.2. This information is needed in the subsequent detection process.

After describing the data model in detail, we demonstrate the application of the model to a sample function log. For the sake of brevity, we will use a simplified, slightly modified and minimised function log presented in Appendix A.3 (Listing A.1). Although the function log is minimised and slightly modified to demonstrate the capability of the common data model, it origins from a real-world analysis. We suppose that this text-based input function log is passed to the function log parser. The resulting function log object accords to the common data model and is illustrated in Appendix A.3 (Figure A.1) as UML object diagram. While the representation of the function log and the process in the function log object is straightforward, the API function call representation and especially its argument objects need further explanation. As seen in the input function log, the argument value of the `lpSystemTimeAsFileTime` parameter is a nested pointer structure. The outer pointer is obvious since it is the argument value of the outbound `lpSystemTimeAsFileTime` itself. This memory pointer points to the `FILETIME` structure [81] as described in the official documentation of the `GetSystemTimeAsFileTime` API documentation [83]. According to the API documentation, this structure, in turn, consists of two parameters – `dwLowDateTime` and `dwHighDateTime`. The inner pointer is seen in the `dwHighDateTime` argument value (object a2 in the UML object diagram). Since the inner pointer has no further structure but only points to an atomic value, the argument of the inner pointer (argument object a3) is not named but *anonymous*. It has to be considered that the API function call object ac only has one argument object in the object model as the in- and outbound argument `lpSystemTimeAsFileTime` from the input function log can be consolidated. Since the argument value `0x1aff38` of the inbound argument corresponds to the pointer address of the outbound argument the consolidation can take place without loss of information. Generally, the consolidation of arguments can be applied if arguments are passed to the API call by reference, i.e. as memory pointer.

3.3 dynmx Generic Function Log Format

The common data model introduced in the previous section is not just the basis for the function log object but also for the initiative to introduce a *generic function log format*. *Generic* in this context means that the function log format abstracts from the source malware analysis platform from that the function log originated. The function log formats of individual malware analysis platforms differ vastly and thus only justify the design of a plugin-based function parser (see Section 3.2.1). But the vastly different function log formats not only hamper automatic processing in the context of dynmx but additionally complicate manual sighting and analysis by domain experts. Further, a subset of the function log formats are not machine-readable like the text-based VMRay funtion log format. Firstly, this complicates the development of appropriate and reliable parsers and secondly this prevents leveraging the valuable information from the function log in automated processes outside of the context of dynmx.

To overcome the described issues caused by the variety of function log formats, we introduce the *dynmx generic function log format* in this thesis. This generic function log format reflects the common

data model and therefore the function log object in a serialised text-based form. In contrast to the generic function log format provided by VMRay Analyzer sandbox (see Section 2.6.3), this format does not abstract from API function calls by mapping them to generic function calls. Instead, our generic function log format abstracts from the actual specific format and therefore keeps the abstraction layer provided by the input function log. For the textual representation of the function log object we leverage JSON. The main reason we chose JSON is that it is machine-readable by definition and has a lightweight markup. Of course the human-readability of a JSON formatted text in contrast to a pure text-based format optimized for human-readability is worse but we weight the machine-readability higher in terms of our use case. Additionally, domain experts are often used to read machine-readable file formats as they need this ability to analyse malware properly. A further advantage of using JSON as general data representation syntax is that it is widely adopted in today's programming and scripting languages. Standard libraries for parsing JSON formatted files exist in a wide variety and allow the domain expert to use function logs following the dynmx format with low effort.

File Size [bytes]	No. of Lines	Avg. Runtime JSON Serialisation [s]	Avg. Runtime YAML Serialisation [s]	Factor
674,059	8,021	0.1862	4.5321	24.34
1,477,347	17,465	0.7893	20.9601	26.56
6,058,532	61,704	1.7169	50.072	29.17
11,088,095	86,293	3.0658	91.7178	29.92
91,453,829	527,193	31.9446	1014.9063	31.77

Table 3.2: Runtime comparison for serialising a converted input function log to JSON and YAML

Generally, the YAML data representation would have been another suitable option for the dynmx function log format since it is also machine-readable despite providing a good human-readability. But we choose JSON over YAML mainly because of JSON's efficiency when it comes to serialise large nested data structures (in this case dictionary data structures). We therefore briefly evaluated the serialisation of the internal dynmx function log format representation as nested dictionaries to JSON and YAML in Python 3.7. We used the standard Python JSON implementation (package `json`) and the third-party Python package PyYAML (version 5.2) [117] for the comparison. For the same input function log with a file size of 6,058,532 bytes (61,704 lines), the serialisation to JSON takes around 1.7 seconds (arithmetic mean value of four consecutive runs) while the serialisation to YAML takes around 50.07 seconds (arithmetic mean value of four consecutive runs). Hence, the evaluation shows that the serialisation to YAML for this particular input function log takes around 29 times as long as the serialisation to JSON. As shown in Table 3.2, the factor increases the larger the input function log gets. In order to reduce the bias of the measurement, the runtime is again the arithmetic mean value of four consecutive runs.

Input Files	No. of Files	Avg. File Size Increase Factor without Compression	Avg. File Size Increase Factor with GZIP Compression
Complete Data Set	1,327	3.0513	0.4436
File size > 100 kB	676	4.6508	0.0769
File size > 1 MB	191	7.6024	0.1086

Table 3.3: File size comparison of input function logs to converted function logs with and without compression

The downside of designing the dynmx function log format machine-readable and leveraging JSON, is the increased file size of the resulting converted function log in comparison to a pure text-based

format like the text-based VMRay function log format. Please note, that we focus on the text-based VMRay function log format rather than the XML-based VMRay function log format since the file size increase is expected to be larger. This larger file size increase is expected due to the missing markup in the text-based VMRay function log format. In order to provide reliable figures, we analysed and compared the change in file size for a data set consisting of 1,327 text-based VMRay function logs. Therefore, we converted the function log data set to the dynmx function log format and calculated the factor of file size increase for each converted function log. The average (arithmetic mean) factor of file size increase we calculated is around 3. Hence, the file size of the converted function log in the dynmx format is around 3 times larger compared the input file size. This average factor increases for larger input function logs. For instance, for function logs with a size larger than 100 kB (676 files of the aforementioned data set) the factor is around 4.6. The highest measured factor is 13.4. Please note, that the file size increase factor is not just correlated with the input file size but also with the structure and content.

To overcome the file size increase we make use of GZIP compression. By compressing the converted function log the file size of the same becomes typically less than the file size of the input function log. In order to provide reliable figures, we converted the same data set of text-based function logs with added GZIP compression. The arithmetic mean factor of file size decrease we calculated is 0.4 while the maximum factor for this data set is 2.1. The maximum factor is due to an empty input function log since the compression itself increases the file size. In contrast to uncompressed converted function logs, the average factor improves with input function logs larger than 100 kB to 0.08 and with function logs larger than 1 MB to 0.1. Table 3.3 gives an overview of the file size evaluation with and without compression.

```
# dynmx generic function log
# converted from: <file_path>
# converted on: 2020-04-10 11:37:01.997282
```

Listing 3.7: Header information provided in a dynmx formatted function log

After explaining our design principles and decisions, we introduce the dynmx function log format in detail. Please note, that the following explanation omits the compression but assumes that the converted function log has already been decompressed. The basic structure of a function log formatted according to the dynmx function log format is divided into two parts. The first lines of the function log contain header information like the timestamp when the input function log has been converted and the path of the input function log. This information is incorporated into the dynmx function log format to provide context information to the domain expert (see Listing 3.7).

Since JSON does not support comments the header information needs to be stripped off before continuing to parse the actual function log object. The second part contains the actual function log formatted as multi-line JSON. The function log is represented in a nested structure. Hence, in contrast to the XML-based VMRay function log format, semantic relationships between the different information categories are not mapped using identifiers but instead using a nested structure with lists.

```
{
  "flog": {
    "version": "1.0",
    "sandbox": "VMRay Analyzer",
    "sandbox_version": "2.1.1",
    "analysis_ts": "11.04.2020 10:08:00.387000",
    "processes": [
      {
        "os_id": 1,
        "name": "p1",
        "file_path": "path",
        "cmd_line": "cmd line",
        "owner": "owner",
        "api_calls": [
```

```
15            {
16              "flog_index": "9e3757a7-7295-42f6-a910-53↩
                   f31cfb5c68",
17              "function_name": "function",
18              "time": 1.000,
19              "arguments": [
20                {
21                  "name": "arg1",
22                  "is_in": false,
23                  "is_out": false,
24                  "value": 0
25                },
26                ...
27              ],
28              "return_value": null
29            },
30            ...
31          ]
32        },
33        ...
34      ]
35    }
36  }
```

Listing 3.8: Basic structure of a dynmx formatted function log

Listing 3.8 shows the basic structure of the dynmx function log format. As seen in the listing, the key `flog` corresponds to the entity `FunctionLog` of the common data model. The JSON object does not contain all of the attributes defined in the common data model since some of the attributes are set dynamically in the parsing process like the function log format and the file path. The `processes` key of the flog JSON object contains the list of processes related to the function log. Each entry in this list is a JSON object that corresponds to the `Process` entity of the common data model. The API function calls made from the process are found as list in the key `api_calls` of the process JSON object. Again, each entry of this list corresponds to the `ApiCall` entity in the common data model. For the sake of traceability, we generate a universally unique identifier (UUID) for each converted API call (attribute `flog_index`). This UUID is needed since typical JSON parsers do not provide the possibility to determine the actual line number for a parsed key. Last but not least, the semantic relationship between API function calls and their arguments is realised with the key `arguments` of the API call JSON object.

To demonstrate the format in more detail, we converted the text-based sample function log from Appendix A.3 to the dynmx function log format. The result is shown in Appendix A.4. The interesting part of this sample function log converted to the dynmx format is the nested pointer structure in the `lpSystemTimeAsFileTime` argument value. With reference to lines 23 and 36 of the converted sample function log sample, a pointer is indicated by the value of the `value` key being a JSON object instead of an atomic value. Further, more comprehensive function logs in the generic format are found in the digital appendix on the enclosed flash drive[1].

In order to use the converted function logs as input function logs in the detection process a parser for the dynmx function log format has been developed. This parser reads the converted function log, strips off the header lines and then leverages JSON to parse the function log information. The parsed information is afterwards mapped to the common data model. Consequently, the parser works similar to the parser for the XML-based VMRay function log format with the difference that relationships do not need to be reconstructed based on the correlation of identifiers.

[1]The converted generic function logs are found in the directory `Digial_Appendix/3_Design/dynmx_Generic_Flog/`.

3.4 Access Activity Model

Based on the function log object introduced in Section 3.2.3, we extract the resource usage and interaction of a certain process which is also referred to as the *access activity model* [63] in the remainder. As a result, the access activity model enriches the function log object by adding a resource-centric view. This resource-centric view of the function log object is valuable for the signature definition and detection process as malware typically and heavily uses resources of the underlying operating system like files or network connections. With reference to the background Section 2.4.2, the management of resources is one of the main tasks of modern operating systems. Hence, manipulating resources can make up a large part of mostly recurring API call sequences in the function log. So, as for the signature definition, the signatures become more concise since the resource-centric view abstracts from these recurring API calls used for manipulating resources. Hence, the domain expert can focus on defining the actual resource manipulation instead of considering all possible low level API calls for certain resource manipulations. In the subsequent detection algorithm, we can then match the resource definitions from the signature against the access activity model without the need of detecting characteristic API call sequences used for manipulating resources.

The access activity model was first introduced and defined by Lanzi et. al in [63]. We adopt their general approach of deducing interactions with OS resources from characteristic system call sequences in our work. Nevertheless, we need to adapt the definition of Lanzi et al. to our needs and to our use case of detecting specific malicious behaviour based on signatures. Our use case differs fundamentally from the use case of Lanzi et al. which is differentiating benign from malicious behaviour based on the access activity model. First of all, we intentionally define the resource activity model as process-centric instead of system-centric. This adaptation is conform with our general detection approach being process-centric. Further, Lanzi et al. only consider system calls from the Windows Native API. Since the VMRay sandbox uses adaptive monitoring (see Section 2.6.1), not all system calls are traced in the function log. Therefore, we need to extend the access activity model by considering WinAPI function calls used for manipulating resources in addition to system calls. A further extension of the initial access activity model definition is needed in terms of the resource representation. Lanzi et al. abstract from the actual file or registry resource by just taking folders and Registry keys into account. For our use case this abstraction is counter-productive as we want to specifically detect the resources manipulated by a process. For this sort of detection the full resource path including the file name and Regisry values of a certain key are of importance. As today's malware heavily relies on network connectivity, e.g. for communicating with their command and control infrastructure, we also take network resources in addition to file and Registry resources into account. As a further refinement of the initial model definition which defines the access operations `read`, `write` and `execute`, we add the access operations `create` and `delete`. In summary, we adapt the access activity model defined by Lanzi et al. as follows:

- The access activity model is process-centric instead of system-centric.
- In addition to system calls, we also consider WinAPI function calls.
- Our lowest layer of abstraction for the resource representation is the actual file, registry value or network resource instead of just folders and Registry keys.
- In addition to file and Registry resources, we also consider network resources.
- We refine the access operations by adding the `create` and `delete` operation to the initially defined operations `read`, `write`, `execute`.

In the following sections, we will describe the resource extraction and give an overview of the considered API calls. After that, we will present the model derivation based on the previously extracted resources.

3.4.1 Resource Extraction

Generally, the outcome of the resource extraction is a list of resources that were accessed and manipulated by a certain process. Resources are thereby represented by their location. For files and Registry resources this is the absolute file or Registry path and for network resources this is the URL. In addition to the resource, we also keep track of the access operations, i.e. how the resource has been accessed. In order to extract resource interactions based on the function log object, we first need to identify relevant API function calls. Therefore, we consider the system calls and WinAPI function calls separately. Due to the lack of reliable and precise documentation, we do not consider kernel routines. As for the system calls, unfortunately, Lanzi et al. do not provide a comprehensive list of system calls that they have taken into account for their access activity model in their publication. Consequently, we identify and categorise the system calls on our own. To get a list of available system calls, we extracted the SSDT from the memory dump of a Windows 7 (x86, SP1, build 23418) system using memory forensics (see Section 2.4). This resulted in 1,229 available system calls. Based on this set of available system calls, we extracted the relevant system calls for manipulating file and Registry resources in a manual process. To our knowledge, there are no system calls for manipulating network resources. Thus, network resources are out of scope for the system call categorisation. Please note, that the SSDT only contains the `Nt` versions of system calls and we thus need to add the `Zw` system calls manually. Table 3.4 provides statistics on the identified system calls.

Resource Category	No. of Relevant API Calls	No. of Relevant System Calls	No. of Relevant WinAPI Calls
File System	147	50	97
Registry	67	28	39
Network	38	0	38

Table 3.4: Statistics on identified API calls for manipulating resources categorised by the resource category

As for the WinAPI function calls, we extracted the API calls from 1,327 function logs in order to obtain a representative list of used WinAPI function calls. The function logs taken into account were generated based on the analysis of malicious and benign software samples. By removing system calls and kernel routines from this set, the extraction resulted in 2,197 WinAPI function calls that were manually assessed in order to identify API function calls relevant for manipulating the considered resource categories. In addition, we used the API call categorisation provided by Plohmann in [121] from his ApiScout publication [123] which states a set of, in sum, 5,225 API calls. This shows the immense complexity of the Windows API which, in turn, makes the identification of resource manipulation hard as a large set of API function calls exists for the exact same operation. We would like to emphasize at this point that our aim is not to strive for a complete set of all possible WinAPI function calls for the individual categories, but rather to identify a relevant set of used API calls. Table 3.4 provides statistics on the identified WinAPI function calls that we take into account. For a comprehensive list of the API calls that we use for extracting resources, please refer to the digital appendix of this thesis which is found on the enclosed flash drive[2].

As introduced in the background on resources (see Section 2.4.2), user-mode processes use handles in order to reference resources. Hence, a large subset of the identified API calls expect handles to be passed to the function to manipulate a certain resource. Consequently, we need keep track of the handles referenced in a certain user-mode process to extract meaningful and consistent information of resource manipulations based on API call sequences from that process. For instance, we temporarily store opened handles returned by certain API calls like `NtOpenFile` or `RegOpenKey`. As a result, we can map subsequent API calls referencing previously opened handles for manipulation operations like `WriteFile` or `RegDeleteKey` to the appropriate resource. This generally corresponds

[2]The list of API calls used for extracting resources is found in the directory `Digial_Appendix/3_Design/Access_Activity_Model/` on the enclosed flash drive.

to the approach described by Lanzi et al. in [63]. So, for each relevant API call identified in the previous step, we need to add the following information:

- The argument that returns the resource handle, if the API call returns a resource handle.
- The argument that is used for referencing a resource with a handle, if the API call references a resource.
- The argument that contains location information of the resource.

As for file resources, the API calls mainly provide absolute path information. But for Registry resources, we need to reconstruct the absolute path based on several API calls that typically open Registry keys and values relative to an already opened base path. Please note the example given in Listing 3.9. The `RegCreateKeyA` references the handle `0x80000001` in the `hKey` argument which is the built-in handle to the registry hive `HKEY_CURRENT_USER` [77] and opens the sub-key `Software\Microsoft\Windows\CurrentVersion\Run`. The subsequent API Call `RegSetValueExW` creates the value Microsoft in the previously opened key by referencing the handle returned by the API call `RegCreateKeyA` in the argument `phkResult`. Hence, the full location of the Registry resource is `Software\Microsoft\Windows\CurrentVersion\Run\Microsoft`.

```
1 [0023.701] RegCreateKeyA (in: hKey=0x80000001, lpSubKey="↩
      Software\\Microsoft\\Windows\\CurrentVersion\\Run", ↩
      phkResult=0xdfe28 | out: phkResult=0xdfe28*=0x9c) ↩
      returned 0x0
2 [0023.701] RegSetValueExW (in: hKey=0x9c, lpValueName="↩
      Microsoft", Reserved=0x0, dwType=0x1, lpData="C:\\Users\\↩
      user\\AppData\\Roaming\\2C46A1.exe", cbData=0x5a | out: ↩
      lpData="C:\\Users\\user\\AppData\\Roaming\\2C46A1.exe") ↩
      returned 0x0
3 [0023.705] RegCloseKey (hKey=0x9c) returned 0x0
```

Listing 3.9: Resource manipulations on a Registry Resource

The same is also applicable for some API calls used to manipulate network resources. For example, the `WinHttpOpenRequest` API call requests a certain HTTP resource from a web server to that a connection was established using the `WinHttpConnect` API call.

Resource Category	Stored Attributes	Applicable Access Operations
File System	• File path	• Create • Delete • Read • Write • Execute
Registry	• Registry path	• Create • Delete • Read • Write
Network	• IP address • DNS name • Destination port • URL	• Read • Write

Table 3.5: Information stored and applicable access operations for the considered resource categories

In addition to the handle table, we monitor DNS requests of the process during the network resource

extraction and thus create a temporarily DNS cache. Based on this DNS cache, we can map used IP addresses in API calls to their corresponding DNS names and thus reconstruct the network behaviour. This is especially helpful for API calls that expect IP addresses as arguments as these API calls do not resolve DNS names on their own[3].

By keeping track of resource handles, we can precisely monitor resource manipulations and thus access operations. We associate these access operations with the corresponding resource in the last step of the resource extraction process. Similar to Lanzi et al., we only take API calls and thus real operations on resources into account rather than the desired level of access to a certain resource. As a result, we get a precise picture of resource operations that can be used in the subsequent detection process. Table 3.5 summarises the resource extraction by giving an overview of the information extracted for the considered resource categories as well as the applicable access operations for the same.

3.4.2 Limitations

The proposed access activity model faces limitations that we will discuss in this section. First of all, we do not check the return value of the API calls in our current design. As a consequence, we cannot determine whether a resource is successfully manipulated. So, to be more precise our adapted access activity model does not reflect precisely the actual used resources of a certain process but rather the *possibly* used resources. We intentionally decided to leave the return value out of scope in order to broaden the detection scope and to avoid that certain circumstances of the sandbox falsify the extracted resources. For example, if a malware sample requests a certain Registry key that is available on real workstations but not in the sandbox, this resource would have not be extracted as the return value of the API call would have indicated that the access was not successful.

A further limitation of the access activity model is caused by the ambiguity of Windows API calls. For instance, API calls used to create new resources like `CreateFile` or `RegCreateKey` can also be used to open an already existent resource. Hence, we cannot determine whether the real access operation is `create` or `read` for resources that are accessed using this kind of API calls solely based on the function log object. Further information of the actual file system or Registry state would be needed in order to identify the correct access operation. In this case, we always use the access operation `create` and the domain expert needs to interpret the `create` access operation as "possibly created if not existent".

With reference to network resources, the proposed model faces another limitation. If the malware sample uses low-level sockets instead of higher-level API calls for HTTP communication, we cannot determine the actually accessed HTTP resource, i.e. the complete URL. Since the process needs to construct relevant application layer headers when using low-level network sockets the needed HTTP header containing the resource is only available in buffers that are sent over the network to the remote server. With a certain probability the headers can be extracted from previous string operations that are often used to construct the header. Nevertheless, for our proposed model this is out of scope.

3.5 Signature Definition

Following the dynmx detection process introduced in Section 3.1 (Figure 3.1), dynmx signatures are the second input – besides the function log – needed for the detection algorithm. Signatures play a key role in the whole detection process, as they precisely bundle the characteristics of previous, mostly manual, analysed malware samples and thus contain the actual intelligence. The quality of a signature-based detection in general, stands and falls with the quality and specificity of the signatures used. Expressed more formally, the quality of a signature-based detection is not

[3] For example low level Berkeley compatible Windows sockets exported by `ws2_32.dll` [146]

just correlated with the quality of the used signatures but is causally related since the detection algorithm can only detect what is defined in the underlying signatures. Consequently, the signature definition is one of the key design components and mainly substantiates the extendibility of the detection approach presented in this thesis.

According to the definition which we set out in the background Section 2.3, dynmx signatures fall in the category of malware-based signatures since they describe characteristics of malware. In contrast to YARA rules, dynmx signatures specify *behavioural characteristics* instead of static ones. In terms of this thesis, these behavioural characteristics that can be characterised in dynmx signatures include

- the usage of API function calls and
- the acquisition and usage of operating system resources.

By leveraging behavioural instead of static characteristics for defining malware-based signatures, dynmx signatures become immune to common obfuscation techniques targeting the static appearance of the malware sample as outlined in the background on dynamic malware analysis provided in Section 2.2. Notwithstanding, sophisticated obfuscation techniques that alter the malware behaviour, in particular, in terms of the interaction with the operating systems' API and resource management, can cause the absence of detection. Although, known obfuscation techniques altering the malware behaviour [144, 182] mainly target the instruction sequence instead of the interaction with the operating system API [112]. Supposing that the malware behaviour is altered at the API to the operating system due to an obfuscation technique this technique faces fundamental limitations since certain API calls need to be called in a certain sequence in order to comply with the API. For instance, a file cannot be written using the WinAPI call `WriteFile` [88] before it has been opened using the `OpenFile` API function call [85] or a similar call that returns a valid file handle. We can even apply this schema to whole function blocks of malware samples or in terms of this thesis *malware features*. For example, the malicious code injection method "Process Hollowing" [65, 68] is considered a malware feature and needs to follow a certain sequence of API function calls to replace memory regions of a benign process with malicious code as described in detail in [65]. For this reason, we rely the definition of dynmx signatures mainly on sequences of API function calls in order to characterise and detect malware features. In addition, the usage of operating system resources which has been derived from the function log as access activity model can be incorporated into the signature.

For defining such signatures, we introduce an external DSL. As the DSL needs to be parsed in order to be executed by an execution engine on the target platform as described in Section 2.7 (Figure 2.12) it should be machine-readable. We therefore leverage the well-known and, in practise, well adopted general data representation syntax YAML. In this case, we choose YAML over JSON as signatures should be concise by definition. Hence, the efficiency factor described in Section 3.3 is negligible but the human-readability is all the more important to make the domain expert's life easier in understanding and working with the signatures. In the upcoming Section 3.5.1 we present the basic structure of dynmx signatures and give a first overview of the possibilities provided by the DSL. We refine this first overview in Section 3.5.2 by presenting the DSL features in detail. In this section, the different language constructs that can be used to define behavioural malware characteristics provided by the DSL are in focus. The way the signatures defined in the DSL are parsed and passed to the detection algorithm is treated in Section 3.5.3.

3.5.1 Basic Structure

As YARA is an already well-adopted signature language in the malware analysis domain, we adopt the structure of YARA rules in the design of the DSL. Hence, as illustrated in Figure 3.6, a dynmx signature is organised in a defined hierarchy and can be roughly separated in three sections. While the first *meta* section provides meta data like a signature name, a description and an author, the second section defines the actual *detection*. In the third section the condition of the signature is defined. The detection section, consists of independent *detection blocks* where each block is identified by a unique *detection block key*. Independent means in this context that each detection

block is evaluated for its own in the detection process (please refer to the description of the detection algorithm in Section 3.6 for detailed information). We break down the detection block into individual *detection steps*. Generally, each individual detection step in the block needs to be detected in order to consider the detection block detected. Thus, the relationship between the detection steps within a block can be thought of as a boolean AND connective. The detection section basically corresponds to the `strings` section of a YARA rule. In order to define when a signature is considered detected, a condition needs to be defined. This condition relates the independent detection blocks from the detection section by using simple boolean connectives (conjunction, disjunction and negation). To reference the detection blocks in the condition the detection block key is used. The condition also defines how the detection blocks should be handled in the detection process.

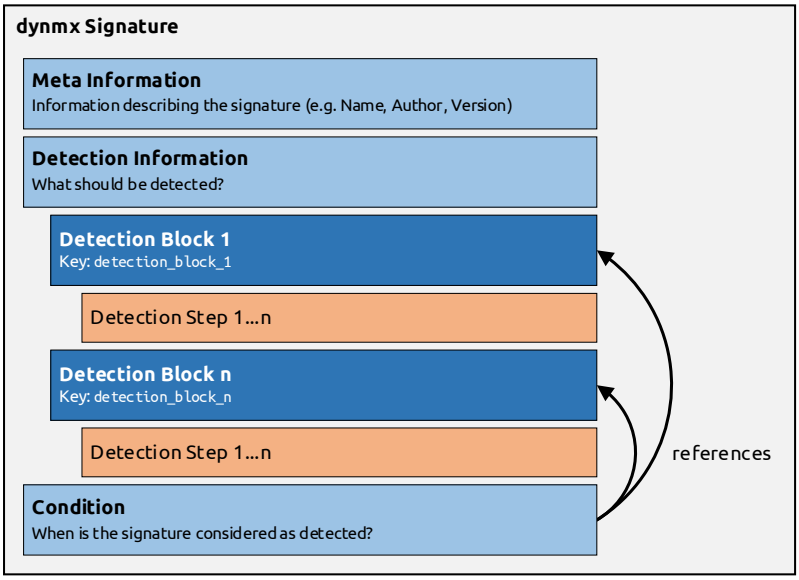

Figure 3.6: Basic structure of a dynmx signature

After we described the basic structure of a dynmx signature in an abstract way, we will apply this abstract description to an example signature. Hence, we leave the abstract description layer and focus on the technical representation. As already introduced before, dynmx signatures are defined in a DSL that leverages YAML as general data representation syntax. Thus, the signature, at first hand, has to comply with the YAML specification referenced in [12]. Furthermore, the signature has to comply with the basic structure shown in Figure 3.6. By combining these specifications, the dynmx signature has the format shown in the minimal example in Listing 3.10. As seen in the example, every signature begins with the YAML node `dynmx_signature`. The three basic sections containing meta information, the detection and the condition as described before, are child nodes of this first YAML node. In this example, one detection block with the key `example_detection_block` is defined. This detection block consists out of two detection steps. Since the steps have to be parsed and processed in a fixed order they are defined as list. In this minimal example, function logs containing processes that call the `WriteProcessMemory` API function after the `CreateProcessA` function are detected. In the condition, it is defined that the API calls have to occur in a sequence with the statement `as sequence`. Furthermore, the condition tells us that the signature is considered detected if the detection block `example_detection_block` is successfully detected. Even though this signature is a minimal example, it is derived from a signature detecting process hollowing. The complete signature for detecting process hollowing is found in the digital appendix of this thesis (please refer to Section 4.2).

```
1 dynmx_signature:
2     meta:
```

```
3           name: example_signature
4           title: Example signature
5           description: Example signature description
6           author: Simon Jansen
7           version: 0.1
8       detection:
9           example_detection_block:
10              - api_call: "CreateProcessA"
11              - api_call: "WriteProcessMemory"
12      condition: example_detection_block as sequence
```

Listing 3.10: Minimal example of a dynmx signature

3.5.2 DSL Features

As we are now aware of the basic signature structure and introduced a minimal signature example, we proceed by describing certain features and capabilities of the DSL in this section. Generally, these features are language constructs allowing the domain expert to express certain malware characteristics precisely. Please consider that the examples given in the following sections are only excerpts and thus not represent complete valid dynmx signatures.

API Call Detection

One of the central features of the designed DSL is the definition of API function calls as detection steps. Our aim in designing the detection of API function calls has been to allow the domain expert to define the detection as precise as needed. As shown in the example signature in Listing 3.10 lines 10 and 11, the most basic definition of an API function call is the definition of the function name. The function name can also be a regular expression or an array of valid function names. This basic definition of the API function by name may be further refined by using the `with` keyword. Generally, the `with` keyword is used to define certain parameter and return values that must be met in order consider the step detected. In the example shown in Listing 3.11, the API call with the function name `VirtualAllocEx` or `VirtualAlloc` should be detected if the value of the parameter `flProtect` is equal to `0x40`. Furthermore, the return value of the function needs to be unequal to zero. With reference to the Windows API documentation, this definition detects the successful allocation of a memory page with the permission read, write and execute [87].

```
1  - api_call: ["VirtualAllocEx", "VirtualAlloc"]
2    with:
3        - argument: "flProtect"
4          operation: "is"
5          value: 0x40
6        - return_value: "return"
7          operation: "is not"
8          value: 0
```

Listing 3.11: Refined API function call detection using the `with` keyword

As shown in the listing, several with-conditions can be defined for a single detection step. The list of with-conditions is handled in the subsequent detection process as they are related to each other with the boolean AND connective. To compare the parameter value from the input function log with the target value defined in the signature, several operations can be used. Table 3.6 gives an overview of the operations that can be used in dynmx signatures. Similar to the specification of the API function name, the parameter name can also be defined as list or as regular expression in order to consider different attribute names in a single with-condition. The same is also applicable to the specification of the target value for a parameter with the difference that regular expression patterns can only be matched using the `regex` operator. This makes the signature, on the one hand, more concise

as different valid parameter and value combinations can be defined in a single with-condition. On the other hand, the definition gets more complex as different detection combinations must be considered. Generally, there are $l \cdot m \cdot n$ valid detection combinations where l is the number of valid API function calls, m the number of valid parameter names and n the number of valid target values.

As seen in line 6 of Listing 3.11, the return value of the function call is referenced with the abstract name `return`. We introduced this abstract name `return` since the return value is anonymous and can also contain a memory pointer with further arguments. In order to reference a specific parameter of a returned memory pointer the name can be changed to the parameter name that should be detected. Basically, arguments of memory pointers can be directly referenced by separating arguments of different hierarchy with colons. For example, the parameter name `name:sin_addr` addresses the `sin_addr` parameter in the `name` memory pointer.

Operation	Description
`is`	Parameter value is equal to target value
`is not`	Parameter value is unequal to target value
`is greater`	Parameter value is greater than the target value
`is less`	Parameter value is less than to target value
`flag is set`	Target binary flag is set in the parameter value
`flag is not set`	Target binary flag is not set in the parameter value
`contains`*	Parameter value contains target value
`contains not`*	Parameter value does not contain target value
`startswith`*	Parameter value starts with target value
`startswith not`*	Parameter value does not start with target value
`endswith`*	Parameter value ends with target value
`endswith not`*	Parameter value does not end with target value
`regex`*	Parameter value matches the target regular expression pattern

Table 3.6: Valid operations in dynmx signatures (*) only applicable for string values)

Resource Detection

In order to detect resources accessed by a certain process, the domain expert can define *resource detection steps*. These detection steps are not directly detected in the function log object but instead in the access activity model that has been deduced from the function log as described in Section 3.4. The definition of a resource detection step generally follows the example of defining API call detection steps. As shown in Listing 3.12, the domain expert defines the resource category (mandatory information, can be `filesystem`, `registry` or `network`) at first hand and can additionally define the access operation as well as further conditions on resource attributes like the location. Please refer to Section 3.4.1 Table 3.5 for available attributes per resource category. The evaluation of the defined with-conditions is the same as for API call detection steps. Hence, all with-conditions must be met in order to consider the detection step detected. In the example shown in Listing 3.12, the resource detection step is used to detect the execution of `*.tmp` files.

```
1 - resource:
2   category: "filesystem"
3   access: "execute"
4   with:
5       - attribute: "location"
6         operation: "endswith"
7         value: ".tmp"
```

Listing 3.12: Signature definition to detect a file resource

Store Directives

Store directives are helpful if a specific argument value or return value of a previously detected API function call should be incorporated in downstream detection steps. This behaviour is especially desirable if we think of operating system resources. As described in detail in the background Section 2.4.2 on resources, processes need to obtain a valid handle in order to work with resources. Consequently, this resource handle needs to be tracked in the course of the signature detection in order to be able to detect and correlate operations on the same resource. As the resource handle is dynamically obtained by the process during runtime and is thus not known during the signature definition, we need to dynamically store and reference the handle in the detection. Generally, stored variables are only valid within the context of a detection block.

```
1  - api_call: ["CreateProcessA", "CreateProcessW"]
2    with:
3        - argument: "dwCreationFlags"
4          operation: "flag is set"
5          value: 0x4
6    store:
7        - name: "hProcess"
8          as: "proc_handle"
9        - name: "hThread"
10         as: "thread_handle"
11 - api_call: "NtUnmapViewOfSection"
12   with:
13       - argument: "ProcessHandle"
14         operation: "is"
15         value: "$(proc_handle)"
```

Listing 3.13: Storage and reference of variables in signatures

Listing 3.13 demonstrates the storage as well as the reference of stored values based on the signature used to detect the process hollowing malware feature. In the first detection step a process created in suspended mode. The second detection step is used to unmap a memory section of this created process. This is needed to write malicious code to the created process at the correct offset. With reference to lines 6-10 of the listing, the process handle as well as the thread handle returned by the API function call for process creation is stored. While the `name` node defines the name of the parameter that should be stored, the `as` node specifies the variable name that can be referenced in the subsequent detection steps (`proc_handle` and `thread_handle` in this case). Line 15 of the listing shows how to reference a stored parameter by using the defined variable name. In this case, the variable is used to keep track of handles in order to ensure that the API calls of the sequence operate on the same resource.

Path Variants

As briefly introduced in the background Section 2.4.1, the API provided by Microsoft Windows is comprehensive and due to the large scope complex. One of the main drivers of the complexity is the mixture of low-level and high-level API functions with a vast variety of function parameters [155]. As a consequence of this mixture and complexity, there exist multiple ways in the API to achieve the same functionality. For example, copying a file can be achieved, on the one hand, by reading a file using `ReadFile` and writing this read file to a new location using `WriteFile`. On the other hand, the copy operation can be performed using the higher-level function `CopyFile`.

To cope with the complexity offered by the Windows API, we designed the possibility to define *path variants* within a detection block. In terms of this work, path variants are defined as disjunct detection paths or sequences. The path variant feature helps the domain expert to define different API call sequences for the same operation within a detection block. Basically, disjunct detection paths can also be defined using different detection blocks connected to each other using the boolean

OR connective in the condition. But this way of definition leads to unnecessary long and complex signatures, especially if the path variant occurs in the middle of a longer sequence. In this case, the subsequence occuring before and after the path variant needs to be copied to the other detection block in order to keep the overall sequence that should be detected. With the path variant DSL feature, the disjunct detection paths can be defined exactly where they are needed in the overall sequence. This basically adopts the well-established *don't repeat yourself* (DRY) programming principle.

We will give an example in order to illustrate this issue and emphasize the advantage of the path variant feature. For the sake of brevity, this example is shortened and kept simple. If we think of a simple malware downloader, this kind of malware downloads and executes malicious code from a network operation. Network connections to remote endpoints can be implemented using different function sequences. On the one hand, a network connection can be established using Berkeley compatible sockets with the API functions `socket` and `connect` from the Winsock DLL `Ws2_32.dll`. On the other hand, the higher-level WinINet API from the DLL `Wininet.dll` can be used to establish a network connection [146]. We assume that our sample downloader first creates a mutex to indicate that it was already executed on the system, then downloads the sample from the Internet and afterwards runs the downloaded malicious code as a new process. The sample detection block detecting this kind of downloader and considering the two API call sequence variants to establish a network connection is defined in Listing 3.14.

```
detection:
  simple_downloader:
    - api_call: "CreateMutex[AW]"
    - variant:
      - path:
        - api_call: "socket"
        - api_call: "connect"
      - path:
        - api_call: "InternetOpen[AW]"
        - api_call: "InternetConnect[AW]"
        - api_call: "HttpOpenRequest[AW]"
    - api_call: "CreateProcess[AW]"
condition: simple_downloader as sequence
```

Listing 3.14: Detection block with path variants

As shown in the listing, the path variant feature can be used with the keyword `variant`. Every valid detection path is then prefaced with the `path` keyword. While the first path variant in the sample detection block detects the establishment of a network connection based on Berkeley compatible sockets, the second path variant takes care of network connections implemented with the WinINet API. The advantage of the definition using the path variant DSL feature compared to the definition of separate detection blocks in this specific example is that the first and last API call of the sequence do not have to be defined redundant. The same signature defined by using separate detection blocks is shown in Listing 3.15 for comparison.

```
detection:
  simple_downloader_a:
    - api_call: "CreateMutex[AW]"
    - api_call: "socket"
    - api_call: "connect"
    - api_call: "CreateProcess[AW]"
  simple_downloader_b:
    - api_call: "CreateMutex[AW]"
    - api_call: "InternetOpen[AW]"
    - api_call: "InternetConnect[AW]"
    - api_call: "HttpOpenRequest[AW]"
    - api_call: "CreateProcess[AW]"
```

```
13 condition: simple_downloader_a as sequence or ↩
       simple_downloader_b as sequence
```

Listing 3.15: Detection block with path variants

Condition

The condition section of the signature is used to define the exact state under that the signature is considered detected by relating the detection blocks with boolean connectives. This design is equivalent to YARA conditions where different string identifiers are related to each other in a boolean expression that defines when a rule matches. Detection block keys are used as identifiers in the condition of dynmx signatures. Examples for valid conditions are already given in the previous section, for example in Listing 3.15. Generally, the basic boolean connectives AND, OR and NOT can be used in the condition to relate detection blocks. The precedence of the boolean operators follows thereby the generally accepted precedence where NOT has the highest precedence, followed by the AND and OR connective with the lowest precedence [151]. The Equation 3.1 exemplifies the precendece in boolean expressions by using parenthesis to explicitly define the evaluation order.

$$\begin{aligned} R &= \neg a \wedge b \vee c \wedge d \\ &= ((\neg a) \wedge b) \vee (c \wedge d) \end{aligned} \tag{3.1}$$

Of course the precedence has to be considered in the definition of the signature condition to get the expected detection behaviour. By allowing the domain expert to use parenthesis in the condition the order of evaluation can be refined and changed in addition to the precedence of the operators. Thus, complex boolean expressions can be used to carefully define under which specific circumstances a certain signature is considered detected.

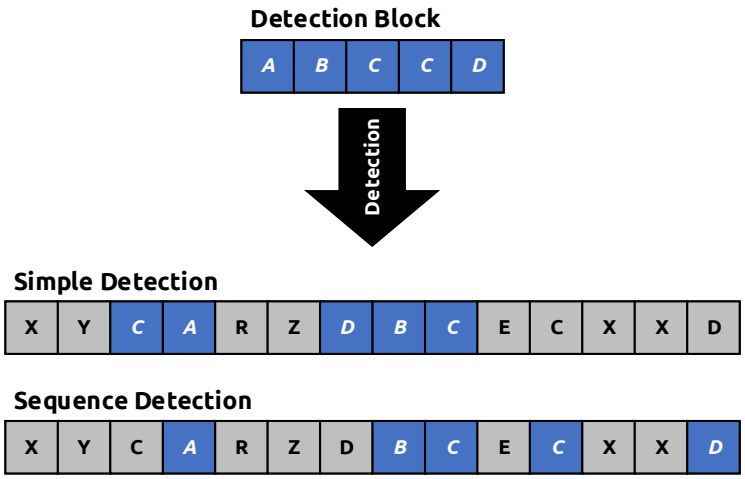

Figure 3.7: Abstract illustration of the detection type

Furthermore, the condition is used to define how the detection blocks should be handled in the subsequent detection process also referred to as the *detection type*. To define the detection type, the keyword `as` is used after each detection block key referenced in the condition. Basically, the following detection types can be used in the condition.

`simple` The steps in the detection block have to be present in the function log independent from the order they occur in. Hence, the `single` detection type only detects the presence of individual steps.

`sequence` The steps in the detection block should be handled and detected as sequence. This means that each step has to be present after its predecessor in the function log.

The detection type is a mandatory information that has to be defined in the condition. Hence, the detection block key always has to be followed by the `as` keyword and the detection type in the condition. As the `sequence` detection type is the more specific one, it is obvious that the same detection block would have also been detected with the type `simple` if it wdetected with the `sequence` detection type. Figure 3.7 illustrates the basic principle of the detection types in an abstract form. For the sake of simplicity, we define the detection steps as well as the API call sequence of the function log as capital letters.

3.5.3 Signature Parser

As briefly introduced in Section 3.1 (Figure 3.1), the signature parser is needed to parse the text-based dynmx signatures. With reference to the background section on DSLs (Section 2.7, Figure 2.12), the signature parser is part of the interpreter as it prepares the signature data for the detection algorithm described in Section 3.6. Generally, the aim of the signature parsing process is to provide a signature object that can be efficiently and effectively used by the detection algorithm.

Since we have chosen the machine-readable data representation syntax YAML as basis for the DSL, parsing the basic structure of the signature is trivial as we can leverage a standard YAML parser[4]. The result of this basic YAML parsing process is a dictionary containing the key value pairs of the text-based input signature. After parsing the basic structure, we perform basic sanity checks to ensure that all needed sections are part of the signature and basic meta data is defined. Further, a valid dynmx signature needs to have at least one detection block with at least one detection step. After the basic sanity checks are passed, we continue with parsing the detection blocks as well as the condition. As the detection and condition section of the signature are highly relevant for the detection algorithm, we transform and parse them to a format that is efficient for the detection. These steps are described in detail in the following sections. Figure 3.8 gives an overview of signature parsing process.

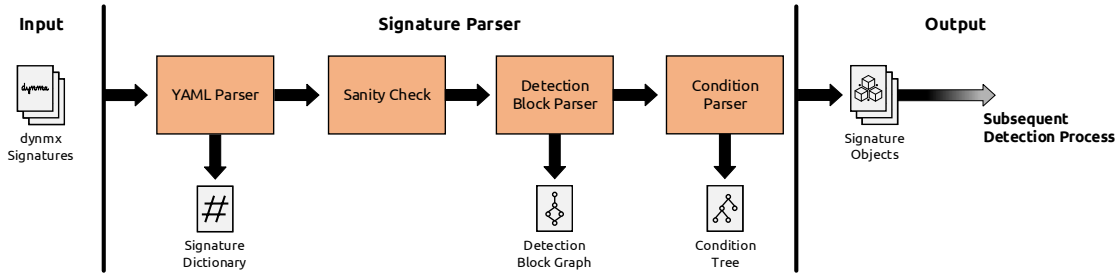

Figure 3.8: Signature parsing overview

Detection Block Transformation

Based on the introduction of the DSL in the previous sections, we know that each detection block is basically independent from other detection blocks in the signature despite they are brought into relation in the condition. But from the perspective of the detection, each detection block is detected individually as described in detail in Section 3.6 on the detection algorithm. Further, we know that each individual detection block consists out of several ordered detection steps (see Figure 3.6). So, the basic idea behind the detection block transformation is that we can imagine the detection block as a graph as illustrated in Figure 3.8 (output of the detection block parser). More formally, we define the *detection block graph*, based on the general definition of a graph [22], as a set of vertices $V = \{v_1, v_2, ...\}$ where each vertex represents a detection step together with a set of edges $E = \{e_1, e_2, ...\}$ that connect adjacent detection steps. The adjacency of detection steps is simply

[4]In our case, we leverage the YAML parser PyYAML [117]

derived based on the sequence in that they are defined in the detection block. For example, if the detection step v_2 is defined right after the detection step v_1 in the detection block, the detection steps are considered adjacent and are thus connected by an edge. Hence, an edge is defined as a tuple of adjacent detection steps $e_n = (v_x, v_y)$. As the detection steps are ordered in a sequence in the signature, we can further define that the detection block graph is directed [22] meaning that each edge has a direction. Since our DSL provides no feature or keyword to define cycles in signatures like jump statements or loops, we can further deduce that the detection block graph is acyclic. Consequently and more precisely, the detection block graph is a directed acyclic graph (DAG) representing the order of detection steps in a detection block.

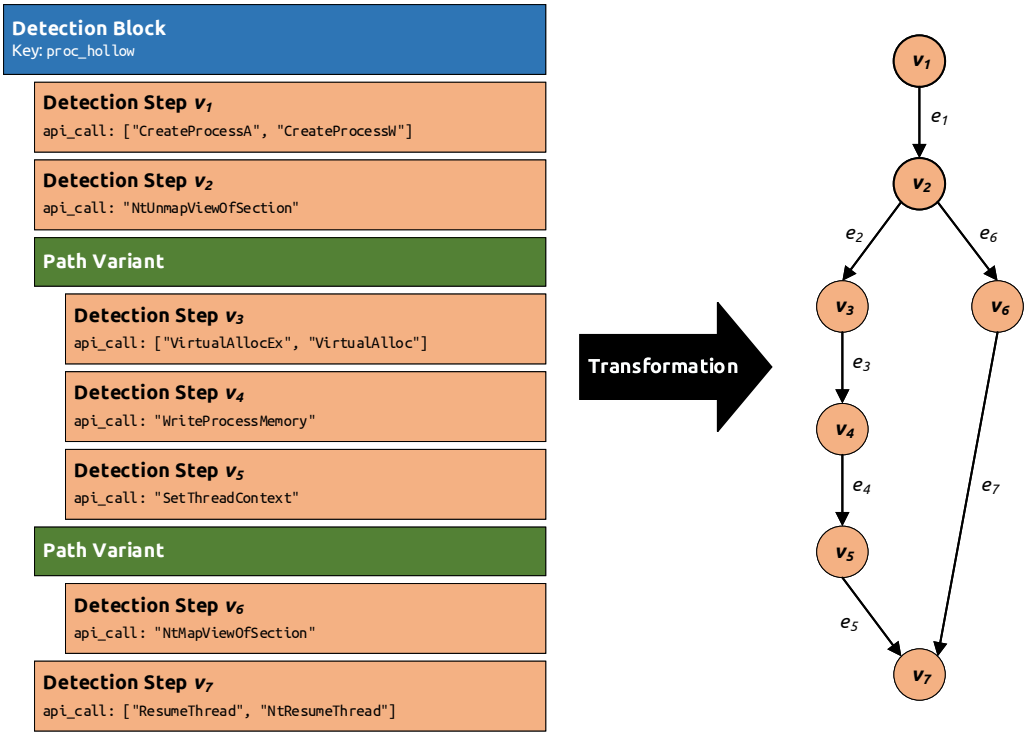

Figure 3.9: Detection block graph transformation

In order to demonstrate the transformation of a detection block to a detection block graph, we leverage a simplified example detection block shown in Listing 3.16 used to detect process hollowing. Please note, that this example is a more detailed version of the signature shown in Listing 3.10. Following the definition of the detection block graph, each detection step represents a vertex in the graph (see left side of Figure 3.9). Therefore, we have a set of vertices V with a cardinality of $|V| = 7$ ($V = \{v_1, ..., v_7\}$). In order to define the edges of the graph, we have to consider that the detection block contains two path variants. As defined in Section 3.5.2, path variants represent disjunct detection paths in the detection block. To represent path variants in the graph, we need to add a junction. As illustrated in Figure 3.9, the detection step v_2 has the two outgoing edges e_2 and e_6 representing the junction and thus the two detection paths. If we follow edge e_2 of the graph, the following vertices v_3 to v_5 represent the first detection path (lines 6-9 in Listing 3.16). The same is applicable for the second detection path that can be followed using the edge e_6 at the junction. Generally, junctions in the detection block graph happen before and after the path variants. Of course, path variants can be nested leading to more complex graphs.

```
1 detection:
2   proc_hollow:
3     - api_call: ["CreateProcessA", "CreateProcessW"]
4     - api_call: "NtUnmapViewOfSection"
5     - variant:
```

```
6         - path:
7             - api_call: ["VirtualAllocEx", "VirtualAlloc"]
8             - api_call: "WriteProcessMemory"
9             - api_call: "SetThreadContext"
10        - path:
11            - api_call: "NtMapViewOfSection"
12    - api_call: ["ResumeThread", "NtResumeThread"]
```

Listing 3.16: Simplified detection block to detect process hollowing

Condition Parsing and Transformation

Analogous to the detection block transformation, we pursue the aim to provide the condition in a suitable format that can be efficiently used in the downstream detection process by parsing and transforming the same. Hence, the condition parser performs a two-step process consisting of the parsing as well as the transformation of the condition. As for the first step and with reference to Section 3.5.2, the condition of a dynmx signature contains three major pieces of information:

1. the detection block keys,
2. the detection block type defining how a detection block should be detected and
3. binary operators relating the detection blocks and thereby defining the circumstance under which the signature is considered detected.

In order to parse this information from the condition string, we use a simple tokeniser. Generally, a tokeniser or lexer splits an input string into semantically related units – the tokens. Applied to our signature condition, the semantically related units are the detection block keys together with their type. The second relevant tokens are the binary operators.

Tree data structures and more precisely *binary expression trees* [62] have become an established representation of textual binary expressions, which also include the condition used in our signatures. Binary expression trees are especially beneficial in terms of the evaluation as the tree represents the operations of the expression in an already ordered form. Thus, we transform the parsed signature condition to a binary expression tree – the so called *condition tree* in terms of this thesis (see Figure 3.8). This condition tree is evaluated in the end of the detection process as further described in Section 3.6.3.

In the following, we demonstrate the condition parsing process using the following example condition expression.

```
1 condition: key_1 as sequence AND (key_2 as simple OR key_3 as↩
      sequence) AND NOT key_4 as sequence
```

Listing 3.17: Example condition

As illustrated in Figure 3.10, the condition expression is first parsed using the tokeniser. Based on our token definition stated before, this tokenisation process results in 8 tokens. These tokens are then transformed to the binary expression tree. As seen in Figure 3.10, this tree generally consists of two node types. While the operator nodes (green) represent the boolean connectives, the condition leaf nodes (orange) represent the basic operands. The condition leaf nodes contain the detection block key as well as the detection type. The illustration of the binary expression tree shown in the figure makes the evaluation order obvious. Please consider, that the usage of parantheses changes the default operator precedence in this particular expression. Otherwise, the boolean AND connective would have a higher precedence resulting in a different binary expression tree and thus to a different evaluation result.

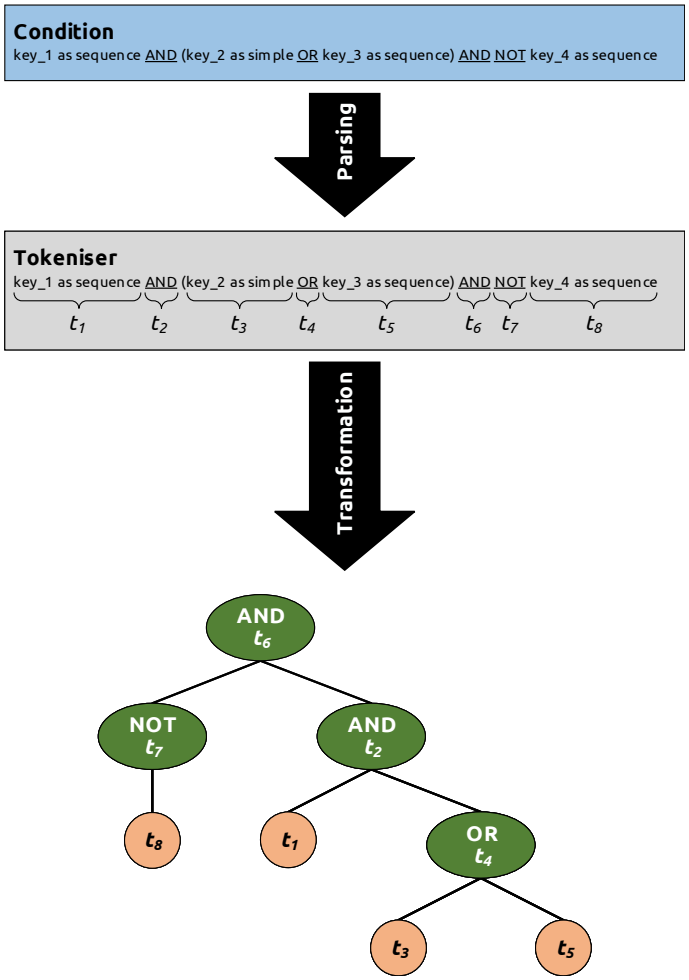

Figure 3.10: Condition parsing and transformation

3.6 Detection Algorithm

As we have described the major preparation and pre-processing steps in the previous sections according to the detection process overview in Section 3.1, we have imparted the mandatory knowledge to finally present the main detection process step – the detection algorithm. As introduced in Figure 3.1 and the previous sections, the three main inputs for the detection algorithm are

1. the function log objects derived from the parsed input function logs (Section 3.2),
2. the access activity models deduced from the function log object representing the resources of a certain process and
3. the signature objects derived from the parsed dynmx signatures defining what malicious behaviour should be detected and how it should be detected (Section 2.7).

The output of the detection algorithm is the detection result. Besides the information which signature matched in which function log, the detection result also provides detailed information where certain steps of the signature have been detected enabling the domain expert to exactly understand why the detection happened. The responsibility of the detection algorithm is obviously to detect the defined detection steps of a certain dynmx signature in a certain function log. Hence, according to the provided background on DSLs (Section 2.7), the detection algorithm forms the second part of the interpreter that executes the DSL on the target platform. Figure 3.11 gives an

overview of the individual detection algorithm steps.

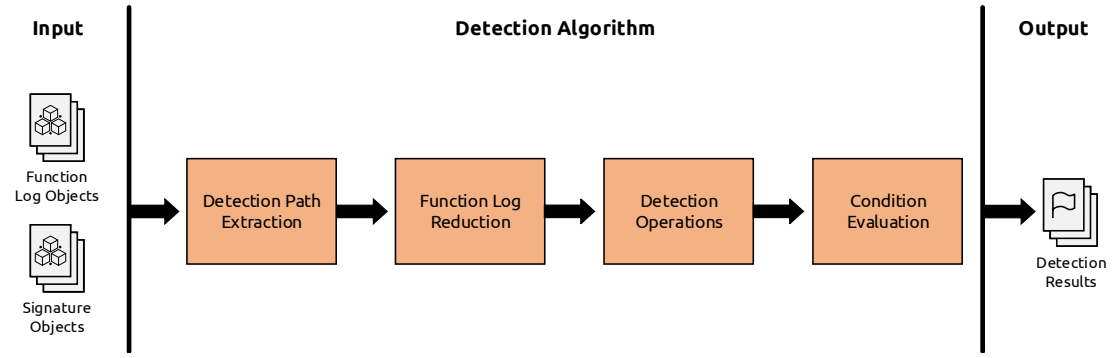

Figure 3.11: Detection algorithm overview

In the following, we first present the mode of operation of the detection algorithm in Section 3.6.1 to give an overview of the algorithm functionality. In this section, we also present the detection procedure for the `simple` detection type. As the input function logs can reach file sizes up to 100 MB and more, the detection algorithm has to be efficient in terms of matching API call sequences. Therefore, we leverage the well-known *Longest Common Subsequence* (*LCS*) algorithm and adapt it to our needs. This adaptation as well as the detailed description of the identification and matching of relevant API call sequences is the focus of Section 3.6.2. We conclude the detection algorithm by explaining the signature condition evaluation (Section 3.6.3).

3.6.1 Mode of Operation

Following the information provided in Section 3.1, the detection algorithm is designed process-centric. This generally means, that the dynmx signature is detected individually in each process of the input function log. Therefore, a certain malware feature that is realised in cooperation of several processes leveraging for example IPC cannot be detected using our proposed algorithm. We have consciously chosen the process-centricity since the input function logs provide the chronologically ordered API function calls categorised by the process that caused the function calls. As the precision of the timestamp information that is provided together with each individual API function call in the function log is not sufficient, we cannot reconstruct the real chronological order of API function calls beyond the process limit. For clarification, please consider the example function log given in Listing 3.18. The listing contains excerpts of two processes originating from the same function log. The first process contains three API function calls with the same timestamp `19.991`. As the sandbox knows the exact order in that the API function calls happened, we can assume that the function call order within a certain process is consistent. To a certain extend, this can also be deduced from the sequence in this example as the `GetCurrentProcess` returning a handle to the current process has to happen before the `TerminateProcess` function call that uses the returned handle. But beyond the scope of a process, we cannot bring the API function calls in consistent order since the second process of the function log also contains function calls with the exact same timestamp. So as a conclusion, if we would like to detect malware features beyond the limit of processes, the timestamp precision needs to be improved or a counter that reflects the consistent overall API function call sequence has to be introduced in the input function log.

```
1 Process:
2     id = "1"
3     image_name = "1de7834ba959e734ad701dc18ef0edfc.exe"
4     [...]
5
6     [0019.991] CloseHandle (hObject=0x5e4) returned 1
7     [0019.991] GetCurrentProcess () returned 0xffffffff
```

```
8       [0019.991] TerminateProcess (hProcess=0xffffffff, ↩
            uExitCode=0x0)
9
10  Process:
11      id = "2"
12      image_name = "svchost.exe"
13      [...]
14
15      [0019.991] GetProcAddress (hModule=0x77c10000, lpProcName↩
            ="NtReadVirtualMemory") returned 0x77c562f8
16      [0019.991] GetCurrentProcessId () returned 0x9f0
```

Listing 3.18: Example function log with ambigious timestamps

In addition to the process-centricity of the detection algorithm, we know from Section 3.5.1 that each detection block of the signature is detected independently. Consequently, the total set of detections D that have to be made to detect a certain signature in a certain function log is the cartesian product $D = P \times B$, where $P = \{p_1, ...p_n\}$ represents the set of processes from the input function log and $B = \{b_1, ...b_n\}$ represents the set of detection blocks defined in the signature. Figure 3.12 illustrates this detection relationship between detection blocks and processes and thus reflects the cartesian product D.

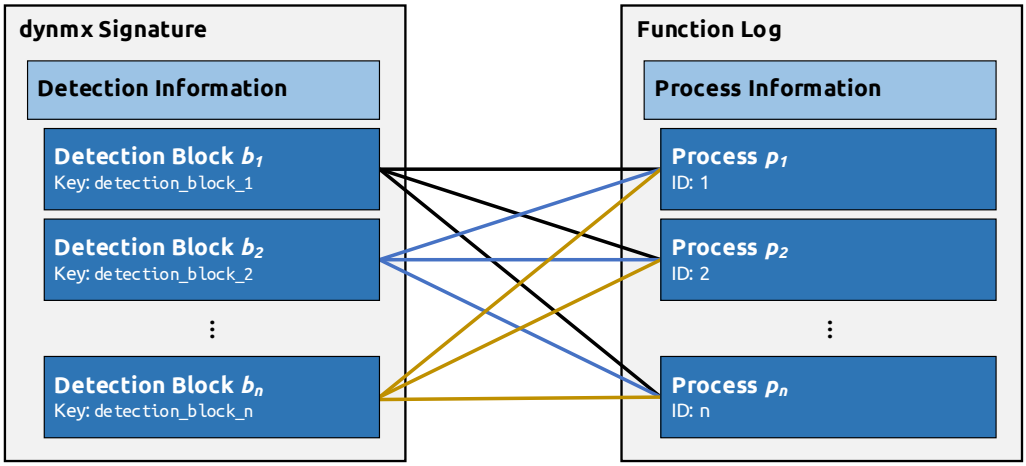

Figure 3.12: Detection relationship between detection blocks and processes

Detection Path Extraction

As we are now aware of the high-level mode of operation of the algorithm, we continue to describe the course of a single detection operation, i.e. the detection of a certain detection block in a certain process. Referring to the introduction of the detection block graph in Section 3.5.3, the graph represents all possible detection paths of a block. We consider a block detected in terms of this thesis, if at least one of the possible detection paths is found in the API function call sequence of the process. Therefore, we first need to extract the possible detection paths from the graph and store them as sequences (lists) of consecutive detection steps. From an algorithmic perspective, in order to find all possible paths of the detection block, the graph is walked completely from the start to the end vertex. At every junction vertex, new detection paths are identified since we can walk different paths to reach the end vertex of the graph. Figure 3.13 illustrates the detection path extraction based on the example detection block graph introduced in Figure 3.9. In the illustrated example graph, beginning from the start vertex v_1, the end vertex v_7 can be reached using two distinct paths.

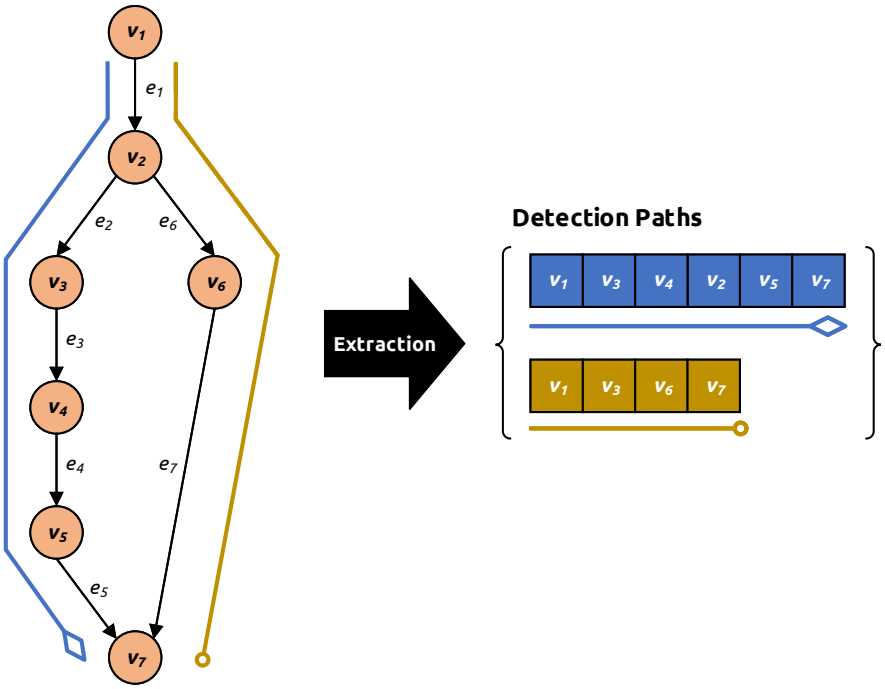

Figure 3.13: Extraction of detection paths from the detection block graph

Function Log Reduction

Based on these extracted detection paths we can determine the relevant API function call candidates. Please consider that the function log reduction is only applicable for detection steps that are used to detect API calls rather than resources. As each detection step from the path that is used to detect an API function call has to define at least the API function name, we can extract a list of relevant API functions from the detection path. At this point of time, this is just a list of possible API function call candidates since the detection step can include further constraints in addition to the function name like specific parameter values in a comprehensive with-condition. With the help of this candidate list, in turn, we are able to reduce the API call sequence of the processes from the function log to only relevant calls based on their function name. This reduction of the process' API call sequence sustains the aim to develop an efficient detection algorithm as the length of sequences that have to be matched is decreased. To make the reduction even more efficient, we build a trivial lookup table of API function calls for each process during the function log parsing process. This lookup table keeps track of the API functions called by the process and their position in the API call sequence of the process. Hence, we can identify API function call candidates and their index in the sequence by leveraging the lookup table instead of iterating over the whole sequence. According to the common data model defined in Section 3.2.3, each parsed API function call contains its index as attribute. Based on this index, we can reconstruct the original order of function calls in the reduced sequence.

Path Detection for the `simple` Detection Type

Finally, after the function log reduction has taken place, we can detect each individual step of the detection path in this reduced API function call sequence. At this point in the detection process, we have to consider the detection type (see Section 3.5.2) defined for the block. As the detection for the `sequence` detection type leverages an adapted version of the LCS algorithm which is described in the upcoming Section 3.6.2, we will focus on the `simple` detection type in this section. Nevertheless, the definition of a successful detection applies to both of the detection types. Thus, generally, we define that a detection path is considered detected if each step of the path has been detected in the

API call sequence and resources. Furthermore, we consider the overall detection block detected if at least one of the valid detection paths has been successfully detected.

We separate the detection path into two separate paths – one path for detecting API calls and one path for detecting resources. As for the API call detection path, we take the first step from the detection path and iterate over the reduced API call sequence of a certain process starting from the beginning of the sequence. In order to find a matching API call for a certain step, two criteria has to be met:

1. the API function name of the call has to match and
2. the API function call has to met all with-conditions defined in the step.

Thereby, all parameter names defined in the with-conditions are detected in all nested pointer structures unless the parameter name is explicitly referenced with a colon-separated notation as introduced in Section 3.5.2. As the operation (see Table 3.6) as well as the expected parameter value need to be defined in the with-condition, the actual matching process is trivial and straightforward. As soon as the first detection step is successfully detected in the API call sequence, the same process is applied for the subsequent steps of the detection path. With reference to the definition of the `simple` detection type in the aforementioned Section 3.5.2, only the presence of the detection block steps in the API call sequence is of relevance. Hence, we always start from the beginning of the API call sequence to find a matching API call for a certain detection step. Formally, we define the path detection result R_{sim_path} for the reduced function log $X = \{x_1, ..., x_m\}$ and the detection path $Y = \{y_1, ..., y_n\}$ as shown in Equation 3.2.

$$R_{path\_sim} = \begin{cases} \text{true}, & \text{if } \forall y \in Y, \forall x \in X : \texttt{detect}(x, y) = \text{true}, \\ \text{false}, & \text{if } \exists y \in Y, \forall x \in X : \texttt{detect}(x, y) = \text{false} \end{cases} \tag{3.2}$$

Thereby, we introduce the `detect` function that matches a detection step with an API function call based on the two criteria introduced before. Formally, the function is defined as shown in Equation 3.3.

$$\texttt{detect}(x, y) = \begin{cases} \text{true}, & \text{if x is detected by y}, \\ \text{false}, & \text{if x is not detected by y} \end{cases} \tag{3.3}$$

As for the resource detection path, the detection follows the same schema. We iterate over the detection steps used to detect resources and try to find a matching resource in the access activity model of the process. The matching process itself is straightforward as we only have to identify the access operation and the attribute values defined in the signature.

3.6.2 Longest Common Subsequence Algorithm Adaptation

Similar to the `simple` detection type, we separate the detection path into the API call and the resource path based on the detection step type. But in contrast to the `simple` detection type, the order of detected API function calls is of relevance for the detection of blocks that are of the `sequence` type. In order to match the sequence of steps in the detection path with the sequence of API function calls in the reduced function log, we leverage the LCS algorithm [13].

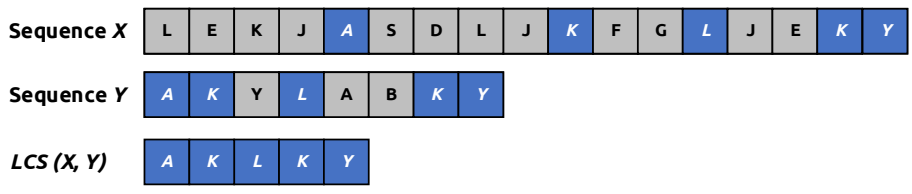

Figure 3.14: Sample LCS calculation based on two input strings

Typically, the LCS algorithm finds its application in the domain of molecular biology for comparing and matching DNA or protein sequences [129, 145]. But the LCS algorithm or related forms of this algorithm have also been successfully applied for matching API call sequences in the malware research domain as presented, for example, in [23, 58].

Generally, the LCS between two sequences is defined as the sequence of maximum length that occurs in both of the input sequences [13]. For example, the LCS of the two strings $X =$ `LEKJASDLJKFGLJEKY` and $Y =$ `AKSLABKY` is $LCS(X, Y) =$ `AKLKY` (see Figure 3.14). Based on this example, it is obvious that the items of the LCS remain the original order but do not have to be next to each other in the input sequence. Further, in contrast to this example, it is noted that there can be multiple LCS for certain input sequences. More formally, for two strings X and Y the traditional LCS algorithm is defined as shown in Equation 3.4 (based on [58]).

$$
\begin{aligned}
X &= \{x_1, ..., x_m\} \\
Y &= \{y_1, ..., y_n\} \\
X_i &= \{x_1, ..., x_i\} \\
Y_j &= \{y_1, ..., y_j\} \\
LCS(X_i, Y_j) &= \begin{cases} \emptyset, & \text{if } X = \emptyset \text{ or } Y = \emptyset, \\ LCS(X_{i-1}, Y_{j-1}) + \text{common character} & \text{if } x_i = y_j, \\ max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}
\end{aligned} \tag{3.4}
$$

By applying this general definition of the LCS to our problem, we define the reduced function log as input sequence X and the detection path as sequence Y. Consequently, the LCS of these two input sequences $LCS(X, Y)$ is defined as the longest subsequence of detection steps that wdetected in the reduced function log. In order to reasonably apply the LCS algorithm to our problem, we need to adapt the traditional algorithm to our needs. First of all, we cannot match the two sequences by simply checking for the equality of two items in the sequence since detection steps and API function calls are not of the same type. Therefore, we leverage the `detect` function introduced in the previous Section 3.6.1 which is used instead of checking for the equality. Furthermore, the sequences are not interchangeable since we always have to match detection steps to API function calls. If we consider the example given before, Y can be defined with the string of X and X with the string of Y and the LCS will remain the same. Hence, X and Y are interchangeable in the example. This is not applicable to our LCS adaptation. Therefore, we need to strictly define that the first sequence X passed to the algorithm is always the reduced function log and the second sequence Y is the detection path. Based on this adaptations, we introduce the LCS_{API} and formally define it as shown in Equation 3.5.

$$
\begin{aligned}
X &= \{x_1, ..., x_m\} \text{ (reduced API call sequence)} \\
Y &= \{y_1, ..., y_n\} \text{ (detection path)} \\
LCS_{API}(X_i, Y_j) &= \begin{cases} \emptyset, & \text{if } X = \emptyset \text{ or } Y = \emptyset, \\ LCS_{API}(X_{i-1}, Y_{j-1}) + \text{API call} & \text{if } \texttt{detect}(x_i, y_j), \\ max(LCS_{API}(X_i, Y_{j-1}), LCS_{API}(X_{i-1}, Y_j)) & \text{if not } \texttt{detect}(x_i, y_j) \end{cases}
\end{aligned} \tag{3.5}
$$

Following our definition of a successful detection of a certain detection path for the `simple` detection type given in Section 3.6.1, we define for the `sequence` detection type that a certain path is considered detected if the length of the found LCS based on the aforementioned algorithm corresponds to the length of the detection path, i.e. all detection steps of a certain were successfully detected. This is formally defined as intermediate result for the detection path R_{path_seq} in Equation 3.6.

$$
R_{path\_seq} = \begin{cases} \text{true}, & \text{if } |LCS_{API}| = |Y|, \\ \text{false}, & \text{if } |LCS_{API}| \neq |Y| \end{cases} \tag{3.6}
$$

We apply our adapted LCS algorithm based on an example function log and signature in the following. For our example, we assume that we would like to detect the malicious behaviour of creating a file in the `AppData` folder of a Windows profile. Since the `AppData` folder is a popular location for malware to persist [52], such a signature is valuable for incident response purposes. In order to detect such malicious behaviour we define the signature shown in Listing 3.19.

```
1  dynmx_rule:
2    meta:
3      name: appdata
4      title: AppData file creation
5      description: Detection of file creation in the AppData ↩
           folder of a Windows profile
6      author: Simon Jansen
7      version: 1.0
8    detection:
9      appdata_win32:
10       - api_call: "CreateFile[AW]"
11         with:
12           - argument: "lpFileName"
13             operation: "regex"
14             value: "[a-z]:\\users\\.*\\appdata\\.*"
15         store:
16           - name: "return"
17             as: "file_handle"
18       - api_call: "WriteFile"
19         with:
20           - argument: "hFile"
21             operation: "is"
22             value: "$(file_handle)"
23       - api_call: "CloseHandle"
24         with:
25           - argument: "hObject"
26             operation: "is"
27             value: "$(file_handle)"
28   condition: appdata_win32 as sequence
```

Listing 3.19: dynmx Signature to detect file creations in `AppData`

The signature shown in the listing consists of three consecutive steps:

1. Create file in the `AppData` folder (line 10),
2. Write content to the created file (line 18) and
3. Closing the file handle (line 23).

By keeping track of the file handle returned by the `CreateFile` API call in the signature, we ensure that the operations are performed on the same file. Based on this signature, we extract the reduced function shown in Listing 3.20 from a certain input function log. The API calls that will be detected by our signature are already emphasized by a bold font face.

```
1  [0013.571] CreateFileW (lpFileName="\\??\\globalroot\\↩
       systemroot\\system32\\tasks\\40c5fec0" (normalized: "c:\\↩
       windows\\system32\\tasks\\40c5fec0"), dwDesiredAccess=0↩
       x1f01ff, dwShareMode=0x7, lpSecurityAttributes=0x0, ↩
       dwCreationDisposition=0x3, dwFlagsAndAttributes=0x0, ↩
       hTemplateFile=0x0) returned 0x130
2  [0013.575] WriteFile (in: hFile=0x130, lpBuffer=0x230000*, ↩
       nNumberOfBytesToWrite=0xb02, lpNumberOfBytesWritten=0↩
       x12e6a4, lpOverlapped=0x0 | out: lpBuffer=0x230000*, ↩
       lpNumberOfBytesWritten=0x12e6a4, lpOverlapped=0x0) ↩
       returned 1
```

```
3 [0013.580] CreateFileW (lpFileName="C:\\Users\\user\\AppData\\↩
      Local\\Temp\\\\setup2760884928.exe.manifest" (normalized:↩
       "c:\\users\\user\\appdata\\local\\temp\\setup2760884928.↩
      exe.manifest"), dwDesiredAccess=0x1f01ff, dwShareMode=0x1↩
      , lpSecurityAttributes=0x0, dwCreationDisposition=0x2, ↩
      dwFlagsAndAttributes=0x0, hTemplateFile=0x0) returned 0↩
      x110
4 [0013.581] WriteFile (in: hFile=0x110, lpBuffer=0x1232e80*, ↩
      nNumberOfBytesToWrite=0x1ac, lpNumberOfBytesWritten=0↩
      x12f0ec, lpOverlapped=0x0 | out: lpBuffer=0x1232e80*, ↩
      lpNumberOfBytesWritten=0x12f0ec, lpOverlapped=0x0) ↩
      returned 1
5 [0013.582] CloseHandle (hObject=0x110) returned 1
6 [0013.585] CloseHandle (hObject=0x130) returned 1
```

Listing 3.20: Reduced function log based on the signature to detect file creations in `AppData`

In order to solve the problem of finding the longest sequence between the reduced API call sequence X and the detection path Y based on the proposed LCS_{API} algorithm, we leverage a two step process:

1. Calculate the length of the LCS based on a tabular notation and
2. Reconstruct the actual LCS items based on the tabular.

The rows of the table used for calculating the LCS represent the API call sequence and the columns the steps of the detection path. Hence, we get a table of $m \times n$ size as shown in Figure 3.15.

X \ Y	∅	y_1	y_2	y_3
∅	∅	∅	∅	∅
x_1	∅	0	0	0
x_2	∅	0	0	0
x_3	∅	*1*	1	1
x_4	∅	1	*2*	2
x_5	∅	1	2	*3*
x_6	∅	1	2	3

Figure 3.15: Tabular notation to solve the LCS problem for our example detection

The first row and column in the table are needed to handle the edge case where empty sets are compared. Generally, the table reflects the LCS length for each detection operation. We highlighted the table cells that represent the detection of the next consecutive step of the detection path. After all detection operations have taken place, the length of the LCS is found in the table cell (x_m, y_n). In our example, the length of the LCS is three. Based on our definition of the intermediate detection result in Equation 3.6, the detection path was successfully detected as the LCS length is equal to the detection path length. In order to reconstruct the detected sequence based on the tabular notation, we need to find the highlighted cells by iterating backwards column by column through the table and find the first row in a certain column that has a lower value than the previous cell. Thus, in our example, the reconstructed LCS is $LCS_{API} = \{x_3, x_4, x_5\}$ which corresponds to the emphasized API calls in Listing 3.20.

The resources are detected independent from the LCS algorithm and thus from the API calls. The detection process for resources is the same as for the `simple` detection type described in the previous section.

For signatures that make use of store directives (see Section 3.5.2), finding the LCS is more complex. To illustrate the problem, we give the following example. The detection block shown in Listing 3.21 uses a store directive to keep track of the process handle returned by the API function call `OpenProcess`. This variable is used by subsequent detection steps in order to ensure that all of API calls are related to the same process identified by the process handle. The detection block shown in the listing can be used to detect code injection techniques. The malicious process obtains a handle to an arbitrary process, allocates memory with read, write and execute permissions within the process memory of this arbitrary process and consequently writes code to this allocated memory.

```
1  code_injection:
2    - api_call: "OpenProcess"
3      store:
4        - name: "return"
5          as: "proc_handle"
6    - api_call: "VirtualAllocEx"
7      with:
8        - argument: "hProcess"
9          operation: "is"
10         value: "$(proc_handle)"
11       - argument: "flProtect"
12         operation: "is"
13         value: 0x40
14   - api_call: "WriteProcessMemory"
15     with:
16       - argument: "hProcess"
17         operation: "is"
18         value: "$(proc_handle)"
19 condition: code_injection as sequence
```

Listing 3.21: Detection block to identify simple code injection into a running process

This detection block should be detected in the function log excerpt shown in Listing 3.22.

```
1 [0029.681] OpenProcess (dwDesiredAccess=0x47a, bInheritHandle↩
      =0, dwProcessId=0x740) returned 0x0
2 [0029.681] GetLastError () returned 0x5
3 [0029.681] OpenProcess (dwDesiredAccess=0x47a, bInheritHandle↩
      =0, dwProcessId=0x754) returned 0xf8
4 [0029.681] VirtualAllocEx (hProcess=0xf8, lpAddress=0x0, ↩
      dwSize=0x310, flAllocationType=0x3000, flProtect=0x40) ↩
      returned 0x120000
5 [0029.962] OpenProcess (dwDesiredAccess=0x47a, bInheritHandle↩
      =0, dwProcessId=0x180) returned 0xf9
6 [0029.962] VirtualAllocEx (hProcess=0xf9, lpAddress=0x0, ↩
      dwSize=0x310, flAllocationType=0x3000, flProtect=0x40) ↩
      returned 0x4df0000
7 [0029.962] WriteProcessMemory (hProcess=0xf9, lpBaseAddress=0↩
      x4df0000, lpBuffer=0x438a5e, nSize=0x310, ↩
      lpNumberOfBytesWritten=0x12fcb0)
```

Listing 3.22: Excerpt of a function log that contains code injection behaviour

The naive handling of store directives in the detection process, i.e. overwriting the variable value of `proc_handle` every time the `OpenProcess` API call is identified, would lead to an incorrectly extracted sequence based on the calculated LCS matrix. For instance, the sequence would consist of the API calls in line 1, 4 and 7. It is obvious that this sequence is incorrect, since the API calls

are not correlated based on the process handle they use. Instead, the correct sequence would consists out of the API calls in lines 5-7. We can even construct example cases that lead to false positive detections. In order to solve this problem, we introduce the concept of *variable contexts*. In principle, a variable context defines which API calls are allowed to store variables in a certain LCS calculation. Consequently, the LCS matrix needs to be calculated for each possible variable context. The algorithm needed to identify the possible variable context is rather complex. At first hand, we identify the API calls in the reduced function log that correspond to detection steps storing variables. Based on the combination of detection steps storing variables and their corresponding API calls, we can extract the information which variable identified by its name is stored by which API calls relatively to the detection step. The set of possible variable contexts is then the cartesian product of these API calls storing a certain variable in a certain detection step. With reference to the example, we identify three variable contexts – one context for each `OpenProcess` API call. In this particular case, the last variable context leads to correct sequence. In order to enhance the detection runtime, we stop the detection if a variable context is found that leads to a successful detection of the path.

With reference to the described algorithm to identify possible variable contexts, it is obvious that the extensive use of variables in signatures can lead to a large set of contexts and consequently to many LCS calculations. Especially, this is the case if there are many API calls that correspond to detection steps storing variables. Hence, variables should be used with care in signatures.

3.6.3 Condition Evaluation

Based on the intermediate results of the path detection R_{path_sim} and R_{path_seq}, we can directly derive the intermediate detection result of the block which is needed for evaluating the condition of the signature. Following our previously introduced definition, a detection block is considered successfully detected as soon as on of the detection paths extracted from the block was successfully detected. Formally, we define the set of intermediate detection path results of a certain block as $R_{paths} = \{r_1, ..., r_i\}$ where i corresponds to the number of detection paths extracted from that particular block. The intermediate detection block result R_{block} is then defined as shown in Equation 3.7.

$$R_{block} = \begin{cases} \text{true}, & \text{if } \exists r \in R_{paths} : r = \text{true}, \\ \text{false}, & \text{if } \forall r \in R_{paths} : r = \text{false} \end{cases} \tag{3.7}$$

By applying this definition to all detection blocks included in the signature, we get a set of intermediate detection block results $R_{blocks} = \{r_1, ..., r_i\}$ where each element of the set represents the intermediate result of a certain detection block. With the help of this set, we can evaluate the signature condition. With reference to the description of the condition tree in Section 3.5.3, the elements of the set R_{blocks} represent the result of the condition leaf nodes. Hence, by replacing the condition leaf nodes with their corresponding result from R_{blocks}, the condition tree can be evaluated as the expression contains boolean values connected by boolean connectives. So, for evaluating the signature condition, we simply traverse the condition tree with the in-order traversal method and evaluate each operator node by connecting the child elements with the boolean connective defined in the operator node. The result of the condition evaluation is the detection result of a certain signature detected in a certain process and is defined as R_{sig}.

Detection Block Key	Detection Block Result R_{block}
`key_1`	`true`
`key_2`	`true`
`key_3`	`false`
`key_4`	`false`

Table 3.7: Assumed detection block results for the condition shown in Figure 3.10

We demonstrate the condition evaluation based on the example condition introduced in Section 3.5.3 (Figure 3.10). In order to evaluate the condition we assume that the detection of the detection blocks in a certain process brought up the results shown in Table 3.7.

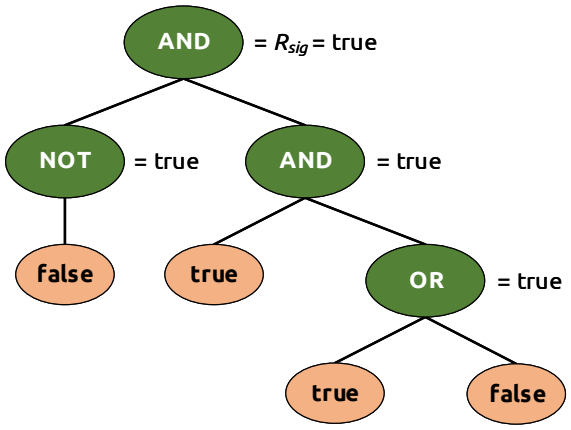

Figure 3.16: Condition evaluation

By replacing the leaf nodes of the condition tree as described before, we get the binary expression tree as shown in Figure 3.16. We traverse the tree beginning from the root node and evaluate all operator nodes from the bottom of the tree to the top. Hence, the last operation in the evaluation is always the root node of the condition tree. In this example case, the signature is successfully detected as the result of the condition evaluation is $R_{sig} =$ true.

3.7 Summary

In this chapter, we presented the key design concepts of our signature-based detection approach of behavioural malware features called *dynmx*. For the detailed introduction of the approach, we followed the overall detection process outlined in the first section of this chapter. Beginning from parsing the function logs, which build the source of information for the detection, into the common data model, we continued to introduce the generic function log format. This generic format is basically the serialised form of the function log object which, in turn, corresponds to the common data model. For serialising the object, we leverage the JSON syntax. In order to store function logs storage efficiently, we propose the usage of compression.

By following the detection process further, the resource usage is deduced based on the function log object in the next process step. This resource extraction is based on the research on access activity models published by Lanzi et al. in [63]. The basic idea behind access activity models is the hypotheses that resource usage can be deduced based on API call traces by following resource handles. We extend this model by considering userland WinAPI calls in addition to system calls for the resource extraction. Another important extension is the extraction of network resources additionally to file and Registry resources.

The main contributions of this chapter are found in Sections 3.5 and 3.6 that explain the signature DSL and the corresponding detection algorithm. We present the basic structure of dynmx signatures which relies on independent detection blocks related in a boolean condition as well as several features of the DSL. For the textual representation of signatures, we make use of the general data representation syntax YAML which is concise and machine-readable. In order to make this textual signatures available to the detection algorithm in a suitable format, we describe the signature parser. The parser is responsible for transforming each of the signatures' detection blocks to detection block graphs. This graph reflects the distinctive detection paths of a block as directed acyclic graph. The vertices of this graph represent the actual detection steps that are connected by edges and

thereby bring the individual steps into a detection sequence. The function log objects and the parsed signatures build the main inputs to the detection algorithm. The basic mode of operation of this algorithm is process-centric which means that the detection happens in each process of the input function log individually. We differentiate between the `simple` and `sequence` detection type. As for the `simple` detection type, detection steps are detected despite of the sequence in that they are defined in the signature. In contrast, the `sequence` detection type is used to detect steps in the sequence in that they are defined in the signature. For the sequence detection, we adopt the LCS algorithm to our specific use case. In particular, instead of matching elements in terms of their equality we introduce the formal `detect` function. This function is used to match a detection step with an API call. In order to enhance the efficiency of the detection, we reduce the problem set. Therefore, the function log object is reduced to only relevant API calls based on the detection path extracted from the detection block graph.

The first part of the subsequent chapter will briefly describe the protoype implementation of the design concepts presented in this chapter. In the second part of the chapter, we will explain the scope of our prototype signature library. This library provides several signatures for generic and specific malware features defined in our proposed signature DSL.

4 PROTOTYPE IMPLEMENTATION

In order to show the practical feasibility of our proposed approach, we have, on the one hand, implemented the *dynmx prototype* in the course of this thesis. Generally, this prototype is capable of detecting signatures in VMRay Analyzer function logs according to the design and the detection process (see Section 3.1) proposed in the previous chapter, most notably the detection algorithm described in Section 3.6. Although, the prototype is implemented mainly for processing VMRay Analyzer function logs, we consistently kept the sandbox-agnostic approach throughout the implementation as described in the design chapter. On the other hand, we have developed a prototype signature library that includes several signatures that were written in our proposed DSL (see Section 3.5.2 for detailed information) to detect certain behavioural malware features. These signatures contain the actual intelligence needed to detect malicious behaviour based on characteristic API call sequences and the usage of operating system resources.

In the remainder of this chapter, we will give a brief overview of the code structure and usage of the dynmx prototype in Section 4.1. We will additionally focus on the detection implementation in more detail in this section. The next Section 4.2 describes the developed signature library and explains which malware features we have considered for our prototype.

4.1 dynmx Prototype

The dynmx prototype is a Python console application that implements the whole detection process as described in Section 3.1 et seq. from parsing input function logs and signatures over deducing the access activity model from function logs and finally detecting signatures in the same. This console application can be used manually by the domain expert or in automated processes. Thus, the application does not need a specific configuration since all relevant information for execution is passed as command line parameters. The Python code of the dynmx prototype is attached to this

thesis in digital form and can be found on the enclosed flash drive[1]. In the following, we will briefly describe the code structure and the usage of the prototype. In Section 4.1.3, we will focus on the detection of signatures in detail and will present certain details of the concrete implementation.

4.1.1 Code Structure

Generally, our prototype is developed object-oriented as far as necessary and reasonable. The classes are organized using Python modules. Table 4.1 gives an overview of the developed Python modules and thereby sets out the high-level code structure of our dynmx prototype. Generally, the modules developed for the prototype are combined in the wrapper module `dynmx` which can be imported by other applications as library.

Python Module	Description
`dynmx`	Wrapper module for the modules specifically developed for the dynmx prototype. Other applications can use this module as framework in order to import certain functionality like parsing function logs.
`dynmx.converters`	The `converters` module contains the classes needed to convert an input function log to the generic function log format presented in Section 3.3.
`dynmx.core`	Within the `core` module basic entity classes for common objects used by all other modules are defined. Generally, this module implements the entities defined in the common data model introduced in Section 3.2.3 as well as the resource entities needed for the access activity model (see Section 3.4).
`dynmx.detection`	The actual detection algorithm as well as the needed preparation steps for the detection like parsing signatures or building the access activity model is contained in the `detection` module.
`dynmx.flog_parsers`	This module comprises the function log parsers as defined in Section 3.2 for processing VMRay function logs as well as function logs in the generic function log format introduced in this thesis.
`dynmx.helpers`	Generic helper functions needed amongst multiple modules like regular expressions or the configuration of the logging are combined in the `helpers` module.

Table 4.1: Overview of the Python modules developed for the dynmx prototype

The main class of our prototype that contains the actual detection logic is the `Signature` class which is defined in the signature.py file (`dynmx.detection` module). The `Signature` class represents, on the one hand, a parsed signature and thus implements the definition of the signature object. Further, the `Signature` class implements our adapted LCS detection algorithm as proposed in Section 3.6. We will explain the implementation of the detection more detailed in Section 4.1.3.

In order to solve generic problems like parsing XML structured files or working with trees we make use of well-established external Python packages. These packages are needed in order to run our prototype. Overall, we use the following external packages in our prototype:

- `PyYAML` [117]
- `anytree` [16]
- `lxml` [128]

[1] The Python source code of the dynmx implementation is found in the directory `Digial_Appendix/4_Implementation/dynmx_Source_Code/` on the enclosed flash drive.

- `pyparsing` [55]
- `stringcase` [109]
- `multiprocessing-logging` [133]

4.1.2 Usage

In order to use the dynmx prototype certain prerequisites must be met. First of all, the previously stated external packages need to present on the target system. All of the packages can be conveniently installed with the help of the Python package manager `pip` as described in detail in [125]. If the packages should not be installed system-wide the usage of virtual environments [126] is beneficial and recommended. Please note, that the prototype is written in Python 3 and thus needs a working Python 3 installation (at least version 3.7) on the target system. Further, the packages need to be installed for this Python 3 installation. As the prototype was developed on a Unix-like operating system, it is recommended to also use a Unix-like operating system like MacOS or Linux.

As soon as the prerequisites are met, the prototype can be used. The main entry point of the dynmx prototype is the Python file `dynmx.py` found in the root directory of the application. Generally, the prototype provides several commands, e.g. for converting input function logs to the generic function log format or detecting signatures. Since these commands can be explored in detail by calling the help with the `-h` command line parameter, we will not describe them in detail. In the following, we will give a brief overview of the commands provided by the prototype.

`detect` This command is used to detect signatures in a given set of input function logs.

`check` The `check` command is used to perform as basic check of a given set of dynmx signatures.

`convert` The command converts a given set of input function logs to the generic function log format and writes the result to a defined output directory.

`stats` The command is used to provide some basic statistics of a given set of input function logs.

`resources` With the help of this command the resources are extracted from input function logs.

As for the `detect` and `convert` commands, we make use of parallelism with concurrent processes in order to reduce the overall runtime. Since the main functionality provided by the prototype lies in detecting signatures defined following our proposed DSL (see Section 3.5.2), we will focus on the usage of the `detection` command in the following by giving an usage example. Therefore, we assume that we would like to detect the signature `dynmx_signature_formbook.yml` in the input function log `flog.txt`. The command to accomplish this task together with the output of the command is shown in Listing 4.1. Please note, that the output is shortened for the sake of brevity.

```
$ python3 dynmx.py detect --input flog.txt --sig ↩
    dynmx_signature_formbook.yml

    |
  __|          _  _    _  _  _
 /  |  |    | / |/ |  / |/ |/ |  /\/
 \_/|_/ \_/|/  |  |_/  |  |  |_/ /\_/
          /|
          \|

 Ver. 0.5 (PoC)


2020-05-25 22:59:30,398 [INFO] (__main__): Parsing dynmx ↩
    signature 'dynmx_signature_formbook.yml'
[...]

RESULT
```

```
17 =======
18 formbook    flog.txt
```

Listing 4.1: Usage of the detect command

The output of the command can be generally divided into two section – the output of the run log (line 13) and the results (line 16). The run log can also be written to a log file for the purpose of traceability. As shown in the result section of the example, output the signature has been successfully detected in the input function log. By defining the output format of the result using the `-format` command line parameter, the domain expert can get a more detailed view on the result (see Listing 4.2). The result section of the detailed output is interpreted as follows: the findings reflect the detection blocks from the signature that were detected in the stated process of the function log; for each finding the detection block key and the concrete API call together with the concrete line number of the function log that caused the detection of the detection block is presented. Thereby, we allow the domain expert to understand the detection in detail. If the result should be further processed, e.g. storing historic results in a database or further examining a result, we also give the option to output the detection result as machine-readable JSON string providing full details of the detected API call sequence including argument and return values.

```
1  $ python3 dynmx.py --format detail detect --input flog.txt --↩
       sig dynmx_signature_formbook.yml
2
3
4  [...]
5
6  RESULT
7  =======
8  Function log: flog.txt
9  Signature: formbook
10   Process: 30af1533f252a90589ee251eff493253.exe (PID: 4948)
11   Number of Findings: 3
12     Finding 0
13       strings : API Call CharUpperBuffW (Flog line 37294)
14       strings : API Call CharUpperBuffW (Flog line 57534)
15       strings : API Call CharUpperBuffW (Flog line 60290)
16       strings : API Call CharUpperBuffW (Flog line 60291)
17     Finding 1
18       shell_timeout : API Call ShellExecuteExW (Flog line ↩
             60301)
19     Finding 2
20       section_load : API Call GetLongPathNameW (Flog line ↩
             2204)
21       section_load : API Call LoadLibraryExW (Flog line 2220)
22       section_load : API Call FindResourceExW (Flog line ↩
             2228)
23       section_load : API Call LoadResource (Flog line 2229)
24       section_load : API Call ISequentialStream:RemoteWrite (↩
             Flog line 2232)
```

Listing 4.2: Usage of the detect command

4.1.3 LCS Detection Procedure

After we have shown the signature detection from the perspective of the domain expert using the dynmx prototype, we will now take a deeper look on how the detection command is actually implemented in the prototype. Following our detection process overview shown in Figure 3.1 (Section 3.1) and our detection algorithm overview in Figure 3.11 (Section 3.6), we assume in the following that the function log as well as the signature have already been parsed. Furthermore,

we assume that a detection block with the `sequence` type should be detected. If not otherwise stated the following Python code origins from the files `signature.py`, `detection_block.py` and `detection_step.py` which are part of the `dynmx.detection` module.

In preparation for the actual detection using our adapted LCS algorithm, we need to first extract the potential detection paths of the detection block that should be detected. This is done in the first code lines of the method `DetectionBlock._detect_sequence()` (see appendix A.5 for the full Python code of the method) by finding all paths of the detection block graph (line 11-12 in the aforementioned listing). Based on the extracted detection paths the API call sequence of the process is reduced to only relevant API calls as proposed in Section 3.6.1 (line 24 in the listing).

With this two main inputs the calculation of the longest common sequence can take place (line 27 in the listing). Whether the detection of the path has been successful is determined in line 32 according to our definition in Equation 3.6 (Section 3.6.2). The detection of resources takes place in lines 36-41 in the listing. The rest of the method is mainly focused on gathering the needed information for the detection result if the detection was successful. For the sake of completeness, we have also included the Python code of our LCS algorithm adaptation in appendix A.5. The implementation is based on a reference implementation of the traditional LCS algorithm published in [105]. The formal `detect` function introduced in Section 3.6.1 is represented by the `DetectionStep.detect()` method in the implementation view. Hence, the method returns a boolean value that indicates whether the step was successfully detected.

It needs to be noted, that we stop the detection of a signature for a certain function log if the signatures was detected in one of the processes. We thereby reduce the detection runtime. This implementation was needed in order to cope with large and complex function logs. Nevertheless, we added a command-line option `-detect-all` to search in all processes and get more complete results. In practice, this option should be used with care.

4.2 Signature Library

Besides the actual detection approach and its concrete implementation as prototype presented in this thesis, the signature library is one of our main contributions. As the signatures included in this library define the malware features considered in this thesis in a sandbox-agnostic and reusable manner on the API call level, these signatures contain the actual intelligence of the detection approach as already introduced in the previous Section 3.5.2. The references together with representative sample hashes used to define the signatures presented in the following are stated within the signature file in the meta data sections `references` and `samples` respectively. Therefore, these resources are not referenced in this thesis as they were solely used for the implementation of the signatures.

From the implementation point of view, the signature library is an internal Git repository containing the text-based signatures leveraging the DSL introduced in this thesis. The repository provides a central storage with version control for the text-based signatures. Changes to this repository can be simply managed in a collaborative manner by following the common understanding of using Git repositories for software development as described in [29]. i.e. by using branching and merge requests. The usage of Git repositories to store signatures is well adopted in the IT security community, e.g. for Yara rules [75, 131] or for Snort rules [25]. Moreover, public repositories can be used to share and collaboratively work on signatures within the IT security community.

Of course, the signature library delivered together with this thesis on the enclosed flash drive[2] is only a starting point and does not aim to be complete – if this state can ever be reached in the everlasting race between detection capabilities and tools, techniques and procedures of adversaries. Instead, we would like to present the feasibility of defining reoccurring and known malware features based

[2]The signature library is found in the directory `Digial_Appendix/4_Implementation/Signature_Library/` on the enclosed flash drive. The library is categorised by generic and specific malware features.

on API call sequences with the help of our DSL. Thus, in the following sections, we will first present the generic malware features that we have considered for the initial signature library delivered together with this thesis. In addition to the concrete features, we will describe our methodology to define this kind of signatures. After focusing on generic malware features, we will present the application of our approach on specific malware features of certain malware families. Again, in addition to state the concrete malware families taken into account, we will describe our underlying methodology of defining signatures for specific malware features.

4.2.1 Generic Malware Features

Following our definition from Section 3.1, generic malware features are used in a wide variety of malware. Consequently, defining signatures for this kind of features is valuable for domain experts as they can deduce certain recurring functionality in unknown malware samples based on the detection of these signatures. This, in turn, can help the domain expert to initially assess and triage an unknown sample without the need to statically reverse engineer the sample. Of course, today's malware sandboxes already detect a wide variety of generic malware features but we have noticed that certain generic behaviour is not detected precisely or even remains undetected (please refer to our evaluation Section 5.2.1 for detailed results). Furthermore, by considering our approach to be sandbox-agnostic, the detection of generic malware features differs between different sandboxes, e.g. [3]. Consequently, we consider the incorporation of certain generic malware features into our signature library beneficial.

For the signature library delivered together with this thesis, we consider the generic malware features of the following categories:

- Malicious code injection and
- Persistence mechanisms

In order to identify common malware features of the aforementioned categories we found the MITRE ATT&CK® Matrix [94] to be a valuable resource. The ATT&CK® Matrix is a collaborative approach to collect and publish adversary techniques in a central knowledge base. In addition, Sikorski and Honig describe a wide variety of malicious behaviour in [146]. Based on these resources, a list of the considered generic malware features together with their corresponding technique ID from the ATT&CK® Matrix is shown in Table 4.2. Based on the category (tactic ID) and the technique ID, one can find further information by navigating through the ATT&CK® Matrix beginning with the category in [100].

Category	Considered Generic Malware Features
Malicious code injection (TA0004, TA0005)	• Process hollowing / process replacement (T1093) [96] • DLL injection (T1055) [97] • Direct code injection (T1055) [97] • Import address table (IAT) hooking (T1179) [93]
Persistence mechanisms (TA0003)	• Windows registry run keys (T1060) [98] • Winlogon notify (T1004) [91] • AppInit DLLs (T1103) [92] • Service installation (T1050) [95] • Scheduled task (T1053) [99]

Table 4.2: Overview of the generic malware features considered in this thesis

As we have stated now the generic malware features taken into account, we will focus on the methodology to define signatures for these kind of malware features in the following. Generic

malware features are typically used by a wide variety of malware and aim to implement a certain functionality. Hence, the sequence of API calls used to achieve this certain functionality is mostly fixed and predefined by the API. Further, most of these malware features are well researched. But a not inconsiderable subset of the published analysis whitepapers for malware focuses on statically reverse engineering samples and on the assembler layer instead of considering the API calls from a behavioural perspective, e.g. [138, 154]. Nevertheless, publications on certain malware features are a valuable resource and starting point for finding characteristic API call sequences. The refinement of this initial signature is an iterative process of

- finding representative malware samples (e.g. based on sample hashes published together with analysis whitepapers),
- manual analysis of the function log of these representative samples and
- manually extract the relevant API call sequence.

The challenging part in defining robust and precise signatures for a generic malware feature lies in handling the complexity of the Windows API, i.e. finding all possible API call candidates for performing the exact same functionality. This, in turn, requires a good knowledge of the Windows API by the domain expert defining the signature.

4.2.2 Specific Malware Features

From a practical perspective, the industry standard way to detect whether a malware sample belongs to a certain malware family is to use YARA rules that define static characteristics of a family's representative malware samples. As already mentioned in the introduction of this thesis, this approach can lead to false positives and can often be easily circumvented by the adversary. So, defining behavioural signatures on the API call level for specific malware features of a certain malware family can be beneficial as the API call sequences are considered more stable than the static representation [123]. By defining signatures for specific malware features, we provide indications for answering the question whether a certain malware family can be characterised by their behaviour on the API call level.

In order to provide these indications, we have taken the following malware families into account:

- Emotet (Geodo, Heodo) [37],
- Remcos (RemcosRAT, Remvio) [36],
- DarkComet (DarkKomet, Fynloski, Breut, klovbot) [38] and
- Formbook [39]

First of all, we have chosen this set of malware families since the Cyber Defense Department of Deutsche Telekom AG is actively hunting for these samples. Hence, we have a good data basis for evaluating and writing our signatures (see Section 5.2.2). Furthermore, these malware families represent a considerable threat to the industry as they mostly aim to steal confidential information from their victims. As a consequence, reliably detecting these samples is beneficial in order to deduce actionable IoCs from the samples. These IoCs, in turn, help to identify infections in the infrastructure and pro-actively defend new infections. As a consequence, the malware families are carefully chosen in terms of their high practical relevance.

In contrast to generic malware features, characteristic API call sequences are typically not known for certain malware families as most of the research and analysis focuses on defining YARA rules. In order to identify such characteristic API call usage, we have applied the same iterative approach as introduced for generic malware features in the previous section. Therefore, we manually analysed function logs of samples belonging to a certain family, identified outliers not detected by the signature and adapted it accordingly. In addition to our own sample dataset, we found the Malpedia project [122] a valuable resource for identifying representative samples and research for a certain

malware family. Based on our examination of function logs for defining specific malware features, good candidates for signatures are

- string formatting API calls like `_snwprintf` that are used for build a string based on a defined format,
- the creation of mutexes with a defined name,
- the load of certain sections from an executable,
- decryption and parsing of the malware's configuration and
- resources created based on a fixed schema.

The challenge in defining a reliable signature is the runtime of the malware in the sandbox is very limited (typically 2 minutes for our sandbox configuration). Thus, we can only rely our signatures on behaviour that happens within this time frame. Furthermore, malware families evolve over time which can lead changes also on the API call level. For example, strings formatted based on a fixed schema can change and lead to missing detections.

4.3 Summary

We briefly presented the dynmx prototype implementation as Python console application in the first part of this chapter. In general, we outlined the overall code organisation as well as external dependencies of the application. Moreover, we showed the usage of the application from the standpoint of a domain expert who uses the prototype. Efficient detection runtimes for large function logs data sets are achieved by leveraging parralelisation with concurrently running processes. In particular, we focused on the implementation of the LCS detection algorithm.

The second part of this chapter dealt with the prototype signature library developed in the course of this thesis. We presented 9 signatures for generic malware features used by malware to inject malicious code into legitimate processes and to gain persistence on an infected system. We related these generic malware features to their equivalent techniques according to the MITRE ATT&CK® Matrix. Furthermore, we presented 4 signatures for specific malware features of the malware families DarkComet, Emotet, Formbook and Remcos. We described the iterative process used to define and refine the proposed signatures.

In the following chapter, we will evaluate or detection approach based on the implementation presented in this chapter. The evaluation consists of three distinct parts. The first part will deal with the evaluation of the detection quality in terms of generic an specific malware features. The detection quality of generic malware features will be compared to the built-in detection capabilities of the VMRay Analyzer sandbox. The focus of the second part of the evaluation lies on signature DSL. Again, we will compare our DSL with custom VTI rules which can be incorporated into the VMRay Analyzer sandbox detection process. The last part of the evaluation aims at indicating the overall performance of our proposed detection approach.

5 EVALUATION

After we have presented the theoretical concept of our approach in Chapter 3 as well as the practical feasibility by implementing the dynmx protoype (Chapter 4) we will focus on finding answers to the open question on the practical relevance and effectiveness in this chapter. Therefore, we will evaluate our approach in the following three dimensions:

1. the detection of generic and specific malware features,
2. the definition of signatures using our proposed DSL compared to VMRay VTI rules and
3. the performance in terms of the detection as well as the proposed generic function log format compared to the VMRay function log formats.

In the following sections, we will first describe the data sets which formed the basis together with the implemented dynmx prototype for evaluating the detection as well as the performance. The detection, in turn, will be the focus of Section 5.2. In this section, we present how well generic and specific malware features are detected in the outlined data sets. Furthermore, as the VMRay Anlayzer sandbox also provides information on detected generic malware feature, we correlate our detection result of generic malware features with the results provided by VMRay Analyzer. The detection evaluation is based on the malware features that we have taken into account for our prototype signature library. Please refer to Section 4.2 for detailed information. The results of this section will mainly contribute to the assessment of the practical relevance and effectiveness of our proposed approach. After that, Section 5.3 will compare the capabilities of our proposed signature DSL with the VMRay feature of defining custom VTI rules. We evaluate to what extent it would have been possible to implement the signatures of our prototype signature library with custom VTI rules. Thereby, we will work out the advantages and disadvantages of the two approaches to define signatures for detecting certain malware features based on API calls. The performance is the last aspect of our evaluation and is outlined Section 5.4. In this section, we will present runtimes of certain steps in the detection process. Additionally, we present performance indications in terms of parsing and storage consumption of our proposed generic function log compared to the function log formats provided by VMRay Analyzer. The chapter concludes with a summary and discussion of the evaluation results.

5.1 Data Sets

Generally, we use two distinct data sets in order to evaluate our proposed detection approach – a malicious and a benign one. These data sets comprise of function logs that origin from the VMRay Analyzer sandbox. The actual sandbox instance from that the function logs origin, is the on-premise sandbox used by the Cyber Defense Center of Deutsche Telekom Group (DTAG CDC). The DTAG CDC is responsible for handling IT security incidents in the Deutsche Telekom Group and their subsidiaries, improve detection capabilities and assessing the threat landscape by leveraging Threat Intelligence methodologies [26]. This on-premise sandbox instance is used by domain experts of DTAG CDC for dynamically analysing malware samples. Furthermore, the VMRay sandbox instance is used in several fully automated processes for detecting and researching on malware. By using the REST API provided by the VMRay Analyzer sandbox [51], we collected the analysis archives (see Section 2.6 for further details) of certain samples and extracted the function logs from this analysis archives. These extracted function logs build the data sets outlined in the following. Generally, we collect at least the following attributes for each sample in our data sets:

- VMRay job ID identifying the exact dynamic analysis,
- SHA-256 hash sum,
- Sample type,
- Sample file size,
- the VM used for the dynamic analysis and
- the duration of the dynamic analysis run.

Further information regarding our data sets is attached to this thesis in digital form and can be found on the enclosed flash drive[1].

5.1.1 Sandbox Configuration

Since we have collected function logs beginning from 11th December 2019 the version of the VMRay Analyzer sandbox is not consistent. The function logs collected origin from the VMRay analyzer versions 3.1.1 and 3.2.0. As we have not faced any parsing issues in the course of the evaluation, with high confidence the function log format did not change between this sandbox versions. Hence, we expect that the inconsistency in the version is not affecting our detection results. We provide the version information for the sake of completeness and traceability.

It is further noted, that the VMRay Analyzer sandbox is configured to use an internal FakeNet instead of a direct connection to the Internet. This internal FakeNet simulates certain network services like DNS and HTTP and is specifically designed for the dynamic analysis of malware. Consequently, malware samples that specifically check for the presence an active connection to the Internet by expecting specific information from real services in the Internet may not run in this environment.

Generally, our internal VMRay Analyzer sandbox can use two distinct VMs for analysing malware. These VMs are operated with the Windows 7 Enterprise x64 and Windows 10 Enterprise x64 operating system respectively. In both of the VMs the samples are executed with administrative permissions. Hence, only malware that relies on a specific operating system may not run in our VMs. But this seems to be more of a theoretic than a practical issue as one of the main goals of malware is to run on the target system. At least for that kind of malware that we have taken into account for our evaluation which is typically not targeted to a specific system. Nevertheless, we cannot preclude that certain samples that are part of our data set have altered execution paths based on the VM in that the sample was executed. By default, the Windows 7 VM is chosen for the dynamic analysis.

[1] The data set information is found in the directory `Digial_Appendix/5_Evaluation/Data_Sets/` on the enclosed flash drive.

We provide detailed information on the VMs used for the analyses on which the function logs are based in the following sections.

5.1.2 Malicious

In order to build the malicious data set, we have leveraged information from the internal malware database of DTAG CDC. This malware database automatically submits malware samples to the internal VMRay Analyzer sandbox to collect behavioural information and store them in the database. We extracted a list of submitted malware samples and used this list to collect the relevant analysis archives to extract the function logs following the aforementioned process. In order to preclude possibly benign samples from the data set, we only collected the analysis archives of the samples that were classified with the severity "malicious" by the VMRay Analyzer sandbox. The source for the malware samples in the malware database can be various and include, amongst others, VirusTotal [163], VirusShare [162] as well as manual uploads by domain experts.

The second source of information that we have used for building the malicious data set, is a further internal tool called Malware Config Extractor (MCE). This tool statically extracts configuration parameters of malware samples that belong to certain malware families. The source of these samples is VirusTotal where YARA rules are deposited for identifying relevant samples. As soon as someone uploads a new sample to VirusTotal and this uploaded sample matches one of the YARA rules, the sample is downloaded and submitted to the internal VMRay Analyzer sandbox. The MCE tool is hooked into the sandbox detection process and statically extracts the malware's configuration parameters based on memory dumps created by the sandbox. Only if the extraction of the configuration is successful, the sample is added to another internal database that stores the extracted configuration parameters. This step is necessary as the deposited YARA rules also match against false positives. However, the static extraction process is very specific for a certain malware family as it has to meet the exact deobfuscation and decryption parameters of this family. Hence, we expect a low false positive rate in this classification. This classification also builds the ground truth for our evaluation of specific malware features presented in Section 5.2.2.

Malware Family	Number of Function Logs	Percentage	Unique Samples	MCE	Malpedia
DarkComet	491	1.53%	491	491	0
Emotet	5,771	18.08%	5,093	5,764	7
Formbook	121	0.37%	121	117	4
NjRAT	1,165	3.65%	1,165	1,164	0
Quasar	101	0.32%	101	101	0
Remcos	612	1.92%	612	611	1
XtremeRAT	16	0.05%	16	16	0
Unknown	23,622	74.08%	23,020	0	0
Overall	31,898	100%	30,619	8,264	12

Table 5.1: Overview of malicious data set categorised by malware family

The third and last source of information for our malicious data set is Malpedia [122]. Malpedia is a collaborative effort introduced by Plohmann et al. to provide a quality-assured and curated corpus of malware families. Generally, Malpedia provides representative samples, information resources as well as YARA rules for certain malware families. Since the malware families that we have taken into account for our prototype signature library are also available on Malpedia (version 5843 as of 11th June 2020 10:27:16), we incorporate these samples into our malicious data set. Therefore, we manually downloaded the samples provided by Malpedia and submitted them to the internal VMRay Analyzer sandbox in order to collect the function logs. Since the Malpedia corpus is curated and stands for quality over quantity we do not expect false positives in this data set. In order to

comply with the Malpedia terms of usage we have only incorporated packed and unpacked samples that are known in VirusTotal.

By combining the function logs originating from all of these three data sources, we get a comprehensive malicious data set consisting of 31,898 function logs in sum. As for the distribution regarding the used virtual machine in the VMRay Analyzer sandbox, 77.4% of the samples were analysed in a Windows 7 VM and 22.6% in a Windows 10 VM respectively. Please note, that one sample is not necessarily represented by exactly one function log since there can be multiple analysis runs for one one sample. Table 5.1 provides a more detailed overview of this data set. As shown in the table, the largest subset of our malicious data set is not classified according to the aforementioned data sources and is therefore part of the "Unknown" category. Nevertheless, this does not preclude that samples of this subset belong to a certain malware family named in the table. For example, if the malware authors change the obfuscation and encryption technique for a certain family, the MCE tool needs to be adapted to these changes in order to successfully extract the malware's configuration.

Sample File Type	Number of Function Logs	Percentage
Windows Exe (x86-32)	31,227	97.89%
Windows Exe (x86-64)	325	1.02%
Windows DLL (x86-32)	186	0.58%
Windows DLL (x86-32)	59	0.19%
Java Archive	33	0.1%
RTF Document	27	0.08%
Microsoft Word Document	18	0.06%
Microsoft Excel Document	17	0.05%
MSI Setup	5	0.02%
PDF Document	1	0.003%

Table 5.2: Distribution of file types in the malicious data set

The function logs in our malicious data set origin from different sample file types. The file type is determined by the VMRay Analyzer sandbox during the submission process. Table 5.2 gives an overview of the different sample types present in our malicious data set. As seen in the table, the majority of samples submitted by the internal malware database are PE executables with a share of 98.91%. Since the VMRay Analyzer sandbox is also capable of running DLL files by running relevant exported functions of the same using a heuristic fuzzing approach [161], this sample file type can also produce meaningful function logs. Malicious documents leveraging techniques like macros or the DDE exploit [56, 57] only make up a small share of in sum 0.19%.

5.1.3 Benign

To find out whether our developed signatures, especially the signature for specific malware features, produce false positives in benign software we have set up a benign data set. Our benign data set consists out of function logs derived from a sample set that was individually collected based on the following four sources:

- Windows PE executables collected from a Windows 10 x64 system,
- a toolkit used for forensic live analysis,
- portable applications published in [64] and
- benign documents found on VirusTotal.

Our focus on collecting the benign samples lies on using only directly executable software that has no need for being installed on the target system. Therefore, we expect to find the actual behaviour

of the benign samples in the function logs instead of API calls that origin from installing a certain benign software.

As for the first source of benign samples, we have recursively searched for files with the ending `*.exe` in the virtual disk of a Windows 10 x64 VM. These samples contain for example tools like the Windows command shell `cmd.exe` or the text editor `wordpad.exe`. The forensic live analysis toolkit contains well-known forensic software like the Windows Sysinternals tools [136] or the FTK Imager Lite application [2]. The samples originating from the third source were scraped from the website referenced in [64]. The web page provides direct download links to benign portable applications. By iterating over all download links of the web page and filtering for URLs that lead to executable files or ZIP archives, we were able to fetch the samples. As for the last source of benign samples, we have collected documents (Microsoft Office and PDF) on VirusTotal that have a zero detection rate over all anti-virus products present in the VirusTotal scan report. Based on a zero detection rate in the scan report we assume the document to be benign.

Source	Number of Function Logs	Percentage
Windows 10 x64 Executables	821	57.78%
Downloaded Portable Apps	302	21.25%
Forensic Tools	285	20.06%
Documents from VirusTotal	13	0.91%
Overall	1,421	100%

Table 5.3: Distribution of benign samples regarding the considered sources

In sum, we have collected 1,421 benign samples that were analysed in the same internal VMRay Analyzer sandbox from that the malicious data set origins. The process for gathering function logs based on these sample set is analogous to the process described for the Malpedia samples. 57.8% of the samples were analysed in a Windows 10 VM and 42.2% in a Windows 7 VM. The distribution of the collected function logs based on the mentioned source is as shown in Table 5.3.

Analogous to the malicious data set, we provide an overview of the sample types present in benign data set in Table 5.4.

Sample File Type	Number of Function Logs	Percentage
Windows Exe (x86-32)	888	62.49%
Windows Exe (x86-64)	498	35.05%
Windows DLL (x86-32)	18	1.27%
Windows DLL (x86-64)	3	0.16%
Windows Driver (x86-64)	1	0.01%
Microsoft Word Document	4	0.28%
Microsoft Excel Document	1	0.01%
PDF Document	8	0.56%

Table 5.4: Distribution of file types in the benign data set

5.2 Detection

In the subsequent sections, we will describe the evaluation of our detection approach in detail. As introduced in Section 3.1, we categorise malware features into generic and specific features. Hence, we also separate the detection evaluation based on this categorisation. We have developed a methodology for evaluating each feature type which will be presented together with the evaluation

results in the following. If not otherwise stated, we have used function logs in the dynmx format. Therefore, we converted our malicious and benign data set using the dynmx prototype.

The basis for the detection evaluation is the dynmx prototype implementation as well as the prototype signature library (both presented in Chapter 4). For the sake of traceability and transparency, we provide the relevant log and result files of our evaluation runs as well as relevant Python scripts used in the course of the evaluation in the digital appendix. The files can be found on the enclosed flash drive provided together with this work.[2]

5.2.1 Generic Behavioural Malware Features

The general aim of our evaluation of the generic malware feature detection is to identify discrepancies between the detection of generic features based on our proposed approach and signatures in comparison to the detection information that the VMRay Analyzer sandbox provides. As we have no custom VTI rules in place, the detections of the VMRay Analyzer sandbox reflect the built-in detection capabilities of the sandbox. With reference to the background section on signature-based malware detection (see Section 2.3), we obviously can only detect such features for that a signature exists. Hence, only the generic malware features mentioned in 4.2.1 are in the scope of our evaluation.

We would like to emphasise that the results presented in the following need to be interpreted in the light that our detection approach does not aim to replace but rather complement the detection of the VMRay Analyzer sandbox. Hence, we are not aiming for completeness in terms of the detection of generic malware features but for intelligently enhancing the overall detection capabilities and thus provide a benefit from the viewpoint of a domain expert.

Following responsible disclosure practices, the detection issues identified in the course of the evaluation of generic malware features have been reported to VMRay and were accepted for a fix in future versions.

Methodology

As introduced in Section 4.2.1, we have categorised our signatures according to the MITRE ATT&CK® Matrix. Because the VMRay Analyzer sandbox also provides information of detected techniques according to the MITRE ATT&CK® Matrix, we can correlate our detection results with the techniques detected by the VMRay Analyzer sandbox. The VMRay Analyzer sandbox maps the detected threat indicators to the MITRE ATT&CK® Matrix since version 3.1 [71]. As the lowest version found in our data set of function logs is 3.1.1, we expect every sample in our data set to have the mapping available. The correlation happens on basis of the technique ID specified by the MITRE ATT&CK® Matrix. As a result of this correlation, we can identify discrepancies in the detection between our proposed approach and the detection capabilities of the VMRay Analyzer sandbox.

The technique IDs identified by the VMRay Analyzer sandbox are denoted in the `summary.json` file which is part of the analysis archive (in the `logs` directory of the archive). More specifically, the technique IDs are found in the key `mitre_attack` and the subsequent `techniques` key (see Listing 5.1) of the JSON-formatted file. By iterating over all analysis archives and extracting the technique IDs from the JSON-formatted file content, we get an overview of the generic malware features detected by the VMRay Analyzer sandbox related to the malware samples that are part of our data sets.

```
1 [...]
2 "mitre_attack": {
```

[2] The raw information regarding the detection evaluation is found in the directory `Digial_Appendix/5_Evaluation/Detection/` and its subdirectories on the enclosed flash drive. It is noted, that the files are typically compressed due to the large file size.

```
3    "matrix_version": "2019-04-25 20:53:07.719000",
4    "techniques": [
5    {
6     "description": "NTFS File Attributes",
7     "id": "T1096",
8     "ref_vtis": [
9       {
10        "ref_id": "built_in._defense_evasion.↩
              _obscure_file_origin.↩
              vmray_delete_zone_identifier_by_delete_file",
11        "ref_source": "summary",
12        "ref_type": "vti_rule_match",
13        "type": "reference",
14        "version": 1
15      }
16    ],
17    "tactics": [
18      "Defense Evasion"
19    ],
20    "technique_version": 1.0,
21    "type": "mitre_attack_technique",
22    "version": 1
23   },
24 [...]
```

Listing 5.1: Techniqe ID in the `summary.json` file

After we have collected the technique IDs for all relevant samples from their analysis archive, we can continue with detecting our signatures in our data sets. Therefore, we start detection runs for each signature in our library that detects generic malware features. As we know the technique ID for each signature according to the MITRE ATT&CK® Matrix as well as the relation between function logs and the analysis archive from that they originate, we can correlate our detection results for a certain malware sample and a certain generic malware feature with the previously collected technique IDs from the VMRay analysis archive. The primary key used for the correlation is the SHA-256 hash value of the sample as well as the VMRay job ID. Please note, that the direction of the correlation is of relevance. We take our detections as basis and verify the detected technique IDs with the techniques detected by VMRay Analyzer. Unfortunately, a correlation in the other direction starting from the detection results of VMRay analyzer is not possible since the definition of techniques in the MITRE ATT&CK® Matrix is too unspecific for our use case. For instance, as shown in Table 4.2 (Section 4.2.1), the technique with ID 1055 refers to two distinc generic malware features in terms of our definition (DLL injection and direct code injection). As a consequence, we cannot identify missing detections for a certain technique in our results.

Results for the Malicious Data Set

By carefully applying the previously described methodology and thus matching our proposed signatures for generic malware features against the malicious data set, we got the results presented in Table 5.5. The results are categorised by the technique ID. The column “dynmx Detections” reflects the number of function logs that were detected by our signature for the generic malware feature. The column “VMRay Analyzer Detections” shows the result of the previously described correlation of function logs detected by dynmx with the collected technique IDs from the analysis archives based on the sample hash. The discrepancy between these two figures is shown in the last two columns of the table.

As seen in the results table, certain generic malware features, especially malicious code injection techniques, are not or only partly detected by VMRay Analyzer according to the technique ID. For instance, process hollowing is not detected by VMRay since the technique ID T1093 is not found for

any of the samples which are part of our malicious data set. In order to verify the correctness of the detections based on our signature, we have randomly examined a subset of our detection results and did not find any false positives. By manually analysing the VMRay analysis report for certain samples that were detected by our process hollowing dynmx signature, we noticed that the VMRay Analyzer sandbox generally detects the code injection with the VTI engine (see Section 2.6.3) but is not mapping this identified behaviour to a corresponding technique in the MITRE ATT&CK® Matrix.

Generic Malware Feature	dynmx Detections	VMRay Analyzer Detections	Detection Discrepancy	Detection Discrepancy in %
Process hollowing / process replacement (T1093)	5,130	0	5,130	100.00%
DLL injection (T1055)	1,267	643	624	49.25%
Direct code injection (T1055)	2,236	645	1,591	71.15%
IAT hooking (T1179)	65	8	56	86.15%
Windows registry run keys (T1060)	9,685	9,313	372	3.84%
Winlogon helper DLLs (T1004)	1,816	0	1,816	100.00%
AppInit DLLs (T1103)	96	0	96	100.00%
Service installation (T1050)	7,234	7,227	7	0.01%
Scheduled task (T1053)	3,813	3,624	189	4.96%

Table 5.5: Evaluation result for the detection of generic malware features in the malicious data set

As shown in Listing 5.2, the VTI technique `vmray_modify_memory` and `vmray_modify_memory_system`[3] representing the code injection is not mapped to any technique ID of the MITRE ATT&CK® Matrix (key `ref_mitre_attack_techniques` has an empty list). In addition, the VMRay sandbox detects the VTI techniques

- `vmray_modify_control_flow_non_system` and `vmray_modify_control_flow_system`[4] and
- `vmray_allocate_wx_page`[5]

for samples that leverage the process hollowing technique which are also not mapped to a technique of the MITRE ATT&CK® Matrix. By correlating our detected function logs based on all of the previously named VTI rules instead of the technique ID from the MITRE ATT&CK® Matrix, we found 4,508 samples and thereby verified our result to a certain extent. The missing 622 samples only had a subset of the named VTI detections and not all four. This is caused by the circumstance that process hollowing can be achieved with different API call sequences that are handled by our signature. Nevertheless, we state that VMRay Analyzer does not detect the process hollowing malware feature specifically but only certain steps of the technique with the VTI engine.

```
[...]
"category": "_injection",
"category_desc": "Injection",
"operation": "_modify_memory_non_system",
"operation_desc": "Writes into the memory of a process ↩
    running from a created or modified executable",
"ref_gfncalls": [
{
```

[3]This VTI technique corresponds to the API call `WriteProcesMemory`.
[4]This VTI technique corresponds to the API call `SetThreadContext`
[5]Corresponds to the API call `VirtualAllocEx`

```
8     "ref_id": "gfn_6503",
9     "ref_source": "glog",
10    "ref_type": "gfncall",
11    "type": "reference",
12    "version": 1
13  }
14  ],
15  "ref_mitre_attack_techniques": [],
16  "rule_classifications": [],
17  "rule_score": 2,
18  "rule_type": "built_in",
19  "rule_version": 1,
20  "technique": "vmray_modify_memory",
21  "technique_desc": "\"c:\\users\\user\\desktop\\5↩
        a2a43a3ddb9f5ac7253fed7cfe7c5425322411c1df407577e5a2a11b90e353a↩
        .exe\" modifies memory of \"c:\\users\\user\\desktop\\5↩
        a2a43a3ddb9f5ac7253fed7cfe7c5425322411c1df407577e5a2a11b90e353a↩
        .exe\".",
22  "technique_path": "built_in._injection.↩
        _modify_memory_non_system.vmray_modify_memory",
23  "type": "vti_rule_match",
24  "version": 3
25  [...]
```

Listing 5.2: VTI rule match for code injection (excerpt from the `summary.json` file)

The same conclusion is also applicable to the other code injection techniques shown in Table 5.5. The detections by VMRay Analyzer for the techniques DLL injection (T1055), direct code injection (T1055) and IAT hooking (T1179) are false positive correlations caused by the unspecificity of the technique definition in the MITRE ATT&CK® Matrix. We identified the false positives by manually assessing the analysis report for the correlated detections of VMRay Analyzer.

For instance, the IAT hooking technique T1179 also includes the hooking of procedures with the `SetWindowsHookEx` API call [89, 93]. This technique is for example used by keyloggers to intercept user inputs like keystrokes or mouse movements. It is exactly this behaviour that was detected by VMRay instead of the IAT hooking technique. As a result, eight of the samples that were identified by our signature for IAT hooking also hook certain procedures for intercepting information.

By manually reviewing our detection results for the IAT hooking feature, we classified all of our detections as false positives. This clearly shows a limitation of our approach. The IAT hooking feature is very unspecific on the API call level since common API function calls (dynamic library load, string comparison and memory allocation) are leveraged to implement this feature [48]. Hence, the API call sequence used in our dynmx signature is to unspecific for a precise detection. For a precise detection, we would need to check for valid Windows API call function names in the string comparison detection step which cannot be defined in our proposed signature DSL. Furthermore, with our detection approach it cannot be determined if the allocated memory lies within the IAT of a certain process.

As for the DLL injection and direct code injection technique T1055, we classify the found correlations as false positives since the actual API call that is related by VMRay Analyzer to this detected technique is also the `SetWindowsHookEx` API call. This relation can be traced in the `summary.json` file of the concerned sample[6]. As shown in Listing 5.3, two VTI detections are referenced. By following these references and the subsequent references in the VTI detections to the actual function calls in the VMRay generic function log (see Listing 5.2 lines 7-13 for a sample reference to the generic function log), the API call `SetWindowsHookEx` is identified for both of the VTI detections. Because the API function call `SetWindowsHookEx` is referenced in the hooking technique T1179 according to the MITRE ATT&CK® Matrix, we consider this mapping incorrect.

[6] SHA-256 hash value `6a4da7e983c492b9291e6e768451929a3f5169d98a4251fb0d4ede3e5d355b90`

```
[...]
{
  "description": "Process Injection",
  "id": "T1055",
  "ref_vtis": [
    {
      "ref_id": "built_in._injection._inject_file_system.↩
          vmray_inject_file_system",
      "ref_source": "summary",
      "ref_type": "vti_rule_match",
      "type": "reference",
      "version": 1
    },
    {
      "ref_id": "built_in._injection._inject_file_non_system.↩
          vmray_inject_file_non_system",
      "ref_source": "summary",
      "ref_type": "vti_rule_match",
      "type": "reference",
      "version": 1
    }
  ],
[...]
```

Listing 5.3: Mapping of T1055 technique to VTI detections (excerpt from the `summary.json` file)

Analogous to the process hollowing technique, we identified the VTI detections `vmray_modify_memory` (or `vmray_modify_memory_system`) and `vmray_create_remote_thread_system` (or `vmray_create_remote_thread_non_system`)[7] for the verification of the dynmx detection results for the techniques DLL injection and direct code injection (T1055). A large subset of our detected samples for these techniques also showed these VTI detections (99.4% of DLL injection detections, 99.6% of direct code injection detections). Seven of our detections for the DLL injection technique did not show the mentioned VTI detections although the API calls that cause the VTI detections were present in the corresponding function logs. Overall, we are confident that our detection results are correct. It is noted, that the DLL injection technique is specialised form of the direct code injection technique. The actual dynmx signature for DLL injection defines two more API calls than the signature for direct code injection. Hence, all of the function logs that have been detected by the DLL injection signature are also detected by the direct code injection signature. This sustains the same limitation of our approach explained for the IAT hooking technique. The DLL injection and direct code injection techniques are too similar on the API call level since the exact API call sequence used to detect the direct code injection technique is also part of the DLL injection signature. Consequently, both of the signatures are detected.

The results for persistence techniques are similar to the previously described results for code injection techniques. For instance, the persistence techniques Winlogon helper DLLs (T1004) as well as AppInit DLLs (T1103) are not detected and mapped to the MITRE ATT&CK® Matrix by VMRay Analyzer as these IDs are not found for any sample in our malicious data set. Furthermore, there are no detections by the VTI engine. For these techniques we verified our results by leveraging the information on accessed resources (artifacts in terms of VMRay Analyzer) provided by the sandbox. The information on accessed resources can be found in the `artifacts` key which is also part of JSON-formatted `summary.json` file. The verification resulted in no false positive dynmx detections.

```
"artifacts": {
  "registry": [
    {
      "operations": [
        "access"
```

[7] This VTI technique corresponds to the `CreateRemoteThread` API call.

```
6         ],
7         "reg_key_name": "HKEY_LOCAL_MACHINE\\System\\↩
              CurrentControlSet\\Control\\Session Manager",
8         "type": "registry_artifact",
9         "version": 1
10      },
11  [...]a
```

Listing 5.4: Accessed resources in the `summary.json` file)

As for the most common persistence technique based on the Registry run keys (T1060), we verified 96.16% our detections. The discrepancy of 372 detections is caused by the circumstance that VMRay Analyzer only detects a subset of the Registry keys and values mentioned for the technique in [98].

The same is applicable for gaining persistence by installing a new service (T1050). Over 99.9% of our findings were also detected by VMRay Analyzer. The seven missing detections were manually assessed. It turned out that they leverage the tool `installutil.exe` [84] for persistence or install a kernel driver as service. The latter behaviour is detected by VMRay Analyzer but not mapped to the corresponding technique T1050 in the MITRE ATT&CK® Matrix. By manually assessing our detection results, we identified one false positive detection in a sample[8] that deleted the tool `installutil.exe` rather than using it for gaining persistence.

Generic Malware Feature	dynmx Detections	VMRay Analyzer Detections	FP Correlations	Detection Discrepancy	Detection Discrepancy in %	FP dynmx Detections
Process hollowing / process replacement (T1093)	5,130	0	0	5,130	100.00%	0
DLL injection (T1055)	1,267	643	643	1,267	100.00%	0
Direct code injection (T1055)	2,236	645	645	2,236	100.00%	0
IAT hooking (T1179)	65	8	8	65	100.00%	65
Windows registry run keys (T1060)	9,685	9,313	0	372	3.84%	0
Winlogon helper DLLs (T1004)	1,816	0	0	1,816	100.00%	0
AppInit DLLs (T1103)	96	0	0	96	100.00%	0
Service installation (T1050)	7,234	7,227	0	7	0.01%	1
Scheduled task (T1053)	3,813	3,624	0	189	4.96%	0

Table 5.6: Revised evaluation result for the detection of generic malware features in the malicious data set

Again, with reference to the detection results of installing a new scheduled task (T1053), the detection discrepancy of 4.96% is caused by the fact that VMRay Analyzer does not detect certain ways of installing a new scheduled tasks. We assessed the detection discrepancy manually and did not identify any false positive dynmx detections.

By incorporating the findings on false positive correlations and detections, we revise and refine our results overview as shown in Table 5.6. In addition to the initial results overview, we added the columns for false positive (FP) correlations and FP dynmx detections. Please note, that the table is based solely on the correlations based on the MITRE ATT&CK® Matrix and not on VTI detections.

[8] SHA-256 hash `81459bde729a97eef8c49d187695b977bba5da2c40ee19abb21d823126c1f5b1`.

Since we used several information sources for verifying our detection results, Table 5.7 gives an overview of the different verification sources that we have taken into account. These verification sources are part of the VMRay analysis archive. In addition to the stated verification sources, for all of the detections we verified our results manually to a certain extent depending on the quantity of detections. If the number of detections was too high for manually verifying them, we verified a random subset of the same.

Generic Malware Feature	**Verification Source**
Process hollowing / process replacement (T1093)	VTI engine detections (VTI rules `vmray_modify_memory` or `vmray_modify_memory_system`, `vmray_modify_control_flow_non_system` or `vmray_modify_control_flow_system`, `vmray_allocate_wx_page`)
DLL injection (T1055)	VTI engine detections (VTI rules `vmray_modify_memory` or `vmray_modify_memory_system`, `vmray_create_remote_thread_system` or `vmray_create_remote_thread_non_system`)
Direct code injection (T1055)	VTI engine detections (VTI rules `vmray_modify_memory` or `vmray_modify_memory_system`, `vmray_create_remote_thread_system` or `vmray_create_remote_thread_non_system`)
IAT hooking (T1179)	VTI engine detections (VTI Rule `vmray_allocate_wx_page`)
Windows registry run keys (T1060)	MITRE ATT&CK® Matrix and registry resources
Winlogon helper DLLs (T1004)	Registry resources
AppInit DLLs (T1103)	Registry resources
Service installation (T1050)	MITRE ATT&CK® Matrix
Scheduled task (T1053)	MITRE ATT&CK® Matrix

Table 5.7: Sources used for the verification of detected generic malware features in the malicious data set

Results for the Benign Data Set

By applying the exact same evaluation methodology to the benign data set, we got the results shown in Table 5.8. As presented in the table, our signatures were not detected in the benign data set. Consequently, there is no detection discrepancy.

5.2.2 Application on Malware Families

Following on from the question asked in Section 4.2.2 whether a certain malware family can be characterised by their behaviour on the API call level, we will present our evaluation results and thus give indications for the answer to this question. Analogous to the previous section, we will first outline the evaluation methodology and present the results afterwards. As explained in Section 4.2.2, the definition of signatures for specific malware features is an iterative manual process. Therefore, the set of malware families chosen for the evaluation is not extensive but carefully chosen and of high practical relevance.

Generic Malware Feature	dynmx Detections	FP Detections
Process hollowing / process replacement (T1093)	0	0
DLL injection (T1055)	0	0
Direct code injection (T1055)	0	0
IAT hooking (T1179)	0	0
Windows registry run keys (T1060)	0	0
Winlogon helper DLLs (T1004)	0	0
AppInit DLLs (T1103)	0	0
Service installation (T1050)	0	0
Scheduled task (T1053)	0	0

Table 5.8: Evaluation result for the detection of generic malware features in the benign data set

Methodology

Since we know with high confidence the family affiliation for certain samples that are part of our malicious data set based on the information provided by the MCE tool (see Section 5.1.2), we have a ground truth data set. This ground truth data set is used for classifying our detection results and thus calculate the actual detection rate. These results origin from detecting our signatures for specific malware features in our malicious and benign data set. For instance, by correlating our detection results with the ground truth data set for a certain family as well as with the ground truth data set for other known families, we can verify our signatures by identifying true positive, false positive and false negative detections. The primary key used for correlating the detections with the ground truth data set is the SHA-256 hash sum of the sample. For our detection evaluation of specific malware features for a certain family, we apply the following classification schema:

True positive (TP) The detected sample belongs to the correct family covered by the signature and to no other families in the ground truth data set.

False positive (FP) The detected sample belongs to other families in the ground truth dataset not covered by the signature.

False negative (FN) The sample belongs to the family covered by the signature based on the ground truth data set but remains undetected by the signature.

Based on the result of this correlation and classification, we can measure the accuracy of our proposed dynmx signatures for the malware families named in Section 4.2.2 with the F_1 *score* (also referred to as F-score) [35, 69]. The F_1 score is well-adopted for evaluating the accuracy and quality of classifiers. The calculation of the F_1 score requires the *precision* P and the *recall* R to be calculated beforehand. Equation 5.3 shows the calculation of the F_1 score.

$$P = \frac{TP}{TP + FP} \tag{5.1}$$

$$R = \frac{TP}{TP + FN} \tag{5.2}$$

$$F_1 = \frac{2 \cdot P \cdot R}{P + R} \tag{5.3}$$

In addition to the F_1 score calculation, we identify additionally detected samples in the overall data set. These additionally detected samples are not part of any malware family based on the information provided by MCE and Malpedia. Hence, they belong to the "Unknown" category shown in Section 5.1.2 Table 5.1. In order to find false positive detections we assess these additionally detected sample set in a further step of the evaluation. Therefore, we leverage the scan report provided by VirusTotal. For instance, we fetch the scan report for each additionally detected sample and extract the scan results (threat names) of the following anti-virus products:

- Avira,
- ESET-NOD32,
- F-Secure,
- Kaspersky,
- Microsoft,
- McAfee and
- ZoneAlarm by Check Point.

If three of the seven chosen anti-virus products detected the sample with a threat name that contains the family name that our detection suggests, we automatically classify the sample as TP. Samples that are not detected by the anti-virus products remain unclassified. We classify a sample as FP if at least three of the seven products detect the sample with the same family name which is different from the family that our detection suggests. Generic threat names are thereby not considered as family name. It is noted, that the classification based on threat names can only serve as an indicator and does not prove that a certain sample belongs to a family.

Results for the Malicious Data Set

By applying the aforementioned methodology to our malicious data set, we obtained the results shown in Table 5.9. The shows the absolute TP, FP and FN figures together with the calculated precision, recall and F_1 score. The last column of the table presents the absolute number of additionally detected samples. Please consider, that the detections stated in the table below are unique per SHA-256 sample hash. Consequently, the number of detections, TP, FP, FN and additionally detected samples shown in Table 5.9 does not directly correspond to the number of function logs but instead to the number of unique samples shown in Table 5.1.

As shown in the results table, our proposed signatures caused no false positive detections in the malicious data set. Hence, the calculated precision P of the signatures is throughout all considered malware families 1. This result is promising and interesting on its own, as it suggests that samples which belong to a certain malware family can be precisely characterised based on the API call usage. The recall R is with values between 0.9919 and 0.9984 also on a high level. This generally means, that only a small portion of the known samples for a certain malware family remain undetected. Expressed in percentage, the rate of undetected samples varies between 0.16% (Remcos) and 4.31% (Formbook). As a result of the high precision and recall, our signatures reach a high F_1 score that varies between 0.9789 and 0.9992.

Malware Family	**dynmx Detections**	**TP**	**FP**	**FN**	P	R	F_1	**Additionally detected**
DarkComet	2,428	487	0	4	1.0	0.9919	0.9959	1,941
Emotet	5,724	5,069	0	24	1.0	0.9953	0.9976	655
Formbook	493	116	0	5	1.0	0.9587	0.9789	377
Remcos	2,774	610	0	1	1.0	0.9984	0.9992	2,163

Table 5.9: Evaluation result for the detection of specific malware features in the malicious data set

We manually assessed the function logs of the samples that were classified as false negative in order to find reasons for the missing detections. As for the Emotet family, the FN samples showed none of the characteristics defined in our signature and one of the Malpedia samples could not be executed in the sandbox environment. As some of the FN samples were obtained from the Malpedia corpus and thus have a timestamp, we deduced that our signature only detects recent Emotet samples starting from the beginning of 2020. Older samples from 2018 and earlier remained undetected

during our evaluation. In addition, some samples that were first seen on VirusTotal in the end of 2019 and origin from our MCE tool could not be detected by our signature. As a result, we conclude that our Emotet signature detects samples that were seen in the wild from beginning of 2020 onwards.

As for the DarkComet family, our analysis of the FN samples shows that very recent samples (first seen on VirusTotal beginning from May 2020) remained undetected. Some of these samples showed certain parts of the characteristic behaviour defined in our signature.

Since a relevant subset of the Formbook samples showed little activity in the VMRay Analyzer sandbox based on the function logs, developing a signature was challenging. Nevertheless, the detection result shows that even with little activity during the dynamic analysis in the sandbox characteristics can be found in order to define a stable signature. The VirusTotal upload timestamps for the Formbook FN samples, in contrast to the other families, drawed no clear picture as there were samples that were very recent as well as samples that were uploaded to VirusTotal six months ago.

Malware Family	Additional dynmx Detections	TP	FP	Unclassified
DarkComet	1,941	1,824	0	118
Emotet	655	627	0	28
Formbook*	377	125	0	253
Remcos*	2,163	1,335	0	828

Table 5.10: Evaluation result for the additionally detected samples in the malicious data set (*) lower confidence)

After we have outlined the detection result with reference to the ground truth data set, we will focus on the additionally detected samples in the following. As described in the methodology, we obtained the VirusTotal scan reports for all of the additionally detected samples identified by their SHA-256 hash sum. The following Table 5.10 shows the classification results of the additionally detected samples categorised by the malware family. While the approach of using the threat names to classify the additionally detected samples worked satisfactorily for the DarkComet and Emotet malware family, the approach did not work for the classification of additionally detected Formbook and Remcos samples. The threat names for the Formbook and Remcos samples have been diverse and drawed no clear picture. Hence, we changed our initial methodology to classify samples that have at least one threat name that contains “Formbook” or “Remcos” automatically as TP. Thereby, we could reduce the number of unclassified samples by accepting that the results have a lower confidence level. This is indicated by the * character in Table 5.10. Further, we manually assessed some of the additionally detected Formbook samples by unpacking the samples with the UnpacMe service [177] and searched for the hash of the unpacked sample on VirusTotal.

Results for the Benign Data Set

Analogous to the detection of our signatures for generic malware features, the signatures for specific malware features produced no false positive detection in the benign data set. The detailed results are shown in Table 5.11.

5.3 Signature DSL

After we have presented the evaluation results of our detection approach, we will focus in our proposed DSL that is used to define the dynmx signatures used for the detection in the following. In

Malware Family	dynmx Detections	FP Detections
DarkComet	0	0
Emotet	0	0
Formbook	0	0
Remcos	0	0

Table 5.11: Evaluation result for the detection of specific malware features in the benign data set

order to motivate the development of our proposed DSL instead of leveraging the already in-place possibility to define custom VTI rules for the detection of malware features, we analysed whether our proposed signatures could have been implemented as custom VTI rules within VMRay Analyzer sandbox. In the following, we will present the methodology as well as the results of this evaluation.

Methodology

With reference to the background Section 2.6.3 on the VTI engine, the VTI rules are matched against the generic function log. This generic function log[9] abstracts from the actual function log by mapping multiple API function calls to an abstracted form of the function call – the generic function call. Hence, in order to answer the question whether a certain dynmx signature could have been implemented as VTI rule, we need to find an appropriate generic function call in the generic function log for each detection step in our dynmx signature. If certain API function calls used in our dynmx signature are not mapped to a generic function call and are thus not part of the generic function log, we could have not used a VTI rule for the detection as the sequence is not complete in the generic function log. To identify relevant samples and, consequently, relevant generic function logs, we use the detection results presented in the previous section. For instance, we choose detected samples for each of our developed signature, collect the generic function log for these samples and perform the described analysis.

Results

The detailed results of our signature DSL evaluation are shown in Table 5.12. The column “Implementable with VTI Rule” in the table shown, indicates whether the dynmx signature could have been defined with a VTI rule. Thus, this column reflects whether all of the used API calls in the dynmx signature are also part of the generic function log. If a signature could not have been implemented with a VTI rule, the column “Missing API Calls” states the exact API calls that are not part of the generic function log, i.e. the API calls that are not mapped to a generic function.

The result shows that only four of the thirteen proposed dynmx signatures can be implemented as VTI rules. Please note, that we used a strict evaluation schema and the results have to be interpreted in this light. Hence, if only one of the API calls used in the dynmx signature is not part of the generic function log we classify the signature as not implementable with a corresponding VTI rule. In particular, for generic malware features which can be implemented with different API call sequences leveraging different API function calls, the dynmx signature is partially implementable. For example, a signature for a DLL injection implementation that uses the API call `VirtualAllocEx` instead of `VirtualAlloc` can be implemented with a VTI rule since the `VirtualAllocEx` is mapped to a generic function.

With reference to specific malware features, our proposed signatures could not have been implemented with VTI rules since elementary parts of the signatures are not found in the generic function log. For instance, API calls used to format strings, certain cryptographic functions, functions to find and load resources from executables and uncommon NTAPI system calls.

[9] The generic function log is represented by the file `glog.xml` which is part of the VMRay analysis archive.

dynmx Signature	Feature Type	Implementable with VTI Rule	Missing API Calls
Process hollowing / process replacement (T1093)	Generic	No	`VirtualAlloc`
DLL injection (T1055)	Generic	No	`VirtualAlloc`
Direct code injection (T1055)	Generic	No	`VirtualAlloc`
IAT hooking (T1179)	Generic	No	`VirtualAlloc`
Windows registry run keys (T1060)	Generic	Yes	
Winlogon helper DLLs (T1004)	Generic	Yes	
AppInit DLLs (T1103)	Generic	Yes	
Service installation (T1050)	Generic	Yes	
Scheduled task (T1053)	Generic	No	
Emotet	Specific	No	`_snprintf`, `_swnprintf`, `CryptExportKey`, `CryptGetHashParam`
DarkComet	Specific	No	`FindResourceA`, `LoadResource`
Formbook	Specific	No	`RtlQueryEnvironmentVariable_U`, `RtlDosPathNameToNtPathName_U`
Remcos	Specific	No	`FindResourceA`, `LoadResource`

Table 5.12: Signature DSL evaluation result

But not only whether a signature could have been implemented with a custom VTI rule has to be taken into account for the evaluation of our dynmx signature DSL. In contrast to custom VTI rules that need to be implemented as Python code within the VMRay Analyzer framework, our proposed signature DSL abstracts from the underlying programming language allowing the domain expert on defining the behaviour to be detected rather than on the actual implementation. Furthermore our signatures are sandbox-agnostic and only require the input function log to be compliant with the MSDN library in terms of the API function and the parameter naming. This can be beneficial in practical situations where several different sandboxes are used for the dynamic analysis of malware.

Since our signature DSL is based on the function log that reflects the whole complexity of the Windows API a profound knowledge of the same is required in order to define good signatures. This is especially the case for generic malware features since there are multiple ways, i.e. API call sequences, of implementing the exact same feature. From this point of view, the VTI rule definition has advantages over our signature DSL as the signature author can take advantage of the generic function log which already abstracts from the complexity of the Windows API by mapping several function calls to one generic function call. However, as explained above, this leads to a reduced specificity in comparison to our proposed DSL.

5.4 Performance

To conclude the evaluation, we review the performance of our dynmx prototype implementation. The first part of this section will provide figures on the overall detection performance and addresses certain performance issues that we identified during the detection evaluation presented in Section 5.2 in the beginning of this chapter. Afterwards, we evaluate the performance of our proposed

generic function log format in terms of the storage consumption and the parsing process. We will begin this section by outlining the environment that we used for our evaluation. The raw information the figures presented in the following rely on are provided together with this thesis in digital form and can be found on the enclosed flash drive[10].

5.4.1 Environment

We used a standalone virtual machine deployed on a VMware vSphere cluster (VMware ESXi version 6.7.0 13006603) for our evaluation. The detailed system information of this virtual machine is shown in Table 5.13. We chose Ubuntu 18.04.4 LTS as operating system for our evaluation VM. In order to meet the installation requirements of our dynmx protoype, we installed Python version 3.7.7 and the required external Python packages as described in Section 4.1.1.

Virtual CPUs	32
Memory	128 GB
Storage	4 TB (SSD storage)
VMware VM Version	13

Table 5.13: Evaluation VM system information

Details of the host system that runs the VMware ESXi hypervisor and thus the evaluation VM is outlined in Table 5.14.

CPUs	2 x Intel Xeon Gold 6138 (20 cores per CPU, 2.00 GHz per Core)
Memory	1 TB
Storage	RAID6 SSD storage (11 x Intel SSD Model SSDSC2KB960G7)

Table 5.14: Evaluation host system information

Please note, that the host system is part of a shared platform and does not exclusively run our evaluation VM. Further, there is no resource and performance guarantee on this platform. Consequently, the CPU and storage consumption of other VMs on the host system can impact our detection performance.

5.4.2 Detection

For the sake of brevity, we present the runtimes measured for the detection of dynmx signatures in the malicious data set. The benign data set is out of scope for evaluating the detection performance since it is smaller and less diverse than the malicious data set. The runtimes were measured with the function logs in the generic function log format proposed in Section 3.3 of this thesis. Table 5.15 presents the detection runtimes for the malicious data set categorised by the signature. The presented runtimes are based on the usage of 30 parallel running worker processes. The table outlines the overall runtime as well as the average detection and resource extraction runtime per function log in seconds. The detection runtime is measured for the whole detection algorithm and includes also the function log reduction. Since resources are only extracted if a signature detects behaviour based on resources, the extraction runtime is not stated for each signature.

[10] The raw information regarding the performance evaluation is found in the directory `Digial_Appendix/5_Evaluation/Performance/` and its subdirectories on the enclosed flash drive. It is noted, that the files are typically compressed due to the large file size.

With reference to the overall runtime, the Formbook dynmx signature took the least time for detection in the malicious data set. This is caused by the circumstance that the Formbook signature contains a relatively short API call sequence consisting of uncommon API functions. Overall, this leads to a small number of API call candidates in the detection process which, in turn, leads to a small LCS matrix. In contrast, the longest runtime was consumed for the detection of the DLL injection malware feature. The long runtime was mainly caused by five function log outliers that had detection runtimes over 1,000 seconds.

dynmx Signature	**Feature Type**	**Overall runtime [s]**	**Avg. Detection runtime [s]**	**Avg. Resource Extraction runtime [s]**
Process hollowing / process replacement (T1093)	Generic	2,665.8992	0.0502	-
DLL injection (T1055)	Generic	15,826.759	1.393	-
Direct code injection (T1055)	Generic	2,601.5829	0.015	-
IAT hooking (T1179)	Generic	3,105.9019	0.1714	-
Windows registry run keys (T1060)	Generic	6,432.3313	1.3406	1.3289
Winlogon helper DLLs (T1004)	Generic	4,209.413	0.52	1.271
AppInit DLLs (T1103)	Generic	4,990.0892	0.3645	1.305
Service installation (T1050)	Generic	8,389.8004	0.5824	-
Scheduled task (T1053)	Generic	5,354.9568	0.2734	-
Emotet	Specific	2,609.1099	0.0426	-
DarkComet	Specific	4,551.6954	0.2206	1.2985
Formbook	Specific	2,544.4426	0.0010	-
Remcos	Specific	4,742.4910	0.4411	-

Table 5.15: Detection runtimes of dynmx signatures in the malicious data set (30 worker processes)

These function logs causing long running detections typically contain a large number of processes where each process contains a large number of API call candidates for a certain signature but not the actual sequence defined in the signature. A further driver of complexity in the detection process is the usage of variables in signatures. For instance, a single function log[11] in the DLL injection run caused a detection runtime of 13,704.1443 seconds (3 hours 48 minutes). This function log contains 69 processes where 45 of these processes contain at least 64 API call candidates. Furthermore, at least 55 variable contexts were identified for each of the 45 processes. In sum and under consideration that the detection path for the DLL injection feature consists out of six steps, this leads to $45 \cdot 55 \cdot 6 \cdot 64 = 950,400$ detection operations. Figure 5.1 shows the 1000 longest detection runtimes for the DLL injection signature in the malicious data set. Please consider the logarithmic scale of the y-axis. Only the first 64 function logs have a detection runtime above 60 seconds and the first 102 function logs have a detection runtime above 10 seconds. This sustains our hypotheses that only a small portion of the function logs that meet certain criteria described before cause extensive overall runtimes. For 99.997% of the function logs in the malicious data set the detection runtime was below 10 seconds and thus in an acceptable range.

[11] Sample with SHA-256 hash value `c9c9a49167ab5a5dfb45e9165321747d2e3d78ad547fa9cbcc4ef5aa30cd296b`

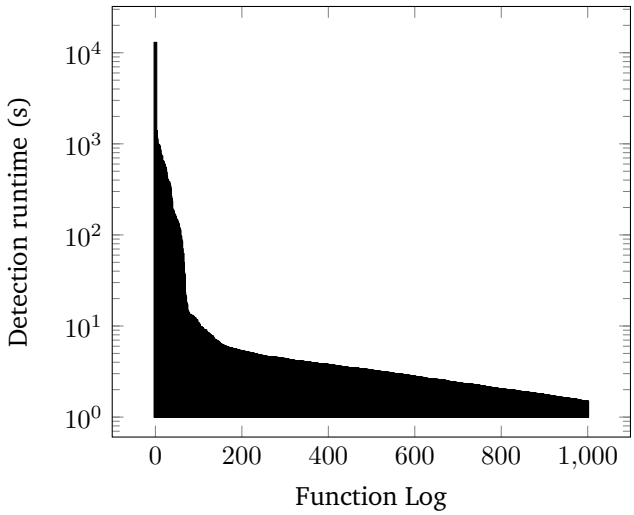

Figure 5.1: Distribution of runtimes of the DLL injection signature detection in descending order (limited to 1000 runtimes)

5.4.3 Generic Function Log Format

In order to evaluate the performance of our proposed generic function log format, we compare the storage consumption together with the time average needed to parse a function log with the two function log formats provided by VMRay Analyzer. Table 5.16 shows the results of this evaluation. Again, the figures shown in the table are based on the malicious data set described in Section 5.1.2.

Function Log Format	Storage Consumption [bytes]	Avg. Parsing runtime [s]	Median Parsing runtime [s]	Max. Parsing runtime [s]	Min. Parsing runtime [s]
VMRay text-based	101,275,620	2.087	1.1504	171.588	0.0006
VMRay XML	421,835,452	4.2073	1.7186	393.2799	0.0004
dynmx	10,435,876	2.2113	1.0525	163.0603	0.0004

Table 5.16: Function log format evaluation

As shown in the table, our proposed generic function log format only consumes 2.47% (XML-based VMRay function log format) resp. 10.3% (text-based VMRay function log format) of the storage space compared to the VMRay function log formats. This reduction of storage consumption is achieved by leveraging compression for our proposed function log format. Of course, the function logs provided by VMRay could have also be stored compressed in order to reduce the consumed storage but we compare the formats as they are provided by the sandbox. With reference to the average parsing time the dynmx function log format was parsed almost two times faster than the same function logs in the XML-based VMRay function log format and slightly slower than the text-based function log format of VMRay. Especially for large and complex function logs, as seen on basis of the maximum parsing time, our format was parsed faster than the other formats provided by VMRay by still being machine-readable. However, the median parsing runtime is the more interesting figure for comparing runtimes since the set of recorded parsing runtimes varies greatly as indicated by the maximun and minimum runtime stated for each function log format. Complex and long function logs with many monitored processes lead to disproportional long parsing times that, in turn, skew the average parsing time. Hence, the median runtime represents the typical parsing

runtime more precisely than the arithmetic mean. With reference to the median parsing runtime, our proposed function log format was performing best. Nevertheless, it has to be taken into account, that we designed the generic function log format particularly for our use case. Consequently, we incorporate only needed information rather than aiming for completeness.

For the evaluation of the memory usage for each function log format, we chose to measure the detection run of the Registry run keys dynmx signature (T1060). The reasons for choosing this signature in particular are that this signature produced the most detections in our malicious data set. Furthermore, the access activity model needs to be derived from the function log in order to detect the signature. Consequently, we expect a higher memory usage in comparison to signatures that do not leverage resources for detection and that do not produce that much positive detections. For profiling the memory usage, we chose the python tool `memory-profiler` [114]. The results of the memory profiling are shown in Figure 5.2.

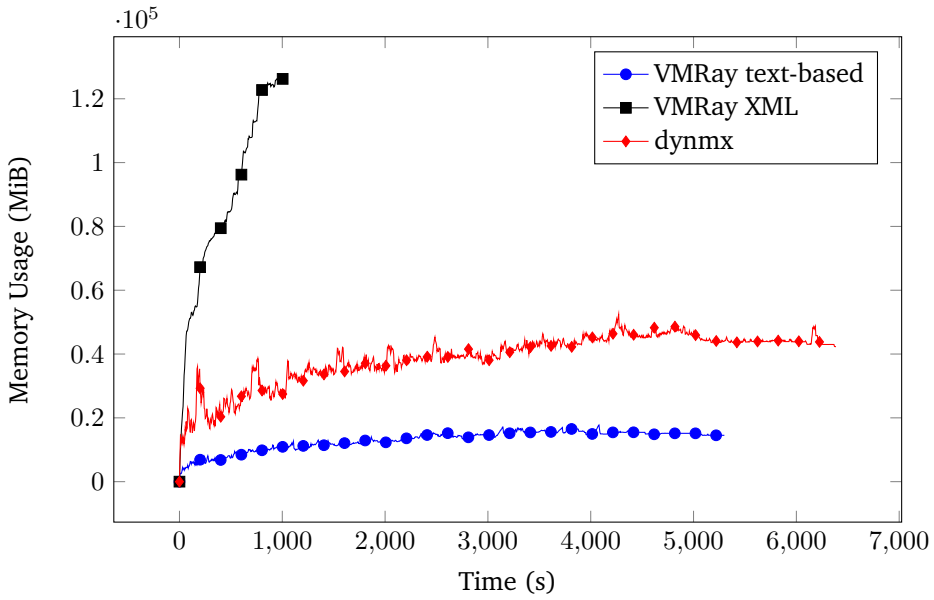

Figure 5.2: Memory usage per function log format (30 worker processes, Registry run keys dynmx signature)

In terms of the memory usage, the performance of the text-based VMRay function log format was best with a highest memory usage of 17,759 MiB during the measured detection run. Our proposed dynmx function log format used 52,454 MiB memory at maximum. The detection run for the XML-based VMRay function logs could not be completed since the whole 128 GB memory of the virtual machine was used after 1,018 seconds and caused the detection run to terminate. In order to complete the detection run on our evaluation VM we needed to reduce the amount of worker processes to 20. With this configuration the whole memory was used during the detection but the process did not terminate prematurely.

5.5 Discussion

In the previous sections we evaluated our signature-based detection of generic and specific malware features in three different dimensions. While the first and most important dimension evaluated the actual detection quality of our proposed prototype implementation as well as our signatures for certain generic and specific malware features, the second dimension focused on our proposed DSL by working out the advantages in comparison with custom VTI rules. In the third dimension, we outlined several performance indications and thus sustained the practicability of our prototype implementation.

As for the first dimension, our evaluation results clearly show that the majority of generic malware features are reliably detected within a set of 31,898 function logs originating from the dynamic analysis of known malware in the VMRay Analyzer sandbox. With reference to the nine generic malware features taken into account, only the IAT hooking malware feature could not be detected by our proposed signature due to the unspecificity of the API call sequence that characterises this feature. The other eight signatures for the detection of generic malware features produced reliable detection results at an almost zero false positive rate. We identified only one false positive detection for the service installation persistence mechanism. Our initial approach of correlating our detection results with the detection results of VMRay Analyzer based on the MITRE ATT&CK® Matrix could only be applied reliably for three of nine generic malware features since VMRay Analyzer does not map all of the detected malicious behaviour to corresponding techniques defined in the MITRE ATT&CK® Matrix. Further, we identified mappings that we consider wrong based on the description provided for the corresponding technique. Hence, for the verification of our detection results, we leveraged additional information provided in the analysis archive by the VMRay Analyzer sandbox. For instance, we used VTI detections as well as information on resources acquired by the monitored processes.

Compared to the built-in detection capabilities of the VMRay Analyzer sandbox, we detected more samples that implement a certain generic malware feature in all cases. The reason for the broader detection lies in our proposed signatures that detect more ways of implementing a certain feature than the built-in detection capability of the VMRay Analyzer sandbox. Furthermore, we classify our detection results more precise according to the MITRE ATT&CK® Matrix, especially for the malicious code injection category. In conclusion, we improved the overall detection capabilities with our proposed dynmx protoype implementation together with the prototype signature library. As a result, the precise detection of generic malware features helps a domain expert to initially assess and triage an unknown malware sample. For instance, in relation to our malicious data set, the detection results can help to characterise the unknown portion of function logs.

With reference to specific malware features, we precisely detected samples belonging to a certain malware family with zero false positives resulting in a high F_1 score for all of the four malware families taken into account. These results are promising for the classification of malware and suggests that there is API call and resource usage which is characteristic of a certain malware family. The information provided by the MCE tool was beneficial for our evaluation since we had a ground truth data set that we used to quantify the quality of our detection approach. For all of the four malware families, we identified additional samples in the malicious data set. These additionally detected samples were assessed based on scan reports collected from VirusTotal in order to identify true positive and false positive detections. This evaluation methodology was not applicable for all malware families since the overall detection for certain families was not sufficient for a reliable classification. We provided the data set of additionally detected samples to the MCE tool developers to improve the automatic malware config extraction for these families.

The second dimension of our evaluation was focused on our proposed signature DSL. We demonstrated that our signature DSL has certain advantages compared to the built-in feature of the VMRay Analyzer sandbox to define custom VTI rules. As VTI rules are detected in the generic function logs which is an abstracted form of the function log, they cannot be as precise as dynmx signatures. Following this circumstance, our evaluation methodology focused on answering the question whether the proposed signatures could have been implemented with VTI rules based on the presence of function calls in the generic function log that we use in our signatures. The result of this evaluation was that only four of our thirteen proposed signatures for generic and specific malware features could have been implemented with VTI rules in the exact same form. But, as a downside of our proposed signature DSL, to operate this specificity, a well-founded knowledge of the Windows API is required compared to the definition of VTI rules. But exactly this specificity is needed for the detection of malware families based on specific malware features. Benefits of our proposed signature DSL are seen in the sandbox-agnostic approach that abstracts from the actual implementation. Consequently, domain experts can focus on defining the behaviour without prior knowledge of implementation specifics of certain sandbox solutions.

By evaluating the performance of our proposed prototype implementation, we sustained the practical relevance. In the worst case scenario, the detection of a signature in the complete malicious data set consisting of 31,898 function logs took 4 hours and 23 minutes on a single machine with thirty parallel worker processes. In the best case scenario the detection took 43 minutes. Drivers for long runtimes are first of all a high number of processes in the function log where each of the process contains a large number of candidates for the API function calls that are defined in a certain signature. Some of the function logs contained over 1,000 processes that were initiated by the malware. This scenarios are challenging for our proposed detection algorithm, as the LCS matrix that is calculated for each combination of detection block and process becomes large. This, is turn, leads to a high number of expensive detection operations driving the long detection runtime. Furthermore, certain malware categories that use a lot of resources, e.g. ransomware, can lead to long runtimes in certain situations where the dynmx signature uses resource to detect malicious behaviour. In terms of the scalability, detection approaches that purely rely on static analysis of the malware sample directly perform obviously better than our approach since they do not require the sandbox analysis which is time consuming. Nevertheless, they lack of semantics incorporated into the detection process and thus precision. However, both of the approaches have their right to exist and need to be chosen flexible with respect to the use case.

With reference to our proposed generic function log format, we demonstrated in the course of the evaluation that our proposed format is more efficient in terms of the storage consumption compared to the function log formats used by VMRay Analyzer. This efficiency is achieved at a lower median parsing time. As for the memory usage, the detection based on the generic function log format uses less than half of the memory that is used by the detection based on the VMRay XML function log format while also being machine-readable. Consequently, more worker processes can be executed concurrently leading to lower overall runtimes for the detection. However, in order to classify our results, it is indispensable to consider that we developed the generic function log format specially for our use case. The function log formats used by VMRay contain more information that can be beneficial in other situations beyond this work.

6 CONCLUSION

In this thesis, we presented the signature-based detection approach for behavioural malware features based on Windows API call sequences called *dynmx*. We introduced a novel, practice-oriented signature DSL together with a suitable detection algorithm based on the solution to the LCS problem. Our detection approach is characterised by being sandbox-agnostic and flexible in terms of the specificity of known malware features that can be detected. In contrast to related research, the integral part of our detection approach, i.e. the definition of signatures for malware features, behoves to the domain expert rather than aiming to fully automate the signature generation. In general, our detection approach can be classified as pendant to YARA rules for behavioural malware features. The generic function log format presented in this thesis helps in storing function logs in an efficient, sandbox-agnostic and machine-readable manner which allows further usage scenarios beyond this thesis. We implemented our detection approach as fully working proof-of-concept. Signatures for real-world generic malware features as well as specific malware features characterising malware families have been defined with the help of our proposed DSL and bundled in the form of a prototype signature library. We provide both of these implementations together with this thesis.

The main objective of this thesis, as outlined in the first chapter (Section 1.2), was the introduction of a signature DSL that can be used by domain experts for the definition of behavioural characteristics. This signature DSL based which is text-based and makes use of the general data representation syntax YAML is described in detail in Section 3.5. Summarised, the signature DSL gives the domain expert the possibility to define malicious behaviour (generic and specific malware features in terms of this thesis) based on API call sequences and the usage of operating system resources. In general, these characteristics are organised in independent detection blocks that can be related to each other in the signature's condition in the form of a binary expression. Each of these detection blocks consist of detection steps. This steps, in turn, define the actual characteristics of an API function call or resource that should be detected. For instance, this can be the API function name, parameter values, the return value, access operations on resources and the resource location. Several atomic detection matching operations for comparing values are provided which allows the signatures to become more flexible. Furthermore, the DSL offers more elaborate features like

the storage of variables which can be accessed in the course of the detection or path variants which make the signatures more concise. In order to make the signature available to the detection algorithm, we designed a signature parser which transforms the signature to a directed acyclic graph. The simplicity of our proposed text-based signatures allows domain experts to share developed signatures collaboratively. Furthermore, domain experts are allowed to concisely define analysis results based on the examination of malware behaviour with the help of our signature DSL. These developed signatures can be detected automatically afterwards which, consequently, eliminates the repetitive manual assessment of function logs for already known and examined malicious behaviour.

The second objective of this thesis was the design of a suitable detection algorithm that is needed in order to identify malware features in function logs based on signatures defined in our proposed DSL. We leveraged the well-known LCS algorithm as a basis for our detection algorithm (see Section 3.6) and adapted it to our needs. For instance, instead of comparing two elements in terms of their equality, we introduced the formal `detect` function which matches a given detection step with an API calls. In order to reduce the problem set for our detection algorithm, the function log is reduced based on the signature that should be detected prior to the LCS calculation. To detect signatures that make use of variables correctly, we introduced the concept of variable contexts. These contexts ensure that detections happen on the exact same variable values. As for the detection results, we focused on maintaining the traceability. As a result, domain experts can verify the detected malware features in detail based on the raw input function log since the actual line number or UUIDs of the detected API calls are provided.

The extraction of resources from the function log is based on the access activity model published by Lanzi et al. in [63]. We applied and extended the proposed approach in different ways. First of all we extended the scope of the extraction from only system calls to userland API calls. Moreover, we extended the extracted resource categories by network resources which play a vital role for present malware. We refined the access operations taken into account which can lead to a more fine-granular detection capability.

Based on the common data model for parsed function logs proposed in this work, we introduced a generic function log format in Section 3.3. In general, this generic function log is the serialised form of the common data model persisted as JSON-formatted string. By leveraging compression, large function logs can be stored efficiently. As the line numbers can not be determined by typical JSON parser libraries, we add UUIDs for API calls in order to remain the traceability. A further objective of our thesis was the implementation of the proposed detection approach. Therefore, we prototypically implemented the proposed design concept in form of a Python console application. This application is a fully working proof of concept implementation. Hence, all of the concepts described beforehand are implemented. For instance, the detection of signatures in function logs, the conversion of function logs to the generic function log format, the extraction of resources from function logs and gaining basic statistics of function logs. By leveraging parallelism with multiple processes for the detection and conversion process, runtimes are dramatically reduced. Thus, even large data sets like our malicious data set used in the evaluation can be processed in reasonable time depending on the number of processes that can be efficiently handled by the underlying hardware. together with the Python implementation, we developed a prototype signature library. This library provides, overall, 13 signatures for generic and specific malware features that were defined based on the manual assessment of function logs from relevant real-world malware samples. The generic malware features taken into account are malicious code injection and persistence techniques commonly used by malware authors. As for specific malware features, we considered the malware families DarkComet, Emotet, Formbook and Remcos as rather prominent representatives of today's malware landscape.

During our evaluation, we identified several generic malware features that remained undetected by the built-in detection capabilities of the VMRay Analyzer sandbox. Hence, we improved the overall detection capability with our proposed signatures and detection algorithm and gained valuable insights of the malicious data set that built the basis for our evaluation. We verified our detection results to a large extent by comparing them to several sources of information provided by the sandbox analysis results. Our detection results are characterised by a very low false positive

detection rate and are thus considered precise. As for specific malware features, our detection approach achieved high F_1 scores between 0.9789 and 0.9992 at the highest possible precision (zero false positive rate) for all of the malware families taken into account. Overall, these are promising detection results.

As for the signature DSL, we proved in the course of the evaluation that our proposed DSL can be used to define behavioural characteristics more accurate than the VTI rule engine provided by the VMRay Analyzer sandbox. This is caused by the circumstance that we base our detection on the actual function rather than the generic function log which abstracts from this function log by mapping multiple API function call used for the same operation to generic function calls. This mapping is, due to the high complexity if the Windows API, not complete. As a consequence, implementations of malware that leverage uncommon API function calls which are not part of the mapping, remain undetected by VTI rules. Our proposed generic function log format is more efficient in terms of the storage consumption as it makes use of compression and shows better median parsing times than the built-in function log formats provided by VMRay Analyzer sandbox while still being machine-readable. Our generic function log format keeps the original abstraction layer of the input function logs and therefore does not abstract from individual API calls. The overall performance of our proposed prototype implementation for such a large and diverse evaluation data set was satisfactory and sustains the practical relevance.

In the course of the present thesis, we identified several limitations of our detection approach that we do not want to leave unmentioned. First and foremost, we designed our detection algorithm process-centric. In general, we can therefore not detect malicious behaviour implemented by several independent processes that leverage interprocess communication for synchronisation. Furthermore, malicious behaviour that is implemented with several threads can only be partially detected. As our objectives already suggest, our detection approach relies on raw and unprocessed information provided by sandboxes. It is obvious, that the detection can only be as qualitative as the information provided by the sandboxes. In the course of this thesis this was no issue since the VMRay Analyzer sandbox provides comprehensive high-quality function logs. First analyses of other sandboxes revealed the quality of provided function logs varies and can become an issue since our detection approach aims to be sandbox-agnostic. During the evaluation we identified one generic malware feature that was not reliably detectable based on our proposed signature since the API call sequence characterising the feature was too unspecific and lead to false positives. In terms of the detection performance we identified certain function logs that lead to long running detection runtimes. These function logs typically contain a large amount of processes which, in turn, contain a large number of API call candidates for a certain signature.

To summarise our results, we refer to the research question posed in the motivation of our thesis. We clearly conclude, that the signature-based detection of runtime behaviour based on the API and resource usage can be beneficial for the malware analysis process in general. Our evaluation results suggest that our proposed detection approach has the potential to improve the efficiency of the analysis process from the viewpoint of domain experts by eliminating recurring manual tasks with the help of precise signatures. This is applicable for the detection of generic as well specific malware features. Furthermore, we allow domain experts to collaboratively share identified malicious behaviour in a standardised and accurate manner by leveraging our proposed signature DSL. This, in turn, can help domain experts to cope with the rising complexity and amount of malware arising in today's threat landscape.

7 Future Work

In the course of this thesis, we identified several research areas that are worth of addressing in future work. These are improvements to our proposed approach as well as novel detection approaches based on our research. As the enhancement of the signature library is an obvious future, work we will not address this topic in detail in the following.

Improvement of the Detection Algorithm

As shown the evaluation of the detection performance, a small fraction of function logs cause long detection run times. In order to clearly understand the dependence of our proposed algorithm to the complexity of the input function log a more detailed analysis of this correlation is needed at first hand. While first indications why these long running detection times occur are given in this thesis, a refinement of these indications is needed to substantiate the revision of our proposed algorithm. Furthermore, several improvements of the LCS algorithm in general are proposed in publications (e.g. [14, 32, 70, 132, 145]) which can be evaluated in terms of enhancing the detection time based on our proposed detection algorithm. To address the problem from another perspective, practical guidelines for domain experts in terms of defining effective signatures that minimise long running detections can be deduced from the evaluation of the problem.

Improvements of our detection algorithm can also be found in the limitations of this thesis. For instance, to detect malicious behaviour independent of the process or thread boundary, API calls of the function logs need to be merged into a single consistent sequence somehow. This merging process has to happen independent of the concrete time information information provided by input function logs as this time information is too unprecise for a naive merging approach. Instead, API calls need to be merged based on semantic context information. For instance, certain sequences need to follow a given order to be semantically meaningful. These semantically relevant orders can be applied in the merging process based on a heuristic algorithm. Another, possibly additional approach to this problem, is tracking inter-process communication. The malicious processes implementing a certain malicious feature in cooperation have the need for synchronisation. This synchronisation

activities can be typically identified in the function log and can therefore be leveraged to reconstruct the original order in that the API calls occurred independent from the process or thread boundary.

Integration of Further Sandboxes

In order to sustain our approach to be sandbox-agnostic, we aim to integrate further sandboxes or more precisely the behavioural runtime information provided by these sandboxes. Our initial research in this area suggests that the quality of this information varies both in terms of the coverage as well as the quality. As for the coverage, the VMRay Analyzer identifies almost all function calls used by an analysed malware sample since it leverages VMI. Other sandboxes rely on API hooking which leads to less coverage since almost all DLLs need to be hooked in order to get an image of the API call usage that as broad as it is provided by VMRay. This has to be considered in the signature creation process as it can happen that not all of the function calls are available in all sources of information. To solve this issue to some extent, available sandboxes need improvements in terms of the API function calls hooked by default. For example, the Cuckoo sandbox only hooks `ntdll.dll` by default. Consequently, only system calls are traced and incorporated into the analysis result. In terms of the quality, some sandboxes do not follow the official MSDN documentation regarding the naming of function call parameters. As a result, a harmonisation layer is needed to obtain function logs that comply with the MSDN documenatation and are thus usable in our detection approach. By omitting such a harmonisation layer, inconsistencies can also be handled within the signatures but this approach is not desirable since it makes signatures unnecessary complex.

Detecting Malicious Behaviour in Memory Images with dynmx

A rather interesting research area is the application of our detection approach on static sources of information. For instance, we take memory images into account as they typically contain the malware in unpacked form, at least to a certain extent. This, in turn, makes the analysis less complex than using the actual mostly packed malware samples. The motivation for this novel approach is the improvement of the identification of malware or in general malware features in the memory of a running system. This improvement would be helpful for forensic examiners, especially in incident response scenarios. To the best knowledge of the authors, such an approach is currently not existing in the form we describe it in this section. In order to implement this detection approach, API calls used by the malware need to be extracted from the memory images. Possible solutions to this problem are delivered by Plohmann, Enders and Padilla in [123] as well as by the volatility plug-in named APIFinder [50]. The approach of Plohmann et al. is preferable and promising in this case since it also considers dynamically loaded DLLs and functions. However, the research needs to be extended in order to also extract parameter values in addition to the API function name. Since return values are only available during runtime, they can not be deduced based on static information which limits the detection. After extracting the function calls they need be brought into a meaningful order. Therefore, the concept of control flow graphs can be leveraged [30, 33]. By combining this research, the extraction of function log like information based on a memory image seems possible and can thereby allow the application of the dynmx approach on static information. Even more advanced would be the application directly on malware samples but this approach would require an efficient solution to the problem of generically unpacking malware samples which is not being solved so far [123].

Classifying Malware Based on Detected Generic Malware Features

As outlined in the related work Section 1.4, the classification of malware is an important research domain. Therefore, the detection results provided by our approach can be a valuable source of information for the classification of malware. While we are proposing manually defined signatures for specific malware features that characterise certain families, the classification can possibly also be achieved in a fully-automated way on the basis of detected generic malware features. Since the

classification approaches known to the authors are based on behavioural information automatically identified despite their actual semantics in terms of the functionality that lies behind the sequence, improvements of classification results by incorporating semantics into the process seem to be within the realms of possibility.

A

APPENDIX

A.1 Contents of the Enclosed Flash Drive

The flash drive provided together with this thesis contains the thesis document itself, figures, the source code of the prototype implementation as well as additional information in digital form. The directory structure of the flash drive is as follows.

Document This directory contains the master thesis in PDF format together with the LaTeX source code of the document in the subfolder `MA_Thesis`.

Figures This directory contains the figures of the master thesis categorised by the chapters.

Digital_Appendix This directory contains the digital appendix of the master thesis categorised by the chapters. Files in this folder and its subfolders are specifically referenced by the corresponding chapters and sections of this work.

A.2 Sample Function Logs

Text-based Function Log Sample

```
# Flog Txt Version 1
# Analyzer Version: 1.8.0
# Analyzer Build Date: Nov  2 2015 15:01:39
# Log Creation Date: 09.12.2015 11:47

Process:
    id = "1"
    image_name = "excel.exe"
    filename = "c:\\program files\\microsoft office\\office12\\excel.exe"
    page_root = "0x7ee40400"
    os_pid = "0xbc0"
    monitor_reason = "analysis_target"
    parent_id = "0"
    os_parent_pid = "0x0"
    cmd_line = "\"C:\\Program Files\\Microsoft Office\\Office12\\EXCEL.EXE\" /e \"\""
    cur_dir = "C:\\Windows\\system32\\"

Region:
                  id = 144
           start_va = 0x10000
             end_va = 0x1ffff
        entry_point = 0x0
        region_type = pagefile_backed
               name = "pagefile_0x0000000000010000"
           filename = ""

Region:
                  id = 145
           start_va = 0x20000
             end_va = 0x22fff
        entry_point = 0x0
        region_type = pagefile_backed
               name = "pagefile_0x0000000000020000"
           filename = ""

[...]

Thread:
    id = 1
    os_tid = 0xb50

    [0048.309] KeGetCurrentIrql () returned 0x927bf200
    [0048.309] MmIsAddressValid (VirtualAddress=0x77848008) returned 1
    [0048.309] NtOpenKey (KeyHandle=0x12f9ec, DesiredAccess=0x80000000, ObjectAttributes=0x12f9cc*(Length=0x18, RootDirectory=0x0, ObjectName="\\Registry\\Machine\\System\\CurrentControlSet\\Control\\Session Manager\0", Attributes=0x40, SecurityDescriptor=0x0, SecurityQualityOfService=0x0))
    [0048.488] KeGetCurrentIrql () returned 0x927bf200
    [0048.488] MmIsAddressValid (VirtualAddress=0x77848c80)
```

```
        returned 1
47      [0048.488] NtOpenKey (KeyHandle=0x12f870, DesiredAccess=0↩
            x3, ObjectAttributes=0x7786e4b8*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\Registry\\MACHINE\\↩
            System\\CurrentControlSet\\Control\\SafeBoot\\Option↩
            \0", Attributes=0x40, SecurityDescriptor=0x0, ↩
            SecurityQualityOfService=0x0))
48      [0048.488] KeGetCurrentIrql () returned 0x927bf200
49      [0048.488] MmIsAddressValid (VirtualAddress=0x77848be0) ↩
            returned 1
50      [0048.489] NtOpenKey (KeyHandle=0x12f880, DesiredAccess=0↩
            x20019, ObjectAttributes=0x778dddc0*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\Registry\\Machine\\↩
            System\\CurrentControlSet\\Control\\Srp\\GP\\DLL\0", ↩
            Attributes=0x40, SecurityDescriptor=0x0, ↩
            SecurityQualityOfService=0x0))
51      [0048.489] KeGetCurrentIrql () returned 0x927bf200
52      [0048.489] MmIsAddressValid (VirtualAddress=0x77848b40) ↩
            returned 1
53      [0048.489] NtOpenKey (KeyHandle=0x12f88c, DesiredAccess=0↩
            x1, ObjectAttributes=0x7786e4d8*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\Registry\\Machine\\↩
            Software\\Policies\\Microsoft\\Windows\\Safer\\↩
            CodeIdentifiers\0", Attributes=0x40, ↩
            SecurityDescriptor=0x0, SecurityQualityOfService=0x0)↩
            )
54      [0048.489] KeGetCurrentIrql () returned 0x927bf200
55      [0048.489] MmIsAddressValid (VirtualAddress=0x262a80) ↩
            returned 1
56      [0048.489] NtOpenKey (KeyHandle=0x12f88c, DesiredAccess=0↩
            x1, ObjectAttributes=0x12f848*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\REGISTRY\\USER\\S↩
            -1-5-21-3463283166-1737944678-1226249386-1003\\↩
            Software\\Policies\\Microsoft\\Windows\\Safer\\↩
            CodeIdentifiers\0", Attributes=0x40, ↩
            SecurityDescriptor=0x0, SecurityQualityOfService=0x0)↩
            )
57      [0048.984] KeGetCurrentIrql () returned 0x927bf200
58      [0048.984] MmIsAddressValid (VirtualAddress=0x12e954) ↩
            returned 1
59      [0048.984] NtOpenKey (KeyHandle=0x12ed74, DesiredAccess=0↩
            x20019, ObjectAttributes=0x12e92c*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\Registry\\Machine\\↩
            System\\CurrentControlSet\\Control\\Nls\\Sorting\\↩
            Versions\0", Attributes=0x40, SecurityDescriptor=0x0,↩
             SecurityQualityOfService=0x0))
60      [0048.985] KeGetCurrentIrql () returned 0x927bf200
61      [0048.985] NtOpenKey (KeyHandle=0x76296b40, DesiredAccess↩
            =0x2000000, ObjectAttributes=0x12f3fc*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\REGISTRY\\MACHINE↩
            \0", Attributes=0x40, SecurityDescriptor=0x0, ↩
            SecurityQualityOfService=0x0))
62      [0048.985] KeGetCurrentIrql () returned 0x927bf200
63      [0048.985] MmIsAddressValid (VirtualAddress=0x75b520f8) ↩
            returned 1
64      [0048.985] NtOpenKey (KeyHandle=0x12f750, DesiredAccess=0↩
            x1, ObjectAttributes=0x75b50940*(Length=0x18, ↩
            RootDirectory=0x0, ObjectName="\\Registry\\MACHINE\\↩
            System\\CurrentControlSet\\Control\\Session Manager↩
```

```
        \0", Attributes=0x40, SecurityDescriptor=0x0, ↩
        SecurityQualityOfService=0x0))
65      [0049.036] KeGetCurrentIrql () returned 0x927bf200
66      [0049.036] MmIsAddressValid (VirtualAddress=0x7646e7d0) ↩
        returned 1
67      [0049.036] NtOpenKey (KeyHandle=0x12f3c0, DesiredAccess=0↩
        x20019, ObjectAttributes=0x12f388*(Length=0x18, ↩
        RootDirectory=0x0, ObjectName="\\Registry\\Machine\\↩
        System\\CurrentControlSet\\Control\\Error Message ↩
        Instrument\\\0", Attributes=0x40, SecurityDescriptor↩
        =0x0, SecurityQualityOfService=0x0))
68      [0049.036] KeGetCurrentIrql () returned 0x927bf200
69      [0049.036] MmIsAddressValid (VirtualAddress=0x12f314) ↩
        returned 1
70      [0049.036] NtOpenKey (KeyHandle=0x12f2fc, DesiredAccess=0↩
        x20019, ObjectAttributes=0x12f2dc*(Length=0x18, ↩
        RootDirectory=0x0, ObjectName="\\Registry\\Machine\\↩
        Software\\Microsoft\\Windows NT\\CurrentVersion\\↩
        GRE_Initialize\0", Attributes=0x40, ↩
        SecurityDescriptor=0x0, SecurityQualityOfService=0x0)↩
        )
71      [0049.036] KeGetCurrentIrql () returned 0x927bf200
```

XML Function Log Sample

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <analysis log_version="2" analyzer_version="3.1.0" ↩
       analysis_date="28.09.2019 12:50:33.543">
3  <monitor_process ts="40421" process_id="1" image_name="pgv.↩
       exe" filename="c:\\users\\user\\desktop\\pgv.exe" ↩
       page_root="0x16f600000" os_pid="0x6d0" os_integrity_level↩
       ="0x3000" os_privileges="0x60800000" monitor_reason="↩
       analysis_target" parent_id="0" os_parent_pid="0x0" ↩
       cmd_line="&quot;C:\\Users\\user\\Desktop\\pgv.exe&quot; "↩
        cur_dir="C:\\Users\\user\\Desktop\\" os_username="↩
       MACHINE\\user" bitness="64" os_groups="&quot;MACHINE\\↩
       Domain Users&quot; [0x7], &quot;Everyone&quot; [0x7], &↩
       quot;MACHINE\\user&quot; [0x7], &quot;BUILTIN\\↩
       Administrators&quot; [0xf], &quot;BUILTIN\\Users&quot; [0↩
       x7], &quot;NT AUTHORITY\\INTERACTIVE&quot; [0x7], &quot;↩
       CONSOLE LOGON&quot; [0x7], &quot;NT AUTHORITY\\↩
       Authenticated Users&quot; [0x7], &quot;NT AUTHORITY\\This↩
        Organization&quot; [0x7], &quot;NT AUTHORITY\\Logon ↩
       Session 00000000:0000f328&quot; [0xc0000007], &quot;LOCAL↩
       &quot; [0x7], &quot;NT AUTHORITY\\NTLM Authentication&↩
       quot; [0x7]"/>
4  <monitor_thread ts="40795" thread_id="1" process_id="1" ↩
       os_tid="0x6a4"/>
5  <new_region ts="40795" region_id="1" process_id="1" start_va↩
       ="0x10000" end_va="0x2ffff" monitored="1" entry_point="0↩
       x0" region_type="private" name="↩
       private_0x0000000000010000" filename="" ↩
       normalized_filename=""/>
6  <new_region ts="40795" region_id="2" process_id="1" start_va↩
       ="0x30000" end_va="0x31fff" monitored="1" entry_point="0↩
       x0" region_type="private" name="↩
       private_0x0000000000030000" filename="" ↩
       normalized_filename=""/>
7  [...]
```

```
<fncall ts="43122" fncall_id="71" process_id="1" thread_id="1" name="GetSystemTimeAsFileTime" addr="0x762d3509" from="0x40f384">
    <in>
        <param name="lpSystemTimeAsFileTime" type="unknown" value="0x18ff78"/>
    </in>
    <out>
        <param name="lpSystemTimeAsFileTime" type="ptr" value="0x18ff78">
            <deref type="container">
                <member name="dwLowDateTime" type="unsigned_32bit" value="0x69472910"/>
                <member name="dwHighDateTime" type="unsigned_32bit" value="0x1d575fb"/>
            </deref>
        </param>
        <param name="ret_val" type="void"/>
    </out>
</fncall>
<fnret ts="43122" fncall_id="71" addr="0x40f38a" from="0x77118c8e"/>
<fncall ts="43122" fncall_id="72" process_id="1" thread_id="1" name="GetCurrentProcessId" addr="0x762d11f8" from="0x40f399">
    <out>
        <param name="ret_val" type="unsigned_32bit" value="0x6d0"/>
    </out>
</fncall>
<fnret ts="43122" fncall_id="72" addr="0x40f39f" from="0x7711ee9c"/>
<fncall ts="43122" fncall_id="73" process_id="1" thread_id="1" name="GetCurrentThreadId" addr="0x762d1450" from="0x40f3a5">
    <out>
        <param name="ret_val" type="unsigned_32bit" value="0x6a4"/>
    </out>
</fncall>
<fnret ts="43122" fncall_id="73" addr="0x40f3ab" from="0x77122b21"/>
<fncall ts="43122" fncall_id="74" process_id="1" thread_id="1" name="GetTickCount" addr="0x762d110c" from="0x40f3b1">
    <out>
        <param name="ret_val" type="unsigned_32bit" value="0x1163afe"/>
    </out>
</fncall>
<fnret ts="43122" fncall_id="74" addr="0x40f3b7" from="0x77118cc9"/>
<fncall ts="43122" fncall_id="75" process_id="1" thread_id="1" name="QueryPerformanceCounter" addr="0x762d1725" from="0x40f3c1">
    <in>
        <param name="lpPerformanceCount" type="void_ptr" value="0x18ff68"/>
    </in>
```

```
45      <out>
46          <param name="lpPerformanceCount" type="ptr" value="0↩
                x18ff68">
47              <deref type="signed_64bit" value="24343572339"/>
48          </param>
49          <param name="ret_val" type="bool" value="1"/>
50      </out>
51  </fncall>
52  <fnret ts="43187" fncall_id="75" addr="0x40f3c7" from="0↩
        x777f88da"/>
```

A.3 Application of the Common Data Model to a Sample Function Log

Sample Text-based Input Function Log Sample

```
# Flog Txt Version 1
# Analyzer Version: 2.1.1
# Analyzer Build Date: Aug 24 2017 11:10:37
# Log Creation Date: 06.11.2017 22:08:00.387

Process:
    id = "2"
    image_name = "cmd.exe"
    filename = "c:\\windows\\syswow64\\cmd.exe"
    page_root = "0xecf6000"
    os_pid = "0xdc0"
    monitor_reason = "child_process"
    parent_id = "1"
    os_parent_pid = "0x13fc"
    cmd_line = "C:\\Windows\\System32\\cmd.exe /k powershell ↩
        -NoP -sta -NonI -w hidden $e=(New-Object System.Net.↩
        WebClient).DownloadString('http://ryanbaptistchurch.↩
        com/KJHDhbje71');powershell -e $e "
    cur_dir = "C:\\Users\\user\\Desktop\\"

Thread:
    id = 10
    os_tid = 0x1358

    [0061.488] GetSystemTimeAsFileTime (in: ↩
        lpSystemTimeAsFileTime=0x1aff38 | out: ↩
        lpSystemTimeAsFileTime=0x1aff38*(dwLowDateTime=0↩
        xce14f500, dwHighDateTime=0x1affff*=0x1d3574b))
```

Listing A.1: Simplified input function log

Resulting Function Log Object

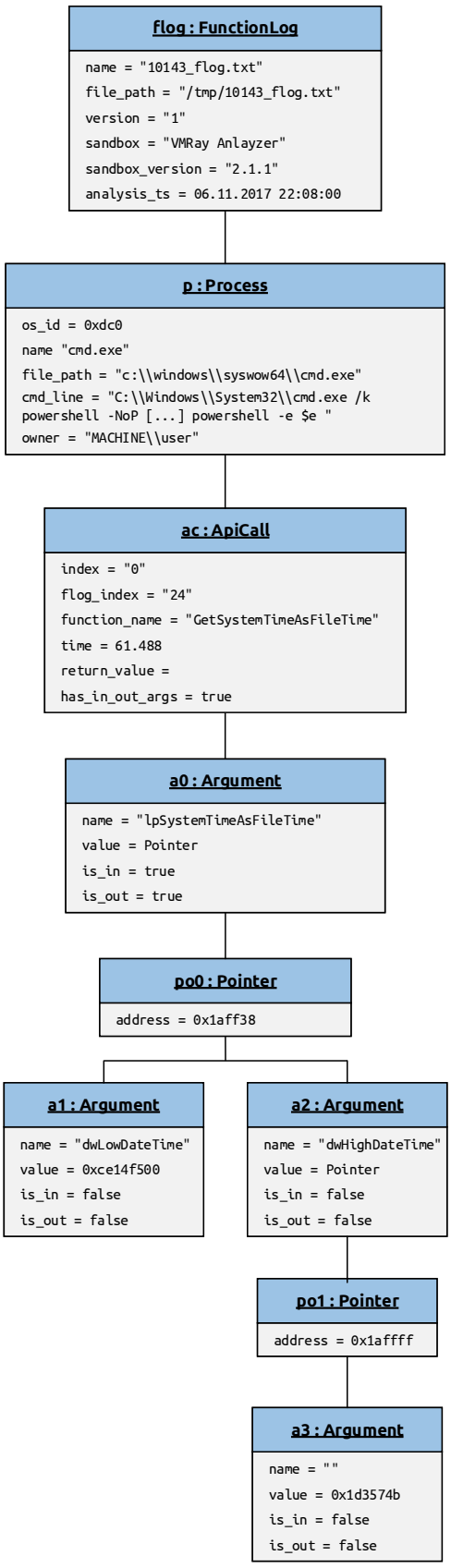

Figure A.1: Function log object derived from the input function log in Listing A.1 as UML object model

A.4 Sample Function Log Converted to dynmx Function Log Format

```
# dynmx generic function log
# converted from: test_flog.txt
# converted on: 2020-04-10 14:02:40.120371

{
  "flog": {
    "processes": [
      {
        "os_id": 3520,
        "name": "cmd.exe",
        "file_path": "c:\\windows\\syswow64\\cmd.exe",
        "cmd_line": "C:\\Windows\\System32\\cmd.exe /k powershell -NoP -sta -NonI -w hidden $e=(New-Object System.Net.WebClient).DownloadString('http://ryanbaptistchurch.com/KJHDhbje71');powershell -e $e ",
        "owner": null,
        "api_calls": [
          {
            "flog_index": "2c979198-8793-4f43-b0d4-ef692b62483c",
            "function_name": "GetSystemTimeAsFileTime",
            "time": 61.488,
            "arguments": [
              {
                "name": "lpSystemTimeAsFileTime",
                "is_in": true,
                "is_out": true,
                "value": {
                  "address": 1769272,
                  "arguments": [
                    {
                      "name": "dwLowDateTime",
                      "is_in": false,
                      "is_out": false,
                      "value": 3457479936
                    },
                    {
                      "name": "dwHighDateTime",
                      "is_in": false,
                      "is_out": false,
                      "value": {
                        "address": 1769471,
                        "arguments": 30627659
                      }
                    }
                  ]
                }
              }
            ],
            "return_value": null
          }
        ]
      }
    ]
```

```
51      }
52  }
```

A.5 Python Code Listings

Method `DetectionBlock._detect_sequence()`

```
def _detect_sequence(self, process, var_storage, potential_detection_paths):
    """
    Detects a sequence detection block
    :param process: Process to detect the detection block in
    :param var_storage: VariableStorage of the signature
    :param potential_detection_paths: Previously found detection paths
    :return: DetectedBlock object if the detection was successful, False if the detection was not successful
    """
    flog_path = process.flog_path
    # Check all detection paths if no detection paths were determined in advance
    if not potential_detection_paths:
        potential_detection_paths = self.graph.find_all_paths()
    detection_result = DetectedBlock()
    # Iterate over the potential detection paths and try to detect them
    detected_api_calls = list()
    detected_resources = list()
    detected = False
    for detection_path in potential_detection_paths:
        self._logger.debug("Detection path={}".format(detection_path))
        self._logger.debug("Detection path length={}".format(len(detection_path)))
        # Get API call detection steps for detecting API call sequence
        api_detection_path = self._get_detection_steps_by_type(DetectionStepType.API_CALL, detection_path)
        # Find suitable API call candidates by their function name based on the steps in the detection path
        reduced_api_calls = self._reduce_api_calls(flog_path, process, api_detection_path)
        if len(api_detection_path) and len(reduced_api_calls):
            # Calculate the longest common subsequence between the API call candidates and the detection path
            detected_sequence = self._find_lcs(flog_path, reduced_api_calls, api_detection_path, var_storage)
            self._logger.debug("LCS length={}".format(len(detected_sequence)))
            self._logger.debug("LCS={}".format(detected_sequence))
            # The detection path is successfully detected if the longest common subsequence has the length of the
            # detection path
            path_detected = len(detected_sequence) == len(
```

```
                api_detection_path)
33          else:
34              path_detected = False
35          # Detect resources
36          resource_detection_path = self.↩
                _get_detection_steps_by_type(DetectionStepType.↩
                RESOURCE, detection_path)
37          if len(resource_detection_path):
38              detected_res = self._detect_resources(process, ↩
                    resource_detection_path, var_storage)
39              path_detected &= len(detected_res) > 0
40          else:
41              detected_res = []
42          if path_detected:
43              # Add the API calls of the detected sequence to ↩
                    the list of detected API calls for this block↩
                     if
44              # the detection was successful
45              for api_call in detected_sequence:
46                  if api_call not in detected_api_calls:
47                      detected_api_calls.append(api_call)
48              for resource in detected_res:
49                  if resource not in detected_resources:
50                      detected_resources.append(resource)
51              self._logger.debug("Path␣detected␣in␣process␣{}␣(↩
                    PID:␣{})".format(process.name, process.os_id)↩
                    )
52          else:
53              self._logger.debug("Path␣not␣detected␣in␣process␣↩
                    {}␣(PID:␣{})".format(process.name, process.↩
                    os_id))
54          detected |= path_detected
55      # Build the DetectedBlock object
56      detection_result.detection_block_key = self.key
57      detection_result.api_calls = detected_api_calls
58      detection_result.resources = detected_resources
59      return detection_result if detected else False
```

Method `DetectionBlock._find_lcs()`

```
1   def _find_lcs(self, flog_path, api_calls, detection_path, ↩
        var_storage):
2       # Calculate LCS length
3       # LCS length is found in the result matrix c in index c[↩
            len(api_calls)][len(detection_path)]
4       # See https://en.wikipedia.org/wiki/↩
            Longest_common_subsequence_problem for algorithm
5       # See https://rosettacode.org/wiki/↩
            Longest_common_subsequence#Python for implementation
6       var_contexts = self._get_variable_contexts(flog_path, ↩
            api_calls, detection_path)
7       # If there are variable contexts, calculate LCS matrix ↩
            for each context
8       if var_contexts:
9           self._logger.info("Found␣{}␣contexts".format(len(↩
                var_contexts)))
10          self._logger.info(
11              "Len␣detection␣path={},␣Len␣API␣calls={}".format(↩
                    len(detection_path), len(api_calls)))
```

```
12          for context in var_contexts:
13              c = self._calculate_lcs_matrix_for_var_context(↩
                    flog_path, api_calls, detection_path, ↩
                    var_storage,
14                                                                 context↩
                                                                       )↩
15              lcs_length = c[-1][-1]
16              # We can stop the detection if a certain context ↩
                    produces a matrix that reflects the length of↩
                     the
17              # detection path
18              self._logger.info("LCS␣Length={}".format(↩
                    lcs_length))
19              if lcs_length == len(detection_path):
20                  break
21      else:
22          self._logger.info(
23              "Len␣detection␣path={},␣Len␣API␣calls={}".format(↩
                    len(detection_path), len(api_calls)))
24          # If no variable contexts are available calculate the↩
                 LCS without considering variable contexts
25          c = self._calculate_lcs_matrix(flog_path, api_calls, ↩
                detection_path, var_storage)
26      # Extract the detected sequence from the calculated ↩
            matrix
27      detected_sequence = self._extract_lcs(c, api_calls, ↩
            detection_path)
28      return detected_sequence
```

Method DetectionBlock._calculate_lcs_matrix()

```
1   def _calculate_lcs_matrix(self, flog_path, api_calls, ↩
        detection_path, var_storage):
2           c = [[0] * (len(detection_path) + 1) for _ in range(↩
                len(api_calls) + 1)]
3           last_detected_index = None
4           for i, api_call in enumerate(api_calls):
5               for j, step in enumerate(detection_path):
6                   if step.detect(flog_path, api_call, ↩
                        var_storage, last_detected_index):
7                       c[i + 1][j + 1] = c[i][j] + 1
8                       last_detected_index = api_call.index
9                   else:
10                      c[i + 1][j + 1] = max(c[i + 1][j], c[i][j↩
                             + 1])
11          return c
```

Method DetectionBlock._extract_lcs()

```
1   def _extract_lcs(self, c, api_calls, detection_path):
2       # Reconstruct the detected sequence based on the LCS ↩
            matrix
3       # Go diagonal through the matrix beginning from the last ↩
            element and find the first element in the column that
4       # has a lower value than the previously found length; the↩
             first element that has a lower value is a member of
5       # the longest common subsequence
```

```
6       detected_sequence = []
7       i = len(api_calls)
8       j = len(detection_path)
9       last_length = c[-1][-1]
10      while j > 0:
11          while i > 0:
12              if c[i-1][j] == last_length-1:
13                  detected_sequence.append(api_calls[i - 1])
14                  i -= 1
15                  last_length -= 1
16                  break
17              i -= 1
18          j -= 1
19      detected_sequence.reverse()
20      return detected_sequence
```

BIBLIOGRAPHY

[1] YARA - The pattern matching swiss knife for malware researchers, 18.06.2018. URL https://virustotal.github.io/yara/. Accessed 03.07.2018.

[2] AccessData. FTK Imager Lite version 3.1.1 | AccessData, 2020. URL https://accessdata.com/product-download/ftk-imager-lite-version-3-1-1. Accessed 4.6.2020.

[3] Muhammad Ali, Stavros Shiaeles, Maria Papadaki, and Bogdan V. Ghita. Agent-based Vs Agent-less Sandbox for Dynamic Behavioral Analysis. In *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–5. IEEE, 2018. ISBN 978-1-5386-7272-3.

[4] Victor M. Alvarez. YARA - The pattern matching swiss knife for malware researchers, 2020. URL http://virustotal.github.io/yara/. Accessed 19.01.2020.

[5] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4):247–258, 2011.

[6] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[7] AV-Test. Security Report 2018/19, 2019. URL https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf. Accessed 17.06.2020.

[8] AV-Test. Malware Statistics & Trends Report | AV-TEST, 2020. URL https://www.av-test.org/en/statistics/malware/. Accessed 17.6.2020.

[9] Thomas Ball. The Concept of Dynamic Analysis. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Oscar Nierstrasz, and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-66538-0.

[10] Chris Balles and Ateeq Sharfuddin. Breaking Imphash, 17.09.2019. URL http://arxiv.org/pdf/1909.07630v1.

[11] Tao Ban, Ryoichi Isawa, Shanqing Guo, Daisuke Inoue, and Koji Nakao. Efficient Malware Packer Identification Using Support Vector Machines with Spectrum Kernel. In *2013 Eighth*

Asia Joint Conference on Information Security (Asia JCIS), pages 69–76. IEEE, 2013. ISBN 978-0-7695-5075-6.

[12] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML Ain't Markup Language (YAML™) Version 1.2, 2020. URL https://yaml.org/spec/1.2/spec.html. Accessed 18.4.2020.

[13] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Seventh international symposium on string processing and information retrieval*, pages 39–48. IEEE Computer Society, 2000. ISBN 0-7695-0746-8.

[14] Ratul Bhowmick, Md Ibrahim Sadek Bhuiyan, Md. Sabir Hossain, Muhammad Kamal Hossen, and Ahsan Sadee Tanim. An Approach for Improving Complexity of Longest Common Subsequence Problems using Queue and Divide-and-Conquer Method. In *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT 2019)*, pages 1–5. IEEE, 2019. ISBN 978-1-7281-3445-1.

[15] Michael Brengel, Michael Backes, and Christian Rossow. Detecting Hardware-Assisted Virtualization. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 9721 of *Lecture Notes in Computer Science*, pages 207–227. Springer International Publishing, Cham, 2016. ISBN 978-3-319-40666-4.

[16] c0fec0de. c0fec0de/anytree GitHub Repository, 2020. URL https://github.com/c0fec0de/anytree. Accessed 25.05.2020.

[17] Alejandro Calleja, Juan Tapiador, and Juan Caballero. A Look into 30 Years of Malware Development from a Software Metrics Perspective. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, volume 9854 of *Lecture Notes in Computer Science*, pages 325–345. Springer International Publishing, Cham, 2016. ISBN 978-3-319-45718-5.

[18] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In Mats Heimdahl, editor, *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM Digital Library, page 122. ACM, 2012. ISBN 9781450314541.

[19] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting Android malware using sequences of system calls. In Aharon Abadi, Shah Rukh Humayoun, and Henry Muccini, editors, *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile - DeMobile 2015*, pages 13–20. ACM Press, 2015. ISBN 9781450338158.

[20] Raymond Canzanese, Spiros Mancoridis, and Moshe Kam. System Call-Based Detection of Malicious Processes. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 119–124. IEEE, 2015. ISBN 978-1-4673-7989-2.

[21] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 133–138. IEEE Comput. Soc, 2001. ISBN 0-7695-1040-X.

[22] David R. Chesney. EECS 281: Data Structures and Algorithms: Graphs and Graph Algorithms. URL https://www.eecs.umich.edu/courses/eecs281/f04/lecnotes/19-intrograph.pdf. Accessed 29.04.2020.

[23] In Kyeom Cho and Eul Gyu Im. Extracting representative API patterns of malware families using multiple sequence alignments. In Esmaeil S. Nadimi, Tomas Cerny, Sung-Ryul Kim, and Wei Wang, editors, *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, pages 308–313. ACM, 2015. ISBN 9781450337380.

[24] ClamAV. Creating signatures for ClamAV, 2020. URL https://www.clamav.net/documents/creating-signatures-for-clamav. Accessed 23.01.2020.

[25] codecat007. codecat007/snort-rules, 2020. URL https://github.com/codecat007/snort-rules. Accessed 26.5.2020.

[26] Deutsche Telekom AG. Cyber Defense Center, 2015. URL https://www.telekom.com/en/corporate-responsibility/data-protection-data-security/security/details/cyber-defense-center-362648. Accessed 2.6.2020.

[27] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In Peng Ning, Paul Syverson, and Somesh Jha, editors, *Proceedings of the 15th ACM conference on Computer and communications security*, page 51. ACM, 2008. ISBN 9781595938107.

[28] Hermann Dornhackl, Konstantin Kadletz, Robert Luh, and Paul Tavolato. Malicious Behavior Patterns. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 384–389. IEEE, 2014. ISBN 978-1-4799-3616-8.

[29] Roger Dudler. git - the simple guide - no deep shit!, 2017. URL https://rogerdudler.github.io/git-guide/. Accessed 26.5.2020.

[30] Thomas Dullien and Rolf Rolles. *Graph-based comparison of executable objects*. 2005.

[31] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. The Cornell commission: on Morris and the worm. *Communications of the ACM*, 32(6):706–709, 1989.

[32] Mohamed Elhadi and Amjad Al-Tobi. Refinements of Longest Common Subsequence algorithm. In *2010 ACS/IEEE International Conference on Computer Systems and Applications*, pages 1–5. IEEE, 2010. ISBN 978-1-4244-7716-6.

[33] G. Erd'elyi and Erasmo Carrera. *Digital genome mapping: ad-vanced binary malware analysis*. 2004. URL https://www.researchgate.net/publication/238218509_Digital_genome_mapping_ad-vanced_binary_malware_analysis.

[34] European Union Agency for Cybersecurity. Cybersecurity in the healthcare sector during COVID-19 pandemic, 2020. URL https://www.enisa.europa.eu/news/enisa-news/cybersecurity-in-the-healthcare-sector-during-covid-19-pandemic. Accessed 17.6.2020.

[35] Tom Fawcett. ROC Graphs: Notes and Practical Considerations for Researchers. *Pattern Recognition Letters*, 31(8):1–38, 2004. URL https://www.researchgate.net/publication/284043217_ROC_Graphs_Notes_and_Practical_Considerations_for_Researchers.

[36] Frauenhofer FKIE. Emotet (Malware Family), 2020. URL https://malpedia.caad.fkie.fraunhofer.de/details/win.emotet. Accessed 29.5.2020.

[37] Fraunhofer FKIE. DarkComet (Malware Family), 2020. URL https://malpedia.caad.fkie.fraunhofer.de/details/win.darkcomet. Accessed 29.5.2020.

[38] Fraunhofer FKIE. Formbook (Malware Family), 2020. URL https://malpedia.caad.fkie.fraunhofer.de/details/win.formbook. Accessed 29.5.2020.

[39] Fraunhofer FKIE. Remcos (Malware Family), 2020. URL https://malpedia.caad.fkie.fraunhofer.de/details/win.remcos. Accessed 11.6.2020.

[40] Martin Fowler. Domain-Specific Languages Guide, 2020. URL https://www.martinfowler.com/dsl.html. Accessed 14.02.2020.

[41] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Proceedings of the network and distributed systems security symposium*, 3: 191–206, 2003. URL https://www.ndss-symposium.org/wp-content/uploads/2017/09/A-Virtual-Machine-Introspection-Based-Architecture-for-Intrusion-Detection-Tal-Garfinkel.pdf. Accessed 30.01.2020.

[42] Will Gibb and Devon Kerr. OpenIOC: Back to the Basics, 2013. URL https://www.fireeye.com/blog/threat-research/2013/10/openioc-basics.html. Accessed 18.6.2020.

[43] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. A Study of the Packer Problem and Its Solutions. In Richard Lippmann, editor, *Recent advances in intrusion detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 98–115. Springer, Berlin and Heidelberg and New York, NY, 2008. ISBN 978-3-540-87402-7.

[44] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur. Malware Characterization Using Windows API Call Sequences. *Journal of Cyber Security and Mobility*, 7(4):363–378, 2018.

[45] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. *IEEE Transactions on Software Engineering*, 43(2):164–184, 2017.

[46] Konrad Hinsen. Domain-Specific Languages in Scientific Computing. *Computing in Science & Engineering*, 20(1):88–92, 2018.

[47] Andrew Hopper and John McCanny. *Progress and Research in Cybersecurity - Supporting a resilient and trustworthy system for the UK*. 2016. ISBN 978-1-78252-215-7. URL https://pure.qub.ac.uk/en/publications/progress-and-research-in-cybersecurity-supporting-a-resilient-and.

[48] Ashkan Hosseini. Ten process injection techniques: A technical survey of common and trending process injection techniques, 2017. URL https://www.elastic.co/de/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process. Accessed 2020-06-12T20:55:43.542Z.

[49] Ted Hudek and Tim Sherer. Using Nt and Zw Versions of the Native System Services Routines - Windows drivers, 2017. URL https://docs.microsoft.com/de-de/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines. Accessed 27.01.2020.

[50] Itaykr. volatilityfoundation/community/apifinder.py, 2016. URL https://github.com/volatilityfoundation/community/blob/master/itayk/apifinder.py. Accessed 20.6.2020.

[51] Nils Jannasch. How to Create Easy and Open Integrations with VMRay's REST API, 2020. URL https://www.vmray.com/cyber-security-blog/how-to-create-easy-open-integrations-vmray-rest-api/. Accessed 06/02/2020 20:36:03.

[52] Monnappa K A. *Learning malware analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware*. Packt Publishing, Birmingham, UK, 2018. ISBN 9781788392501. URL http://proquest.tech.safaribooksonline.de/9781788392501.

[53] Michael Kan. Upgraded Mirai botnet disrupts Deutsche Telekom by infecting routers, 2016. URL https://www.pcworld.com/article/3145449/upgraded-mirai-botnet-disrupts-deutsche-telekom-by-infecting-routers.html. Accessed 17.01.2020.

[54] Kaspersky. Kaspersky Security Bulletin '19: Statistics, 2019. URL https://securelist.com/kaspersky-security-bulletin-2019-statistics/95475/?utm_source=securelist&utm_medium=blog&utm_campaign=gl_ksb-stats_ay0073&utm_content=banner&utm_term=gl_securelist__ay0073_banner_blog_ksb-stats. Accessed 17.06.2020.

[55] Cengiz Kaygusuz, Katherine Crowson, Brian Graham, Jon Dufrense, and Paul McGuire. pyparsing/pyparsing, 2020. URL https://github.com/pyparsing/pyparsing. Accessed 25.05.2020.

[56] Migo Kedem. Malware Embedded in Microsoft Office Documents | DDE Exploit (MACROLESS), 2018. URL https://medium.com/@migokedem/malware-embedded-in-microsoft-office-documents-dde-exploit-macroless-4f197387ddbd. Accessed 04.06.2020.

[57] Swati Khandelwal. Unpatched Microsoft Word DDE Exploit Being Used In Widespread Malware Attacks, 2017. URL https://thehackernews.com/2017/10/ms-office-dde-malware-exploit.html. Accessed 4.6.2020.

[58] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. A Novel Approach to Detect Malware Based on API Call Sequence Analysis. *International Journal of Distributed Sensor Networks*, 11(6): 659101, 2015.

[59] Sangwoo Kim, Seokmyung Hong, Jaesang Oh, and Heejo Lee. Obfuscated VBA Macro Detection Using Machine Learning. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 490–501. IEEE, 25.06.2018 - 28.06.2018. ISBN 978-1-5386-5596-2.

[60] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep Learning for Classification of Malware System Call Sequences. In BYEONG H. O. KANG, editor, *AI 2016*, volume 9992 of *Lecture Notes in Computer Science*, pages 137–149. SPRINGER INTERNATIONAL PU, 2017. ISBN 978-3-319-50126-0.

[61] Kento Kono, Sanouphab Phomkeona, and Koji Okamura. An Unknown Malware Detection Using Execution Registry Access. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 487–491. IEEE, 2018. ISBN 978-1-5386-2666-5.

[62] Michael Langer. COMP 250 Lecture 21: Binary Tress, Expression Trees, 2017. URL http://www.cim.mcgill.ca/~langer/250/21-binarytrees-slides.pdf. Accessed 30.04.2020.

[63] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. AccessMiner: using system-centric models for malware protection. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*, page 399. ACM Press, 2010. ISBN 9781450302456.

[64] Andrew Lee. Latest entries - The Portable Freeware Collection, 2020. URL https://www.portablefreeware.com/. Accessed 4.6.2020.

[65] John Leitch. Process Hollowing, 2011. URL http://www.autosectools.com/process-hollowing.pdf. Accessed 16.04.2020.

[66] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[67] Michael Ligh, Matt Richard, and Steven Adair. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. John Wiley & Sons, 2010. ISBN 978-0-470-61303-0.

[68] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting malware and threats in Windows, Linux, and Mac memory*. Wiley, Indianapolis, IN, 2014. ISBN 978-1-118-82509-9.

[69] Zachary Chase Lipton, Charles Elkan, and Balakrishnan Narayanaswamy. *Thresholding Classifiers to Maximize F1 Score*. 2014. URL https://www.researchgate.net/publication/262996346_Thresholding_Classifiers_to_Maximize_F1_Score.

[70] Jiamei Liu and Suping Wu. Research on longest common subsequence fast algorithm. In *2011 International Conference on Consumer Electronics, Communications and Networks*, pages 4338–4341. IEEE, 2011. ISBN 978-1-61284-458-9.

[71] Chad Loeven. MITRE ATT&CK: A Rosetta Stone for the Cyber Security Ecosystem, 2019. URL https://www.vmray.com/cyber-security-blog/mitre-attack-rosetta-stone-cyber-security-ecosystem/. Accessed 06/06/2020 14:23:01.

[72] Huabiao Lu, Baokang Zhao, Xiaofeng Wang, and Jinshu Su. DiffSig: Resource Differentiation Based Malware Behavioral Concise Signature Generation. In Khabib Mustofa, editor, *Information and communication technology*, volume 7804 of *Lecture Notes in Computer Science*, pages 271–284. Springer, Berlin, 2013. ISBN 978-3-642-36817-2.

[73] Aleksandar Maričić, Kumar Bhattacharyya, Koen van den Dool, Erik Frinking, and Katarina

Kertysova. *Cybersecurity: Ensuring awareness and resilience of the private sector across Europe in face of mounting cyber risks*. EESC, Brussels, 2018. ISBN 9283041054.

[74] Joseph Marks. Analysis | The Cybersecurity 202: Coronavirus pandemic has not stopped cyberattacks on hospitals and other vital infrastructure. *The Washington Post*, 2020, 20.4.2020. URL https://www.washingtonpost.com/news/powerpost/paloma/the-cybersecurity-202/2020/04/20/the-cybersecurity-202-coronavirus-pandemic-has-not-stopped-cyberattacks-on-hospitals-and-other-key-targets/5e9caee2602ff10d49ae8640/.

[75] Jaume Martin, mmorenog, jovimon, and j0sm1. Yara-Rules, 2020. URL https://github.com/Yara-Rules/. Accessed 22.01.2020.

[76] Jaume Martin, mmorenog, jovimon, and j0sm1. rules/packer.yar at master · Yara-Rules/rules · GitHub, 2020. URL https://github.com/Yara-Rules/rules/blob/master/packers/packer.yar. Accessed 22.01.2020.

[77] Microsoft. Registry Cmdlets: Working with the Registry, 2015. URL https://devblogs.microsoft.com/scripting/registry-cmdlets-working-with-the-registry/. Accessed 18.05.2020.

[78] Microsoft. LookupPrivilegeValueA function (winbase.h) - Win32 apps, 2018. URL https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-lookupprivilegevaluea. Accessed 29.3.2020.

[79] Microsoft. NtWriteFile function (ntifs.h) - Windows drivers, 2018. URL https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntwritefile. Accessed 27.01.2020.

[80] Microsoft. CreateFileA function (fileapi.h) - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea. Accessed 24.01.2020.

[81] Microsoft. FILETIME (minwinbase.h) - Win32 apps, 2020. URL https://docs.microsoft.com/de-de/windows/win32/api/minwinbase/ns-minwinbase-filetime. Accessed 7.4.2020.

[82] Microsoft. GetLocalTime function (sysinfoapi.h) - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getlocaltime. Accessed 22.3.2020.

[83] Microsoft. GetSystemTimeAsFileTime function (sysinfoapi.h) - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemtimeasfiletime. Accessed 7.4.2020.

[84] Microsoft. Installutil.exe (Installer Tool), 2020. URL https://docs.microsoft.com/en-us/dotnet/framework/tools/installutil-exe-installer-tool. Accessed 8.6.2020.

[85] Microsoft. OpenFile function (winbase.h) - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-openfile. Accessed 16.4.2020.

[86] Microsoft. Programming reference for the Win32 API - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/. Accessed 24.01.2020.

[87] Microsoft. VirtualAlloc function (memoryapi.h) - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc. Accessed 20.4.2020.

[88] Microsoft. WriteFile function (fileapi.h) - Win32 apps, 2020. URL https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile. Accessed 16.4.2020.

[89] Microsoft. SYSTEMTIME (minwinbase.h) - Win32 apps, 2020. URL https:

//docs.microsoft.com/de-de/windows/win32/api/minwinbase/ns-minwinbase-systemtime. Accessed 22.3.2020.

[90] MISP Project. MISP - Open Source Threat Intelligence Platform & Open Standards For Threat Information Sharing (formely known as Malware Information Sharing Platform), 2020. URL https://www.misp-project.org/. Accessed 18.6.2020.

[91] MITRE. AppInit DLLs, Technique T1103 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1103/. Accessed 16.6.2020.

[92] MITRE. Hooking, Technique T1179 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1179/. Accessed 6.6.2020.

[93] MITRE. MITRE ATT&CK®, 2020. URL https://attack.mitre.org/. Accessed 27.5.2020.

[94] MITRE. New Service, Technique T1050 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1050/. Accessed 16.6.2020.

[95] MITRE. Process Hollowing, Technique T1093 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1093/. Accessed 16.6.2020.

[96] MITRE. Process Injection, Technique T1055 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1055/. Accessed 16.6.2020.

[97] MITRE. Registry Run Keys / Startup Folder, Technique T1060 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1060/. Accessed 8.6.2020.

[98] MITRE. Scheduled Task, Technique T1053 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1053/. Accessed 16.6.2020.

[99] MITRE. Tactics - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/tactics/enterprise/. Accessed 29.5.2020.

[100] MITRE. Winlogon Helper DLL, Technique T1004 - Enterprise | MITRE ATT&CK®, 2020. URL https://attack.mitre.org/techniques/T1004/. Accessed 16.6.2020.

[101] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. AMAL: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251–266, 2015.

[102] Asit More and Shashikala Tapaswi. Virtual machine introspection: towards bridging the semantic gap. *Journal of Cloud Computing*, 3(1):0133, 2014.

[103] Jens Müller, Vladislav Mladenov, Dennis Felsch, and Jörg Schwenk. PostScript Undead: Pwning the Web with a 35 Years Old Language. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, volume 11050 of *Lecture Notes in Computer Science*, pages 603–622. Springer International Publishing, Cham, 2018. ISBN 978-3-030-00469-9.

[104] Gary Nebett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publ, Indianapolis, Ind., 1. print edition, 2000. ISBN 1578701996.

[105] Mike Neurohr. Longest common subsequence - Rosetta Code, 2020. URL https://rosettacode.org/wiki/Longest_common_subsequence#Python. Accessed 26.5.2020.

[106] Stavros D. Nikolopoulos and Iosif Polenakis. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques*, 13(1):29–46, 2017.

[107] Tomasz Nowak and Antoni Sawicki. NTAPI Undocumented Functions, 2019. URL https://undocumented.ntinternals.net/. Accessed 27.01.2020.

[108] OASIS Cyber Threat Intelligence Techical Comittee. Introduction to STIX, 2020. URL https://oasis-open.github.io/cti-documentation/stix/intro. Accessed 18.6.2020.

[109] Taka Okunishi. okunishinishi/python-stringcase, 2020. URL https://github.com/okunishinishi/python-stringcase. Accessed 25.5.2020.

[110] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Computing Surveys*, 52(5):1–48, 2019.

[111] H. Orman. The Morris worm: a fifteen-year perspective. *IEEE Security and Privacy Magazine*, 1(5):35–43, 2003.

[112] Chinmaya Kumar Patanaik, Ferdous A. Barbhuiya, and Sukumar Nandi. Obfuscated malware detection using API call dependency. In R. Chandrasekhar, P. Venkat Rangan, and Andrew S. Tanenbaum, editors, *SecurIT 2012*, ICPS, pages 185–193. Association for Computing Machinery, 2012. ISBN 9781450318228.

[113] Bryan D. Payne, Steven Maresca, Tamas K. Lengyel, and Antony Saba. Introduction to LibVMI, 2015. URL http://libvmi.com/docs/gcode-intro.html. Accessed 06.02.2020.

[114] Fabian Pedregosa. memory-profiler, 2020. URL https://pypi.org/project/memory-profiler/. Accessed 18.6.2020.

[115] Xabier Ugarte Pedrero. PyREBox, a Python Scriptable Reverse Engineering Sandbox, 2017. URL https://blog.talosintelligence.com/2017/07/pyrebox.html#more. Accessed 14.07.2018.

[116] Xabier Ugarte Pedrero. Cisco-Talos/pyrebox: Malware Monitor, 2018. URL https://github.com/Cisco-Talos/pyrebox/tree/master/mw_monitor. Accessed 10.02.2020.

[117] perlpunk. https://pyyaml.org, 2020. URL https://pyyaml.org/. Accessed 10.4.2020.

[118] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In Angelos Stavrou, editor, *Proceedings of the 1st ACM workshop on Virtual machine security*, page 1. ACM, 2009. ISBN 9781605587806.

[119] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 Architecture to Derive Virtual Machine State Information. In *2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, pages 166–175. IEEE, 2010. ISBN 978-1-4244-7517-9.

[120] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-Based System Call Tracing for Virtual Machines. In Tetsu Iwata and Masakatsu Nishigaki, editors, *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-25140-5.

[121] Daniel Plohmann. danielplohmann/apiscout: winapi_contexts.csv, 2019. URL https://github.com/danielplohmann/apiscout/blob/master/apiscout/data/winapi_contexts.csv. Accessed 18.5.2020.

[122] Daniel Plohmann, Martin Clauß, Steffen Enders, and Elmar Padilla. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape: 1-19 Pages / The Journal on Cybercrime & Digital Investigations, Vol 3 No 1 (2017): Botconf 2017. 2017.

[123] Daniel Plohmann, Steffen Enders, and Elmar Padilla. ApiScout: Robust Windows API Usage Recovery for Malware Characterization and Similarity Analysis. In *The Journal on Cybercrime & Digital Investigations*, volume 1, pages 1–16. 2018. URL https://journal.cecyf.fr/ojs/index.php/cybin/article/view/20/23. Accessed 18.05.2020.

[124] proofpoint. The Cybercrime Economics of Malicious Macros, 2015. URL http://index-of.es/Viruses/The%20Cybercrime%20Economics%20of%20Malicious%20Macros.pdf. Accessed 30.12.2019.

[125] Python Software Foundation. 12. Virtual Environments and Packages — Python 3.8.3 documentation, 2020. URL https://docs.python.org/3/tutorial/venv.html. Accessed 25.5.2020.

[126] Python Software Foundation. Installing Python Modules — Python 3.8.3 documentation, 2020. URL https://docs.python.org/3/installing/index.html. Accessed 25.5.2020.

[127] Ralph Langner. To Kill a Centrifuge: A Technical Analysis of What Stuxnet's Creators Tried to Achieve, 2013. URL https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf. Accessed 30.12.2019.

[128] Stephan Richter. lxml - Processing XML and HTML with Python, 2020. URL https://lxml.de/. Accessed 25.5.2020.

[129] S.A.M Rizvi and Pankaj Agarwal. A time efficient algorithm for finding longest common subsequence from two molecular sequences. In *2007 IEEE 33rd Annual Northeast Bioengineering Conference*, pages 302–306. IEEE, Jan. 2007. ISBN 978-1-4244-1032-3.

[130] Scott J. Roberts and Rebekah Brown. *Intelligence-driven incident response: Outwitting the adversary*. O'Reilly Media, Sebastopol, CA, first edition edition, 2017. ISBN 9781491934944. URL http://proquest.tech.safaribooksonline.de/9781491935187.

[131] Florian Roth. Neo23x0/signature-base, 2020. URL https://github.com/Neo23x0/signature-base. Accessed 26.5.2020.

[132] R. Devika Rubi and L. Arockiam. Positional_LCS: A position based algorithm to find Longest Common Subsequence (LCS) in Sequence Database (SDB). In *IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), 2012*, pages 1–4. IEEE, 2012. ISBN 978-1-4673-1344-5.

[133] Javier Ruere. multiprocessing-logging, 2020. URL https://pypi.org/project/multiprocessing-logging/. Accessed 24.6.2020.

[134] Mark E. Russinovich. Sysinternals Freeware - Inside the Native API, 2004. URL https://web.archive.org/web/20050807083555/www.sysinternals.com/Information/NativeApi.html. Accessed 24.01.2020.

[135] Mark E. Russinovich. Inside Native Applications - Windows Sysinternals, 2006. URL https://docs.microsoft.com/en-us/sysinternals/learn/inside-native-applications. Accessed 24.01.2020.

[136] Mark E. Russinovich. Windows Sysinternals - Windows Sysinternals, 2020. URL https://docs.microsoft.com/en-us/sysinternals/. Accessed 4.6.2020.

[137] Mark E. Russinovich, Alex Ionescu, and David A. Solomon. *Windows internals: Part 1*. Microsoft Press, Redmond, Wash, 6th ed. edition, 2012. ISBN 978-0-7356-4873-9. URL http://proquest.tech.safaribooksonline.de/9780735671294.

[138] Marc Salinas and José Miguel Holguin. Malware Report: Evolution of Trickbot, 2017. URL https://www.securityartwork.es/wp-content/uploads/2017/07/Trickbot-report-S2-Grupo.pdf. Accessed 29.05.2020.

[139] Om Prakash Samantray, Satya Narayan Tripathy, and Susanta Kumar Das. A study to Understand Malware Behavior through Malware Analysis. In *2019 IEEE International Conference on System, Computation, Automation and Networking (ICSCAN)*, pages 1–5. IEEE, 2019. ISBN 978-1-7281-1525-2.

[140] Michael Satran. Unicode in the Windows API, 2018. URL https://docs.microsoft.com/en-us/windows/win32/intl/unicode-in-the-windows-api. Accessed 27.01.2020.

[141] Michael Satran and Mike Jacobs. Windows Coding Conventions - Win32 apps, 2018. URL https://docs.microsoft.com/en-us/windows/win32/learnwin32/windows-coding-conventions. Accessed 24.01.2020.

[142] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons Incorporated, New York, 2015. ISBN 1119096723. URL https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=4875261.

[143] Sebastian Schrittwieser and Stefan Katzenbeisser. Code Obfuscation against Static and Dynamic Reverse Engineering. In Tomáš Filler, Tomáš Pevný, Scott Craver, and Andrew

Ker, editors, *Information hiding*, volume 6958 of *Lecture Notes in Computer Science*, pages 270–284. Springer, Berlin and Heidelberg, 2011. ISBN 978-3-642-24177-2.

[144] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Network and Distributed System Security Symposium*, 2008. URL https://www.ndss-symposium.org/wp-content/uploads/2017/09/Impeding-Malware-Analysis-Using-Conditional-Code-Obfuscation-paper-Monrul-Sharif.pdf.

[145] Amit Shukla and Suneeta Agarwal. A relative position based algorithm to find out the longest common subsequence from multiple biological sequences. In Ieee, editor, *2010 International Conference on Computer and Communication Technology*, pages 496–502. IEEE, 2010. ISBN 978-1-4244-9033-2.

[146] Michael Sikorski and Andrew Honig. *Practical malware analysis: The hands-on guide to dissecting malicious software*. No Starch Press, San Francisco, 2012. ISBN 978-1-59327290-6. URL http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10574799.

[147] Karan Sood and Shaun Hurley. NotPetya Technical Analysis – A Triple Threat: File Encryption, MFT Encryption, Credential Theft, 2017. URL https://www.crowdstrike.com/blog/petrwrap-ransomware-technical-analysis-triple-threat-file-encryption-mft-encryption-credential-theft/. Accessed 29.12.2019.

[148] Sophos Ltd. Macro Viruses: What They Are, and How to Avoid Them, 2019. URL https://home.sophos.com/en-us/security-center/articles/2019/04/macro-viruses.aspx. Accessed 30.12.2019.

[149] Murugiah Souppaya and Karen Scarfone. *Guide to Malware Incident Prevention and Handling for Desktops and Laptops*. National Institute of Standards and Technology, 2013.

[150] Deepa Srinivasan and Xuxian Jiang. Time-Traveling Forensic Analysis of VM-Based High-Interaction Honeypots. In Muttukrishnan Rajarajan, Fred Piper, Haining Wang, and George Kesidis, editors, *Security and Privacy in Communication Networks*, volume 96 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 209–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31908-2.

[151] Stanford University. Operator Precedence, 2016. URL http://intrologic.stanford.edu/glossary/operator_precedence.html. Accessed 25.4.2020.

[152] Stichting Cuckoo Foundation. Configuration — Cuckoo Sandbox v2.0.7 Book, 2019. URL https://cuckoo.sh/docs/installation/host/configuration.html. Accessed 14.02.2020.

[153] Stichting Cuckoo Foundation. Cuckoo Sandbox - Automated Malware Analysis, 2019. URL https://cuckoosandbox.org/. Accessed 10.02.2020.

[154] Talos. Talos Blog || Cisco Talos Intelligence Group - Comprehensive Threat Intelligence: NavRAT Uses US-North Korea Summit As Decoy For Attacks In South Korea, 2018. URL https://blog.talosintelligence.com/2018/05/navrat.html. Accessed 29.5.2020.

[155] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. Prentice Hall and Pearson, Boston, MA, 4th ed. edition, 2015. ISBN 978-0-13-359162-0.

[156] Benjamin Taubmann and Hans P. Reiser. Secure Architecture for VMI-based Dynamic Malware Analysis in the Cloud. In Matthieu Roy, Javier Alonso Lopez, and Antonio Casimiro, editors, *Fast Abstract in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN2016-FAST-ABSTRACT, 2016. URL https://hal.archives-ouvertes.fr/hal-01316519.

[157] The Snort Project. SNORT Users Manual 2.9.15.1, 2020. URL http://manual-snort-org.s3-website-us-east-1.amazonaws.com/. Accessed 19.01.2020.

[158] Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. A malware instruction set for behavior-based analysis, 2009. URL https://madoc.bib.uni-mannheim.de/2579/.

[159] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[160] Rohan Viegas. Custom Threat Scoring with VTI, 2016. URL https://www.vmray.com/cyber-security-blog/custom-threat-scoring-vti/. Accessed 12.02.2020.

[161] Rohan Viegas. VMRay Analyzer 2.2 – An Improved User Experience for Malware Analysts and Incident Responders, 2018. URL https://www.vmray.com/cyber-security-blog/vmray-analyzer-2-2-improved-user-experience/. Accessed 06/03/2020 23:35:06.

[162] VirusShare. VirusShare.com, 2020. URL https://virusshare.com/. Accessed 2.6.2020.

[163] Virustotal. VirusTotal, 2020. URL https://www.virustotal.com. Accessed 23.01.2020.

[164] Virustotal. VirusTotal - VT Hunting, 2020. URL https://www.virustotal.com/gui/hunting-overview. Accessed 23.01.2020.

[165] VMRay. Malware Analyzer Performance, 2014. URL https://www.vmray.com/cyber-security-blog/malware-analyzer-performance/. Accessed 08.02.2020.

[166] VMRay. VMRay Technology Whitepaper: Hypervisor-based monitoring for malware analysis and threat detection, 2018. URL https://306dlv9g2c4fl4za451pdg1b-wpengine.netdna-ssl.com/wp-content/uploads/2018/09/VMRay-Technology-Whitepaper-2018.pdf. Accessed 04.02.2020.

[167] VMRay. About Us, 2019. URL https://www.vmray.com/company/malware-analysis-company/. Accessed 08.02.2020.

[168] VMRay. VMRay Analyzer Admin Documentation, 2019.

[169] VMRay. VMRay Technology Data Sheet VMRay Analyzer: A Smarter, Stealthier Malware Sandbox, 2019. URL https://www.vmray.com/wp-content/uploads/2019/03/Data-Sheet-VMRay-Analyzer.pdf. Accessed 08.02.2020.

[170] VMRay. Malware Analysis Reports | VMRay, 2020. URL https://www.vmray.com/dfir-resources/malware-analysis-reports/. Accessed 23.6.2020.

[171] Markus Völter. *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, Lexington, KY, 2010-2013. ISBN 9781481218580.

[172] Markus Wagner, Alexander Rind, Gernot Rottermanner, Christina Niederer, and Wolfgang Aigner. Knowledge-Assisted Rule Building for Malware Analysis. In *Forschungsforum der österreichischen Fachhochschulen*, volume 10. 2016.

[173] Markus Wagner, Alexander Rind, Niklas Thür, and Wolfgang Aigner. A knowledge-assisted visual malware analysis system: Design, validation, and reflection of KAMAS. *Computers & Security*, 67:1–15, 2017.

[174] Gary Wang, Zachary J. Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In USENIX Association, editor, *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015. URL https://www.usenix.org/system/files/conference/woot15/woot15-paper-wang.pdf.

[175] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine*, 5(2):32–39, 2007.

[176] Carsten Willems, Ralf Hund, and Thorsten Holz. CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring, 2012. URL https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2012/11/26/TR-HGI-2012-002.pdf. Accessed 16.01.2020.

[177] Sean Wilson and Sergei Frankoff. UnpacMe, 2020. URL https://www.unpac.me/#/. Accessed 12.6.2020.

[178] Davey Winder. Cyber Attacks Against Hospitals Have 'Significantly Increased' As Hackers Seek To Maximize Profits. *Forbes*, 2020, 8.4.2020. URL https://www.forbes.com/sites/daveywinder/2020/04/08/cyber-attacks-against-hospitals-fighting-covid-19-confirmed-interpol-issues-purple-alert/#7d08b9ce58bc.

[179] World Wide Web Consortium. XML Path Language (XPath) 3.1, 2017. URL https://www.w3.org/TR/2017/REC-xpath-31-20170321/. Accessed 24.3.2020.

[180] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2018. URL https://www.w3.org/TR/REC-xml/#syntax. Accessed 24.3.2020.

[181] Rui YANG, Jiang-chun REN, Shuai BAI, and Tian TANG. A Digital Forensic Framework for Cloud Based on VMI. *DEStech Transactions on Computer Science and Engineering*, (cst), 2017.

[182] Ilsun You and Kangbin Yim. Malware Obfuscation Techniques: A Brief Survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300. IEEE Computer Society, 2010. ISBN 978-1-4244-8448-5.

[183] Alexander Zhdanov. Generation of Static YARA-Signatures Using Genetic Algorithm. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 220–228. IEEE, 2019. ISBN 978-1-7281-3026-2.