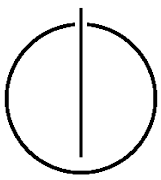
FAKULTÄT FÜR INFORMATIK

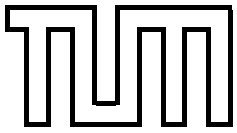
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Behavior-based Malware Detection with Quantitative Data Flow Analysis

Tobias Wüchner





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl XXII - Software Engineering

Behavior-based Malware Detection with Quantitative Data Flow Analysis

Tobias Wüchner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende:	Univ.-Prof. Dr. rer. nat. Claudia Eckert
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. rer. nat. Alexander Pretschner
2.	Univ.-Prof. Dr.-Ing. Felix Freiling, Friedrich-Alexander-Universität Erlangen-Nürnberg

Die Dissertation wurde am 27.01.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.05.2016 angenommen.

Acknowledgments

I would like to thank my first adviser, Prof. Dr. Alexander Pretschner, for giving me the opportunity to almost freely decide on a research topic that I was truly passionate and excited about and for letting me *"play around"* – even though this meant that I got a bit offside the main research profile of the group. Although in particular the beginning of my time as a Phd student, without even a vague vision of where the whole thing should be going to, was sometimes frustrating, in retrospect I am glad having taken this road.

My thanks also go to Prof. Dr.-Ing. Felix Freiling for agreeing to be my second adviser, for providing me with useful feedback in earlier stages of my research, and in particular for supporting me with my research stay in England.

Furthermore, I would like to thank Siemens AG and the Siemens CERT for funding my research. In particular I want to thank Thomas Schreck for stimulating my thoughts and teaching me what really matters in industry.

My sincere thanks go to Prof. Dr. Martín Ochoa for always encouraging me to carry on with my research, even when I was close to giving up. Your continuous belief in me and your tireless support definitely were among the main reasons for me to push further – thank you a lot for this!

Also, I shall thank my office-mate Dr. Enrico Lovat and my flat-mate Dominik Holling for enduring my frustrations, for endless and fruitful discussions, and for always being open and critical.

I am very grateful to Hannah Brosch, Benjamin Lautenschläger, Alei Salem, and Enrico Lovat for reviewing my thesis and for providing valuable feedback that definitely improved the quality of this work.

Thank you also Andrea Catalucci, Aleksander Cisłak, and Gaurav Srivastava for supporting me in the development of my research prototypes and for helping me to turn my often half-baked ideas into usable software.

I would further like to express my gratitude to my parents and family for always having supported me in this endeavor, both financially and motivational.

Last but most important, thank you Monika for always backing me up and supporting me through all these years, even though my decisions for sure did not make your life much easier. Thank you for your empathy and understanding when I was not as much there for you as I should have been. Thank you for forcing me to sometimes hold on, reflect on my goals and plans in life, and for always reminding me that there is a life beyond work and research. I will not forget this!

Abstract

Malware remains one of the biggest IT security threats, with available detection approaches struggling to cope with a professionalized malware development industry. The increasing sophistication of today's malware and the prevalent usage of obfuscation techniques renders traditional static detection approaches increasingly ineffective. This thesis contributes towards improving this situation by proposing a novel effective, robust, and efficient concept of leveraging quantitative data flow analysis for behavior-based malware detection.

We interpret system calls, issued by monitored processes, as quantifiable flows of data between system entities, such as files, sockets, or processes. We aggregate multiple flows as quantitative data flow graphs (QDFGs) that model the behavior of a system during a certain period of time. We operationalize this model for behavior-based malware detection in four different ways by either detecting patterns of known malicious behavior in QDFGs of unknown samples, or by profiling and identifying malicious behavior with graph metrics on QDFGs.

The core contribution of this thesis is the demonstration that quantitative data flow information improves detection effectiveness compared to non-quantitative analyses. We establish high detection effectiveness, obfuscation robustness, and efficiency by evaluations on a large and diverse malware and goodware data set.

Zusammenfassung

Schadsoftware stellt nach wie vor eines der größten Probleme im Bereich IT-Sicherheit dar. Die Effektivität verfügbarer Erkennungsmethoden, insbesondere kommerzieller Anti-Virus-Produkte, ist in zunehmendem Maß durch die ansteigende Professionalisierung der Entwickler von Schadsoftware bedroht. Vor allem die stetig steigende Komplexität und der mittlerweile nahezu flächendeckende Einsatz von Obfuskations- und Anti-Analysetechniken durch Standard-Schadsoftware stellen insbesondere statische Erkennungsansätze vor große Probleme. Die vorliegende Arbeit leistet einen Beitrag hin zu einer Verbesserung dieser Situation mit einem neuen, effektiven, robusten und effizienten Ansatz zur Erkennung von Schadsoftware basierend auf quantitativer Datenflussanalyse.

Dazu interpretieren wir von Prozessen ausgelöste Funktionsaufrufe als quantifizierbare Datenflüsse zwischen verschiedenen Systemressourcen, etwa Dateien, Netzwerkschnittstellen, oder anderen Prozessen. Mehrerer solcher Datenflüsse werden dann in Form sogenannter quantitativer Datenflussgraphen (QDFGs) aggregiert, die das Systemverhalten während eines bestimmten Zeitraums modellieren. Wir stellen vier verschiedene Ansätze zur Operationalisierung dieses Modells für die verhaltensbasierte Erkennung von Schadsoftware vor. Diese geschieht entweder durch Definition und Wiederkennung von Mustern bekannten Schadverhaltens in QDFGs unbekannter Prozesse oder über das Erstellen von Profilen typischen Schadverhaltens mit Hilfe spezieller Graphmetriken auf QDFGs.

Der Kernbeitrag dieser Arbeit ist der Nachweis der Nützlichkeit quantitativer Datenflussanalyse zur Verbesserung der Erkennungseffektivität gegenüber nichtquantitativen Analysen. Wir weisen die hohe absolute Effektivität, Robustheit und Effizienz unserer Erkennungsansätze auf Basis einer großen und heterogenen Sammlung bösartiger und gutartiger Software nach.

Contents

Outline of this Thesis

CHAPTER 1: INTRODUCTION In this chapter we discuss problems of current malware detection approaches, derive a set of open research questions, and sketch the contributions of this thesis towards answering them.

CHAPTER 2: BACKGROUND In this chapter we briefly introduce the history and genealogy of malware, give an overview of the two basic categories of malware detection, and briefly discuss the most prevalent detection models and concepts.

CHAPTER 3: SYSTEM MODEL In this chapter we propose a model to represent the interaction between different system entities as quantitative data flows that together constitute a quantitative data flow graph. An earlier version of this model was presented in [144], co-authored as first author by the writer of this thesis. We then sketch how to leverage this model for behavior-based malware detection.

CHAPTER 4: PATTERN-BASED DETECTION In this chapter we discuss how to use QDFG patterns to detect malicious activities. We do so with a deductive approach that detects pre-defined patterns of known malicious behavior in unknown samples and an inductive one that mines behavioral commonalities of malware samples to infer generalized detection patterns. Parts of this chapter are based on published work [144], co-authored as first author by the writer of this thesis.

CHAPTER 5: METRIC-BASED DETECTION In this chapter we propose an approximate notion of QDFG similarity based on graph metrics as a flexible alternative to sub-graph isomorphism-based detection. To this end, we discuss a deductive approach to profile behavior with generic graph metrics and an inductive extension to this where we use genetic programming to generate complex graph metrics that are by design effective. Some parts of this chapter are based on a previous publication [145], co-authored as first author by the writer of this thesis.

CHAPTER 6: ASSESSMENT AND OPERATIONALIZATION In this chapter we critically assess the devised detection approaches with respect to common malware detection quality attributes, carve out strengths and weaknesses, and relate them to our initial research questions. We also show how to operationalize our approaches for different detection settings and reason about their real-world utility.

CHAPTER 7: CONCLUSION This chapter summarizes the conclusions and contributions of this thesis, critically reflects on the gained answers to the initial research questions, and discusses open questions and possible directions for future work in the area of malware detection using quantitative data flow analysis.

1. Introduction

Malicious software (malware) remains to be one of the biggest IT security threats, with hundreds of thousands new malware samples and reported infections appearing on a daily basis [112]. Despite the widespread use of security software like anti-virus scanners or firewalls, the yearly global economic losses caused by malware are estimated to be in the order of several hundred billion dollars [95].

The reason for this can be found on both the attacker and the defender sides. Commercial anti-malware products are still mainly based on signature matching or basic static analysis and thus very limited in detecting unknown malware. This results in an inevitable time interval between the release of a new, unknown malware and the specification and distribution of the corresponding detection signatures [134]. Within this time interval, the malware cannot be detected through signatures and can freely operate, if no other security measures are in place.

At the same time, malware development has become a lucrative business model [22, 154, 59]. A considerable body of today's malware landscape is highly sophisticated, polymorphic, and makes use of a diverse portfolio of advanced obfuscation and anti-analysis techniques [152, 15]. This in particular renders signature-based malware detection mechanisms more and more ineffective as modern polymorphic malware often autonomously creates obfuscated clones, yielding completely different looking malicious binaries with every new malware generation [35].

To cope with the issue of modern malware continuously obfuscating itself and significantly hampering static detection, research on behavior-based malware detection in the past decade gained considerable momentum. Unlike static detection approaches, behavior-based detection approaches do not use the binary of a malware sample for profiling and detection but rather detect malware by specifying or learning typical malicious behavior and then later re-identifying it in unknown samples. In a nutshell, behavior-based detection approaches aim to detect malicious programs on the basis of issued behavior under the assumption that it significantly differs from the behavior of benign programs.

While behavior-based detection approaches can quite significantly differ in terms of used detection methodology, system scope, or deployment concept, almost all of them have in common that they build on top of some sort of behavior model. Behavior models are representations of low-level observable process or system behavior (system or API calls), typically either employing abstractions with rather simple topology, like e.g. n-grams, or more complex topology like trees or graphs.

These approaches make it significantly harder for malware to prevent its detection, as standard static obfuscation techniques typically barely affect behavior.

However, a rather new challenge is posed by advanced obfuscation techniques [116, 115, 6] that aim to confuse *behavior* profiles e.g. by reordering system call[1] sequences, injecting bogus sequences of calls, or alternating between semantically equivalent ones. This has consequences for the detection accuracy of approaches that operate on behavior models at rather low-abstraction levels that e.g. are directly based on raw system call traces. Recent studies [6, 116, 115] showed that many state of the art behavior-based detection concepts, based on system call sequence analysis, are particularly sensitive to certain behavior obfuscation transformations, in many cases rendering them ineffective on highly obfuscated malware.

In contrast, detection approaches that utilize more complex behavior models at higher abstraction levels are intuitively less likely to be confused by behavior obfuscation [6]. This is because many changes to the underlying system call traces often barely or only slightly influence abstracted behavior models. In particular, approaches that detect malware based on induced data flows seem to be more promising. This is because data flows, resulting from processes interacting with system resources, are more stable with respect to alternations of system call traces.

In the context of malware detection, two principle ways of performing data flow analysis have been proposed. *Taint-based data flow analysis* concepts usually leverage full system emulation and explicitly track the flow of data by means of shadow memory and taint propagation [151]. *System call centric data flow approximation* based approaches usually infer potential flows of data by interpreting system calls and their according to their known data flow semantics and, by correlating multiple system calls, infer likely data flows through a system [80].

While taint-based approaches thus by construction are more precise than system call inference based approaches, which usually leads to more accurate malware classification results, they in most cases induce significant performance penalties and are thus often very inefficient [151]. Furthermore, taint-based approaches usually require certain execution environments and thus are complicated to deploy, which limits their real-world practicality and portability.

System call inference based data flow analysis approaches in contrast are usually easier to deploy and induce less computational overhead than taint-based approaches [80]. At the downside, data flow approximation naturally is less precise than exact taint-based tracking, which leads to reduced detection accuracy.

Both data flow analysis concepts in the context of malware detection thus either suffer from bad efficiency and portability or from reduced effectiveness which, depending on the operational context, limits their real-world applicability.

[1] To remain consistent with the terminology used in literature and to avoid naming confusion, we use the terms *Windows API call*, *system call*, and *syscall* synonymously in this thesis.

One specialization of data flow analysis is *quantitative data flow analysis* that does not only tackle the question *whether or not* a flow of data happened between two entities, but also *how much* data has been transferred. Quantitative data flow analysis at system call level thus can be seen as a compromise between precise taint-based analysis and approximate (possibilistic) system call inference based data flow analysis. This is, by considering the quantitative dimension for approximate data flow tracking one can improve analysis precision [89], while still benefiting from high efficiency and portability of system call centric data flow inference.

To leverage this insight for behavior-based malware detection we propose so-called *Quantitative Data Flow Graphs (QDFGs)* to represent quantitative data flows between system entities, where nodes represent the system entities and directed weighted edges represent the quantitative flows between them. QDFGs can be constructed by mapping system events, e.g. system calls, to creations or updates of edges and nodes according to their data flow semantics. Due to their concise and abstracting nature with respect to individual system events, we consider QDFGs a suitable way of representing complete system activities over time and thus to act as basis for detection and analysis of malicious behavior induced by malware. Moreover, the quantitative information inherently encoded in data flow related system call arguments helps to increase analysis precision and adds another source of potentially highly characteristic behavior information to the classification process, which we assume to improve detection accuracy. We are not aware of any approach that explicitly models and uses such quantitative data flow information for behavior-based malware detection.

In this thesis we investigate the utility of *quantitative data flow analysis* at the system call level for behavior-based malware detection. More precisely, we want to investigate if using *quantitative data flow analysis* yields more accurate malware detection results than non-quantitative approaches.

1.1. Research Description

The goal of the work presented in this thesis is to understand the utility of quantitative data flow analysis for behavior-based malware detection. In the context of malware detection, the behavior of one or more processes is typically captured in form of traces of issued system calls. In order to lift this low-level representation of behavior to a higher abstraction level, we translate them into quantifiable data flows. To do so we interpret system calls according to their data flow semantic as inducing quantifiable flows of data between different system entities such as processes, files, or network sockets. We then aggregate these flows in form of so-called quantitative data flow graphs (QDFGs) that represent the flows as edges between nodes that model the source and sink system entities of the flows.

Non-quantitative data flow graphs have already been shown to be useful for malware detection [113, 114, 151]. In the malware detection domain, we can observe a correlation between the abstraction level and granularity of information that is used to model the behavior of malware and the accuracy of respective detection approaches. In particular, building detection approaches upon abstracted behavior models together with few selected abstract context aspects seems to lead to better detection accuracy than directly using low-level behavior artifacts [24].

Our work is thus motivated by the expectation that we can improve detection accuracy by not only taking an abstracted high-level data flow perspective on malware behavior in that we consider malware-induced data flows from an existential or possibilistic perspective, i.e. only reason about whether or not data has been transferred between system entities, but by also taking the *quantitative component* of the respective data flows into account, i.e. how much data has potentially been transferred. More precisely, the intuition is that quantitative data flows induced by malware significantly differ from data flows that result from benign behavior. This gives us a rich and characteristic source of high-level context information that so far has not been tapped by malware detection research. In sum, we hypothesize that by using this high-level context information, i.e. quantitative data flow properties that are inherently embedded in observable program behavior, we can significantly improve detection accuracy.

To investigate this hypothesis we devised multiple behavior-based detection approaches that operationalize quantitative data flow graphs as behavior models for highly accurate, efficient, and robust malware detection. The description and analysis of these detection approaches will be the core components of this thesis.

1.1.1. Tackled Problems and Research Goals

The quality of malware detection approaches is typically measured along the lines of detection effectiveness, efficiency, and robustness: a good malware detection approach should be able to detect malware with good sensitivity and specificity, avoiding false positive and false negative classifications as far as possible, and only imposing a reasonably high performance overhead. Moreover, considering the increasing prevalence of obfuscated malware, a good detection approach should also be robust against obfuscation, providing accurate results even for malware that uses obfuscation or anti-analysis techniques.

The research goals of this work are thus tightly aligned with the following problems that we, based on a thorough literature review and continuous discussions with domain experts, deem most important in the context of behavior-based malware detection:

P1: Behavior-based malware detection approaches yield false positive detection rates (i.e. goodware mistakenly classified as malware) that are often considered too high to be useful in real-world operational settings [2].

P2: The accuracy of many state of the art behavior-based malware detection approaches is significantly impacted by malware obfuscating its behavior [6].

While we by no means claim to be able to solve any of these problems in its entirety, we nevertheless see them as guiding posts for our work. The aim of this thesis is thus to at least partially solve the aforementioned problems and contribute towards the state of the art in behavior-based malware detection.

1.1.2. Research Questions

The aforementioned problems of current malware detection approaches motivate the following set of research questions, which we will answer in the course of this thesis:

RQ1: Can quantitative data flow analysis be used for behavior-based malware detection?

RQ2: Can the use of quantitative data flow analysis for malware detection improve detection accuracy compared to non-quantitative detection?

RQ3: Is our system call based data flow approximation sufficiently precise to yield highly accurate detection results while being very efficient?

RQ4: Does the quantitative data flow abstraction lead to higher robustness towards behaviorally obfuscated malware than using raw system calls?

In sum, these research questions can be subsumed under the guiding research hypothesis tackled by this dissertation, which states that

> **Quantitative data flow analysis yields better malware detection accuracy than non-quantitative analysis.**

Throughout this thesis we will continuously try to generalize the insights gained from the approach-specific evaluations in order to answer the introduced research questions, with the ultimate goal of confirming our main hypothesis.

1.2. Structure and Research Methodology

We approach the aforementioned research goals in five steps in order to answer the research questions from Section 1.1.2 and investigate our main hypothesis.

To put our work in context, we first give an overview of the most prevalent malware categories and recap the most important malware detection concepts. After this, we introduce a generic model to represent system behavior in form of quantitative data flow graphs which are aggregations of quantitative data flows induced by system calls. Subsequently we introduce four different approaches that leverage this model for behavior-based malware detection and discuss how far their evaluation gives answers to our main research questions, stated in Section 1.1.2.

The detection approaches proposed in this thesis follow two main concepts: Pattern-based approaches use fixed detection patterns that refer to known malicious behavior that are then matched against captured behavior of unknown samples to classify them as likely benign or malign. In contrast, metric-based approaches do not define fixed behavior patterns but rather relate certain QDFG properties, i.e. graph metrics, to known malware behavior. Sets of graph metrics are then used to profile recurrent behavior of known goodware and malware with which we then establish a more flexible notion of behavior similarity.

To investigate the research questions mentioned in Section 1.1.2, we further follow a research methodology where we, for each generic detection concept, first explore the general feasibility and then assess the conceptional and operational boundaries building upon the previously gained insights. For the first feasibility analysis of a detection concept we follow a *deductive* approach in that we manually define detection mechanisms, i.e. patterns or metrics referring to known malicious behavior, and then assess their utility to discriminate malicious from benign samples in a defined evaluation set consisting of known malware and goodware.

To further investigate the general utility and properties of the different concepts we then follow an *inductive* approach where we employ advanced machine learning and data mining techniques to automatically infer more accurate and elaborate detection mechanisms from captured behavior of known malware and goodware.

Afterwards, we discuss how the gained insights and detection approaches can be operationalized in real world detection settings. For this, we assess and compare the approaches according to a set of common malware detection quality criteria, reflect on their suitability for concrete detection purposes, and discuss their operational limitations. Finally, we conclude this thesis with a critical reflection on the level of achievement of the previously defined research goals, discuss gained insights and learned lessons from a more macroscopic perspective, and offer an outlook on envisioned future work in this area. Figure 1.1 gives an overview of the structure of this thesis and the employed research methodology.

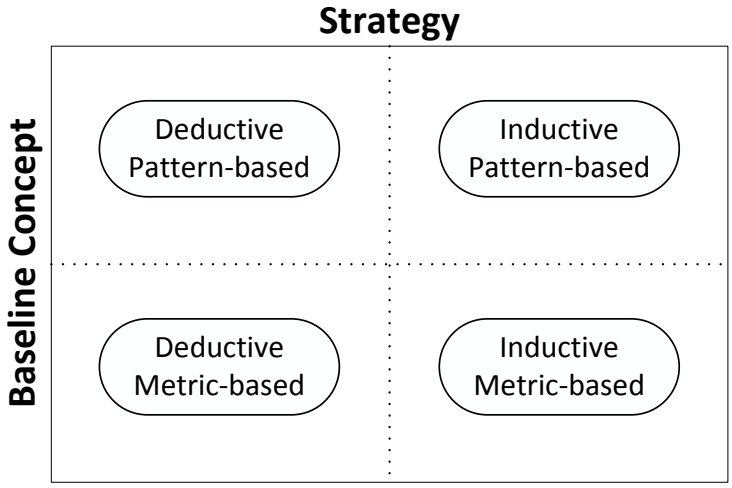


Figure 1.1.: Structure Overview

Some parts of this thesis are based on previous publications co-authored as first author by the writer of this dissertation. Earlier versions of the system model discussed in Chapter 3, which underlies all malware detection approaches presented in this thesis, already have been presented in [144] and partially also in [145].

An earlier version of our basic pattern-based detection approach, described in Section 4.1, has already been published in [144]. However, for this thesis we have extended it with a more elaborate concept to check quantitative properties in detection patterns and present a more comprehensive evaluation. Also, the inductive extension to the basic pattern-based detection concept, described in Section 4.2, is an original contribution of this thesis and is unpublished so far.

The concept of profiling QDFGs with graph metrics, described in Section 5.1, already has been presented in [145]. However, its inductive extension based on genetic programming, discussed in Section 5.2, has not yet been published and thus is an original contribution of this thesis as well.

Finally, an extension of our pattern-based detection concept to visualize and manually analyze potentially malicious activities has been published in [145]. Also, a conceptual continuation of our metric-based detection concept that extends pure detection with risk mitigation means, has already been presented in [143].

Due to unavoidable content overlaps with the author's previous work, quotes from the previously mentioned publications within the respective chapters are not marked explicitly. Instead, in the preface of each chapter we will briefly refer to the author's publications that are relevant for the content of the respective chapter.

1.3. Contributions

To the best of our knowledge, we are the first ones to use quantitative data flow analysis for the behavior-based detection and analysis of malicious software.

This means, we are the first ones to use quantitative data flow information inferred from system call arguments for behavior-based malware detection. In sum, the main contributions of this thesis are:

- A generic model to represent system behavior as quantitative data flow graphs.
- Four distinct detection approaches that operationalize this model for behavior-based malware detection by
  1) manually defining and detecting behavior patterns with quantitative properties that relate to known malicious behavior.
  2) mining quantitative data flow patterns from observed behavior of known malware and goodware samples.
  3) profiling malicious behavior with generic graph metrics on QDFGs.
  4) generating effective detection metrics using genetic programming.
- A data-centric notion of graph compression to measure pattern utility in the context of malware behavior graph mining.
- A machine learning based concept to approximate graph similarity with sets of graph metrics over QDFGs.
- A concept to describe graph metrics as functions over primitive graph properties and a genetic programming scheme to generate metric instances.
- Substantial empirical evidence that quantitative data flow analysis outperforms non-quantitative analysis in terms of detection effectiveness.

2. Background

In this chapter we briefly introduce the history and genealogy of malware, give an overview of the two basic categories of malware detection, and briefly discuss the most prevalent detection models and concepts.

2.1. Malware

Before discussing the core content of this thesis, a novel approach for behavior-based malware detection with quantitative data flow analysis, we need to set the background. As this thesis deals with the detection of malicious software, i.e. malware, we first need to clarify what malware actually is. To this end, we give a brief overview of the history of malware and then discuss a taxonomy, in order to categorize different types of malware.

2.1.1. Malware History

John von Neumann was the first to theoretically postulate the concept of a computer virus in the mid 1960s [140]. In his article, von Neumann discussed the possibility of viral programs, i.e. self-reproducing automata, and thus set the first conceptual baseline for malicious software, long before the first real computer viruses appeared on the scene. Although at this time no technical interpretation was yet conceivable, von Neumann's theories set the baseline for what in the future would be called *computer virus* or more general *malware*.

It took about ten years until the first concrete technical instantiations of von Neumann's abstract concept were developed. The *Creeper* program, developed by Bob Thomas in 1971, is considered the first implementation of von Neumann's concept [33] and could autonomously spread through the internet's predecessor network, the Arpanet. Although *Creeper* was not a malware, i.e. malicious software, in the strict sense (as it was not designed to actually do any harm and even informed the user about its presence), *Creeper* can be considered the technological "father" of all future worms and viruses as it was able to autonomously replicate.

Roughly another decade later, Fred Cohen in his 1984 article [38], and afterwards also in his PhD thesis [39], first coined the term *computer virus* for programs *"[...] that can 'infect' other programs by modifying them to include a possibly evolved copy of itself"* [38].

With a prototypical instantiation of this concept for Unix operating systems, Cohen then developed the first actual malware that was intentionally designed to illegitimately obtain full access rights to a computer system.

This first real malware entailed a so far unbroken series of more or less sophisticated follow-up developments. Chen et al. talk of different waves of malware outbreaks [33]. The first phase, which Chen considers to roughly span from the late 1970s to the early 1990s was mainly dominated by malware that was mostly developed for the sake of curiosity and for experimental purposes. Although supposedly developed with no clear malicious intentions, the first self-spreading internet worm *Morris* then was the first one that actually caused substantial damages and, in the course of a few hours, took down about 10% of all computers that back then were connected to the internet [129]. The Morris worm can thus be considered the first real malware as it was the first program that (although most likely not on purpose) conducted malicious activities.

The second malware wave, that essentially covered the 1990s, was mainly characterized by the upcoming use of polymorphism and encryption by malware to automatically obfuscate malicious binaries with significant effects on the effectiveness of prevalent signature-based anti-virus scanners.

In the third wave, which took place between 1999 and 2001, emails replaced local file-based replication as primary infection and distribution vector. Together with the increasing growth of the internet this yielded previously unseen propagation rates and hundreds of thousands of infections [26]. Popular malware from this era like *Melissa*, *PrettyPark*, or *LoveLetter* also were with the first to include additional functionality to ensure persistent execution, as well as remote system access and to steal sensitive information.

The main characteristic of the fourth malware wave, which roughly lasted from 2001 to 2009, was the rapidly increasing sophistication of malware that started to use multiple vulnerabilities and infection vectors to spread, i.e. through instant messaging or peer-to-peer file sharing applications. Furthermore, malware from the fourth wave started to implement plug-in functionality to dynamically download additional malicious payload and thus adapted to changing malicious goals or new detection mechanisms. Popular malware from this period like *CodeRed*, *Slammer*, or *Nimda* used multiple ways to spread, including replication via email, actively exploiting known remote system vulnerabilities, copying itself to unprotected network shares, or manipulating webservers to drive-by infect visitors.

With *Stuxnet*, which was first detected in 2010 and supposedly developed around 2007 [83] to attack SCADA systems and thus sabotage the Iranian nuclear program, the advent of so-called Advanced Persistent Threats (APTs) began.

We consider this the fifth malware wave which is characterized by targeted, highly sophisticated malware, specifically designed to attack certain victims and typically exceeding the sophistication of commodity malware of previous waves.

Moreover, in contrast to commodity malware, which is primarily developed by amateurs or organized criminals for experimental or economic reasons, APTs like *Stuxnet*, *Duqu*, or *Flame* were likely developed by highly professional development teams, funded by governmental agencies, for espionage or sabotage purposes. APTs typically use multiple attack vectors, unknown 0-day vulnerabilities, and advanced stealth mechanisms to infect specific victims and then prevent detection and analysis at any cost, which makes them particularly hard to detect [137].

Summarizing the development of malicious software from its beginning until today, we can say that there has been a shift in motivation to develop malware. While in the early days malware was mainly developed for sake of curiosity, today's malware landscape is backed by a highly professionalized industry and mainly developed to serve economic purposes [154]. Also, over the years there was a change in infection and propagation models from local replication through file infection to remote propagation via emails or remote exploits [102]. Finally, whereas early malware usually did not conceal its presence, modern malware puts substantial effort into preventing detection, e.g. through passive means like obfuscation, or active countermeasures like rootkits or anti-debugging functionality [15, 152].

2.1.2. Malware Taxonomy

Now that we have given a brief overview of the history of malware, we will discuss the different types of malware in more detail. Besides some loose guidelines [104] and naming schemes [104], there unfortunately there is no commonly accepted malware taxonomy or industry standard and almost all anti-malware vendors have their own schemes to separate different malware categories and families [96, 97].

Typical categories separate malware according to used replication mechanism, malicious functionality, targeted victims, or operational goals. Unfortunately, most modern malware families employ a variety of functionality that often can even get updated or extended at runtime, which further complicates clear categorizations. A clear and unambiguous separation thus often is not possible and the same malware sample can fall into several categories for different anti-virus vendors.

Nevertheless, there is a recurring core of main malware categories proposed by different anti-malware vendors and researchers. Most sources at least agree in the differentiation of commodity malware into *Viruses*, *Worms*, and *Trojans*. Hence, this is what we will focus on in the following and discuss their behavior specificities to allow a better understanding of the later introduced behavior-based detection approaches.

2.1.2.1. Viruses

Viruses are the oldest form of malware [38]. Their main characteristic is that they replicate via file infection. That means, viruses usually only replicate locally and only passively propagate to other systems if removable devices are infected or infected files are accidentally copied to remote destinations. This, for instance, happened with one of the first Microsoft Office macro viruses, named *Concept*, that was accidentally shipped with originally benign Microsoft software installation CDs [33]. Although modern malware and APTs still sometimes use file-based infection vectors for local replication and persistent manifestation, exclusive replication through file infection is rather rare these days.

Viruses usually replicate by either overwriting or patching benign files. Overwriting benign files with the same malicious binaries poses the problem of possibly breaking system functionality, if important system executables are overwritten. Moreover, overwriting legit files with fixed-sized malicious executables makes them easy to detect. Finally, simply replacing legit binaries with the same malicious executable makes it easy to create detection signatures.

An alternative to naive file replacement is the so-called companion strategy where instead of replacing a benign file with a malicious one, the file is copied "next to it" by giving it the same name but a different extension. Exploiting operation-system-specific execution priorities, a user is then lured into executing the malicious binary instead of the benign one.

A more advanced local replication strategy, often adopted by viruses, is patching benign binaries instead of replacing them. This is typically done by adding new code sections to benign binaries to which the malicious code is written to. By bending the code entry point of the binary to first point to the malicious code and only after its execution return to the originally benign content, viruses like *Parite* can preserve system functionality, even if sensitive system binaries are infected.

Patching system binaries also has the advantage that the virus itself does not need to be manually executed in order to perform its malicious actions, but is automatically executed if the patched executable is invoked. More sophisticated patching strategies even manipulate other pointers than the file entry point to refer to malicious payload, injected somewhere into the legit binary. This, in conjunction with binary obfuscation techniques employed on the patched payload, makes it hard to craft generic detection signatures, especially as many signature-based scanners for performance reasons only scan the beginning and end of a file. Even though file-based replication strategies were mainly superseded by network-based propagation, they experienced a renaissance in APTs like *Stuxnet* [83].

From a system call behavior perspective, viruses can be recognized by the fact that they often yield unusually high amounts of file access and write activity to already exiting known benign binaries.

2.1.2.2. Worms

With the advent of the internet and increasing numbers of computers that were connected via network, local replication techniques employed by viruses were increasingly replaced by network-based replication mechanisms. While the propagation rates of viruses were comparably low, as they relied on passive propagation models through external storage devices, infection rates with the first internet-based worms skyrocketed to hundreds of thousands of new infections within hours. The *Sasser* and *Blaster* internet worms [17] in the early 2000s, for instances, replicated autonomously by exploiting unknown vulnerabilities in Microsoft Windows RPC services. As most computers where directly connected to the internet via modems back then, such worms could spread almost uncontrolled and reached propagation and infection of previously unseen dimension.

Although occasionally still employed by more current internet worms such as *Conficker*, which exploits a known vulnerability in the Windows NetBIOS service, such active replication via exploitation of operating system vulnerabilities has become rare in the past years. This is likely due to a loss in effectiveness of such attack vectors, as most home computers these days are no longer directly connected to the internet, but to some extent shielded by home gateway routers. Instead, more recent worms such as *Stration* or *Waledac* [67] rely on propagation via email or drive-by-download infection through hijacked webservers [102].

From a system call perspective, worms, especially those that autonomously replicate through active exploitation of remote vulnerabilities, often issue unusual high amounts of network-related system calls to the same remote ports.

2.1.2.3. Trojans

A more fuzzy malware category are so-called trojan horses, i.e. trojans, often also referred to as backdoors or remote administration tools (RATs). The main commonality of malware that falls into this category is that they usually do not employ active autonomous self-replication means but rather disguise themselves as legit software in order to lure users into executing them. The mainly passive distribution vectors of trojans thus are drive-by-downloads and email attachments.

Most trojans have further functionality to allow remote access to a compromised system. Often, trojans are only the first stage of multi-stage malware infections and e.g. reload additional malicious payload and instructions from command-and-control servers after initial infection. Furthermore, trojans typically implement means to evade detection and disguise their presence, e.g. through rootkits. The *ZeroAccess* trojan, for instance, almost exclusively propagates passively though drive-by downloads and, directly after infection, installs a kernel level rootkit to hide its presence from the user and all user mode processes [148].

A special form of trojans are so-called spyware or information stealers that are specifically designed to steal sensitive data from infected victims, such as banking credentials, credit card numbers, or login credentials. The very popular and widespread *Zeus* [12] trojan construction kit, for instance, was specifically designed for criminal purposes, i.e. for large-scale online banking fraud. Despite less aggressively spreading via direct attacks on vulnerabilities like the early internet worms, trojans like *Zeus* make up for most of today's malware infections [112].

A fairly new trojan variant is so-called ransomware, sometimes also referred to as crypto viruses. In contrast to most other trojans that focus on establishing remote access or stealing sensitive data, ransomware, as the name suggests, tries to force users into paying a ransom to regain access to files that is intentionally prohibited by the ransomware. This is usually done by the ransomware encrypting potentially sensitive files and demanding the payment of a ransom to obtain the decryption key and thus regain access to the files, or by locking the user out of the system and demanding a ransom payment to release the lock.

The famous *CryptoLocker* ransomware, for instance, infects victims through drive-by downloads or comes as additional payload of the *Zeus* trojan. After infection, it searches the victim computer for potentially sensitive files, such as office documents or pictures. *CryptoLocker* then encrypts found sensitive files with a strong asymmetric crypto-scheme to render the files unusable, unless a private key to decrypt them is retrieved through paying a ransom to the malware developers. Other non-cryptographic ransomware like the *Reveton* trojan, instead of encrypting sensitive files, simply block access to infected computers, which is again regained once a ransom has been paid by the user [107].

While trojans, due to their behavioral heterogeneity and lack of autonomous replication functionality, are considered harder to detect than viruses and worms, some behavior specificities nevertheless often reveal their presence. Besides more abstract signs of a trojan infections like overall increased CPU utilization and slower responsiveness, trojans can often be detected through characteristic system call behavior. This can include the manipulation of certain autorun registry keys, connections to remote servers, followed by the download and execution of executable binaries, or unusually high interactivity with sensitive files and related high network interactivity.

2.1.2.4. Advanced Persistent Threats

So-called Advanced Persistent Threats (APTs) are not a malware class of their own in the strict sense, as they typically incorporate functionality and behavioral specificities of other malware classes. Nevertheless, APTs are particularly interesting from a malware detection perspective, because they are specifically designed to infect particular victims, unlike commodity malware that usually spreads rather indiscriminately as wide as possible. This is, while the utility of most commodity malware, e.g. their achieved revenue, is directly bound to the number of infected victims, APTs such as *Stuxnet*, *Duqu*, or *Flame* [137] are typically developed for victim-specific espionage or sabotage purposes and try to avoid collateral damage, as this increases the risk of getting detected and analyzed.

While for commodity malware preventing detection thus only plays a subordinate role and devised anti-detection countermeasures remain rather simplistic, APTs typically put significant effort into preventing detection and e.g. even delete themselves upon inadvertent infection of victims that do not fall into the targeted group. Furthermore, APTs usually employ a wide range of attack and propagation vectors, often including multiple infection stages and using 0-day exploits, whereas untargeted commodity malware in comparison implement rather simplistic distribution models.

From a behavioral perspective, the typical multi-stage and multi-vector distribution models of APTs make their detection extremely difficulat. Moreover, advanced kernel-level root-kit functionality, commonly employed by APTs [133, 137], effectively disguises malicious activities and thus renders post-infection detection particularly hard. In most cases, detection thus needs to happen during the first infection stages, when the system is not yet under full control of the APT.

2.2. Malware Detection

Having introduced the most common malware types and their behavioral specificities, in this section we will discuss how malware detection industry and research tries to cope with the malware threat. As we will see, the increasing sophistication of malware over the past decades stimulated an ongoing arms-race between developers of malware and anti-malware software.

Malware detection approaches can be categorized along various dimensions, like for instance targeted operating system (e.g. Windows or Android), basic detection model (static vs. dynamic), classification paradigm (rule-based vs. statistics-based), or scope (e.g. file, process, or system).

Like others [4, 62, 46] we consider malware detection approaches to be either static or dynamic.

Hence, we will first give an overview of static approaches as they are still predominantly used in commercial anti-virus products. The focus of our overview of different malware detection approaches, however, is on dynamic behavior-based approaches, to which category the approaches proposed in this thesis also belong.

Note that in this section we only give a broad overview on the state of the art in malware detection in order to put our work into context, only mentioning the in our opinion main contributions to the field. A thorough discussion of related work will then follow for each devised detection approach in Chapter 4 and in Chapter 5. Moreover, as there are few reliable resources available that in-detail elaborate on the functionality of commercial anti-malware products, the following overview emphasizes academic work in this area.

2.2.1. Static Detection

Static detection is the oldest way of detecting malicious software, with the first approaches dating back to the late 1980s [134]. In contrast to dynamic detection approaches that need to actually execute suspicious samples to classify them, static detection approaches classify malware samples based solely on their persistent representation, e.g. Windows PE binary files or other file formats that can contain malicious code. This distinction has recently become somewhat blurry as modern static anti-virus engines often employ memory scanning and code emulation functionality and thus, in a sense, also "virtually" execute a sample before classifying it. Nevertheless, scanning suspicious binaries for patterns that refer to known malware remains the core functionality of static analysis approaches.

Again, there is no commonly accepted static malware detection taxonomy and different sources categorize them according to various characteristics, including applied scanning technique (pattern-, or misuse- vs. anomaly-base), operation mode (on-access vs. on-demand), evaluation strategy (lazy vs. eager), or abstraction level (e.g. byte code, control flow, or function call graphs) [62, 46, 134].

We consider these abstraction level the most important and least overlapping characteristic and thus differentiate between three major pillars of static malware detection approaches: the *string search based* category covers the majority of commercially applied detection approaches which operate on the raw binaries without any further abstraction or interpretation; the *semantics based* category is mainly academic and covers approaches that abstract from the concrete binary representation of malware by inferring their semantics, e.g. on basis of abstracted control flow or function call graphs which then are used as generic detection signatures.

In the past decades, a plethora of static detection approaches has been proposed. In the following we will focus our overview on the mentioned main categories that cover the bulk of commercial and academic static detection approaches and mention only the seminal or most characteristic works for each category explicitly.

2.2.1.1. String search based

Early static malware detection approaches were simplistic programs designed to detect the presence of one specific type of malware. The *Creeper* virus, for instance, had the *Reaper* program as counterpart that itself was a self-replicating worm, but instead of performing malicious activities, was solely designed to remove all found *Creeper* instances [33]. The first actual anti-malware tools that were able to detect (and often also remove) different malware types emerged with the advent of the first non-experimental computer viruses for personal computers in the late 1980s and early 1990s [134].

To describe the ensuing development, Peter Szor [134] differentiates between different generations of (commercial) static malware detection approaches.

First Generation The approaches of the first generation of static malware detection all relied on the same basic principle, the so-called *string scanning*. This essentially means that anti-virus software scanned potentially malicious binaries for byte sequences that characteristically related to known malware. To detect an instance of the *Stoned* virus it e.g. was sufficient to scan binaries for the malicious byte code snippet that implemented the boot sector infection functionality of the virus. Such byte sequences were usually distinct enough to prevent false positive matches but generic enough to also capture slight variations of the malware byte code. Therefore, even such a simple scanning paradigm, although demanding substantial manual analysis effort to extract characteristic byte signatures, was sufficient to yield high detection effectiveness and precision for non-obfuscated prevalent viruses at that time.

In order to boost detection efficiency and to avoid having to scan entire files for matching byte signatures, this scheme was later improved by e.g. only scanning the beginning and end of binary files (top+tail scanning), or computing hash functions on individual parts of known malicious binaries and then re-identifying parts in unknown binary samples that had the same hash value.

Although this brough significant efficiency improvements, such lazier scanning approaches led to increased sensitivity towards changes in malware binaries or malicious code not being appended or prepended but injected at pseudo-random places within the infected binary. By simply adding bogus NOP instructions, malware developers could, for instance, effectively break fixed-sequence byte patterns and prevent hash-based matching.

To further improve genericity of detection signatures, fixed byte sequence patterns were thus soon replaced with more flexible patterns that included wildcards and more complex byte grammars in order to also detect updated or slightly modified versions of malicious binaries that did not exactly match a detection signature.

Second Generation With the increasing appearance of malware that used simple obfuscation techniques to alter their binary representation, e.g. by inserting useless NOP sequences or bogus arithmetic operations, the effectiveness of simple fixed byte pattern scanning approaches steadily decreased. To counteract this, static malware detection approaches of the second generation implemented more sophisticated fuzzy byte pattern matching strategies and data normalization. One commonly used technique, for instance, was the removal of useless NOP sequences or other stop-words and the recognition of bogus operations.

Instead of demanding exact matches of byte patterns, more elaborate approaches allowed a certain deviation from theses patterns, using different edit distance metrics to assess the closest similarity of a suspicious binary to a set of known malware byte patterns. This raised detection robustness, at least against simplistic obfuscation techniques, though it brought the cost of risk of increased false positive classifications.

Current Generation While the simple fuzzy matching and stop-word removal techniques of the second generation malware detection approaches indeed increased detection robustness against simple code alternation, the advent of advanced obfuscation techniques put static detection into a severe crisis.

The increasing prevalence of polymorphic and metamorphic malware that used encryption and packing chains to completely alter their binary representation with each replication iteration rendered fixed byte patterns essentially useless.

Current generation static analysis approaches cater to this issue in multiple ways. As many new malware families wrongly implement or use cryptographic algorithms to alter their binaries or use comparably easy to break custom XOR-based encryption schemes, modern anti-virus scanners implement generic decryption routines that e.g. try to automatically infer used encryption keys or brute-force simple keys to circumvent the effects of encryption on the binary structure. Furthermore, many current generation approaches feature advanced emulation and simulation mechanisms in order to "virtually execute" suspicious samples to obtain the unobfuscated malicious payload. However, metamorphic malware that uses self-modifying crypto schemes [109] still significantly challenges modern static detection approaches.

Furthermore, current generation anti-virus scanners, in addition to file-based detection, often also monitor the memory of suspicious processes for decrypted and unpacked known malicious code. As malware, at some point, always needs to decrypt and unpack the payload to be able to execute it, this is a viable strategy to counteract encryption and packing. Unfortunately, in-memory scanning can only be done when malware has already been executed and thus can no longer a-priori prevent infections but rather only ex-post detect them.

Finally, current generation static anti-malware approaches often include malware-specific detection heuristics in addition to family-specific detection signatures. These heuristics scan a binary for code most likely related to malicious functionality, such as encryption or packing chains, or characteristic replication or infection functionality, and thus indicates malicious intents.

Besides these consecutive improvements of the basic byte pattern string scanning strategy, there have also been some, mainly academic, proposals for more advanced static byte level detection concepts. These range from proposals to use entropy analysis on binaries to detect packed and encrypted content that likely indicates malicious intents [19, 90], concepts to use data mining over n-gram-tokenized binaries [127, 123, 99], or function references to create fuzzy classification models [87, 124], to approaches that statically detect malware based on embedded binary resources like strings or pictures [131, 88].

2.2.1.2. Semantics based

The fact that commercial byte sequence pattern-based anti-virus engines continuously lost in effectiveness in the past decade due to increasing sophistication of anti-detection countermeasures [52, 136, 35] stimulated research in more advanced and sophisticated static detection concepts that do not solely rely on syntactic identification or recurrent byte sequences.

One common alternative to string scanning based detection is the profiling of known malicious binaries by means of more abstract structural aspects like *control flow* or *(function) call graphs*. As call graphs and control flow graphs both describe the structure of a program in more abstract and generic terms that is to some extent independent of concrete binary representations, they are considered more robust toward accidental or intentional variations of malware binaries [14]. This means, it is considered *"harder for an attacker to radically change the behavior of a malware than to morph its syntactic structure"* [54].

To leverage control flow or function call graphs for static malware detection, most approaches extract the respective graphs of varying precision from sets of known malware and e.g. use data mining primitives to infer recurring and thus characteristic graph patterns [29, 21, 49, 74]. As control flow and call graphs are typically less influenced by injection of NOP sequences or bogus arithmetic computations, they are more capable of coping with simple binary obfuscation. In a sense, using more abstract and topological call or control flow graphs instead of less topological flat byte sequences for detection, can be considered moving from purely *form-focused* towards a more *semantics-focused* detection paradigm [65].

Mihai Christodorescu was among the first to use formal program analysis and semantic specifications on malware control flow or call graphs for malware detection purposes, and thus coined the term *semantic-based malware detection* [37].

This to a degree extends the idea of malware-specific detection heuristics, but instead of searching binaries for byte sequences that likely relate to generic malicious behavior, semantics-based approaches infer specifications, i.e. program semantics of known malicious binaries through manual specification, abstract interpretation [116], or data mining [54]. Similar to program verification and (hardware) model checking, these malware specifications, often referred to as *Malspecs* [36], can then be matched against unknown binaries to classify them as malicious, iff they satisfy one or more of the manually defined or automatically inferred specifications [75, 36].

Such semantics-based approaches have been shown to significantly outperform state of the art commercial string search anti-virus detection engines in terms of general detection effectiveness and robustness towards simple obfuscation transformations like NOP or bogus operation injection, at the cost of up to ten times lower classification efficiency [37].

2.2.2. Dynamic Detection

In comparison to the field of static malware detection, dynamic malware detection is much younger and less intensively explored, with the first approaches dating back to the late 1990s and early 2000s [62].

Unlike static detection approaches that classify malware only according to their binary representation, the classification decision of dynamic detection approaches, or more specifically, of dynamic behavior-based detection approaches, is based on the observable behavior of malware. This means, static approaches have a *representation-focused* scope, whereas dynamic approaches have a *behavior-focused* scope. Although malware behavior in principle can also be determined statically, for the sake of brevity we in the following use the term *behavior-based* whenever we refer to *dynamic behavior-based detection* concepts.

In contrast to static approaches that can classify malware samples without actually executing them, dynamic malware detection approaches always require samples to be executed and to issue some observable behavior in order to classify them. This is usually done by executing suspicious samples in controlled and isolated sandbox execution environments in order to establish a stable analysis baseline and to prevent analyzed malware from spreading and attacking sensitive assets. Although there exist a couple of commercial sandbox-based dynamic malware systems like LastLine Analyst [63], VMRay Analyzer [139], or the VxStream-Sandbox [126], dynamic analysis is far less common in commercial anti-malware products than static analysis and still mainly prevalent in academic approaches.

The typical behavior-based detection work-flow is to execute a suspicious sample within an analysis sandbox, to monitor and record the behavior of the corresponding process(es), and then revert the sandbox to a clean system state.

Monitored behavior in this context usually means traces of system or API calls whose execution is typically monitored by reference monitors inside [110, 8, 142] or outside the virtual machines of the sandbox environment [45, 69, 44, 103], or even through dedicated non-virtualized bare-metal analysis systems [77, 78].

Unfortunately, this also causes a certain limitation of behavior-based approaches in that they can only detect active malware. Malware that for some reason is not executable within the analysis environment or actively defers behavior can, without additional stimulation measures, not be detected by behavior-based approaches. Nevertheless, behavior-based detection approaches are more robust towards classical static binary obfuscation, as reordering, substitution, or insertion of bogus instructions typically has little impact on the induced system call level behavior [46].

2.2.2.1. System Call Sequence based

Dynamic behavior-based malware detection is very similar to host-based intrusion detection. This is, both concepts typically aim to identify malicious activities based on observed process and system behavior, usually captured in form of system call traces. The main difference between the concepts is rather subtle in that intrusion detection systems mainly focus on detecting motivated attacks, whereas behavior-based malware detection systems are designed to classify suspicious programs. From a technological perspective, however, they usually leverage similar data processing and pattern inference concepts to discriminate benign from likely malicious behavior, with the main difference of behavior-based malware detection approaches being mostly sandbox-based, whereas host-level intrusion detection is usually performed in operational system settings.

This blurry line between the two related concepts makes it hard to clearly determine the origins of the first behavior-based detection concepts. There nevertheless is a certain agreement that Forrest et al. were among the first to bring the notion of *self* as baseline concept for discriminating malicious from benign behavior [1, 62] into discussion. The core idea behind Forrest's early work [51] was loosely based on basic functional principles of biological immunology as it introduced a *sense of self* to operating systems by monitoring normal behavior in terms of syscall traces issued by benign programs. Deviations from such profiles of normal behavior, i.e. traces that are not similar to known benign ones, were then considered malicious.

Although this first anomaly-detection concept was primarily designed for intrusion detection purposes and, due to the relatively basic trace comparison techniques, was fairly imprecise, it still can be seen as the conceptual "father" of all subsequent behavior-based malware detection approaches, as it was one of the first to propose the detection of malicious activities through system call profiling.

The first dedicated behavior-based malware detection concepts then in essence transferred the basic concepts from static malware detection, i.e. string [118] and n-gram-based [119] search, to a system call behavior context.

Although these syscall sequence focused approaches led to increased robustness towards static binary-level obfuscation, recent studies revealed their sensitivity towards certain behavior obfuscation techniques [116, 115, 6]. Hence, dynamic system call sequence based approaches share similar limitations as static byte sequence-based approaches in that the intentional permutation of system call traces, i.e. through injection of bogus calls, system call reordering, or semantic substitution, tremendously decreases detection effectiveness [116, 115, 6]. Moreover, similar to non-semantics-based static detection approaches, system call sequence-based detection approaches lack an obvious semantic dimension, which prevents manual validation of classification results and further-reaching malware analysis.

2.2.2.2. System Call Dependency based

System call dependency based dynamic detection approaches are an evolution of pure system call sequence-based detection approaches and explicitly consider the semantic relation between different system calls for classification. This is, while pure sequence-based approaches usually capture system call dependencies in inferred detection patterns only implicitly, system call dependency based approaches explicitly model semantic dependencies between system calls.

Semantic dependencies between system calls exist whenever they depend on each other output- or parameter-wise, i.e. if results from one system call invocation determine arguments of a later system call. If a process, for instance, issues an OpenFile syscall to obtain a handle to a to-be-read file and then reads data from this file with a ReadFile call, then there exists a semantic dependency between the two calls. This is, because for reading from the opened file, the file handle needs to get passed as argument to the ReadFile system call. Such semantic dependencies between system calls are either approximated through argument and parameter correlation [80, 5], or through precise, but expensive taint tracking [151, 3].

Kolbitsch et al. were then among the first to model semantic dependencies between syscalls as *system call dependency graphs*, representing system calls as graph nodes and semantic dependencies among them as edges [80]. By matching syscall traces of unknown samples with dependency graphs of known malware it was possible to detect variants of known malware families with high effectiveness.

However, this comparably naive matching approach yielded relatively low effectiveness on samples from unknown malware families.

The consideration of dependencies between system calls then allowed for filtering certain non-discriminative information and keeping behavior models, i.e. characteristic behavior graphs, lean, with positive effects on detection precision.

Follow-up work further improved this concept through more elaborate and robust sub-graph isomorphism matching [74], by using graph-mining algorithms to yield more precise and characteristic behavior graphs [50, 49, 32, 54], or through automata inference to improve behavior model precision [3].

2.2.2.3. Resource Dependency based

While the consideration of system call dependencies led to a further improvement of detection effectiveness, system call dependency based approaches were still to some extent sensitive to changes in system call traces like bogus call insertions that can cause changes in the resulting system call dependency graphs.

To address the limitation of syscall-centric approaches of being sensitive towards variability in system call traces, Bailey et al. [4] proposed to focus classification on *consequences* of malicious behavior rather than on the *form* of the behavior itself. This is, they proposed a *resource-centric* view for reasoning about and defining characteristic malware behavior instead of a *system call-centric* view.

The benefits of this new behavior profiling paradigm are intuitively compelling: while it has been shown that there are multiple ways to implement malicious functionality, which can be actively exploited by malware to obfuscate their behavior [116, 115, 6], there are only limited possibilities to alter the set of system resources that need to be accessed in order to achieve a specific malicious goal [4]. Bailey further emphasized the utility of resource-centric behavior patterns for further analysis in that they *"are more invariant and directly useful than abstract code sequences representing programmatic behavior and can be directly used in assessing the potential damage incurred, enabling detection and classification of new threats, and assisting in the risk assessment of these threats in mitigation and clean up"* [4].

The first dynamic resource-centric approaches clustered similar groups of resources, e.g. files, processes, or registry keys, accessed by known malware samples into resource-centric behavior *fingerprints* [4]. Unknown samples were then classified through the distance of their resource fingerprints to those of known malware. Follow-up extensions then e.g. used n-gram analysis on accessed resources to obtain more precise behavior models [84].

Finally, the most elaborate concepts from this pillar of dynamic malware detection approaches use precise but expensive taint-training [151] or less-precise but considerably faster argument-correlation [114] to express more complex multi-step resource dependencies in form of *resource dependency graphs*, or *(resource-centric) data flow graphs*. This way, such approaches also cater to the issue of modern malware distributing malicious behavior among multiple processes by spawning clone processes or hijacking benign processes to confuse process-centric detectors [117].

2.2.3. Gap Analysis and Assessment

Considering published studies on the sensitivity of static detection approaches [98] and dynamic syscall-centric detection approaches [115, 6] against various obfuscation transformations, we agree with Bailey et. al. [4] in that dynamic resource-centric detection approaches at the moment are the best weapon against obfuscated polymorphic and metamorphic malware. This has to some extent also been acknowledged by anti-malware industry, which recently started to move from purely *preventive* towards more *detective and responsive* mechanisms [136].

Nevertheless, current resource-centric approaches still suffer from some issues concerning detection *precision*, obfuscation *robustness*, and classification *efficiency*.

In this thesis we thus aim to improve this situation by proposing a new behavior model based on approximated *quantitative* data flows between system resources and several approaches that leverage this model for precise, robust, and efficient malware detection. This is, current resource-centric detection solutions often yield false positive classifications that are too high for practical purposes, are sensitive against certain behavior obfuscation techniques, and, especially when based on full system emulation-based taint-tracking, induce high computational overheads. In sum, these limitations currently hinder the operationalization of resource-based malware detection concepts.

Our proposed system model and detection approaches belong to the resource-focused pillar of dynamic malware detection in that we also model dependencies, i.e. potential data flows between system entities. However, in contrast to the aforementioned approaches we do so from a *quantitative* rather than an *existential* flow perspective. While taint-based analysis, as discussed before, allows the precise tracking of data flows at the cost of induced considerable computational overheads, bad portability, and thus suffers of limited opertionalizability, system call based data flow approximation is computationally spoken lean and thus comparably easy to deploy in operational settings, but rather imprecise. Both concepts thus suffer from limitations that hinder their applicability in real-world operational settings, either because of complicated deployment and bad efficiency, or because of limited precision.

We expect our concept of system call based data flow approximation *with consideration of quantitative data flow aspects* to yield a good tradeoff between the precision and efficiency which contributes towards the real-world applicability of data flow analysis based malware detection approaches.

To this end we will continuously assess this assumption throughout the course of this thesis and investigate, whether or not our *quantitative perspective* indeed positively influences detection effectiveness, efficiency, and robustness.

3. System Model

In this chapter we propose a model to represent the interaction between different system entities as quantitative data flows that together constitute a quantitative data flow graph. An earlier version of this model was presented in [144], co-authored as first author by the writer of this thesis. We then sketch how to leverage this model for behavior-based malware detection.

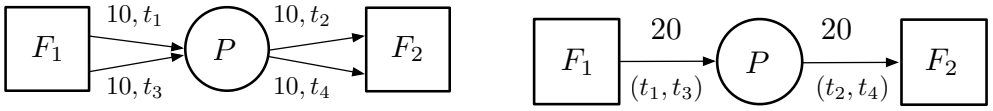
Behavior-based malware detection approaches all have in common that they utilize different forms of abstraction of the observed behavior of to be analyzed samples. As mentioned in Section 2.2, such models and abstraction concepts range from rather simplistic behavior representations in form of raw system call sequences to more sophisticated graph-based concepts like system call dependency or data flow graphs that encode more complex behavior inter-dependencies.

As approaches that directly leverage raw system calls for detection have shown to be sensitive to malware behavior obfuscation [6, 115], we propose a more abstract representation of system behavior, i.e. captured system call traces, as quantitative data flows. While representing behavior as graphs in itself is conceptually not new, we put particular emphasis on incorporating *quantitative data flow aspects* into our graph-based models. This, as we will later show, has a significant impact on detection accuracy and robustness.

The main reason to embody this quantitative component into our model is the intuition that the more information a model captures about the underlying behavior, the better it can be used for discrimination, i.e. malware detection purposes.

In this section we hence present the principal idea of modeling low-level system events, e.g. system calls, as quantitative data flows. The resulting model will then be the conceptual basis of all malware detection and analysis approaches presented in the remainder of this thesis.

To do so, we proceed in two steps. First we introduce a generic model that can be instantiated for various system types, and that provides a lean data structure suitable for quantitative data flows analysis. Then, we discuss how this model can be instantiated for a concrete operational context, which in our case is that of a typical Microsoft Windows operating system environment.



(a) Individual flows (b) Aggregated flows

Figure 3.1.: Data flow abstraction from single system events.

Our core idea is to perform malware detection based on the analysis of system-wide quantitative data flows. System-wide data flows in this context refer to all flows of data between different system entities that happened within a specific time frame e.g. as consequence of the execution of system calls. *System entities* denote here all conceptual sources and sinks of data. In the case of operating systems, this includes resources like processes, sockets, or files. Data flows between system entities are caused by the execution of specific data flow related *events*. An example are file system events that, if called by a process, lead to a (quantifiable) flow of data between the calling process and the involved file.

3.1. Quantitative Data Flow Graphs

Capturing system behavior as sequences of data flows allows us to reason about behavior from a more abstracted data flow perspective. However, traces of data flows are not very handy to process and not ideally suited for being interpreted by human analysts. In addition to that, data flow traces lack a clear topological dimension which complicates more complex analysis tasks.

As next step we hence show how sequences of data flows can be aggregated in form of what we call *Quantitative Data Flow Graphs* (*QDFGs*). The nodes of these graphs represent system resources that were at least once involved in a data flow, whereas the edges represent the quantified data flows between these entities.

To better understand how QDFGs model system behavior, consider the following simple process behavior example. Suppose process P reads 10 Bytes from file F_1, then writes them to file F_2, and it repeats this process twice. An accurate log of this events would keep track of all individual system event and their timestamps, as depicted in Figure 3.1a. As a built-in optimization, we aggregate these flows into single weighted flows, as depicted in Figure 3.1b, together with a record of the timestamps of the contributing system events.

QDFGs are then modeled as elements of the set $\mathcal{G} = \overline{N} \times \overline{E} \times \overline{A} \times ((\overline{N} \cup \overline{E}) \times \overline{A} \rightarrow \mathit{Value}^{\overline{A}})$ for a set of nodes, $\overline{N}$, a set of edges, $\overline{E} \subseteq \overline{N} \times \overline{N}$, a set of attribute names, $\overline{A}$, and a set of labeling functions drawn from $((\overline{N} \cup \overline{E}) \times \overline{A}) \rightarrow \mathit{Value}^{\overline{A}}$ that map an attribute $a \in \overline{A}$ of a node or an edge to a value drawn from set $\mathit{Value}^{\overline{A}}$.

In a QDFG $G = (N, E, A, \lambda) \in \mathcal{G}$, nodes N represent data flow related system entities and edges E data flows between them. Attributes A are needed to keep our model flexible enough to be instantiated for various types of systems. They represent characteristics of data flows and system entities that are important for malware detection and analysis. Edges, for instance, have an attribute $size$ that represents the amount of transferred data of the respective flow. The labeling function λ retrieves the value of an attribute assigned to a node or an edge, in this example the size of the flow that corresponds to an edge.

QDFGs are intuitively to be read as follows: If there has been a flow of data in-between the system entities corresponding to two nodes, there is an edge between these nodes. Data flows are caused by system events. Among other things, this is the case if a process reads from a file; writes to a registry; or writes to a socket. At the level of a QDFG, we are not interested precisely which event has caused the flow. Instead, we model events $(src, dst, size, t, \lambda) \in \mathcal{E}$ as tuples where $src \in \overline{N}$ is the originating system entity, $dst \in \overline{N}$ the destination system entity, $size \in \mathbb{N}$ is the amount of transferred data, $t \in Value^{time}$ is a time-stamp (defined below) and λ holds other attributes of the involved entities. As mentioned above, a runtime monitor will then observe all events at the operating system level, and use the characterization of an event by set $\mathcal{E}$ as a data transfer to update a QDFG. QDFGs hence evolve over time as captured events either cause the creation or the update of nodes or edges in a data flow graph.

In order to define how exactly the execution of an event modifies a QDFG, we need some notation for updating attribute assignments. For any node/edge, attribute pair $(x, a) \in (N \cup E) \times \overline{A}$, we define $\lambda[(x, a) \leftarrow v] = \lambda'$ with:

$$\lambda'(y) = \begin{cases} v & \text{if } y \in dom(\lambda) \\ \lambda(y) & otherwise \end{cases}$$

We further agree on some syntactic sugar to represent multiple synchronous updates:

$$\lambda[(x_1, a_1) \leftarrow v_1; \ldots; (x_k, a_k) \leftarrow v_k] = (\ldots(\lambda[(x_1, a_1) \leftarrow v_1])\ldots)[(x_k, a_k) \leftarrow v_n].$$

The composition of two labeling functions is defined as:

$$\lambda_1 \circ \lambda_2 = \lambda_1[(x_1, a_1) \leftarrow v_1; \ldots; (x_k, a_k) \leftarrow v_n]$$

where $v_i = \lambda_2(x_i, a_i)$ and $(x_i, a_i) \in \mathsf{dom}(\lambda_2)$.

We consider the aggregation of flows and system entities to be one distinct feature of our approach. Such aggregations are needed to keep the resulting graphs within reasonable limits as illustrated in Figure 3.1.

This in particular means that entities with the same name (e.g. multiple running instances of the same program) are represented by the same node.

Furthermore, flows between the same pair of system entities are represented by one edge where we simply sum the resulting transferred amount of data rather than creating two different edges.

In order to aggregate flows caused by similar events between the same entities edges get assigned a set of time-stamps $time \in \overline{A}$ such that $Value^{time} \subseteq 2^{\mathbb{N}}$. With each event that contributes to a flow we thus add this event's time-stamp to the time-stamp list of the respective edge. An example for this, depicted in Figure 3.1b, would be two subsequent *WriteFile* calls between the same process-file pair, one at time step 1 and the other at time step 3, where the resulting edge would then be assigned the set of time-stamps $\{1, 3\}$.

Now we can precisely define how an event updates a QDFG by triggering the graph update function $update : \mathcal{G} \times \mathcal{E} \to \mathcal{G}$, formally specified by:

$$
\begin{aligned}
&update(G, (src, dst, s, t, \lambda')) = \\
&\begin{cases}
\begin{pmatrix} N, \\ E, \\ A \cup \mathsf{dom}(\lambda'), \\ \lambda \begin{bmatrix} (e, size) \leftarrow \lambda(e, size) + s; \\ (e, time) \leftarrow (\lambda(e, time) \cup \{t\}) \end{bmatrix} \circ \lambda' \end{pmatrix} & \text{if} \quad e \in E \\
\begin{pmatrix} N \cup \{src, dst\}, \\ E \cup \{e\}, \\ A \cup \mathsf{dom}(\lambda'), \\ \lambda \begin{bmatrix} (e, size) \leftarrow s; \\ (e, time) \leftarrow \{t\} \end{bmatrix} \circ \lambda' \end{pmatrix} & \text{otherwise}
\end{cases} \\
&\text{where } e = (src, dst) \text{ and } G = (N, E, A, \lambda)
\end{aligned}
$$

3.2. Model Instantiation

To operationalize the concept of malware detection with quantitative data flows for a concrete system context we now only need to instantiate our generic QDFG-based system model. In general this implies mapping the abstract entities, i.e. nodes, of the model to concrete system entities in the targeted operational context. Furthermore, an operationalization of our abstract model requires the instantiation of the generic data flow event set $\mathcal{E}$ to concrete system events.

Although we also successfully instantiated our model in an Android mobile operating system context and showed its utility for the detection of malicious Android apps, this thesis exclusively focuses on the detection of Windows malware. We therefore in the following discuss the instantiation of our generic system model to standard Windows operating systems. For details on the Android instantiation we refer the reader to the respective literature [27].

To operationalize the generic QDFG model to a Windows malware detection context we need to map abstract QDFG nodes to concrete Windows resources like processes, files, sockets, or registry entries. We furthermore need to relate Windows API calls that carry an intuitive data flow semantic to respective data flow events. This Windows instantiation of our generic model will be discussed in the following.

3.2.0.1. System Entities

Through a preliminary study on data flows induced by interactions of different processes within Windows operating systems, we identified a set of system resources that can be considered as sources or sinks of data flows, and thus as entities in our generic model. We use the attribute $type \in \overline{A}$ to describe an entity's type. For the sake of brevity we later will use its first letter to refer to a specific entity type, instead of always using its full name, i.e. P instead of *Process*:

- **Processes** interact with the file system, registry, and sockets and cause data flows from or to these entities. Nodes that represent processes are assigned *(P)rocess* as value of the *type* attribute.
- **Files** persistently store data and can be either read or written to by processes. Nodes that represent files are always implicitly assigned *(F)ile* as value of the *type* attribute.
- **Sockets** are connection points to remote systems. Processes can either read or write data from them, causing data flows from or to the respective remote systems. Socket nodes are always assigned *(S)ocket* as value of the *type* attribute.

- **URLs** represent remote (potentially virtual) systems that a process interacts with through a socket. URLs in essence are our way of modeling data flows between local processes and resources on remote systems. URL nodes are assigned *(U)rl* as value of the *type* attribute.
- **Registry Keys** persistently store settings and other Windows configuration data and can be either read or written to by processes. Nodes that represent registry keys are assigned *(R)egistry* as value of the *type* attribute.

3.2.0.2. System Events

To model interactions that are relevant from a data flow perspective, i.e. have an obvious data flow semantics, we model several Windows API functions by abstract events. These will be used in the update function of Section 3.1, and thus by the runtime monitor, to build the QDFG. Due to the considerable complexity of the Windows API we concentrate on a subset of the Windows API that is commonly used by malware to interact with system resources. For brevity's sake we sometimes simplified some aspects of these functions like parameter types (e.g. use file names instead of file handlers) or enriched them with additional information that is usually not directly given by the function's parameters.

In the following we only describe one representative of each type of events which model all semantically (in terms of induced data flows) equivalent events; e.g. the WriteFile function that represents all Windows API functions (e.g. WriteFileEx, WritefileGather, ...) that induce a data flow from a process to a file. In the actual implementation of our approach we considered and intercepted a wide range of semantically equivalent events for each class.

File System Operations File manipulation operations are a useful source for capturing malware behavior as most malware, except for very advanced in-memory-malware, typically need to conduct some file system access operations to conduct malicious activities. This ranges from simply profiling an infected system to plan further manifestation or persistence steps, stealing sensitive data, or to actually manipulate the infected system's integrity.

- **ReadFile(Ex)** is used by a process to read a specified amount of bytes from a file into its memory. This type of events are interesting, as they are typically employed by malware to steal sensitive information, read dropped binary images, or in general profile and infected system. We model this as a flow of data of a defined amount from the read file to the reading process node.
  Relevant Parameters: Calling Process (P_C), Source File (F_S), ToReadBytes (S_R), File Size (S_F)
  Mapping: $(F_S, P_C, S_R, t, \emptyset[(F_S, size) \leftarrow S_F]) \in \mathcal{E}$

- **WriteFile(Ex)** Using this function a process can write a specific number of bytes to a file. File write events are interesting from a malware perspective as they are used to manipulate an infected system, write dropped malware binaries to the file system for persistence reasons. We model this as a flow of data of a defined amount from a process node to the target file node.
  Relevant Parameters: Calling Process (P_C), Destination File (F_D), ToWriteBytes (S_W), File Size (S_F)
  Mapping: $(P_C, F_S, S_W, t, \emptyset[(F_D, size) \leftarrow S_F]) \in \mathcal{E}$

Registry Operations Captured registry access and manipulation operations of a malware are an important source of information to reason about its basic behavior and potential targets. Malware often profiles a system by crawling the Windows registry to get a better picture of the installed software landscape to e.g. find potential targets for further malicious or sensitive data worth to be stolen. Furthermore, most modern malware after the initial infection step tries to persistently dig into the system by dropping additional malicious binaries and ensuring their execution during start-up through manipulation of respective registry keys.

- **RegQueryValue(Ex)** Using this function a process reads the value of a specific registry key. We mainly consider this function as it is typically used by malware to profile the configuration and state of an infected system. We model it as a flow of a specific size, determined by the length of the read registry value, from the respective registry key node to the calling process node.
  Relevant Parameters: Calling Process (P_C), Source Key (K_S), ToReadBytes (S_R)
  Mapping: $(K_S, P_C, S_R, t, \emptyset) \in \mathcal{E}$

- **RegSetValue(Ex)** Using this function a process can write data to a specific registry key. It is in so far interesting for us in that it is typically used by malware to manipulate certain registry keys to e.g. ensure persistent execution of malicious binaries after system start-up. We model this function as a flow of a certain amount of data, again determined by the size of the buffer holding the data that is to be written to the registry key, from the calling process node to the target registry key node.
  Relevant Parameters: Calling Process (P_C), Destination Key (K_D), ToWriteBytes (S_W)
  Mapping: $(P_C, K_S, S_W, t, \emptyset) \in \mathcal{E}$

Socket Operations Considering that almost all modern malware is internet-based, monitoring and modeling network related behavior is crucial for accurate classification. Sockets, or WinSock functions are the standard interface for user mode processes to communicate with remote systems via network. Malware typically uses them to conduct all sort of malicious activities, ranging from communication with remote command-and-control servers to receive new instructions, report stolen data, or download additional malicious payload, over distribution of spam and manipulation of online advertisement, to active infection of other systems by exploiting network-related vulnerabilities.

- **Recv** Using this function a process can read a specific number of bytes from a network socket. We capture it as it is the primary way of malware to receive instructions and data from command-and-control servers. We model it as a flow of a defined amount, determined by the size of the respective socket buffer, from a socket node to the calling process node.
  Relevant Parameters: Calling Process (P_C), Source Address (IP Port) (A_S), ToReadBytes (S_R)
  Mapping: $(A_S, P_C, S_R, t, \emptyset) \in \mathcal{E}$

- **Send** Using this function a process can send a specific number of bytes to a network socket. It is important for us to intercept it to be able to model network-related self-replication, click fraud, or spamming behavior.
  Relevant Parameters: Calling Process (P_C), Destination Address (IP Port) (A_D), ToWriteBytes (S_W)
  Mapping: $(P_C, A_D, S_W, t, \emptyset) \in \mathcal{E}$

Process Operations Malware often creates complex fork chains, i.e. spawns clone processes of the own or additional downloaded malicious binary images to confuse detection mechanisms and implement update and plug-in functionality. This typically manifests in the creation of additional processes by the malware. Furthermore, malware quite often tries to circumvent host-based detection and firewall systems by hijacking other benign processes, that are usually white-listed by such approaches, by injecting malicious code into them. By doing this, the malware can effectively conduct a parasite strategy in the sense of making the formerly benign process conducting malicious activities that blend into the original benign behavior and thus harden detection.

- **CreateProcess(Ex)** Through this function a process can trigger the creation of another process, using a specific executable file as binary image. Although there are many ways to create new processes in Windows, we subsume the most common one under the umbrella of this function. We mainly capture it to model local self-replication behavior of malware, i.e. the creation of

fork chains. The function is modeled by a flow of a specific amount of data, determined by the size of the binary image of the to be created process, from the parent process node to the newly create child process node.
Relevant Parameters: Caller Process (P_C), Callee Process (P_D), Binary Name (F_B), Binary Size (S_B)
Mapping: $(P_C, P_D, S_B, t, \emptyset[(P_D, size) \leftarrow S_B]) \in \mathcal{E}$

- **ReadProcessMemory** Using this function a process can read a specific number of bytes from the memory of another process. To create hidden interprocess communication channels to coordinate different malicious processes, malware often employs IPC schemes that built on top of process memory read and writes. We model this function as a flow of a fixed amount, determined by the size of the respectively referenced buffer, from the target process node to the calling process node.
  Relevant Parameters: Calling Process (P_C), Source Process (P_S), ToReadBytes (S_R)
  Mapping: $(P_D, P_C, S_R, t, \emptyset) \in \mathcal{E}$

- **WriteProcessMemory** By this function one process can write a specific number of bytes to the memory of another one. This is an often used way of malware trying to write malicious code to other benign processes or implement some function call interposition based rootkit functionality. We model this function as a flow of a fixed amount, determined by the size of the referenced write buffer, from the calling process node to the target process node.
  Relevant Parameters: Calling Process (P_C), Destination Process (P_D), ToWriteBytes (S_W)
  Mapping: $(P_C, P_D, S_W, t, \emptyset) \in \mathcal{E}$

After having associated a, from a malware perspective interesting, set of Windows resources and WindowAPI functions to the respective conceptual counterparts of our abstract QDFG-based system model, we are now set to discuss its utility for malware detection purposes.

3.3. Malware Data Flow Behavior Example

To better understand how we use QDFGs to abstract from low-level system behavior, Figure 3.2 gives a grasp of how a QDFG obtained from monitored behavior of a system that was infected with the *Cleaman* malware looks like. The edge size represents the amount of data transferred between system resources.

The highlighted part of the graph lets us easily visually identify a behavioral pattern that is typical for so-called dropper malware: the executed malware connects to a remote server and downloads additional payload (1); then it executes the downloaded payload (2), which then gets loaded into a new malware child process (3). This malware child process then also connects to the remote server, most likely to receive additional instructions and update information (4).

This behavior yields several data flows of similar size between the initial dropper malware process, the dropped binary file, the remote command-and-control server, and the created malware child process, which manifests into the addition of multiple similar-sized edges to the corresponding system QDFG.

As we can see, the characteristic malicious behavior in this example is mainly captured by the edges with the highest amount of transferred data. We will later see that this is no coincidence and that focusing on edges that represent high amounts of data flows in fact is a good heuristic to isolate characteristic malware behavior.

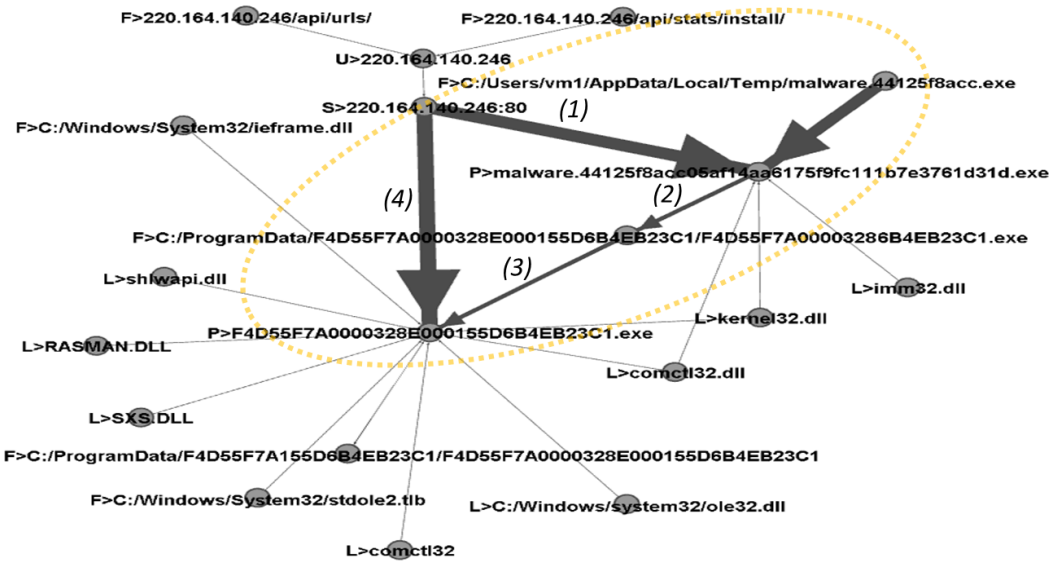
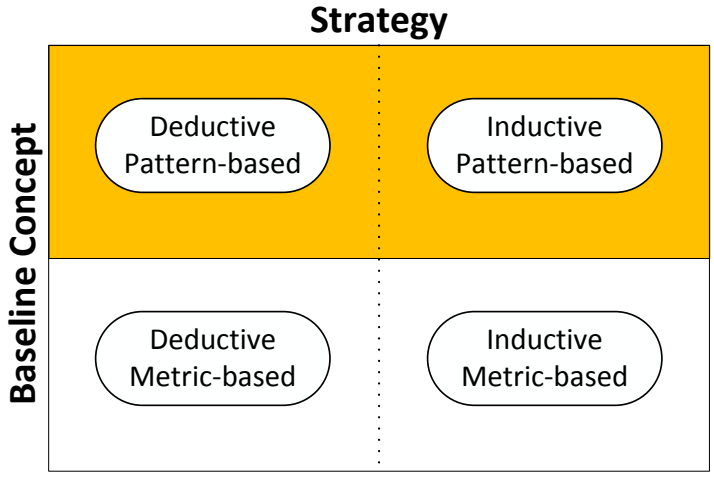


Figure 3.2.: Excerpt of infected system QDFG with highlighted malware behavior.

4. Pattern-based Detection

In this chapter we discuss how to use QDFG patterns to detect malicious activities. We do so with a deductive approach that detects pre-defined patterns of known malicious behavior in unknown samples and an inductive one that mines behavioral commonalities of malware samples to infer generalized detection patterns. Parts of this chapter are based on published work [144], co-authored as first author by the writer of this thesis.



After having introduced our basic idea of modeling low level behavior of a system in form of more abstract quantitative data flow graphs we will now discuss how to use this model for malware detection purposes.

In principle, there are two basic ways of detecting malicious activities in recorded system behavior: through statistical anomaly or pattern-based misuse detection [62]. Anomaly-based detection approaches in essence learn a notion of normal system behavior by monitoring the data flow behavior of known benign programs and then considering any deviation from that benign baseline as being malicious.

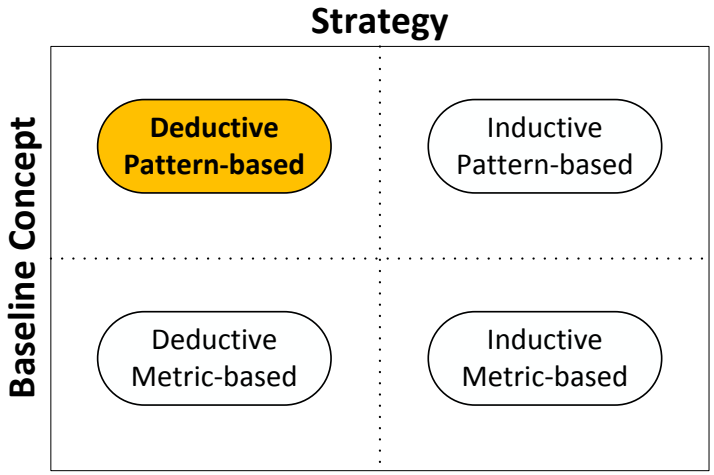
While this concept at first glance seems compelling and various instantiations at network- or host-level have shown promising results, anomaly detection ap-

proaches in highly unstable and noisy operational settings like our anticipated Windows operating system context still suffer from often unacceptably high false positive classification rates [2].

For this reason, in this thesis we focus on the second principal concept of malicious activity detection, which is the identification of known malicious behavior patterns in observed system behavior, i.e. misuse detection. Misuse detection in comparison to anomaly detection is considered to only provide limited capabilities of detect novel malicious behavior. However, the different misuse detection approaches proposed in this thesis cater to this problem by incorporating various fuzzy matching mechanisms that, as we will later see, still to some extent enable us to detect previously unseen malicious behavior, i.e. unknown malware.

Just like most misuse detection concepts, our first pillar of devised detection approaches also leverages patterns of known malicious behavior to classify unknown behavior as malicious or benign. As in our context behavior is modeled as graphs, we consequently specify and re-identify patterns as graphs. While pattern-based detection on graphs has been proposed before, e.g. at the level or control-flow- [21, 28, 29, 37, 85], system-call-dependency- [80, 54, 60, 36], or resource-dependency-graphs [113, 114, 76, 151], we are the first to do this at the level of quantitative data flow graphs.

4.1. Deductive Pattern-based Detection



4.1.1. Introduction

For our first instantiation of our pattern-based malware detection concept we classify unknown samples by comparing their observed behavior with predefined patterns of known malicious behavior.

As we will later see, using QDFGs as behavior model for pattern-based detection renders reordering, injection, and semantic substitution attacks widely ineffective, as long as they do not alter the structure of the resulting quantitative data flow graphs, making this approach more resilient against common behavioral obfuscation attacks than comparable behavior-based detection approaches [6]. By aggregating semantically similar and related flows between system entities during the construction of our data flow graphs, we can furthermore keep the underlying data structures light-weight and efficient. Considering the findings of Fredrikson et al. [55] that malware detection through behavioral pattern matching in general cannot be done efficiently, simplifications as used in our approach are particularly helpful in keeping detection efficiency within reasonable boundaries.

Besides the fact that we use QDFGs as behavior model, the main difference of our approach to related ones that are based on pattern recognition on graphs is that we do not only check whether or not a known malicious pattern matches a

unknown QDFG, but also impose additional quantitative constraints on how exactly the pattern has to match a QDFG in order for it to be considered malicious. The basic idea of this approach thus is to improve the detection precision and reduce false positive classifications by incorporating additional quantitative constraints into the pattern identification process and thus making pattern matching checks more restrictive.

As mentioned in Section 1.2, for the first instantiation of our pattern-based detection concept we follow a deductive methodology in that we first manually specify patterns of well-known malicious behavior and then, together with additional quantitative data flow constraints, use them to classify unknown samples. This is done by executing potentially malicious samples in a customized malware analysis sandbox [110], record the sample's behavior in terms of system call traces, interpret them as QDFGs, and try to identify malicious patterns in them.

The main goal of this first approach was on giving an initial answer to *RQ1*, i.e. whether we can operationalize our QDFG model for malware detection, and *RQ2*, if the usage of quantitative information can improve detection effectiveness.

We will then later discuss an extension to this basic detection concept with data mining and machine learning mechanisms to automatically derive more discriminative patterns from a body of known malware, i.e. follow an inductive instead of a deductive strategy, to evaluate the feasibility boundaries.

The specific contributions of this approach can be summarized as follows:

1. To the best of our knowledge this is the first approach that makes use of quantitative data flow analysis for behavior-based detecting malware.
2. We show the feasibility of pattern-based malware detection on QDFGs.
3. We show that the additional quantitative data flow constraints encoded in the patterns yield a significant improvement of detection effectiveness.

Some parts of this section are based on previous work [144], co-authored as first author by the writer of this thesis.

4.1.2. Approach

Our approach consists of six components that conduct the different data retrieval, processing, and classification tasks. Figure 4.1 depicts a high-level overview of the architecture of our approach. In the following we briefly sketch the different tasks of our approach and associate them to the respective architecture components. In the subsequent section we will than elaborate on their functionality in more detail.

A) *Malware Sandbox*: To obtain behavior profiles of known goodware and malware samples we need to first execute them in a controlled environment and record the system calls induced by the respective processes. We do so using a customized version of the popular open-source malware sandbox Cuckoo [110]. As the vanilla version of Cuckoo is only capable of monitoring the process and its descendants that refers to the executed malware or goodware sample, but for our approach we need a full picture of the complete system behavior, we had to substitute its behavior monitor with an own IAT-patching based user-mode Windows API monitor [146], capable of full system monitoring. Our monitor furthermore features more advanced functionality to annotate intercepted system calls with context information like the size of referenced read or write buffers, which we need for building our QDFGs. After submitting a sample to the sandbox it gets uploaded and executed in one of the sandboxes' virtual machines that are equipped with our monitoring infrastructure. The behavior of the instrumented systems then get monitored for a pre-defined period of time after which the VM is stopped, the recorded API calls written to a log, and the log sent back to the backend of our infrastructure.

B) *Event Parser*: This component implements the data flow semantics described in Section 3.2.0.2 and translates the system calls from the traces it received from the *Malware Sandbox* component to sequences of data flow events that then get forwarded to the *Graph Builder* component. The *Event Parser* also features functionality to map a wide range of semantically similar Windows API calls to the basic set of functions described in Section 3.2.0.2 and their respective data flow semantics. With this we cover a good portion of the Windows API and consider the respective functions for building the QDFGs although we only formalized a very reduced subsets of it. Note that, although by doing so we were able to capture a wide range of possible malware behavior, we by no means claim to be able to intercept all possible Windows API calls. This is because the Windows API in its current form features thousands of different functions that get changed and extended with almost every major Windows update. We thus deem a full interception and modeling of the entire Windows API unfeasible within the scope of this the-

sis. Although the intercepted functionality is sufficiently comprehensive to build highly accurate and robust detection models, we are aware that this still imposes a certain risk of malware circumventing detection.

C) *Graph Builder*: The *Graph Builder* implements our system model and in particular the graph update function described in Section 3.1. For this it takes the sequences of data flow events it gets from the *Event Parser* and incrementally extends a correspondingly built QDFG by continuously invoking the QDFG *update* function. After having processed the entire received data flow event sequence, the *Graph Builder* outputs a QDFG that models the behavior of the corresponding system during the monitored period of time.

D) *Pattern Matcher*: The *Pattern Matcher* can be considered the core component of our approach as it is in charge of identifying patterns of known malicious behavior in the captured behavior of unknown samples in order to classify them. As behavior in our case relates to QDFGs and thus graphs, doing so essentially boils down to finding sub-graphs in the QDFG of a to be classified sample that are isomorphic to the graphs, i.e. patterns, associated with known malicious behavior. This sub-graph isomorphism problem in general is known to be NP-complete and thus computationally hard. However, baring in mind that there do exist efficient sub-graph isomorphism algorithms for specific classes of graphs [48], considering our comparably simple patterns, and anticipating that we had to incorporate more complex quantitative constraints on the actual matching process, for this work we decided to make use of a modified version of the VF2 algorithm [43]. The *Pattern Matcher* itself does not yet consider the quantitative data flow constraints within the pattern properties for matching but rather forwards all potential matching pairs of QDFG sub-graphs and patterns and their yet to be checked quantitative guard properties to the *Quantity Analyzer* component.

E) *Quantity Analyzer*: All our detection patterns, as we will see in the next section, specify additional quantitative data flow constraints that must be satisfied by a sub-graph of a to be classified QDFG in order for it to match the respective pattern. However, our approach allows to enable and disable those properties in order to make the pattern matching more or less restrictive. The *Quantity Analyzer* implements this functionality by taking the matching decisions from the *Pattern Matcher* and evaluating the quantitative data flow properties of the respective patterns on the matching sub-graphs. Depending on a pre-configurable maximum accepted deviation from these constraints, the *Quantity Analyzer* then removes matches from the result set that do not satisfy the quantitative constraints. By allowing slight deviations from the quantitative data flow guarding properties we anticipate the prob-

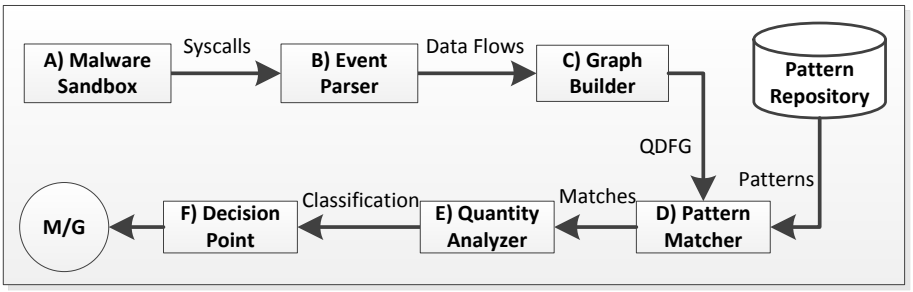


Figure 4.1.: Architecture

lem of the quantitative guard conditions in some cases being too restrictive due to inevitable noise that e.g. results from different file encoding or network management overhead.

F) *Decision Point*: Anticipating that one pattern in itself might not be discriminative enough to accurately differentiate between malign and benign samples, we introduced an additional arbiter component that yields the final classification decision taking all pattern matching results into account. This is, the *Decision Point* only classifies a QDFG, i.e. a sample, as malicious if the respective pattern matches together satisfy a global meta-property. In our current implementation this means, that we only consider a QDFG to be malicious iff at least two patterns from different categories matched it. To again relax or tighten this final classification decision, we make this matching threshold adjustable.

This architecture will build the reference for all subsequently discussed detection approaches and will be customized and extended to fit the specific characteristics of the respective approaches. Keeping the big picture of our detection approach in mind, in the next section we will now discuss how to specify malware-specific behavior patterns as properties over QDFGs to be used for the actual classification of unknown samples.

4.1.2.1. Malware Behavior Patterns

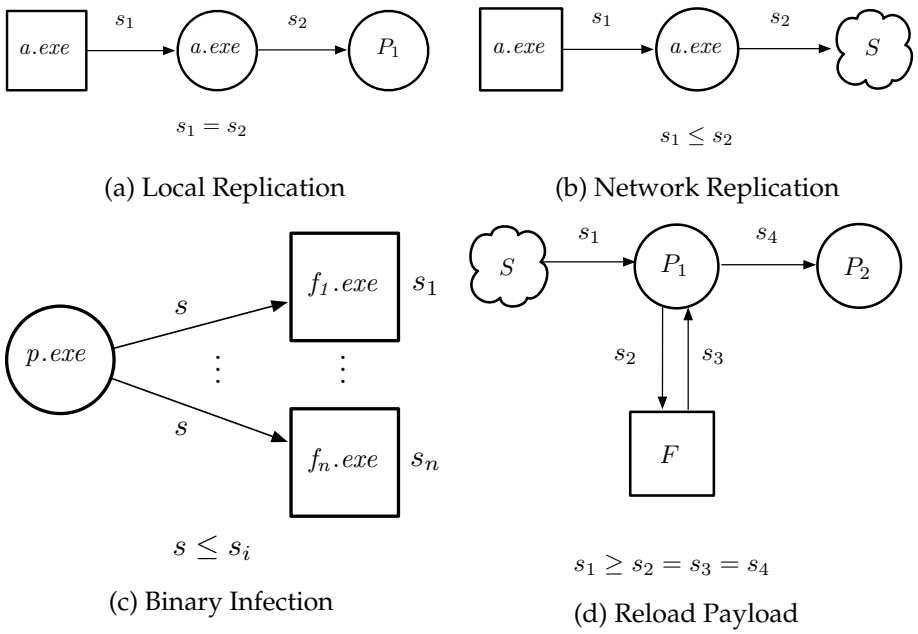
Based on the generic system model introduced in Chapter 3 and its instantiation to Windows operating systems, we can model specific malware detection patterns that make use of (quantitative) data flows and data flow properties. We used these for our prototype to detect potentially malicious processes and for the experiments in Section 4.1.3. These patterns were deductively defined on the basis of commercial malware behavior databases [132], as partially sketched in Section 2.1, and academic malware analysis reports [9]. In the following, we will say that a "pattern matches" if according to this pattern malware is present.

The basic idea of constructing such patterns is to identify characteristic data flows and flow properties that correspond to typical high-level behavior of malware such as "*a malicious process tries to replicate itself by infecting other benign binaries*", like e.g. seen for the Parite worm, or "*a malicious process tries to replicate itself via email*", as e.g. common for email worms like MyDoom. These characteristic data flows or properties than represent a set of potentially malicious activities that can later be re-identified in data flow graphs of the to be analyzed samples.

Not all of these patterns in themselves are sufficiently specific and expressive to always discriminate between malware and goodware. An example for this is a pattern that matches if a process downloads payload from the internet and starts it as a new process. Although such behavior is typical for so-called dropper malware that install additional malware (so-called eggs) using this technique, it also matches the behavior of benign web installers. Yet, by combining several patterns, we can achieve a sufficient specificity and precision do correctly discriminate between good- and malware in most cases.

For formally specifying our detection patterns we need a few auxiliary functions in addition to the graph functions defined in 3.1. Because their definition is standard, we omit a formalization.

Function $pre : \overline{N} \times \mathcal{G} \rightarrow 2^{\overline{N}}$ computes all immediate predecessor nodes of a node of the graph. Conversely, $suc : \overline{N} \times \mathcal{G} \rightarrow 2^{\overline{N}}$ computes the immediate successors of a node. Functions $in, out : \overline{N} \times \mathcal{G} \rightarrow 2^{\overline{E}}$ compute the set of incoming and outgoing edges of a node. If $Path$ denotes the set of all finite sequences, function $paths_{from} : \overline{N} \times \mathcal{G} \rightarrow 2^{Path}$ calculates all paths without loops that originate from a given node n. The intuition is that there is a data flow from n to each node in the path. Conversely, and finally, function $paths_{to} : \overline{N} \times \mathcal{G} \rightarrow 2^{Path}$ computes all those paths without loops in the graph that have the argument node as last element. We further introduce projections $!_i$ with $(x_1, \ldots, x_k)!_i = x_i$ for $1 \leq i \leq k$ to address a specific element y in the i-th position of a sequence Y. To further simplify the description, we define a sub-string function to extend patterns definitions with string comparisons: $ss : \mathcal{S} \times \mathcal{S} \rightarrow \{0, 1\}$ which evaluates to true, iff the first provided string is a sub-string of the second string parameter.



(a) Local Replication

(b) Network Replication

(c) Binary Infection

(d) Reload Payload

Figure 4.2.: Replication patterns

The formalizations of our detection patterns come as predicates $\phi \subseteq \overline{N} \times \mathcal{G}$ that are to be read as follows. If $\phi(n, G)$ is true for some node n in a given QDFG G, then the patterns corresponding to ϕ suggests that malware is present. This means that the evaluation must be performed for each node.

We now have a basis for specifying malware-specific (quantitative) data flow patterns. These patterns will enable us to discriminate between benign and potentially malicious system entities. Typical examples for such patterns include restrictions on orders or specific sequences of flows that are characteristic for certain malware types.

We express such malware data flow patterns as first-order logic predicates on properties of the introduced QDFGs. The set $\Phi_{\text{mw}} := \Phi_{\text{rep}} \cup \Phi_{\text{mpl}}$ contains all specified malware replication and manipulation patterns that model specific classes of malware behavior. The motivation for our separation of patterns is that we want to differentiate between patterns that detect replication and those that detect the system manipulation behavior of a malware. This separation also allows us to increase the detection specificity by combining different types of patterns. Note that the sets of patterns are not necessarily disjoint.

Each of the following pattern specification consists of a natural language description of the corresponding flow pattern, and a brief description of the rationales behind the pattern.

In addition we visually illustrated some patterns to give the reader an understand of their structure (see Figure 4.2). Boxes represent file nodes, circles represent process nodes, and clouds represent socket nodes. We only visualized the more complex replication patterns and omit a visualization of our manipulation patterns as they share the same simple 2-nodes-1-edge structure.

The choice of the the following set of detection patterns essentially is motivated by the goal of capturing the most distinctive behavior characteristics of the common malware classes, as described in Section 2.1.2. Nevertheless we are aware that the huge behavioral heterogeneity of today's malware landscape can likely not entirely be covered by such a fixed-sized pattern set and thus do not claim to be comprehensive in this respect. However, we assume to having captured the most prevalent behavior characteristics and later in Section 4.2 will discuss how to arbitrarily extend this pattern library by means of behavior mining.

Replication Patterns The patterns in this class capture activities targeting infection of a local or remote system with malware. That can e.g. be achieved by appending malicious binaries to a set of benign system programs, injecting malicious code into other processes, or sending binaries to remote network nodes.

- **Local Replication (ϕ_{rep_1})** Matches if a process node has at least one flow from a file node that has the same name as the process, followed by at least one flow to another process. To increase the precision of the pattern the amount of flown data from file to process must be the same as the amount of data that flows between the two processes (as e.g. observed for Trojans like Agent).
  Rationale: This pattern covers characteristic data flows caused by malware trying to replicate by spawning clone processes from the own binary image.
  Formalization:

$$\begin{aligned}&\phi_{\text{rep}_1}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n)\bullet\\ &G.\lambda(n, type) = F \wedge G.\lambda(e_1!_2, type) = P \wedge ss(e_1!_2, n)\wedge\\ &G.\lambda(e_2!_2, type) = P \wedge G.\lambda(e_1, size) = G.\lambda(e_2, size).\end{aligned}$$

- **Network Replication (ϕ_{rep_2})** Similar to the local replication pattern, this pattern matches if there exists at least one flow between a process and a file of similar name and a flow from this process to a network socket. To increase specificity the amount of data sent to the socket must be at least as big as the amount of data read from the file (as e.g. observed for Email Worms like MyDoom).
  Rationale: This pattern covers data flows that are typical for a malware that tries to replicate itself by sending its binary image over the network to infect remote systems.

Formalization:

$$\phi_{\text{rep}_2}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n) \bullet$$
$$G.\lambda(n, type) = F \wedge G.\lambda(e_1!_2, type) = P \wedge ss(e_1!_2, n) \wedge$$
$$G.\lambda(e_2!_2, type) = S \wedge G.\lambda(e_1, size) \leq G.\lambda(e_2, size).$$

- **Binary Infection (ϕ_{rep_3})** This pattern matches if there exist multiple flows (at least two) from a process to executable binary files. To reduce false positives an additional quantitative constraint is as follows. The amount of transferred data to the benign executables must be at least as high as the amount of data from the binary image of the process to the process; and the size of the target binary files must be greater than 0. This to some extent ensures that at least the size of the malware image is appended to already existing binaries.
  Rationale: These data flows resemble malware activities that are targeted at replication through infection of other program's executables. This usually happens when malware tries to append its own malicious code to other benign binaries (as e.g. observed for Viruses like Parite).
  Formalization:

$$\phi_{\text{rep}_3}(n, G) := \{(e_1, \ldots) \in paths_{from}(n) |$$
$$G.\lambda(n, type) = P \wedge G.\lambda(e_1!_2, type) = F \wedge ss(\text{“}.exe'', e_1!_2) \wedge$$
$$G.\lambda(e_1!_2, size) > 0\} \geq 2$$

- **Download and Start Payload (ϕ_{rep_4})** This pattern matches if there is a flow from a socket node to a process node, a flow from this process to a executable binary file node, a flow from this file node back to the process node, and then a flow from this process node to a new process node with a similar name as the file node. To increase detection specificity we increase the additional quantitative constraint that all flows of this pattern must have the same quantities, except for the first flow from the socket node that may also be bigger due to additional meta- and network control data. This to some extent ensures that at least the downloaded size of payload is propagated to a binary file and then to a new malicious process.
  Rationale: This data flow pattern subsumes malware behavior that targets reloading and executing additional malicious payload from the internet (observed for Droppers as e.g. used by the Conficker virus).
  Formalization:

$$\phi_{\text{rep}_2}(n, G) :=$$
$$\exists p = (e_1, e_2, e_3, e_4, \ldots) \in paths_{from}(n) \bullet$$
$$G.\lambda(n, type) = S \wedge G.\lambda(e_1!_2, type) = P \wedge G.\lambda(e_2!_2, type) = F \wedge$$
$$ss(\text{“}.exe'', e_2!_2) \wedge e_3!_2 = e_2!_1 \wedge G.\lambda(e_4!_2, type) = P \wedge$$
$$G.\lambda(e_1, size) \geq G.\lambda(e_2, size) = G.\lambda(e_3, size) = G.\lambda(e_4, size).$$

Manipulation Patterns This class of patterns contains certain flow patterns that correlate with specific high-level semantics for activities that fall under the broad category of manipulation of system integrity or data confidentiality. Such patterns for example include activities that target leaking sensitive data to untrusted locations like the internet, modifications of the registry to e.g. add autostart entries, or opening backdoors for further malicious activities.

- **Leak Cached Internet Data (ϕ_{mpl_1})** Whenever we detect a flow of data from a dedicated internet cache file (in our prototype identified by the absolute file path containing either "*Cookies*" or "*Temporary Internet Files*" as sub-string) to a process that then sends data to a socket, this pattern matches. The specificity of this pattern is increased by demanding that the flow from the leaking process to the remote socket node must be as least as big as the flow from the cache file to the process.
  Rationale: This pattern captures the data flow behavior of a malicious process trying to steal potentially sensitive data like cookies from dedicated internet cache folders by sending them to a remote network location (as e.g. observed for various samples of the generic Infostealer malware family).
  Formalization:

$$\begin{aligned}&\phi_{\mathrm{mpl}_1}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n)\bullet\\&G.\lambda(n, type) = F \wedge G.\lambda(e_1!_2, type) = P \wedge\\&(ss(\text{“}Cookie'', n) \vee ss(\text{“}Temporary\ Internet\ Files'', n)) \wedge\\&G.\lambda(e_2!_2, type) = S \wedge G.\lambda(e_1, size) \leq G.\lambda(e_2, size).\end{aligned}$$

- **Spawn Shell (ϕ_{mpl_2})** This pattern matches if a command line shell process node (in Windows identified by the name of the process node containing the sub-string "*cmd.exe*") has an incoming or outgoing data flow connection to at least one socket. For simplicity's sake we currently only consider processes with an indirect connection with a maximum distance of at maximum two hops to a socket.
  Rationale: This pattern describes data flows caused by malware trying to glue a command shell to a listening socket to open a backdoor (as e.g. observed for Backdoors like Autorun).
  Formalization:

$$\begin{aligned}&\phi_{\mathrm{mpl}_2}(n, G) := \exists n' \in (pre(n) \cup suc(n))\bullet\\&G.\lambda(n, type) = S \wedge G.\lambda(n', type) = P \wedge ss(\text{“}cmd.exe'', n').\end{aligned}$$

- **Deploy System Driver (ϕ_{mpl_3})** This pattern matches if we detect a flow between a process and a system driver binary, identified by its name containing the system driver extension "*.sys*", followed by a flow from this system

driver file to one of Window's driver loading utilities (in our current implementation we consider "*wdreg.exe*" and "*sc.exe*"). To increase the specificity of this pattern we also demand that the flow from the potentially malicious processes to the system driver file and from the file to the driver loading utility node must be of the same size.
Rationale: Today, many sophisticated malware types make use of root kit technology to take over full control of a compromised system and hide their behavior from anti malware software. This pattern thus describes the data flows that correlate with malware attempts to deploy and load malicious system drivers to the Windows kernel to inject its root kit functionality (as e.g. seen for the ZeroAccess root kit).
Formalization:

$$\begin{aligned}&\phi_{\text{rep}_3}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n) \bullet \\ &G.\lambda(n, type) = P \wedge G.\lambda(e_1!_2, type) = F \wedge ss(\text{“}.sys\text{”}, e_1!_2) \wedge \\ &G.\lambda(e_2!_2, type) = P \wedge G.\lambda(e_1, size) = G.\lambda(e_2, size) \wedge \\ &(ss(\text{“}wdreg.exe\text{”}, e_2!_2) \vee ss(\text{“}sc.exe\text{”}, e_2!_2)).\end{aligned}$$

4.1.2.2. Malware Detection

Having a set of malware-specific behavior patterns at hand, using them to classify unknown samples, i.e. operationalize them for behavior-based malware detection, is comparably straight-forward and follows the procedure described in Section 4.1.2:

1. Execute the to be analyzed sample in the malware sandbox and record the API calls issued by all processes running in the monitored system for a defined period of time.

2. Translate the captured API call traces into data flow events and then into a QDFG.

3. Match the previously described detection patterns against the QDFG, i.e. evaluate the function $malicious$ on it as described in the following.

4. Evaluate the deviation between the quantitative data flows in the sub-graphs that structurally matched the patterns and their quantitative constraints.

As mentioned before, in our current prototype for a sample to be considered malicious at least one replication and one manipulation pattern has to match the QDFG that models its behavior, which is formally defined the $malicious$ functioned called with the to be classified QDFG G as argument returning 1:

$$malicious(n, G) = \begin{cases} 1 & \text{if } \exists \phi_{\text{rep}} \in \Phi_{\text{rep}} \; \exists \phi_{\text{mpl}} \in \Phi_{\text{mpl}} \; \exists \phi_{\text{qnt}} \in \Phi_{\text{qnt}} \bullet \phi_{\text{rep}}(n, G) \wedge \phi_{\text{mpl}}(n, G) \\ 0 & \text{otherwise} \end{cases}$$

As mentioned before, in operational contexts there sometimes is some inherent noise added to buffers or files that has an effect on the sizes of the respectively modeled nodes and flows. This typically is consequence of the fact that in real system processes do not only exchange and interact with actual payload, but also need to conduct some non payload-focused management and control flow operations like network protocol or file encoding task. If we recall that any system call that has a data flow semantic contributes to the update of our system model, i.e. QDFGs, it becomes clear that such additional behavior can add noise to the sizes of nodes and edges. If we for example think of a typical network context where a process needs to conduct a series of protocol handshake actions that all lead to some sort of data flow between the remote system and the process, a property that imposes hard equality constraints on respective flows might be, from a semantic perspective, wrongly prevent a pattern from matching.

To cater to this problem, our approach allow slight deviation from quantitative constraints that originally demand hard equality. Depending on the acceptable threshold we thus can effectively control, how restrictive the pattern matching should be enforced, i.e. how much emphasis should be put on the additional quantitative constraints. Proper and suitable values for setting this threshold are then governed by the concrete detection goals, i.e. if we want to put more emphasis on achieving high detection rates or more on preventing false positive classifications.

4.1.3. Evaluation

In general, it is difficult to objectively assess the effectiveness of any behavior-based malware detection technique. This is because experiment outcomes heavily depend on chosen goodware and malware evaluation sets, malware activity during behavior recording, and counter-analysis measures employed by the malware.

To shed light on the general feasibility of our concepts and to evaluate our prototype we thus conducted a proof-of-concept study based on a representative set of goodware and malware. Within this study we investigated the effectiveness and efficiency of our detection approach. In terms of effectiveness we mainly focused

on detection and false positive rates, where detection rate, i.e. true positive rate, is defined as the fraction of correctly detected malware samples in all analyzed malware samples, false positive rate by the ratio of goodware samples, wrongly classified as malware, within the entire set of analyzed goodware samples, and accuracy, which is the faction of correctly classified samples within the overall evaluation data set. In terms of efficiency we analyzed average time it took us to classify QDFGs of different sizes.

4.1.3.1. Evaluation Setup

To foster the comparability between the different approaches and their contributions presented in this thesis we strived to consolidate the methodology, data sets, and operational settings used for the respective evaluation experiments. Therefore all experiments conducted to evaluate the different approaches made use of the same or only slightly deviating evaluation setup.

Evaluation Data Set As data source for our populating the evaluation data set of QDFGs used for our experiments we used about 7000 different known malicious programs and about 500 different known benign applications. The malicious program samples were taken from a subset of the Malicia malware data set, i.e. all samples that were executable in the considered evaluation environment, that comprises of real-world malware samples from more than 500 drive-by download servers [100]. The respective malware set consists of samples from 18 malware families, including popular ones like Zeus/Zbot, Spyeye, and Ramnit.

We decided to use this data set for various reasons. First of all it is publicly available and thus can be used to replicate our experiments and to compare our results with the performance of other approaches that use the same data set. Furthermore, the malware samples were gathered using a semi-automated approach to milk a wide range of malicious web site. With this approach a typical malware infection entry vector via drive-by-downloads is replicated. We thus consider the obtained set of malware a good snapshot of the real-world threat landscape at the time where the malware samples were obtained, which allow us to some extent to reason about the real-world utility of our detection concepts. Finally, the Malicia data set already comes with labels that represent the joint detection and classification results of other malware detection approaches. With this we get a credibly source of ground truth to compare our classification results to.

Our goodware sample set was composed of popular applications that were downloaded from www.download.com and a wide range of standard windows programs, including popular email programs like ThunderBird, browsers like FireFox, video and graphics tools like Gimp, or VLC Player, and security software like Avast.

With this we aimed at replicating a wide range of settings as they would typically occur in normal Windows desktop environments.

The exact composition of our evaluation data set is described in Appendix C.

To generate the QDFGs that model the behavior of the evaluation samples, we then separately executed each sample in a clean virtual machine within the malware sandbox and recorded the induced system behavior for a period of 5 minutes. With this we avoided collusion of malware behavior and ensured a comparable evaluation baseline. To anticipate the fact that goodware usually is more reactive than malware in that it typically reacts on certain user events, whereas malware usually automatically conducts most of its behavior, we employed some simple program stimulation means. To this end we used the standard program stimulation module of Cuckoo [110] which features some very basic user input simulation and e.g. randomly clicks on user controls, displayed by an executed binary. Despite its simplicity, this stimulation strategy was sufficient to yield behavior for most of the executed samples that was rich enough for further processing.

Unfortunately we were not able to obtain a QDFG for all samples as some of them did not execute properly within our evaluation environment. We account this observed effect to compatibility issues between certain malware instances and our execution environment which, without a deep understanding of the implementation details of the malware, can hardly be resolved externally.

The complexity of resulting QDFGs that then made up our evaluation data set ranged between 44 and 1203 edges, with an average of about 500 edges and a standard deviation of about 250. Figure 4.3 depicts the respective cumulative distribution function of the complexity of the QDFGs over all QDFGs in the evaluation data set. As we can see, more than 70% of the QDFGs within the evaluation data set have a size of less than 500 edges and only about 10% are of size 1000 or bigger.

Execution Environment For setting up our execution environment our main goal was to replicate an operational setting that gets as close as possible to the defacto standard of user desktop PCs at the time this thesis was written. To this end we configured the Cuckoo sandbox environment [110] that we used to execute and monitor malware and goodware samples to use virtual machines running Windows 7 SP1 which, at the time this thesis was written, was the globally predominate operating system on the market with a market share of over 52% [130]. Each Cuckoo sandbox VirtualBox instance got assigned two 2,4GHz CPU cores and 2GByte of RAM. The malware sandbox itself was then deployed on an Intel Xeon server powered by 6 physical 2.4GHz cores and equipped with 128GByte of RAM. The actual classification computations and evaluations of this approach finally were conducted on a Intel Xeon server powered by 16 physical 3.4GHz cores and equipped with 128GByte of RAM.

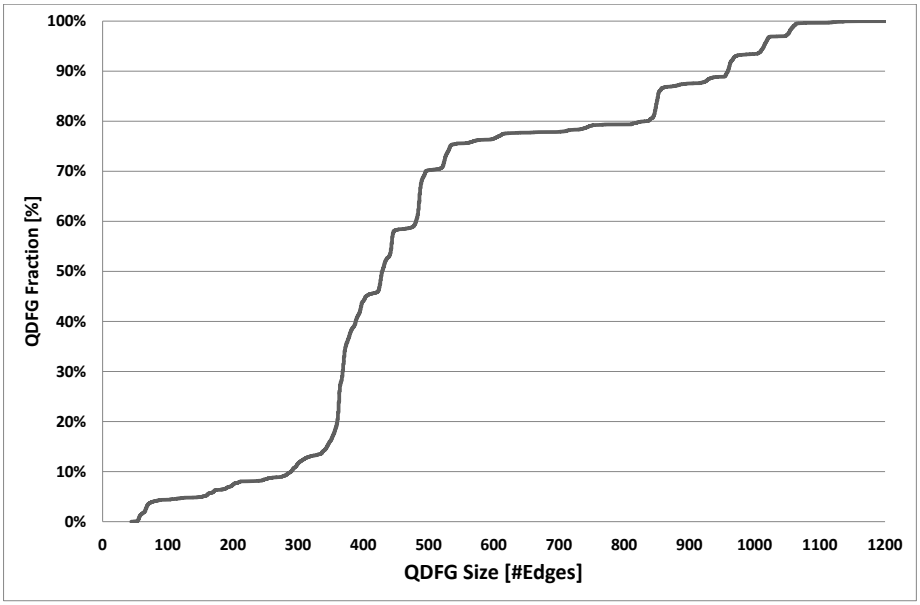


Figure 4.3.: CDF of QDFG edge size

4.1.3.2. Effectiveness

On the QDFGs from our evaluation data set we conducted the analysis steps described in Section 4.1.2, using the patterns described in Section 4.1.2.1 to classify each QDFG into the categories *malicious* and *benign*. To get a baseline for later evaluating the impact of considering quantities on classification effectiveness, for our first experiment we completely ignored the quantitative constraints of the patterns for matching. For this we disabled the *Quantity Analyzer* component and directly forwarded the *Pattern Matcher* results to the *Decision Point* to get a final classification result irrespective of any constraints on the quantitative data flows tackled by the patterns. The result of this experiment is depicted in the left part of Table 4.1.

As we can see from the results of this experiment, the detection effectiveness with about 77% detection rate at 14% false positive rate is not overly good. Nevertheless it shows, that we to some extent can leverage the introduced patterns to do malware detection on QDFGs, which partially answers our initial research question *RQ1*.

	Non-quantitative	Quantitative
Detection Rate	0.769	0.769
False Positive Rate	0.139	0.046

Table 4.1.: Effectiveness with and without quantitative constraints

Of course, the final goal of this work was to investigate if considering quantitative data flow properties for the classification decision has a positive impact on the overall detection precision.

To this end we conducted an additional set of experiments to specifically investigate the effect of the quantitative guard properties within the patterns on the overall detection precision. To do so we again applied our approach on the evaluation data set, but this time we only considered a pattern to match an evaluation QDFG if also its quantitative guard properties were satisfied. To anticipate side-effects and noise due to file encoding and network protocol overhead we step-wise increased the tolerance level of the quantitative properties to also accept not perfectly equal data flows until the tolerance level reached a point where the detection rate started to get compromised.

Comparing the results of the experiments with and without considered quantitative constraints we can see that by introducing quantitative flow properties to the detection properties we can effectively cut the false positives by a factor of $3 \approx \frac{.139}{.046}$ for a targeted detection rate of about 77%.

These insights thus to some extent answer our initial research question *RQ2* in that we could show that, at least for this approach, considering quantitative data flow properties for classification indeed improved detection effectiveness. Note again that this first detection approach was mainly designed to investigate the relative effectiveness impact of quantitative data flow properties and for this reason was not optimized to yield an as high as possible absolute effectiveness. Naturally the absolute detection effectiveness results of this first approach are thus not yet optimal and, as we will see later, can be tremendously improved.

4.1.3.3. Efficiency

A crucial requirement for malware detection approaches is their efficiency as it has a direct impact on the analysis throughput, i.e. the amount of samples that can be analyzed during a specific time frame given a fixed set of computational resources. Considering that anti-malware industry reports having to analyze tens of thousands malware samples every day [112] it becomes clear that efficiency and thus scalability is a major concern.

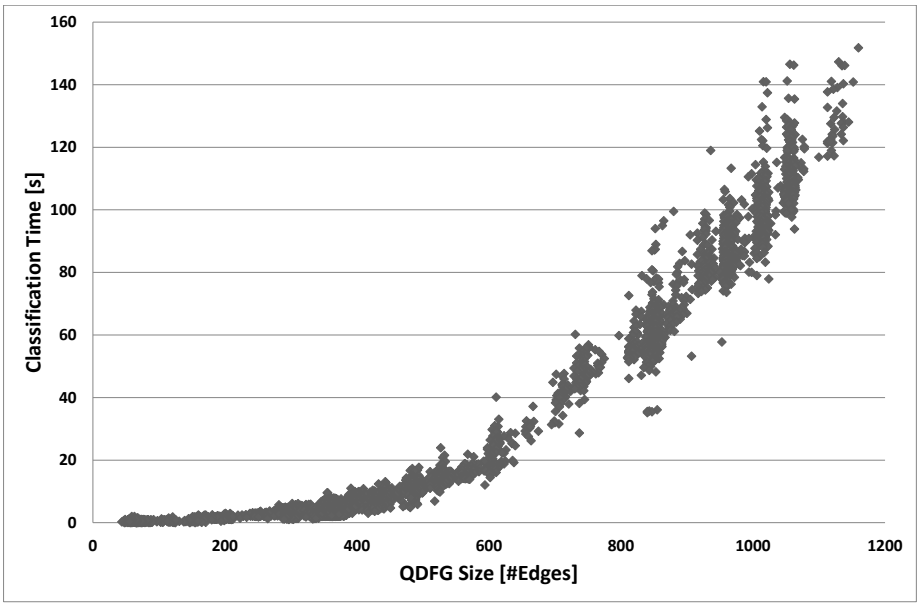


Figure 4.4.: Absolute classification time vs. QDFG size

Figure 4.4 depicts the results of efficiency evaluation of our approach where we for each QDFG in the evaluation data set measured the overall time it took to classify it as malicious or benign. As we can see, the absolute classification time seems to at least quadratically grow in the amount of edges. While for small- to medium-sized QDFGs with less than 500 edges, which make up more than 70% of the complete evaluation data set, the overall classification time only grew slowly and at average remained below 6 seconds, to classify graphs with 500 edges or more, the computation time rapidly increased. In sum, the complete classification procedure at average could be conducted in under 23 seconds.

Further note that by construction our QDFT graphs typically only grow slowly under normal usage because of the applied aggregation steps of our model, with occasional peaks whenever new programs are started or new resources used. This means that, although exponential with respect to graph size in general, in real-world operational settings the maximum actual computational complexity is to some extent bounded.

4.1.3.4. Discussion and Threats to Validity

Although using quantitative data flows as abstraction of malware activities, as we will later see, raises the bar for malware to hide its presence through obfuscation, advanced quantitative malware obfuscation or intentionally delayed behavior might challenge our pattern-based detection concept. We will later discuss the influence of such advanced behavior obfuscation attempts on the detection robustness of our more advanced detection approaches. We furthermore acknowledge the threat arising from the so-called base-rate fallacy [2] on the effectiveness of malware detection approaches, including ours. According to this paradox, even a relatively low false-positive rate of less than 5% can render an approach ineffective if malicious activities are considerably less frequent than benign ones. We will counteract this threat by proposing more complex and robust matching approaches (see Chapter 5) that do not rely on fixed patterns and for which we show that they are to some extent robust against certain types of behavior obfuscation.

In addition, the proposed approach is solely based on static data flow patterns which only allows us to detect malware that has the specified behavioral characteristics. Although we used a wide range of behavioral descriptions of different malware families to derive our patterns, new malware could thus simply achieve its goals in a way that we did not anticipate. To thwart this issue we in Section 4.2 will propose a more elaborate approach to mine characteristic behavior patterns from a corpus of known malware to obtain larger, more diverse, and more specific malware detection patterns.

4.1.4. Related Work

In the past decades, a plethora of work has been published in the broad area of behavior-based malware detection [46] which we already reviewed in Section 2.2.

Since the proposed approach focuses on graph-based malware detection, we in the following limit ourselves to review malware detection approaches that employ some sort of graph model to represent malware behavior.

One of the main lines of research on malware detection using graph-based models bases on the analysis of dependencies and interrelationships between activities like system call invocations of processes. After the extraction of characteristic call dependencies of different knowingly malicious processes they can be used to re-identify certain dependency patterns in unknown behavior graph samples in order to discriminate between malicious and non-malicious processes.

This behavior-focused line of research can be roughly subdivided into the categories: approaches that are based on the derivation and later re-identification of potentially malicious system-call-dependency (sub-)graphs [80, 54, 60], approaches that infer high-level semantics for system-call-dependency graphs [37, 36, 116],

and approaches that tackle the data flow aspects of potentially malicious behavior in terms of interaction with system resources [151, 54, 76, 84, 113, 114].

One of the first call-graph based approaches was proposed by Kolbitsch et al. [80] who introduced the idea of deriving call-graphs from known malware samples that represent dependencies between different runtime executions of system calls, their arguments, and return values and using them as behavioral profiles for later re-identification in unknown executables. This idea was later refined by clustering and deriving near-optimal graphs to increase precision [54], or anticipate malware metamorphism [85].

Although we also base on the construction and analysis of behavioral graphs, we base our graph generations on *quantitative data flows* between system entities rather than on dependencies between single system calls. As we showed in our evaluation the incorporation of quantitative data flow properties in our graphs has a significant impact on detection accuracy. Besides quantitative aspects we furthermore differ from clustering approaches as presented by Park et. al. [113, 114] in that we construct per-system rather than per-process data flow graphs which widens the detection scope to inter-process behavior.

A seminal work of the third large pillar of behavioral malware detection was presented by Christodorescu et al. [37]. The basic idea of these type of approaches is to give high-level semantics to observed malware behavior. Rather than matching behavior on a purely syntactic and structural level, these approaches try to extract and re-identify characteristic behavior at a semantic level. Follow-up work further enriched this idea with formal semantics for identified malware behavior [116]. The semantic perspective of these approaches typically leads to better resilience against more simple obfuscation like call re-ordering. This is because, although resulting in mutations of call-graphs and thus challenging normal call-graph based approaches, such obfuscation attempts do not change the semantic dimension of the malware behavior and therefore cannot easily trick semantics-based approaches.

The main commonality with this line of research is that we also base our analysis on graphs and give high-level semantics for specific of malware behavior patterns. However, in contrast to the aforementioned approaches we base the construction of our behavior graphs on the analysis of *data* rather than *control* flows. As we will later see, this abstraction increases the resilience against advanced obfuscation like permutations between semantically equivalent functionality, which is usually not given by such approaches [115, 6]. The data flow perspective, as abstraction from control flows, furthermore reduces the set of sub-graphs or properties we need to maintain in order to detect malicious activities, which has a positive impact on detection efficiency. Intuitively, this is because one data flow property often subsumes multiple control flow properties.

The last category of related work tackles the data flow or resource dependency perspective of malicious behavior. One of the most prominent examples for this family of approaches is the Panorama system of Yin et al. [151] that leverages dynamic data flow analysis for the detection of malicious activities. The basic idea of Panorama is to profile data flow behavior of malicious processes through fine-grained system-wide taint analysis and match it against manually specified data flow properties. Similarly, our approach uses system-wide data flow graphs to represent process and system behavior and match them against FOL data flow invariants. However, we make use of quantitative flow aspects to increase invariant specificity and thus detection precision. Moreover, we incorporate dedicated graph aggregation, simplification, and abstraction steps which helps to keep data structures lean and maintainable and employ a lean approximate data flow analysis using a user mode system call monitor that at average imposes less than 20 % overhead [146]. This results in comparably low performance overhead, especially when compared to expensive taint-tracking approaches like Panorama that yield performance overheads of up to 2000%. Furthermore, we mainly differ from data dependency graph based work like the one presented by Lanzi et al. [84], Park et al. [113, 114], and Elish et al. [47] in that we leverage *quantitative data flow* aspects for our analysis and have a system- rather than a program-centric scope.

In sum, the main difference between our work and related contributions is that to the best of our knowledge we are the first to leverage quantitative data flow aspects for the behavior-based detection of malware which, as we showed, yields increased detection accuracy.

4.1.5. Discussion and Conclusion

In this section we have proposed a novel approach to leverage quantitative data flow analysis and graph analysis for the detection of malicious activities. Through the specification of patterns in form of malware-specific quantitative data flow patterns, we can then identify behavior that relate to the presence of malware.

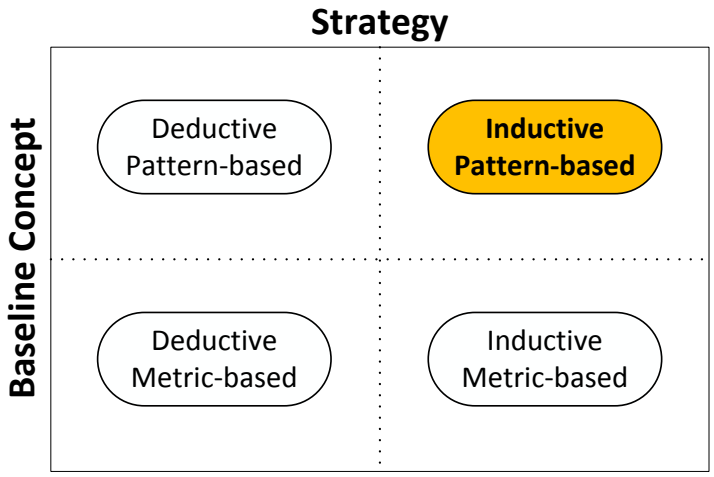
To demonstrate the practicability of our generic system model, we presented an exemplary instantiation for Microsoft Windows operating systems based on a prototype that builds on top of a user-mode monitor, capturing process activities in form of data flows induced by process calls to the Windows API.

Our evaluation on a large and diverse data set showed that our approach is able to effectively discriminate between malware and goodware with a good detection rate of up to 77% while maintaining a reasonably low false positive rate of less than 5%, when using quantitative properties. The evaluation results also indicate a fair efficiency with the average time to classify an unknown sample remaining under 23 seconds.

In sum, with this approach we could give a first answer to our baseline research questions *RQ1*, *RQ2*, and *RQ3* in that we were able to show that QDFGs can be used for behavior-based malware detection, that the consideration of quantities improves detection precision by more than 300%, and that our detection approach is reasonably fast.

Nevertheless, both, the detection effectiveness and efficiency is not optimal and for instance still yields a rather high number of false positives that, depending on the operational context, might be considered too high for real-world usage. This approach is thus more meant as an initial prototype to show the general feasibility of our basic detection concept. In the subsequent section we will extend it by means of more complex pattern retrieval and matching functionality to further improve effectiveness and efficiency.

4.2. Inductive Pattern-based Detection



4.2.1. Introduction

In the last section we have seen that we can operationalize our QDFG-based system behavior model for malware detection by specifying a set of patterns, i.e. graphs, that can be associated to typical malware behavior. By scanning QDFGs of to be classified samples for sub-graphs that are isomorphic to these patterns we can effectively discriminate between likely malicious and benign behavior and thus to some extent detect unknown malware. We further saw that by incorporating additional information about the specific quantitative data flow behavior of the analyzed samples we could significantly boost detection effectiveness.

While the evaluation of our deductive pattern-based detection approach thus already gave first answers to our main research questions *RQ1* and *RQ2*, the proposed concept of using a fixed set of deductively defined detection patterns still suffers from a couple of limitations when it comes to the real-world applicability of the approach. First of all, the achieved detection effectiveness is still too low for most operational settings; even when incorporating quantitative guard properties the false positive rate remained above 4%.

More problematically is the fact that the deductive pattern-based approach entirely relies on a comparably small fixed set of detection patterns, which makes it vulnerable to changes in malware behavior.

If a malware author for instance would learn about the used detection patterns he in theory could devise specific anti-analysis countermeasures and e.g. obscure the behavior and quantitative data flow properties of a malware to prevent it from matching the fixed detection patterns. This limits the generalizability and adaptability of the deductive pattern-based detection approach. Finally, as we saw in Section 4.1.3.3, the approach is comparably expensive in that it needs to perform a costly full-fledged sub-graph isomorphism and quantitative conformance check for every patter-QDFG combination which has a negative impact on its scalability.

To tackle these problems and explore the conceptual boundaries of pattern-based detection using QDFGs we follow our principal research methodology and propose an inductive extension of our pattern-based detection concept. The main difference of this inductive extension to the basic deductive pattern-based approach is that we automatically extract highly discriminative behavior patterns from corpus of QDFGs of known malicious and benign samples instead of manually defining a fixed set of patterns.

The most natural way of establishing such repositories of malicious behavior patterns is to employ some sort of data mining on corpus of known malicious and benign samples and look for recurring behavior that is more typical for the analyzed malware samples than for the benign ones. Considering that we use graphs, i.e. QDFGs, so represent system and process behavior, this essentially boils down to using graph mining algorithms for extracting characteristic behavior patterns. As patterns in our context are (sub-)graphs, for the sake of brevity we in the following use the terms *pattern* and *sub-graph* interchangeably.

Graph mining is a concept that traditionally is mainly used in bioinformatics to e.g. find characteristic molecule properties or for protein prediction. The core idea of graph mining is to determine interesting patterns that are shared by a large portion of the mining data set. Interesting in this context refers to the capability of the respective patterns to be able to effectively discriminating between different types of graphs, e.g. different molecule or protein groups in the bioinformatics domain or malicious or benign samples in our context.

So far, in the domain of malware detection, most approaches that make use of graph mining determine the utility of a pattern mainly from a frequency point of view [32, 66, 91, 30, 111, 70, 93]. This is, the level of utility of a pattern is determined depending on how often it appears in all analyzed malware samples, irrespective of its semantics or structural properties. In consequence, so far, graph mining in the context of behavior-based malware detection is mainly done using adoptions of popular frequency-based algorithms like AGM [64], gFSG [82], or GSpan [150].

Recent results from experiments in the molecule (graph) mining domain however indicated that frequency-based mining in many cases yields significantly less interesting and thus discriminative patterns than compression-based mining approaches [71].

Unlike frequency-based mining approaches that widely ignore the structure and complexity of patterns for determining their utility (i.e. likely discriminative capabilities), compression-based mining approaches do consider the structural complexity of a pattern candidate for determining its utility. Instead of only looking at the pure number of overall occurrences of a pattern, compression-based mining algorithms also consider the capability of the pattern to compress the graphs of the mining set. This is, a pattern that compresses well, i.e. covers a large portion of most graphs in the mining set, but overall occurs less frequently than another less-complex pattern with more limited compression capabilities, might still be more discriminative than the less complex but more frequent one.

To our knowledge, the utility of compression-based graph mining for malware detection has not yet been investigated and we see good reason to believe that the insights gained in the molecule mining domain also generalize to the malware detection domain. This assumption is substantiated by the results we obtained from a preliminary study where we applied a state of the art frequency-based mining approach [150] on QDFGs obtained from a large body of malware samples. The mined patterns, although in principle discriminative, almost entirely referred to rather simplistic behavior like reading certain system libraries or writing specific registry keys. Using such simple patterns for malware detection is somewhat problematic in that they a) likely are very sensitive to even slight changes in the behavior of the profiled malware families, b) for the same reason comparably easy to circumvent, and finally c) might miss important and more complex malware-specific behavioral specificities like e.g. self-replication.

Following these reasoning we thus decided to interweave our basic deductive pattern-based detection approach with advanced compression-based graph mining functionality to replace the originally static and manually set of detection patterns with automatically mined ones. To incorporate the quantitative information encoded in our QDFGs into the mining process we furthermore propose a graph compression concept that leverages quantitative data flow information for determining the utility of a pattern. Having already shown for the deductive pattern-based approach that the incorporation of such quantitative properties can significantly improve detection accuracy (see Section 4.1.3.2) we expected the incorporation of quantitative data flow information into the graph mining process to also improve detection accuracy.

In sum, the specific contributions of our inductive pattern-based approach can be summarized as follows:

- To our best knowledge we are the first to use *compression-based graph mining* for behavior-based malware detection using *quantitative data flow information*.
- We show that inductively obtained detection patterns outperform our deductively specified ones in terms of resulting detection accuracy.

- We show that patterns obtained using a standard frequency-based mining approach at average are less effective than the ones obtained from using our customized compression-based mining approach.
- We show that considering quantitative data flow properties for determining the utility of a pattern yields more discriminative patterns than binding pattern utility only to frequency and structural graph properties.

4.2.2. Preliminaries

Before diving into the technical details of our inductive pattern-based detection approach we first want to provide the reader with a few background information concerning the concept of graph mining which we deem necessary for the understanding of the remainder of this chapter.

Graph mining is a specialization of the more general concept of data mining. Traditionally, data mining mainly focused on extracting rules and regular patterns from unstructured or semi-structured data [141]. However, with the availability of multi-core systems and advent of disciplines like bio-informatics the interest in also extracting interesting patterns from structured data steadily increased in the past years. Structuring data in form of graphs is very common in computer science and has a very natural applicability to problems in chemistry, biology, and medicine. Extending the concept of data mining to extract patterns from structured graph data thus recently received quite some attention. In comparison to data mining on unstructured data, graph mining is a rather young domain with the first dedicated algorithms having been proposed in the early 90's [141]. We can roughly categorize graph mining algorithms along two dimensions: approximate vs. exhaustive and frequency- vs. compression-based.

In a nutshell, exhaustive graph mining algorithms evaluate all possible sub-graph that can be build from a set of to be mined graphs to isolate the ones that describe best the entire data set. This usually involves computing isomorphisms between each pattern, i.e. sub-graph candidates, and each graph of the to be mined set, which boils-down to the NP-complete sub-graph isomorphism problem [43]. Exhaustive graph mining algorithms, although by construction being ensured to find the optimal discriminative patterns, thus usually suffer from bad scalability and are very expensive to be run on huge and complex data sets.

Approximate or greedy graph mining algorithms in contrast try to avoid having to evaluate the entire search space and usually do not check all possible sub-graphs for being possibly interesting patterns. Instead, such approaches typically incorporate domain knowledge or structural heuristics into the search process to early prune parts of the search space that are unlikely to yield interesting patterns and thus reduce the numbers of necessary sub-graph isomorphism calculations.

As consequence of this heuristic reduction of the search space, greedy algorithms can not guarantee to provide optimal results in that they might miss potentially discriminative patterns from parts of the pruned search space.

The common goal of all data mining algorithms is to find *"interesting"* patterns in the training data; in the context of graph mining this relates to finding interesting sub-graphs in a set of training graphs. The concrete interpretation of the notion of interestingness, or *utility*, of a pattern being interesting heavily depends on the used mining algorithm. Frequency-based graph mining algorithms like GSpan [150] or AGM [64] bind the level of utility of a pattern solely on how often it occurs within the training data, also called the the *frequency* or *support* of a pattern. The complexity or other properties of the patterns themselves are usually not considered by frequency-based mining approaches.

Compression-based mining approaches like Subdue [71] or GBI [94] in contrast, while typically also considering pattern frequency, in addition take the structural complexity of a pattern into account for utility assessment. This is usually done by evaluating the factor of *graph compression* that can be achieved when condensing all isomorphic sub-graphs into one single node. The resulting compression factor, typically determined by the ratio between the complexity of the uncompressed and the compressed graph, then determines the pattern's utility. Correspondingly, a compression-based graph mining algorithm thus might sometimes favor a less-frequent but compressive pattern over a more frequently occurring one. Recent work in the domain of molecule mining indicates that correspondingly obtained pattern can be of higher interest than those extracted with purely frequency-based methods [71].

For the inductive extension of our pattern-based detection approach we thus adopted a greedy compression-based graph mining algorithm [71] which we expected to provide a good trade-off between effectiveness, i.e. yield highly discriminative patterns, and efficiency, in that it incorporates domain knowledge and advanced candidate selection to aggressively prune the to be evaluated search space.

4.2.3. Approach

Now that we have laid the conceptual foundations underlying our inductive pattern-based approach we are now set to discuss its technical specificities.

Being an extension to the previously introduced deductive pattern-based approach the inductive approach, at least for the initial steps, follows the same data processing procedure. The main differences to the initial deductive pattern-based concepts are: i) we incorporate a graph mining approach to extract arbitrary complex and large sets of highly discriminative detection patterns, ii) we substitute the simple rule-based matching and detection procedure of the deductive approach with a machine learning based matching concept that is capable of identifying complex relationships between different detection patterns, and iii) we perform the mining and matching on complexity-reduced process-centric reachability graphs instead of using the full QDFGs, which has a positive impact on efficiency.

In essence, the inductive extension of our pattern-based detection approach follows a classical data mining, or more general, soft computing rationale. Like for most behavior-based malware detection approaches, our core assumption is that we can detect new malware based on behavioral similarities to already known malware samples. More precisely, we use a graph mining algorithm to extract characteristic behavioral patterns from known malware QDFGs to define detection patterns that are capable of discriminating unknown malware and goodware with high accuracy. The inductive extension of the pattern-based approach thus in essence consist of the following steps:

3.1) *Data Retrieval*: To generate the input data for the mining step we use the same analysis infrastructure as our deductive pattern-based approach (see Section 4.1.2). However, instead of directly using the full obtained system behavior QDFGs for mining, we first prune them from all behavior that is not directly or indirectly related to the to be classified samples. This is, instead of using the full QDFGs we use the sub-graphs that we obtain from performing a graph reachability analysis on them and only maintain nodes and edges that are directly or indirectly reachable from the process node that relates to the analyzed malware or goodware sample.

3.2) *Pattern Mining*: On the obtained QDFGs we then run a compression-based graph mining approach to extract characteristic patterns, i.e. sub-graphs of known malicious QDFGs. The goal of this step is to establish a repository of graph patterns that capture the essence of the behavior of known malware that, following our baseline assumption, we expect to also appear in unknown malware with a high likelihood. The mined patterns then establish the basis for all subsequent training and detection steps.

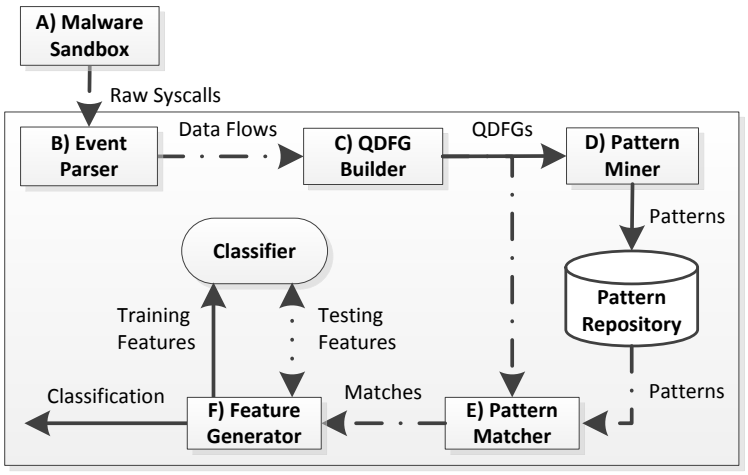


Figure 4.5.: High-level architecture

3.3) *Pattern Matching (3.3.1) and Classifier Training (3.3.2)*: As individual patterns in themselves most likely are not discriminative enough to accurately differentiate between goodware and malware, we then introduce a second learning step. For this we again match the mined patterns on the training set and record which patterns matched on which malware and goodware graphs. Using this information we then train a supervised classifier to infer complex relationships between a graph matching certain patterns and it being malicious or benign.

3.4) *Detection*: The actual classification of unknown samples is then done by again matching the mined patterns against the sample's process-centric reachability graph and then passing the obtained matching information to the trained classifier. Based on the similarity of the unknown sample's matching profile with matching profiles of the known goodware and malware samples from the training set, the classifier then classifies the unknown sample as benign or likely malicious.

Figure 4.5 depicts a high-level overview of the architecture of our approach, i.e. the conceptual components that realize the aforementioned training and detection steps. The solid arrows in the figure mark training activities, the dotted arrows refer to those that are only relevant for detection, and the semi-dotted lines denote activities that are relevant for training and detection.

In the following we will elaborate on the different steps in more detail and associate them with the respective architectural components.

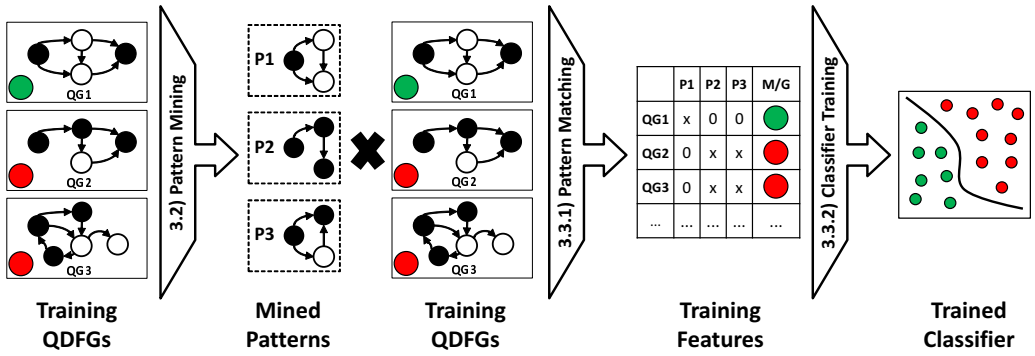


Figure 4.6.: High-level overview of complete training procedure

4.2.3.1. Data Retrieval

As mentioned before, the data retrieval process of the inductive pattern-based approach in wide parts resembles the one of our deductive pattern-based approach in that we use the same Malware Sandbox (*Component A*) and Event Parser ((*Component B*)) to generate the baseline QDFGs. However, different to the deductive pattern-based approach we for performance reasons do not use the full system behavior QDFGs for training and detection but rather introduce an additional filtering step into the the QDFG building step (*Component C*). This step filters out all behavior, i.e. respective edges and nodes, that are not directly or indirectly associated to the main malware or goodware process corresponding. The obtained process-centric reachability QDFGs then build the data basis for all subsequent mining, training, and detection steps.

4.2.3.2. Pattern Mining

Having a sufficiently large set of QDFGs that relate to known malicious and benign samples at hand, we are set to conduct the actual learning part. An overview of the complete learning procedure of the inductive pattern-based approach is depicted in Figure 4.6 and consists of a *Pattern Mining*, a *Pattern Matching* or *Feature Generation*, and a *Classifier Training* phase.

The first learning phase has the goal of extracting interesting patterns from the body generated malware QDFGs. As we want to use these patterns to later detect unknown malware, "interesting" in our context refers to how malware-specific a pattern is in the sense of it more likely capturing characteristic malware behavior than benign one. Subsequently we refer to this interpretation of a pattern being considered interesting whenever we talk about pattern utility.

We will later concretize this notion of pattern utility in Section 4.2.3.2 and Section 4.2.3.2 when we introduce the idea of graph compression.

As we capture and abstract from low-level behavior using QDFGs, the most natural way of obtaining the demanded highly characteristic patterns is to employ some sort of supervised graph mining algorithm on the labeled training data that we obtained in the previous step.

Most related malware detection and classification approaches that leverage graph mining on some soft of behavioral model do so following a frequency-focused rationale [36, 32, 66, 91, 30, 111, 70, 93]. This is, the utility of a pattern is determined by how frequent it appears in the training malware set and how seldom it appears in the goodware set. Properties of the pattern itself like e.g. its structural complexity in most cases are either completely ignored for the computation of the pattern's utility or only play a subordinate role.

While occurrence frequency for sure is a useful and important property to determine the utility of a pattern, we argue that by ignoring or at least not equally considering the structural aspects of the patterns themselves one does not make use of a lot of interesting information that might lead to extraction of even more interesting patterns. This insight is backed by results of experiments conducted in the context of molecule mining [71] that essentially indicate that frequent patterns are not necessarily also very interesting. More precisely, a few slightly less frequent but more complex patterns might be more interesting than many very frequently occurring but less complex ones.

We hypothesize that this also might hold for graph patterns in the context of malware detection. If we recall the basic operation principle of frequency-based mining algorithms like e.g. GSpan [150] typically operate (see Section 4.2.2) and consider that most malware for instance loads similar libraries or manipulate the same registry keys to e.g. ensure persistent execution, it becomes clear that employing frequency-based mining on malware behavior graphs likely yields very simple behavior patterns which might be too specific and too easy to circumvent.

Also, the pattern complexity intuitively has a direct impact on the computationally effort needed to conduct all possible isomorphism checks on sub-graphs of the QDFG. To see why, we consult the example depicted in Figure 4.7. Here the pattern *P2* consists of one black and two white nodes, connected by three edges. Looking at all possible sub-graphs of *QG4* we directly see that there exists only one 3-node sub-graph that has the same number of nodes of the required type as the pattern *P2*, which is a necessary pre-condition for node-induced colored sub-graph isomorphism. In this case we thus only need to conduct one isomorphism check to be sure whether and how often *P2* is contained in *QG4*. If *P2* would be less complex and for instance only consist of one black and one white node connected by one edge, there would be at least 6 sub-graphs of *QG4* with the same node count and type as *P2* that thus potentially could match *P2*.

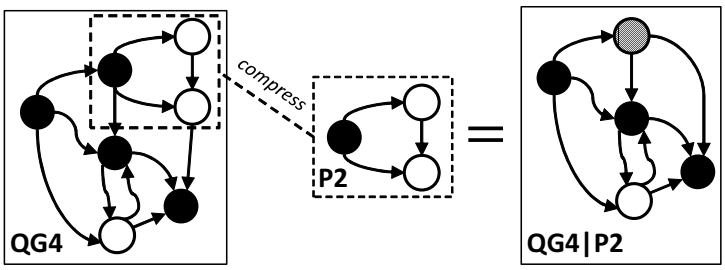


Figure 4.7.: Example: Graph *QG4* compressed by pattern *P2*.

In this case we would thus need to perform six isomorphism checks instead of only one with a likely negative effect on the overall detection costs.

Although this example of course does not generalize to all possible matching scenarios and the mentioned scaling effect highly depends on the structure of the mined patterns and the to be evaluated graph, it nevertheless motivates the utility of pattern-centric graph compression to early prune the search space from patterns of likely little relevance. Moreover, as we will later see, this early pruning of the search space tremendously improves efficiency as it reduces the number of to be conducted expensive sub-graph isomorphism checks.

For this reason, the *Pattern Miner* (*Component D*) that conducts the pattern mining step of our approach implements a compression-based graph mining algorithm instead of a frequency-based one. More precisely, we implemented a variant of the Subdue graph mining algorithm proposed in [42] which we customized to our needs and whose details we will elaborate on in the following.

Unlike exhaustive frequency-based algorithms that scan the entire search space for pattern evaluation, Subdue is an approximate algorithm in that it only considers those parts of the pattern search space that are likely to yield interesting patterns. By construction such approximate algorithms in most cases are faster than approximate ones but might miss some patterns. The choice for using an approximate algorithm thus introduces some indeterminism to the training process. However, as we will later show in Section 4.2.4, this choice is well justified as it yields superior effectiveness with very competitive efficiency.

For brevity's sake we in the following use the terms *positive examples* (T^+) to refer to the malware QDFGs in our training data set and *negative examples* (T^-) to refer to goodware training QDFGs. In the following we describe the details of the mining process as well as the scoring functions used for assessing pattern utility, following the algorithm sketched in Listing 1.

Algorithm 1 Abstracted Mining Algorithm

procedure MINEPATTERNS
  $PC_0 \leftarrow \emptyset$; $i \leftarrow 0$
  for each t^+ in T^+ **do**
    $PC_0 \leftarrow ((\{init\_node(t^+)\}, \emptyset, G.A, G.\lambda), 0)$
  while $i \leq k$ **do**
    $PC_{i+1} \leftarrow PC_i$
    for each p in PC_i **do**
      $p' \leftarrow extend(p)$
      $PC_{i+1} \leftarrow PC_{i+1} \cup \{(p', S(p'))\}$
    $PC_{i+1} \leftarrow \lceil PC_{i+1} \rceil_n$
    $i \leftarrow i + 1$
  return P_k

Node selection To determine the set of most interesting patterns within the the malware part of our training set we first have to make a suitable choice for defining the root nodes of the prospective sub-graph candidates. As we want to mine process-centric detection patterns, the first step of the mining algorithm is thus to determine all process nodes of the malware QDFGs in the training set. Recall that in our data flow model process nodes refer to the monitored processes of a system, including the processes that loaded the executed malicious binaries together with their descendants. As these process nodes refer to the only active entities in a system, and we identify malware based on observed behavior, it is reasonable to initialize prospective sub-graph candidates with process nodes as root nodes. Furthermore, we focus the sub-graph mining to first consider the direct proximity of the process nodes that loaded the initial malicious binaries, that is we restrict the set of initial pattern nodes to the set of all initial processes in the entire training set. In our training data set we respectively mark all process nodes that loaded the analyzed malicious binaries with a special $\lambda(n, init)$ property.

The initial set of pattern candidates $PC_0 = \mathcal{G} \times \mathbb{R}$ is thus determined by creating one singleton pattern per initial process node in the set of all malicious training QDFGs T^+. We index the pattern candidate sets PC_x to denote the corresponding algorithm iteration it was generated in. As each pattern in the candidate set will later be assessed regarding its overall discrimination utility, each element of the set PC is a tuple of a pattern and its utility score.

$$PC_0 = \bigcup_{G \in T^+} \begin{array}{l} \{(G', 0) = ((\{n \in G.N\}, \emptyset, G.A, G.\lambda), 0) \mid \\ G.\lambda(n, type) = Process \wedge G.\lambda(n, init)\} \end{array}$$

Now that we have established an initial population of pattern candidates that only contain one process node and no edges yet, we are set to enter the pattern evaluation and evolution loop. Let us reiterate that each pattern $p \in PC_0$ consists of an initial process node that usually exists multiple times in the training data set, but usually their neighborhoods in the different QDFGs might (and usually does) differ. After some preliminary evaluations on smaller data sets we further decided to only use the 1%, with respect to the subsequently discussed scoring functions best-performing initial process nodes to seed the initial pattern candidates to further improve mining efficiency.

As reasoning about the compression capabilities of one-node-patterns does not make much sense (see Section 4.2.3.2 and Section 4.2.3.2), we needed to come up with an auxiliary way of determining the utility of the initial pattern candidates. To determine the utility of those singleton-patterns we, depending on the subsequently used pattern scoring function, thus either considered the size of the directly connected edges of the respective process nodes or their edge degree, which then defines the initial utility of the respective patterns in PC_0.

After having defined the initial set of pattern candidates we then extend them in each possible direction by adding an additional node and edge from their respective neighborhood in the training QDFGs. This yields a new set of extended pattern candidates that then together form the scored pattern candidate set PC_i of the respective algorithm iteration i. We then filter the pattern set of all duplicates and condense all isomorphic patterns into one surrogate pattern.

At this point, the consideration of pattern candidate structure comes into play. A purely frequency-based mining algorithm would now determine the value of a pattern only based on how often it appears in the positive training graphs (and does not appear in the negative examples). In contrast, our pattern utility assessment strategy considers both, the relative frequency and structure of a candidate pattern. This is, for each pattern candidate $p \in \mathcal{G}$ we determine the pattern's utility through a scoring function S that computes the pattern's utility as real number $(p, S(p)) \in PC_i$. We consider two different pattern scoring functions $S = S_{MDL} \cup S_{MDC}$. The *Minimum Description Length* (*MDL*) scoring function S_{MDL} is the standard scoring function of the original Subdue algorithm and considers both, the frequency a pattern occurs within the training data set and the complexity of the pattern. The *Maximum Data Compression* (*MDC*) scoring function S_{MDC} catches up the same basic idea of considering structural properties to determine pattern utility but further extends it by also considering quantitative data flow properties encoded in the QDFGs. We will discuss the details of those scoring functions in Section 4.2.3.2 and Section 4.2.3.2.

After having evaluated the utility of each pattern candidate on the entire training set we sort the resulting pattern candidate set w.r.t. the pattern scores in descending order and only retain the n best patterns for the next iteration.

As we thus only consider the n best-performing patterns of a pattern set PC_i for computing the patterns set PC_{i+1} of the next iteration, we essentially perform a heuristic beam search, where n is the size of the search buffer. By construction we thus only follow the best-performing extension branches of the initial singleton-node patterns which tremendously cuts down the algorithm's search space.

This process is then repeated until a defined maximum pattern complexity k is reached. As a pattern is extended by an edge in each iteration, the maximum number of iterations to be conducted directly limits the maximum allowed pattern complexity. In the end, after termination, the algorithm only returns the n globally best-performing patterns. Note that the members of the final pattern set can be of different complexity as in some situations simple patterns can outperform more complex ones and vice versa. Finally, after termination of the mining algorithm, the *Pattern Miner* (*Component D*) dumps the final set of scored patterns to the *Pattern Repository* for the use by subsequent training and detection steps.

To speed up the computation of the pattern expansion and evaluation step we could easily distribute the expansion of the different initial singleton pattern instances among different physical processes. As expansion and evaluation of the different instances are independent of each other we can thus almost arbitrarily parallelize the algorithm, even among different machines in a cloud or grid setting. For this, the pattern miner component spawns a new process for each initial singleton pattern instance and continues to expand it.

After all expansions have finished, the main component of the pattern miner simply combines the results of all processes and filters out the duplicates. Even though there might be an overlap between pattern candidates in different processes during the expansion, we have found out that it is less computationally expensive to remove the duplicates at the end rather than running more complex process synchronization mechanisms that avoid redundant pattern exploration (such as those suggested by original Subdue authors [56]). Using this distribution paradigm, the parallelization of the mining approach in principle is thus only constrained by the number of considered initial singleton pattern instances and available computing resources. Still, independent of the employed scoring function and parallelization, determining the utility of a pattern candidate implies to evaluate its occurrence within the positive and negative training graphs.

Checking the presence of a pattern $p = (N, E, A, \lambda) \in \mathcal{G}$ in a training QDFG $G = (N', E', A', \lambda') \in \mathcal{G}$ boils down to the node-induced sub-graph isomorphism problem. This is, a pattern p is *sub-graph isomorphic* to G, i.e. $p \cong_s G$, iff:

1. there exists a sub-graph g of G, i.e. $g \subseteq G$, with:

   - $\exists g = (N'', E'', A'', \lambda'') \in \mathcal{G}$
   - $N'' \subseteq N'$

- $E'' \subseteq (E' \cap (N'' \times N''))$
- $\lambda'' \subseteq \lambda'$

2. and p is isomorphic to g, i.e. $p \cong g$, i.e.:

- $\exists f : N \to N''$ with f being bijective
- $\forall n_1, n_2 \in N'' : (n_1, n_2) \in E$
  $\iff (f(n_1), f(n_2)) \in E''$
- $\forall n \in N, \forall a \in A : \lambda(n, a) = \lambda(f(n), a)$

Using this definition we introduce the function $sg : \mathcal{G} \times \mathcal{G} \to 2^{\mathcal{G}}$ that returns all sub-graphs g of G that are isomorphic to a pattern p, i.e. $sg(p, G) = \{g \subseteq G | p \cong g\}$.

Conducting sub-graph isomorphism calculations is the computationally most expensive step in our approach as the underlying problem is known to be NP-complete [43]. In fact, the average-case computational complexity of the employed VF2 algorithm for computing the sub-graph isomorphism between a given pattern candidate and training graph is only quadratic in the maximum complexity of the training graph and the pattern candidate [43]. Unfortunately, the pattern expansion in all possible directions, at least in theory, demands that we check the sub-graph isomorphism relation for each possible pattern-candidate training-graph pair. The overall theoretical worst-case computational effort for one entire mining run thus unfortunately is exponential in the number of expansion iterations, i.e. is in $\mathcal{O}(v^2 \cdot (|T^+| + |T^-|) \cdot |PC_0|^k)$ with v being the number of nodes of the most complex training graph, i.e. $v = max(\{\|g.N\| \, | \forall g \in T\})$.

In reality, it is not possible extend each pattern candidate to an arbitrary depth. Therefore, the exponential factor in reality is rather c^k with $c \ll |PC_0|$. Fortunately, the maximum number of expansions k is constant and usually chosen rather small, which further reduces the average case complexity. While we have little influence on the maximum training graph complexity v, we can further cut down the overall complexity of the algorithm by only considering a smaller sub-set of the entire training data set for determining the pattern scores. To this end we introduce the approximation ratio σ, which describes the fraction of training graphs that are considered for the isomorphism checks. Each time we need to evaluate the utility of a pattern, i.e. we need to calculate the isomorphic sub-graphs in the training set $T = T^+ \cup T^-$, we only consider a randomly determined sub-set of size $\sigma \cdot |T|$.

By sub-sampling the training set and thus not considering all training graphs for assessing the utility of a pattern we certainly compromise generalizability of the respectively computed pattern scores and thus might wrongly prune potentially interesting pattern candidates.

However, our preliminary evaluations revealed that a not too aggressive down-sampling of the training set barely affects the overall detection accuracy but significantly improves mining efficiency. We explain this by most malware samples from the same family behaving fairly similar. Down-sampling a big enough training data with a fairly uniform distribution of malware families thus mainly removes redundant behavior, resulting in little effect on the respectively computed pattern utility scores.

After having discussed the main steps of our mining phase we now will elaborate on the details of the scoring function, i.e. pattern utility computation.

Minimum Description Length (MDL) In the context of compression-based graph mining scoring functions are used to express the utility of a pattern in terms of describing well a bigger set of graphs. This is, a scoring function assigns a concrete semantics to the notion of a pattern *"describing well"* a graph data set.

The standard scoring function of the employed subdue mining algorithm is called *Minimum Description Length* (MDL). The basic idea behind MDL goes back to the work of Rissanen [120] who, in essence, postulated that the optimal description for a set of data items is the one that encompasses as many and complex commonalities within the data set as possible. This is, a optimal description compresses the data set as good as possible.

Applied to graph mining this means that a pattern is interesting, i.e. describes well a set of graphs, if by removing it from each training graph (i.e. removing all isomorphic sub-graphs) the cumulative complexity of the graph set is reduced. If we encode the structure of a graph in bits, a good pattern hence compresses the full graph set so that its encoding after compression needs less bits than before.

An, with respect to a set of graphs G optimal pattern p thus minimizes the term $\sum_{g \in G} DL(p) + DL(g|p)$, where $g|p$ denotes the graph we obtain when compressing g with p, and DL is a function to encode the structure of a graph (e.g. its edges and nodes) in bits. DL is defined as: $DL(g) = DL_N(g) + DL_E(g)$, with $DL_N(g)$ being the number of bits required to encode the nodes and node labels of a QDFG g, and $DL_E(g)$ the number of bits needed to represent the node interconnections.

Compressing a graph G with a pattern p is then defined by:

$$
\begin{aligned}
G|p = (\ & (G.N \setminus N_{G|p}) \cup \{n' \notin G.N\}, (G.E \setminus E_{G|p}) \\
& \cup \left\{(n_1, n') | \forall (n_1, n_2) \in G.E : n_1 \notin N_{G|p}\right\} \wedge n_2 \in N_{G|p} \\
& \cup \left\{(n', n_2) | \forall (n_1, n_2) \in G.E : n_2 \notin N_{G|p}\right\} \wedge n_1 \in N_{G|p}, \\
& G.A, G.\lambda\ ) \\
& \text{with } N_{G|p} = \bigcup_{g' \in sg(p,G)} g'.N \text{ and } E_{G|p} = \bigcup_{g' \in sg(p,G)} g'.E
\end{aligned}
$$

The actual binary encoding of the nodes of a QDFG G and their connection via edges is then done as follows:

- $DL_N(G) = |G.N| * \log_2 |G.N| + |G.N| * (\log_2 |G.A| + \log_2 |codom(G.\lambda)|)$ encodes the set of nodes N of G and their respective labeling functions.
- $DL_E(G) = \sum_{e \in G.E}(2 * \log_2 |G.N|)$ encodes the edges of a QDFG as list of tuples of node references. As for the node-induced sub-graph isomorphism check we do not use edge labels we do not need to encode them.

The actual graph compression is then done by replacing all instances of a pattern p in g, i.e. all to p isomorphic sub-graphs in g, with a single node while retaining the original edges. A simple graph compression example is depicted in 4.7.

Finally, for calculating the actual score of a pattern we do not only to consider its compression capabilities on the positives graph samples but also its compression effect on the negative examples. A good pattern in this sense should thus highly compress the positive examples while barely or not at all compressing the negative ones. The final MDL scoring function is thus defined by:

$$S_{MDL}(p) = \frac{\sum\limits_{g^+ \in T^+} DL(g^+) + \sum\limits_{g^- \in T^-} DL(g^-)}{DL(p) + \sum\limits_{g^+ \in T^+} DL(g^+|p) + \sum\limits_{g^- \in T^-} (DL(g^-) - DL(g^-|p))}$$

Maximum Data Compression (MDC) The MDL scoring function already allows us to consider more of the information contained in the training QDFGs set for mining than a purely frequency-based mining approach would do. Still we did not yet make use of all potentially discriminatory aspects embodied in the training QDFGs, i.e. we did not yet use the quantitative data flow aspects of the QDFGs. Following the main hypothesis of this thesis of this quantitative information potentially yielding leading to better detection accuracy, we thus came up with an extension of the originally only structural notion of compression to also consider the quantitative data flow information encoded in the training QDFGs.

To this end we propose the *Maximum Data Compression* (MDC) pattern scoring function. MDC like MDL does consider both, pattern frequency and pattern complexity. However, unlike MDL that only considers the *structural complexity* of a pattern to assess its utility, MDC also considers the cumulative *data flow complexity* of a pattern candidate. This is, instead of measuring how many edges (and nodes) get removed by compressing the training graphs by a pattern candidate, we measure how much data flows get compressed by removing those edges.

More precisely, we calculate how big the fraction of the total amount of data flows associated to the edges removed by compressing a training QDFG with a pattern candidate is with respect to the total amount of data of all edges of the uncompressed graph.

The MDC score of a pattern $S_{MDC}(p)$ thus calculates the relative amount of data that is encompassed by the edges removed from a all training graphs when removing all sub-graphs from them that are isomorphic to the pattern. To this end we introduce the function $QC(p, g)$ that returns the fraction of data flows that is removed by compressing a QDFG q with a pattern p, formally defined by:

$$QC(p, g) = \frac{\sum_{e \in g.E \setminus (g|p).E} g.\lambda(e, size)}{\sum_{e \in g.E} g.\lambda(e, size)}$$

Assigning a positive score to patterns that data-wise compress well the positive examples and penalizing the data-wise compression of negative examples, the complete MDC scoring function is thus defined by:

$$S_{MDC}(p) = \sum_{t^+ \in T^+} QC(t^+, g) - \sum_{t^+ \in T^-} QC(t^-, g)$$

In sum, the inductive pattern-based mining approach features two distinct scoring functions to assess the utility of mined pattern candidates. The S_{MDL} scoring function only considers the structural complexity of a pattern candidate to evaluate its expected utility, whereas the S_{MDC} scoring function also considers its inherent data flow complexity.

4.2.3.3. Pattern Matching and Classifier Training

Now that we have established a base of discriminative malware behavior patterns we in theory could directly use them for the classification of unknown samples and, like the deductive pattern-based approach, flag a sample as being malicious, if it contains one or more of the mined malware patterns. However, at this point we have to again recall how these patterns were generated. As we use soft-computing algorithms, i.e. a graph mining algorithm, and several approximation and pruning steps we can not be entirely sure that the mined patterns in themselves are discriminative enough. This in particular means that there is for instance a non negligible risk of the mined malware patterns also appearing in goodware samples which, following such a naive detection strategy, would likely lead to high amounts of false positive classifications.

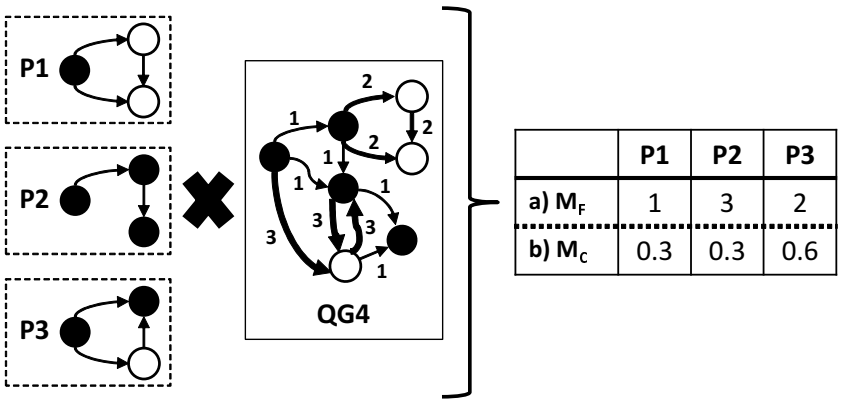


Figure 4.8.: Example for matching procedure.

This observations, together with the insights gained from evaluating the comparably simple matching strategy of our deductive pattern-based approach, suggests that implementing a naive pattern matching strategy that only looks for the existence of a malware pattern in unknown samples likely would yield sub-optimal detection accuracy.

To this end we introduce a second learning phase to our approach that anticipates the potential presence of malware patterns in goodware. The core idea of this second learning phase is to train a classifier on the information obtained from matching the mined malware patterns on all malware and goodware graphs from the training set. By this the classifier can learn more complex dependencies between the occurrence of different patterns in the graph's being malicious or benign. This to some extent compensates the issue that the mined patterns do not necessarily only appear in malware QDFGs. In the following we elaborate on the *pattern matching* step that yields the feature on whose basis we then train a supervised machine learning classifier in the *classifier training* step.

Pattern Matching The mined patterns together with their assessed utility so far only give us an aggregated view on the distribution of occurrences on the different malware and goodware samples in the training graphs. Unfortunately, in order to build more complex detection models that relate the occurrence of different patterns in QDFGs to their respective class we need more precise pattern matching information. To this end we thus again evaluate all mined patterns on all (positive and negative) examples in the training set. This is done by the *Pattern Matcher* (*Component E*) that for each training graph searches for all sub-graphs that are isomorphic to the mined patterns.

More precisely, the pattern matcher provides us with a mapping between the mined patterns and the respective isomorphic sub-graphs in the training graphs. The matching results then allow us to for each pattern training-graph pair record a) how many sub-graphs in the training example where isomorphic to the pattern (*Frequency Match*), and b) which fraction of the overall data flows captured by the training graph can be "compressed" by removing all matching sub-graphs from it (*Compression Match*).

The matching information is then generated by conducting sub-graph isomorphism checks between all patterns $p \in PC_k$ and all training graphs $g \in T$ and then evaluating the following functions $M := M_F \cup M_C$ on the results:

- *Frequency Match*: $M_F(p, g) = |sg(p, g)|$
- *Compression Match*: $M_C(p, g) = QC(p, g)$

While the frequency match function is meant to be used in conjunction with patterns that were mined using GSpan and MDL, the (quantitative data) compression matching is the natural dual to the MDC scoring function.

Figure 4.8 depicts the outcome of the two matching functions when evaluating three different patterns on a simple training graph. As we can see from the example, the scoring functions applied on the same matching scenario do not necessarily always concur in the distribution of weights of the different matched pattern. While the frequency matching in this simple example considers the pattern $P2$ the most important one and thus assigns it the highest value in the result row, the compression matching function considers $P3$ to be the most important pattern. This means that in fact classifiers trained using different matching functions can come to different classification conclusion when facing an unknown sample.

Classifier Training After having matched all mined patterns on the positive and negative training graphs we now have all ingredients for training a supervised machine learning classifier. For this we organize the matching information obtained in the previous step in a training feature matrix, which is done by the *Feature Generator* (*Component F*). Each column of this table captures the matching information for one graph t_i of from the training set T. Each element of the column thus is obtained by evaluating one of the previously mentioned matching functions for one specific pattern p_i from the final mined pattern set PC_k. As we our training set is labeled in the sense of each training QDFG representing the behavior of one known malware or goodware sample we can label the each feature vector with the *class* (benign vs. malicious) of the respective training graph.

The training data generation function is then defined by:

$$gen(PC_k, T) := \begin{bmatrix} M(p_1, t_1) & \cdots & M(p_k, t_1) & class(t_1) \\ \vdots & \ddots & \vdots & \vdots \\ M(p_1, t_{|T|}) & \cdots & M(p_k, t_{|T|}) & class(t_{|T|}) \end{bmatrix}$$

On the so obtained training data we then finally train a standard supervised machine learning classifier. Preliminary evaluations on training data obtained from smaller training sets with different supervised machine learning algorithms indicated that meta learners using decision trees, i.e. RandomForest [18] or ExtraTrees [58] yielded particular good and stable classification results. We thus chose to use a standard RandomForest algorithm as classifier for our approach.

In sum, the final classification model thus consists of the trained classifier along with the set of detection patterns that were used for training.

To give an idea on how a respectively trained classifier in principle looks like, Figure 4.9 depicts an example of a classification model that we obtain from using a simple decision tree algorithm for training. Note that this is a deliberately simplified example as for the actual classification models we use more elaborate meta-classifiers like RandomForest instead of simple decision trees.

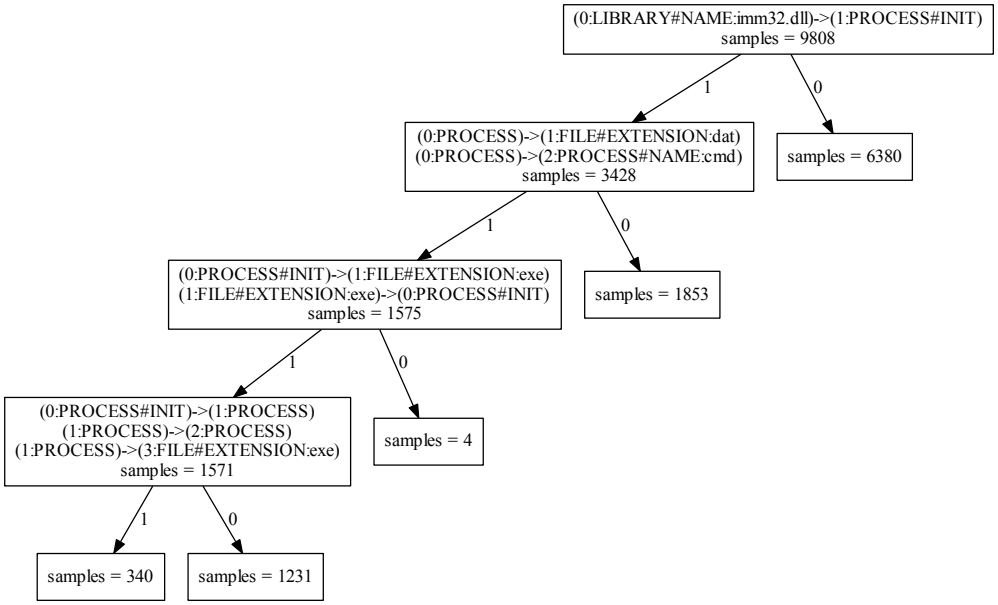


Figure 4.9.: Example for generated classification model

4.2.3.4. Detection

Having mined a set of discriminative patterns and trained a supervised classifier on them, the detection phase of our approach is now rather straight forward.

To classify an unknown sample we first follow the same initial steps as done for the training phase. This is, we take the set of detection patterns contained in a previously generated classification model and match them against the QDFG of the to be classified sample. Just as described in Section 4.2.3.3 we then convert the matching results into a feature vector, using the same matching function that also was used for generating the training features the classifier contained in the classification model was trained on. Finally, we pass the obtained feature vector to the classifier which then returns a classification proposal based on the similarity of the matching profile with matching profiles of known malware and goodware from the training set.

4.2.4. Evaluation

To evaluate the effectiveness of the inductive extension of our pattern-based detection approach used the same evaluation setting and data set as used to evaluate the deductive approach (see Section 4.1.3).

4.2.4.1. Effectiveness

For evaluating the effectiveness of our approach we were mainly interested in investigating two approach-specific research hypotheses:

(H0) Mined pattern significantly outperform the manually defined ones of the deductive approach in terms of detection accuracy.

(H1) Using compression-based graph mining for malware pattern extraction yields better malware detection accuracy then using frequency-based mining.

(H2) Quantitative data flow compression of patterns yields better detection accuracy then structure-only compression.

For investigating hypothesis *H1*, which we consider the main hypothesis that motivates this work, we aimed at comparing the detection effectiveness of our approach with other detection approaches that make use of frequency-based graph mining [36, 32, 66, 91, 30, 111, 70, 93].

Unfortunately, for most of these approaches the used implementations and data sets were either not publicly available or the contacted authors could not provide us with operational prototypes. For our experiments we thus had to fall back to re-implementing the core mining concept shared by most of the approaches.

This is, at least for the ones that mentioned the used graph mining algorithms we could compare the detection performance performance by substituting our compression-based mining component with a component implementing the respective mining algorithm. A closer look at related work that uses graph mining revealed that the frequency-based GSpan[150] mining algorithm by far was the most commonly used one [66, 91, 111, 70, 93] and most other used algorithms were structurally very similar to GSpan. Finally, another major reason for us to (at least partially) re-implement related approaches instead of directly comparing the numbers presented in the respective papers is given by the well-known fact that machine learning based approaches can be highly sensitive to the number and nature of used training samples. A direct comparison of numbers produced on different evaluation data sets would thus likely lead to biased conclusions.

For assessing hypothesis *H1* we thus evaluated the inductive pattern-based detection approach on our baseline evaluation data set, once using our compression-based pattern mining component for pattern utility evaluation, and using a publicly available implementation[1] of the frequency-based GSpan algorithm that was also used by many of the the mentioned related approaches. The individual evaluations were then conducted through typical 10-fold cross validation experiments. For each fold of the experiment we trained the approach on 90% of the evaluation data, i.e. mined interesting patterns and trained the final classifier on this set and used the so resulting detection model to classify the remaining 10% of the data.

In total we evaluated the following combinations of mining algorithms and matching functions: a) frequency-based mining with GSpan and frequency matching; b) compression-based mining with MDL and frequency matching; c) compression-based mining with MDC and (data) compression matching.

Malware quite often uses randomized names for dropped files. Extracted patterns that use the full name of a file would likely be too restrictive and only match very few malware instances. To thwart this issue we thus conducted all aforementioned experiments using only the file extensions part of the file node labels for label equivalence matching instead of the full file names.

To avoid biasing the comparison due to unequal baseline data sets we further set the sub-sampling ratio δ of our mining algorithm and configured our GSpan wrapper to only use 25% of the training part of each fold for training. Considering that this sub-sampling was done randomly we furthermore repeated each cross validation experiment 10 times to weed out noise and cut out random side-effects.

[1] https://www.cs.ucsb.edu/~xyan/software/gSpan.htm

To express the aggregated effectiveness of the approaches, we computed the following quality metrics. True positives (TP) refer to malware samples (MW) that have been correctly classified as malicious, true negatives (TN) to goodware samples (GW) that were correctly classified as benign, false positives (FP) to goodware samples incorrectly classified as malicious, and false negatives (FN) malware samples that were mistakenly labeled as benign:

- *Area under ROC Curve* (*AUC*): The AUC is the integral of the area enclosed by the function graph that we get when plotting the achievable true-positive rate for different false-positive rate thresholds δ. A point on this curve can thus be interpreted as the maximum achievable true positive rate that can be achieved when accepting a certain number of false positive classifications. The AUC thus gives us an idea of the best overall performance of an approach, with 1.0 being the best achievable AUC.

- *Best-case Detection Rate* (*BDR*): With this we capture the best-case true positive rate that can be achieved when fixing the maximum acceptable false positive rate to a threshold of 0.5%. In other words, the BDR is the value of the ROC function at 0.005. While we consider this threshold reasonable in the operational context of a medium-size company, the threshold value essentially is arbitrary. Depending on protection goals and the amount of to be classified samples, 0.5% false positive classifications might still be considered too high, or even too restrictive. The rationales of our threshold choice are based on discussions with domain experts and we thus deem this threshold useful to ensure a fair operational comparison of settings and approaches. In contrast to the AUC that, although better capturing the overall quality of a classifier, does not have an obvious operational interpretation, the BDR is better suited to express operational effectiveness. In a medium-sized company environment with 1000 to be classified email attachments per day this for instance would translate to the question of how many malicious attachments we can correctly identify as malware when accepting an upper bound of 5 emails wrongly put into quarantine due to incorrect classifications.

The average results and standard deviations of the experiment runs are depicted in table 4.2. As we can see, the experiment results support the confirmation of our hypothesis *H1* in that, at least on our evaluation data set, the best compression-based mining approach outperformed the frequency-based mining using GSpan.

More precisely, we at least perform $\frac{AUC_{MDC(M_C)}}{AUC_{GSpan(M_F)}} = \frac{0.988}{0.952} \approx 4\%$ better in terms of AUC when using data compression-based mining instead of frequency-based mining. Interestingly, at least concerning the overall effectiveness in terms of AUC, structural compression-based mining seemed to perform worse than frequency-based mining with GSpan.

	a) $GSpan(M_F)$	b) $MDL(M_F)$	c) $MDC(M_C)$
Avg. AUC	0.952	0.946	0.988
(Std. Dev. σ)	(0.013)	(0.013)	(0.018)
Avg. BDR	0.157	0.368	0.957
(Std. Dev. σ)	(0.069)	(0.149)	(0.042)

Table 4.2.: Average effectiveness

Looking at the individually achievable detection rates when fixing the maximum acceptable false positive rate to 0.5% (BDR), the effectiveness differences between the approaches becomes even more apparent. Concerning the BDR, the quantitative data flow compression-based yielded more then $\frac{BDR_{MDC(M_C)}}{BDR_{GSpan(M_F)}} = \frac{0.957}{0.157} \approx 600\%$ better results than the frequency-based GSpan approach. This means, in particular when targeting low false positive classification rates, our quantitative data flow compression-based mining approach seems to significantly outperform frequency-based mining (*H1*).

If we look at the individual differences in effectiveness between the structural compression-based mining experiments and the experiments where we considered quantitative data flow aspects for mining and matching, we can furthermore see that quantitative data flow compression-based mining at average yields $\frac{MDC(M_C)}{MDL(M_F)} = \frac{0.988}{0.946} \approx 5\%$ better overall effectiveness, i.e. AUC, than structural compression based mining. Again looking at the effectiveness for low false positive rates, data compression-based mining yields $\frac{BDR_{MDC(M_C)}}{BDR_{MDL(M_F)}} = \frac{0.957}{0.368} \approx 260\%$ better detection rates than when only considering structural compression.

These findings confirm our hypothesis *H2* in that considering quantitative data flow aspects for mining indeed seems to improve the quality of mined patterns and overall accuracy of respectively devised detection models. They further confirm our hypothesis *H0* in that at least the quantitative compression mining results yielded better accuracy than the deductive pattern-based approach. Furthermore, these findings again positively answer our main research questions *RQ1* and *RQ2* in that we again successfully operationalized our QDFG-based system model for highly accurate malware detection and in particular showed that the consideration of quantitative data flow properties significantly improves classification accuracy.

	a) $GSpan$	b) MDL	c) MDC
Training Time (Std. Dev. σ)	13.20s (0.10s)	4473.33s (754.35s)	133.98s (6.66s)

Table 4.3.: Average training efficiency

4.2.4.2. Efficiency

As our inductive pattern-based detection approach in contrast to our deductive pattern-based one consists of a computationally independent training and a detection phase, we also separately consider the computational overhead induced by the training and by the detection phase. Whereas the mining and training phase in principle only has to be conducted once and, as discussed in Section 4.2.3.2 can be offloaded to a powerful multi-core server, the detection phase needs to be re-run for every to-be classified sample.

The training time includes the computational effort for mining and scoring a set of detection patterns, as well as training a supervised classifier on them. The detection phase consist of matching the patterns against the testing graphs and evaluating the obtained feature vector on the trained classifier.

Considering that GSpan uses graph encoding and tries to avoid expensive isomorphism checks while our compression-based approach heavy relies on graph-isomorphism verification, we expected the frequency-based GSpan approach to outperform ours in terms of efficiency. In sum we were thus expecting that the gained effectiveness improvement comes at the cost of efficiency. Table 4.3 summarizes the average training and mining times of the different approaches. As we can see, the experiment results confirmed this assumption in that the frequency-based mining with GSpan was more than one order of magnitude faster than the best compression-based mining experiment. Concretely, frequency-based mining using GSpan was almost $\frac{T_{MDC}}{T_{GSpan}} = \frac{133.98s}{12.20s} \approx 11$ times faster than compression-based mining using MDC and more than $\frac{T_{MDL}}{T_{GSpan}} = \frac{4473.33s}{12.20s} \approx 366$ times faster than MDL. We explain the significant difference in performance between MDL and MDC with the same argument as we explain their effectiveness difference: likely MDC is better able to early prune useless pattern candidates from the search space and thus needs to perform significantly less isomorphism checks than MDL.

For assessing the detection overhead induced by the different approaches we measured the overall time consumed for matching the patterns of a detection model against unknown QDFGs of different size. Recall that, unlike the deductive pattern-based approach that uses the entire graph for classification, the inductive pattern-based approaches employs some graph pruning steps and only uses reachability graphs for training and detection (see Section 4.2.3.1).

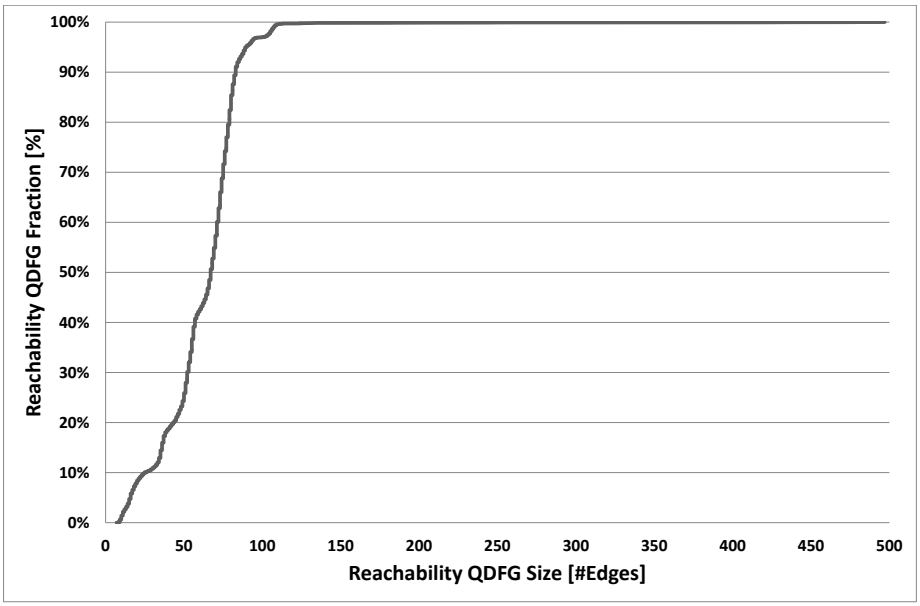


Figure 4.10.: CDF of Reachability-QDFG edge size

This has a considerable impact on the size and complexity of the to be evaluated QDFGs and thus also significantly affects the detection efficiency. Figure 4.10 shows the cumulative distribution of the edge sizes of all reachability-graphs generated on our baseline evaluation data set. As we can see, more than 97% of the reachability graphs have 100 edges or less.

As the detection phase is independent of the training approach used for generating the employed detection model, we only needed to conduct this experiment once for all reachability QDFGs.

As the time needed to match the obtained testing QDFG vectors against a trained classifier with at average below 2ms was far below the computational effort needed to conduct the actual patter matching, we can safely ignore the influence of the classifier on the overall detection overhead.

From the results depicted in Figure 4.11 we can see that the overall detection time, at least on our evaluation set, seemed to linearly increase with the complexity of the to be classified QDFGs and ranged from 6ms to almost 4200ms.

At average, classifying an unknown sample took 102ms.

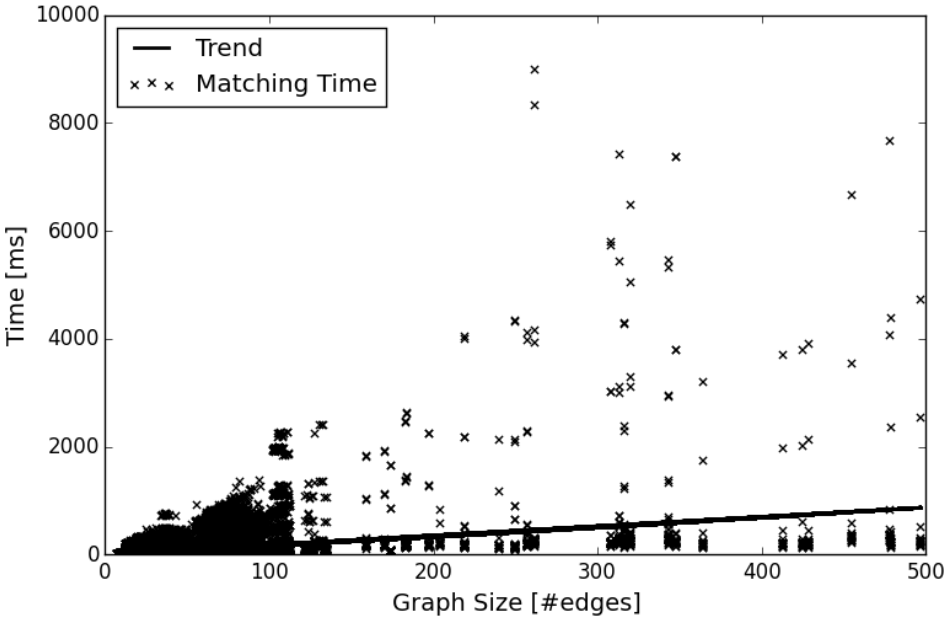


Figure 4.11.: Detection time vs. QDFG size

We account the high efficiency and stability with the comparably low average complexity of the classified reachability-QDFGs (see Figure 4.10). We consider this comparably low classification efficiency overhead a substantial improvement over the initial results gained with our deductive pattern-based approach and thus can also positively answer our main research question *RQ3*.

4.2.4.3. Threats to Validity

Although we put considerable effort into conducting a sound evaluation and comparison with related approaches, there still are some threats to the generalizability of the gained insights.

Firstly, the inductive extension of our pattern-based detection concept makes use of approximate soft-computing algorithms whose overall effectiveness are well-known to be dependent on the amount and structure of the data they were trained on. While our standard evaluation data set is composed of a wide and diverse range of malware samples from different families as well as a representative and diverse selection of widely used goodware, we can only safely claim that our approach works well with respect to this particular set of malware and goodware. The gained insights thus have to be interpreted in the context of our study.

Furthermore, supervised machine learning algorithms can suffer from either over-generalization or over-fitting to the training data. To compensate this threat we consistently employed repeated 10-fold cross validation experiments for evaluating our research hypothesis. We furthermore used a supervised learning algorithm, i.e. RandomForest, for training that is known to be rather robust towards over-fitting to the training data [18].

Finally, we acknowledge that we introduced a certain bias in the comparison between our compression-based mining approach and frequency-based mining approaches in that we did not directly compare the approaches themselves but only indirectly related them based on a comparison of the used mining algorithms. Considering that none of the comparison approaches offers publicly available implementations or data sets it was impossible for us to do so and this indirect comparison on the same standardized data set was the closest we could get towards a fair comparison. Nevertheless, there is a strong conceptual similarity between those approaches and they often employ similar graph mining algorithms.

4.2.5. Related Work

We already discussed the main difference between our approaches and related work in that we are the only ones to use QDFGs to represent malware behavior. We thus restrict the following discussion of related work to approaches that also use graph-based behavior models and employ some sort of data mining.

Christodorescu et al. [36] were within the first to use data mining to obtain graph-based malware detection patterns. They do so by looking for minimal subgraphs of system call dependency graphs of known malware that are not contained in goodware. By this the utility of a mined pattern directly correlates with how often it appears in the malware and does not appear in the benign set. The structural complexity of the pattern itself is not considered as long as it is minimal. Therefore, this approach falls into our category of frequency-based methods.

Chen et al. [32] improve upon this idea by first shrinking the pattern search space through randomized summaries of to be mined system call dependency graphs. Through this approximation they, like us, avoid an expensive evaluation of the entire search space. They thus also make use of graph compression but only use it to reduce the search space; the pattern utility evaluation is still done entirely frequency-based and does not anticipate pattern complexity aspects. We thus also consider their work to fall into the category of frequency-based methods.

The HOLMES detection system proposed by Fredrikson et al. [54] goes along the same lines as it also relies on system call dependency graph mining but introduces an aggressive probabilistic sampling of the pattern search space to further improve detection accuracy.

They further propose to employ concept analysis to combine redundant and semantically similar patterns. To our understanding, their pattern selection does not directly consider the structural complexity of a pattern and we therefore also categorize this work into the frequency-based category.

Besides these works that employed rather custom graph-based pattern mining techniques, there also exists a considerable body of work that in various ways make use of the frequency-based GSPan [150] graph mining algorithm to extract characteristic patterns from bodies of malware [91, 66, 111, 70, 93].

We mainly differ from those approaches in two ways: i) we primarily use the compression capabilities of a pattern to determine its utility, and not its frequency, and ii) we make use of quantitative data flow aspects embedded in syscall traces, both of which we showed to positively contribute to detection accuracy.

Park et al. [113, 114] proposed a malware classification method based on so-called HotPaths, i.e. maximum common sub-graphs on kernel object dependency graphs, to capture characteristic behavior of individual malware families. As this approach per se does not make use of dedicated graph mining techniques to construct a HotPath and thus does not fit our categorization scheme, we wanted to compare its performance against our approach. Unfortunately we were not able to obtain an implementation of the approach from the authors and thus had to fall back to re-implementing it following the descriptions in the respective publications. We then tried to evaluate this implementation on the same data set that we used for evaluating our approach to assure a fair comparison.

Unfortunately, even when running the implementation on the most powerful system that we had access to, we did not manage to get the algorithm conclude and deliver hot-paths for our data set. We account this to the following issues. First of all, the generation of the HotPaths requires to calculate the maximum common sub-graph (MCS), which is a well-known NP-complete problem [57], for a huge number of graphs. The authors in their articles suggest an approximate solution, where the MCS is calculated sequentially for each subsequent pair of graphs. In particular, they suggested to use the McGregor algorithm [41] for doing so. However, using the our implementation of their approach turned out to be computationally infeasible even on a few graphs of less than 50 nodes of size.

Of cause there is a certain chance that we might have wrongly implemented their algorithms or did not employ optimizations that were not mentioned in the article. Nevertheless, even from a purely complexity-theoretical perspective and comparing our experiences with the practical evaluation of maximum common sub-graph algorithms by Conte et al. [41] we are highly concerned regarding the generalizability and scalability of their approach. Also their kernel-object dependency graphs do not take quantitative flows between object into account which, as our evaluations showed, can have a significant effect on detection accuracy.

The main difference between our inductive pattern-based detection approach and related work, besides the obvious aspect of us being the only ones using QDFGs, thus is that we employ a compression-based mining scheme where most related mining approaches use frequency-based algorithms, which we could show yields superior detection accuracy.

4.2.6. Discussion and Conclusion

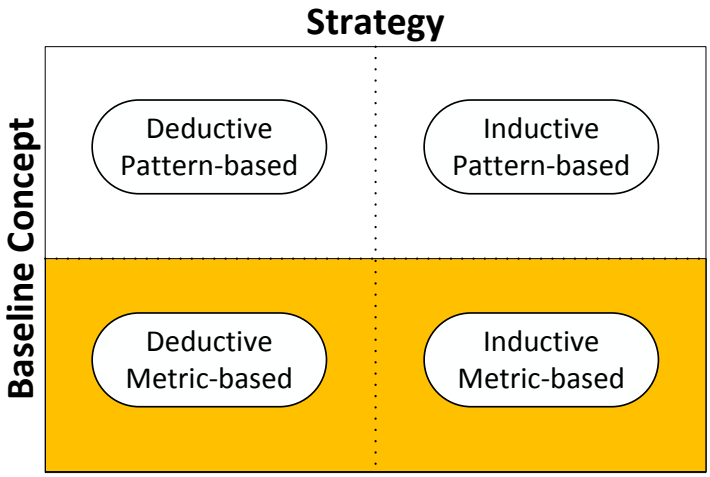
In this section we have proposed a inductive extension of our pattern-based detection approach that replaces the manually defined set of detection patterns of the deductive approach with highly characteristic behavior patterns, automatically extracted from corpi of known malware QDFGs using graph mining. More precisely, we proposed a compression-based graph mining algorithm that considers the quantitative data flow specificities of malware to assess the discriminatory utility of behavior patterns. With this we again successfully operationalized our QDFG-based behavior model for behavior-based malware detection. Furthermore, we showed that inductively mined patterns significantly outperform deductively defined ones in terms of efficiency and effectiveness (*RQ1*,*RQ3*).

Our evaluations showed that the mined patterns significantly outperform the manually defined ones used by the deductive pattern-based approach. We furthermore showed that considering the quantitative data flow compression capabilities of patterns for mining yields a 260% higher detection rate at a 0.5% false positive rate than when only considering structural compression. This positively answers our main research question *RQ2* in that also for mining malware detection patterns we could show that considering quantitative data flow properties can improve detection accuracy. Finally, we could also show that our data compression-based mining approach in comparison to the widely used frequency-based mining algorithm GSpan yields an absolute detection effectiveness improvement of more than 4% at the cost of ten times longer mining times.

In sum, with the proposed inductive extension of our pattern-based detection concept we could further improve the detection effectiveness that can be achieved using QDFGs as behavior models.

5. Metric-based Detection

In this chapter we propose an approximate notion of QDFG similarity based on graph metrics as a flexible alternative to sub-graph isomorphism-based detection. To this end, we discuss a deductive approach to profile behavior with generic graph metrics and an inductive extension to this where we use genetic programming to generate complex graph metrics that are by design effective. Some parts of this chapter are based on a previous publication [145], co-authored as first author by the writer of this thesis.



In the last chapter we have seen how our QDFG-based system model can be operationalized for behavior-based malware detection by a-priori defining or mining characteristic malware behavior patterns and re-identifying them in QDFGs of unknown samples. With this we did not only show the utility of quantitative data flow analysis for accurate malware detection, but also that the consideration of data flow quantities can substantially improve detection effectiveness.

Intuitively, these approaches are also to some extent robust towards noise in captured behavior traces and simple behavior obfuscation since the QDFGs built on top of them are not affected by system call splitting or reordering.

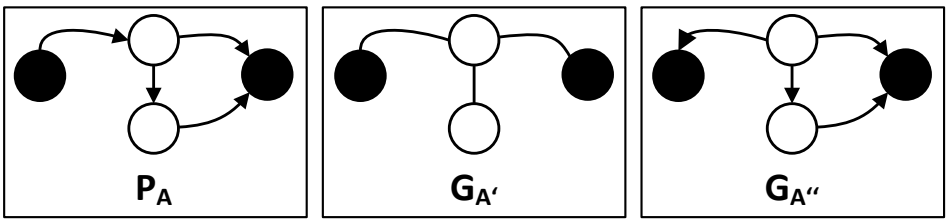


Figure 5.1.: Pattern and QDFGs of mutated variants of the same malware family

Programs that intentionally or unintentionally change their behavior in term of alternations in the sequence of issued system calls usually still yield the same QDFG representations. This is, because our model, i.e. the graph update function, is agnostic to relative orders of system calls. The same holds for situations where programs use several system calls to read a file in multiple small chunks instead of one "large" system call to read the entire file at once. As long as the total flow of transferred data remains the same, this has no direct consequence on the structure of the respectively built QDFGs and the pattern-based detection approaches operating on to of them.

Nevertheless, strictly pattern-based approaches, like the ones introduced in the last chapter, are quite sensitive towards more complex behavior changes that also impact the structure of the corresponding QDFGs. If a QDFG that models behavior of a new version of a malware for instance lacks certain edges or nodes that were present in previous version of that malware for which we mined or manually defined detection patterns for, the old patterns might not fit these new graphs anymore. In consequence, a strict pattern-based detection mechanism might fail to identify known malign behavior patterns in the behavior graphs of new versions of a malware although they in essence relate to very similar behavior.

To get a better understanding of this issue we consider a simplified detection situation, depicted in Figure 5.1, where we want to classify the QDFGs $G_{A'}$ and $G_{A''}$ that model the behavior of two malware samples that are descendents, i.e. extended versions of the same malware family A. Now also consider that we already obtained a highly characteristic detection pattern P_A that we mined from the behavior of multiple samples from the initial generation of the malware family A. So far, the pattern worked well for properly identifying members of family A and yielded good detection accuracy.

At his point we need to recall that our pattern-based detection approaches are based on sub-graph isomorphism checks, i.e. classify new QDFGs by looking for sub-graphs that are isomorphic to detection pattern graphs that resemble known malicious behavior. A closer look at the situation depicted in Figure 5.1 however reveals that the malware graphs $G_{A'}$ and $G_{A''}$ do not have any sub-graphs that are isomorphic to our detection pattern P_A. This is because $G_{A'}$ for instance lacks an edge between the lower two nodes that would be needed to match P_A. $G_{A''}$ in contrast in principle has all necessary edges and nodes, but the edge between the upper two nodes is inverted in comparison to the corresponding one in the pattern P_A. Both cases would lead to the the same detection result: the sub-graph isomorphism check would report that $G_{A'}$ and $G_{A''}$ do not match the detection pattern and thus are not malicious according to our definition.

This situation is somewhat unfortunate in that $G_{A'}$ and $G_{A''}$ indeed are malicious and, from a behavioral perspective, are very close to P_A. Nevertheless, they are not isomorphic to each other and our sub-graph isomorphism based approaches do not cater to such small behavioral deviations. In a sense we can thus say that checking for sub-graph isomorphism in malware detection in such situations might be too restrictive and corresponding detection approaches easy to be fooled by malware that adapts its behavior in a targeted way.

Although this example is of course highly simplified, malware authors typically do not have full flexibility on changing arbitrary parts of the malware behavior and thus cannot arbitrarily tamper with the resulting QDFGs, this still reveals a principal problem with sub-graph isomorphism based detection approaches. Similar situations can happen in reality whenever new versions of a malware add to or restrict the original functionality. This is commonly the case when malware developers use malware construction kits or reuse program code of old malware to create new malware variants.

Note that this is not only an issue for our detection approaches, but a principal problem for all graph isomorphism based detection approaches, irrespective of whether they use control flow, system call, or resource dependency graphs.

Based on this observation we thus saw a need for a more elaborate detection concept that caters to such intentional or unintentional deviations in malware behavior and employs a more flexible notion of behavioral similarity than our previous strict sub-graph isomorphism based approaches.

To this end we again re-visit the simplified detection situation scenario depicted in Figure 5.1. While we already discussed that, at least from a strict isomorphism perspective, the graphs are different, they still share some non-negligible similarities and only differ in a few aspects. All three graphs for instance have the exact same number and type of nodes (represented by the node color). Furthermore, two of the three graphs also have exactly the same number of edges. Finally, all graphs at least share a common sub-set of edges.

This intuitively suggests that partial similarity between graphs can be expressed in terms of graph invariants or metrics like node or edge count.

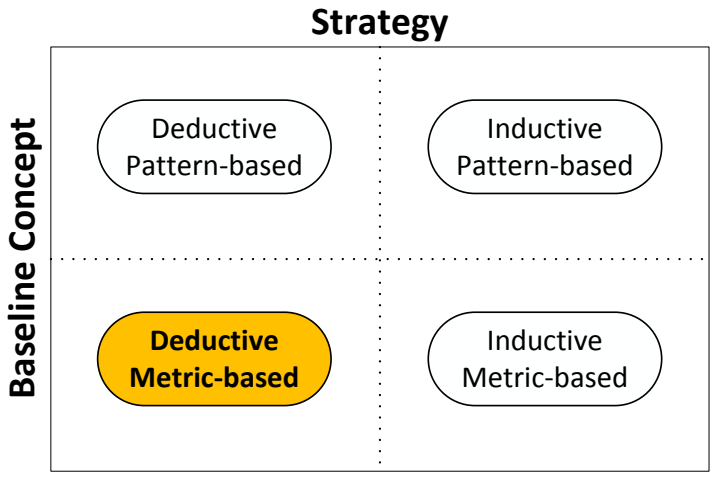
This observation let us to the question of whether or not we can establish a flexible notion of graph similarity based on such graph invariants and metrics that can be incorporated into our malware detection concept and still achieve a sufficiently high detection accuracy, but with a better robustness towards noise and changes in behavior profiles.

Looking at this idea from a purely graph theoretical perspective, approximating similarity with graph metrics also seems natural as the equality of certain graph invariants and metrics between two different graphs is a necessary precondition for graph isomorphism. This also likely has a favorable side effect on detection efficiency as calculating graph metrics typically is of lower computational complexity than checking for full graph isomorphism.

In sum, this in means that we wanted to investigate if we could substitute the sub-graph isomorphism checks in our approaches by more elaborate and flexible graph metric based similarity checks while still maintaining a sufficiently high detection accuracy. We were furthermore interested in investigating in how much such flexible similarity checks lead to an improvement of detection robustness with respect to structural changes in malware QDFGs and how using this notion of approximate similarity impacts detection efficiency.

In this chapter we thus investigate the utility of metric-based graph similarity approximation for doing malware detection on QDFGs. To this end we again follow our basic research methodology in that we first explore the principal utility of a generic graph metric based notion of similarity in Section 5.1, and then assess the conceptual boundaries of this concept using more complex and automatically generated malware-specific graph metrics in Section 5.2.

5.1. Deductive Metric-based Detection



5.1.1. Introduction

Following our general research methodology to first assess the general feasibility of a detection concept with statically defined detection mechanisms before exploring the conceptual boundaries using more elaborate mechanism generation techniques, the main goal of our first metric-based detection concept is to demonstrate the principal feasibility of metric-based detection on QDFGs. This is, with this approach we want to firstly show that graph similarity approximation with graph metrics is sufficiently precise to substitute sub-graph isomorphism verification and thus to be suitable for being used accurate behavior-based malware detection. This again boils down to *RQ1* in that we introduce another concept for utilizing our QDFG-based system model for malware detection.

As the guiding theme of this thesis is to show the utility of quantitative data flow analysis for increasing malware detection accuracy, we also want to give answers to *RQ2* and show that by leveraging the quantitative data flow information encoded in our QDFGs for metric computation we can improve detection accuracy. Furthermore, considering the complexity of the respective algorithms we expect approximate similarity checks with graph metrics to be significantly more efficient than performing fully-fledged sub-graph isomorphism checks.

In the introduction of this chapter we already motivated the use of more flexible and approximate similarity checks on graph metrics with the shortcoming of sub-graph isomorphism based detection approaches to in some cases be too restrictive and sensitive towards changes in observed malware behavior. Besides investigating the accuracy implications of using approximate metric-based similarity checks instead of hard sub-graph isomorphism we are thus also interested in analyzing the effects on detection robustness, i.e. give answers to the respective research question *RQ4*.

Most new malware families found in the wild today employ some kind of functionality to avoid or harden detection through traditional security measures. Examples range from rather simplistic attempts to disable known security software upon infection; over polymorphism and metamorphism techniques to alter and obfuscate the executable binaries of malware in order to harden detection by signature-based approaches; up to more sophisticated behavioral obfuscation techniques, such as mimicry attacks, that aim at altering the runtime behavior to trick behavior-based detection approaches [152]. One of the biggest challenges of malware detection research is thus to cope with the threat of stealthy and obfuscated malware and to counteract their attempts to remain "below the detection radar".

Recent research showed that already comparably simplistic behavior obfuscation techniques can have a tremendous impact on the accuracy of most state of the art behavior-based detection approaches [116, 115, 6]. Like others [115], considering the steadily increasing sophistication of modern malware we thus expect new malware families to soon employ more complex functionally to obfuscate behavior in a targeted way which will impose a severe threat to most behavior-based detection approaches.

One goal that we want to achieve with our research and tackle with the approach proposed in this section thus is to a-priori anticipate potential behavior obfuscation techniques and investigate their impact on the integrity of our QDFG models. Doing so we aim at coming up with a detection approach that we can show to be robust at least against the behavior obfuscation techniques that we anticipated. In sum, with the approach described in the following we thus firstly give answers to our research question *RQ4* in that we want to show that our detection concept is to some extent robust towards simple behavior obfuscation techniques.

As mentioned before, the main functional principle of the proposed approach is to use graph metrics, computed on QDFGs as means to come up with a flexible approximate interpretation of graph similarity which we then use to profile known malware and goodware behavior. By computing similar metric profiles on QDFGs of unknown samples and evaluating their distance to the profiles of known malicious and benign samples we then can classify them as more likely being malicious or benign.

To come up with such profiles we use a fixed set of simple standard and more context-specific graph metrics, which we partially took from social network analysis research and adapted to fit our context. More precisely we use these metric-based profiles of typical data flow behavior of benign and malicious processes to train a machine learning classifier. This classifier can then learn a notion of similarity between those profiles that is sufficiently discriminative to accurately separate malicious from benign QDFGs based on their metric profiles. This is, by matching a set of metric values of an unclassified QDFG, the classifier can decide whether the sample is more likely to be malicious or benign based on its similarity to known goodware and malware metric profiles.

The main difference of this approach to the previous pattern-based ones is thus that it uses approximate similarity checks based on graph metrics instead of subgraph isomorphism verification. As we will later show, this gives us a superior detection robustness, efficiency, and very high accuracy. This concept of approximate similarity checks also differentiates us from other malware detection approaches that uses graph models to represent malware behavior [80, 54, 85, 37, 36, 116, 151, 53, 76, 79, 84, 113, 114]. In sum, the contributions of this approach are:

- To the best of our knowledge, this approach is the first one to combine *quantitative* data flow tracking and graph metrics with machine learning for checking for behavioral similarity of processes in the context of malware detection.
- Our experiments demonstrate the utility of *quantitative* data flow aspects for detection precision. In particular we show that the consideration of quantities in data flow graphs can effectively halve false positive and false negative rates.
- Our evaluations indicate that our approach is more robust against behavioral obfuscation such as reordering or inserting bogus system calls than approaches that build on raw system calls such as n-gram based approaches.
- We show that we are able to detect samples from unknown malware families, i.e. samples from families that our approach was not trained on, with good accuracy.

Most parts of this section are based on an earlier publication [145], co-authored as first author by the writer of this thesis.

5.1.2. Approach

Our initial metric-based detection approach, at least for the first processing steps, in essence follows the same procedure as the previously described pattern-based detection approaches: we also first need to execute a sample in our custom malware sandbox environment, monitor its behavior for a defined time span, translate the captured system calls into data flow events, and finally generate a QDFG from them that models the sample's data flow behavior.

Our core idea is to learn statistical profiles for benign and malicious nodes in QDFGs that represent known infected and non-infected systems. We later use these profiles for matching feature sets of unknown processes against them. The first difference to the pattern-based detection approaches is in the step that precedes the QDFG generation. Unlike the previously described pattern-based detection concepts that directly conduct their training or detection steps on those QDFGs, or only a-priori reduce training graph complexity for performance reasons, for metric-based detection we need to control and normalize the complexity of both, the training and the testing graphs in order to create a compatible noise-reduced training and testing data baseline.

This step is necessary as our metric-based detection concept, i.e. the computation of our graph metrics, is highly influenced by the complexity of the input graphs that is e.g. influenced by the number of processes running in the monitored system. To establish a more reliable training baseline and partially weed out unwanted side-effects from processes that are in no functional relationship to the actual malicious processes that we want to classify, we thus, just like for the inductive pattern-based approach, create process-centric sub-QDFGs from the main QDFG of a sample. This is, for each process node within a QDFG we generate a new QDFG that consists of all nodes and edges directly or indirectly reachable from this process node. By doing so we create process-centric sub-QDFGs that only modeled the behavior that was in direct or indirect relationship to the baseline process.

Such a pre-processing step was not strictly necessary for the pattern-based approaches as their sub-graph isomorphism based detection concept by construction only considers the "potentially interesting" parts of the QDFGs and are less biased by unrelated noise. However, preliminary evaluations of a metric-based prototype where we did not employ such a pre-processing step indicated that some of the graph metrics used for profile generation where too sensitive to noise in the QDFGs and thus generated poor detection results on data sets with unbalanced graph complexities. Especially if we recall that our evaluation data set, which we consider to be representative to real world data sets, indeed varies in terms of QDFG complexity, the necessity of conducting the pre-processing becomes obvious.

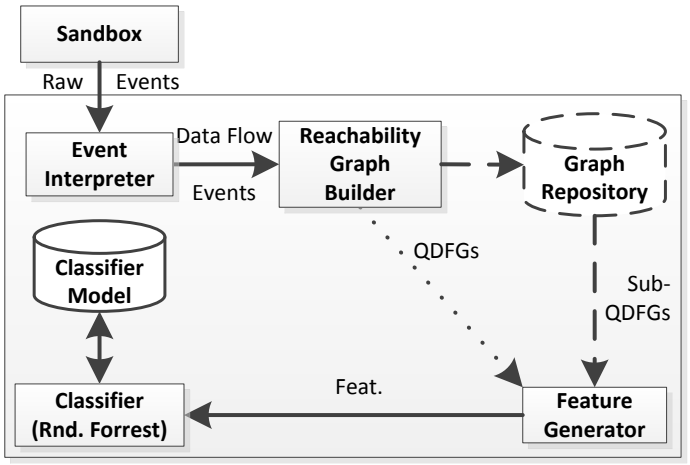


Figure 5.2.: High-level Architecture

Besides this reachability graph pre-processing step, the deductive metric-based detection approach also differs from the pattern-based approaches in that in the following we do not employ any sort of isomorphism-based graph matching with fixed or minded detection patterns but rather compute a set of highly descriptive graph metrics on them whose output values we then feed into a machine learning classifier to infer generic metric-based profiles for known malicious and benign behavior.

The overall architecture of this approach, depicted in Figure 5.2, is thus a variation of our basic architecture, as described in Section 4.1.2. Dashed lines in the architecture mark components and interactions that are only used in the training phase. Dotted lines refer to the ones only relevant for detection.

1) *Base Data Generation*: Just like for the inductive pattern-based approach we first obtain the base QDFGs that model the full behavior of the monitored sandbox system where we executed a sample. To this end, we execute a sample in the instrumented sandbox, capture the system calls issued by all relevant processes, interpret their data flow semantics, and build the corresponding behavior QDFG. For the obtained base QDFG we then extract the set of process node and, for each process node, generate a reachability graph consisting of all reachable nodes and edges. In the training phase of our approach the so obtained process-centric reachability graphs are then put into a training graph repository as baseline for the subsequent metric-based profiling steps.

2) *Feature Generation*: To establish profiles for known malware and goodware processes we then compute a set of graph metrics on the corresponding reachability QDFGs. The respectively computed values for these metrics are then stored as labeled feature vectors that then get forwarded to our classifier training component.

3) *Classifier Training*: We then use the obtained metric vectors, which are labeled as malicious or benign in correspondence to the known nature of the associated process reachability QDFGs, to train a supervised machine learning classifier. This classifier then generalizes from the individual metric profiles of the training reachability QDFGs to infer generic relationships between graph metric profiles and known malicious and benign samples. In a nutshell, the classifier thus learns metric profile commonalities of known malicious QDFGs that significantly differ from those of known benign QDFGs. With this we establish a approximate and flexible metric-based notion of malicious data flow behavior.

4) *Detection*: We now can simply classify QDFGs of unknown samples by matching their metric profiles against the trained classifier. Based on the similarity of the metric profile of the unclassified sample to the ones of known goodware and malware the classifier will then output a prediction of whether the analyzed sample more likely is malicious or benign.

Now that we have sketched the architecture of the deductive pattern-based detection approach, we in the following will elaborate on the used graph metrics and their formal definition, followed by a more detailed explanation of the training and detection steps.

5.1.2.1. Features

After having decided to approximate QDFG similarity with graph metrics, the natural next question was which exact graph metrics to be used for our approach.

While investigating the utility of simplistic graph metrics like node or edge count, as discussed in the intro of this chapter, with mediocre success we saw an analogy between our QDFGs and communication graphs as they typically appear in social networks [108, 16].

The analogy between the two is quite obvious: In social network analysis, members of a network are typically represented as nodes with relations between them, e.g. established via communications, mutual interests, or friendship associations being modeled as edges between the member nodes. These edges can be weighted to represent the degree of connection between two members, e.g. determined by the amount of exchanged messages or mutual interests.

Analogously, in our QDFGs the nodes represent system entities and the edges between them their interaction, i.e. communication, in form of data flows.

Considering this similarity between social network graphs and quantitative data flow graphs we thus came up with the idea of leveraging similar graph metrics that are used for social network analysis for QDFG-based malware analysis. Doing so is intuitively compelling as social network analysis is all about finding interesting properties and sub-structures, i.e. communities within a social network, while our malware detection concept similarly aims at finding discriminative and characteristic properties of malware QDFGs.

The following metrics were thus selected using an inductive and a deductive approach. The inductive selection was done based on a preliminary analysis of graphs of a small set of malware-infected systems where we applied several standard metrics from social network analysis and tried to correlate malware activities with the applied metrics. The deductive selection was performed through an analysis of standard graph metrics where for each metric we tried to correlate its intuition with typical malware behavior or properties. For instance, malware that tries to infect other processes or files results in a high connectivity of the corresponding node to certain types of other nodes. We then assessed whether this hypothesized correlation also holds in reality by again applying the respective metrics on some malware QDFGs and then manually investigating similarities.

As we will later use the evaluated graph metrics as input features for training a supervised machine learning classifier, we in the following will use the terms (graph) *metric* and *feature* synonymously.

We express graph metrics, i.e. features, as functions $\phi \in \Phi$ that map a QDFG node to a real number: $\phi : \overline{N} \times \mathcal{G} \rightarrow \mathbb{R}$. We enrich the QDFG model to store the value of features as attributes of nodes n in graph G, so $\lambda(n, \phi) = \phi(n, G)$. Additionally, we distinguish between two basic types of graph features, *local features* (Φ_l) and *global features* (Φ_g). Local features have a single-hop scope. This means that they only capture the relationship of a node with its direct neighbors. Global features in contrast have a multi-hop scope and represent relationships of one node with all other nodes of a graph. The reason for this distinction is that we wanted to capture both, direct and indirect behavior behavior of a malware, as e.g. caused by malware child processes or hijacked benign processes.

5.1.2.2. Local Features (Φ_l)

To define the features, we need some auxiliary notation. Function $d_\psi : (\overline{N} \times \overline{N} \times \mathcal{G}) \rightarrow \mathbb{R}$ with ψ returns the shortest path between two nodes in a graph, where $\psi : \overline{E} \rightarrow \mathbb{R}$ with $\psi(e) = \lambda(e, size)$ defines the edge distance, or cost, i.e. the amount of data transferred via this edge.

1. **Entropy** $\phi^1 \in \Phi_l$ computes the normalized entropy of the distribution of sizes of all outgoing edges of a process node $n \in N$. The entropy captures the uniformity of the distribution of percental flows of all outgoing edges of a node n.

   Rationale: Viruses like Parite infect other executable binaries or processes by injecting or appending their own binary image. The respective subgraphs tend to have a comparably uniform distribution of specific features of outgoing edges, because the majority of triggered events by that malware are targeted at the infection with roughly the same size of the events as consequence of them relating to reading or writing the same binary image.

   Computation: Let $\overrightarrow{s} = (s_1, \ldots, s_k)$ and $e_i \in out(n)$ in $s_i = \frac{\psi(e_i)}{\sum\limits_{e' \in out(n)} \psi(e')}$. Then we define: $\phi^1(n, G) := NE(\overrightarrow{s})$ where $NE(\overrightarrow{s}) := \frac{-\sum\limits_{i=1}^{k} s_i * \log(s_i)}{\log(k)}$.

2. **Variance** $\phi^2 \in \Phi_l$ expresses the statistical population variance of the distribution of a certain edge feature for all outgoing edges of a node $n \in N$. A low statistical variance indicates that most of the elements of the distribution elements are close to the statistical mean, whereas a high variance indicates a spread of elements. Due to its similar focus on uniformity of underlying input distributions, the variance feature is closely correlated with the entropy feature. First evaluations indicated, however, that the entropy metric performed comparably badly if a node has only a few outgoing edges, but better for larger sets of outgoing edges. The variance metric seemed to exhibit exactly the inverse characteristics.

   Rationale: The motivation is similar to that for entropy: malware often exhibits outgoing edge distribution characteristics different from benign ones.

   Computation:

$$\phi^2(n, G) := \frac{\sum_{e \in out(n)} \left(\psi(e) - \frac{1}{|out(n)|} \sum_{e' \in out(n)} \psi(e') \right)}{|out(n)|}$$

3. **Flow Proportion** $\phi_t^3 \in \Phi_l$ captures the proportion of a certain type of outgoing data flows of a node $n \in N$ w.r.t. all outgoing flows of that node. The type of a flow is determined by the target node's type of the outgoing edge. We define different variants of the proportion feature that consider different edge attributes.

   Rationale: Malware processes often exhibit different flow proportion characteristics than goodware. Examples include ransomware or virus processes that have an irregularly high percentage of outgoing edges that point to file nodes, as they either encrypt several sensitive files, or infect all executable binary files on the hard disk.

   Computation: Let $t \in \{Process, Registry, File, Socket\}$.

$$\phi_t^3(n, G) := \frac{\sum_{e=(src,dst)\in out(n), \lambda(dst,type)=t} \psi(e)}{\sum_{e\in out(n)} \psi(e)}$$

5.1.2.3. Global Features (Φ_g)

Global features represent the relation between one node and—possibly all—other nodes of a graph. In contrast to local features, capture the importance of one node within the overall graph. Note that a crucial feature of global features is the fact that the weight of edges (given by the size of data flows between them) is considered when computing the shortest path between nodes (given by the function $\psi(e)$).

1. **Closeness Centrality** $\phi^4 \in \Phi_g$ for a node $n \in N$ represents the inverse of that node's average distance to all other nodes of the same graph. A high closeness centrality indicates that the respective node is closely connected to all other graph nodes [108].

   Rationale: High connectivity with other nodes indicates a node manipulating or infecting other system resources like processes or executable binaries. Such behavior is typical for viruses like Parite that replicate by infecting other processes and binaries. This leads to a close connectivity of the corresponding malware process node with other process and binary file nodes.

   Computation:

$$\phi^4(n, G) := \frac{|N| - 1}{\sum_{n'\in N\setminus\{n\}} d_\psi(n, n', G)}$$

2. **Betweenness Centrality** $\phi^5 \in \Phi_g$ of a node $n \in N$ represents the relative portion of all shortest paths between all possible pairs of nodes of a graph that pass through that specific node n. A high betweenness centrality means that one specific node is part of a multitude of "communications" between nodes [108].

   Rationale: This metric captures how often a process is part of a multi-step interaction or data flow between other system resources. This is useful to identify malware aiming at man-in-the-middle attacks to e.g. intercept the communication of a benign process with a socket, or to manipulate the information that a benign process reads into memory, to e.g. infect that process with malicious code at runtime. We observe such behavior for backdoors, or more specifically, information-stealers. The respective malware processes typically have a higher betweenness centrality than benign process nodes.

   Computation: The function $sp(x, y, G)$ returns the number of shortest paths between the nodes x and y in a graph G; $sp_z(x, y, G)$ the ones that pass through node z.

$$\phi^5(n, G) := \sum_{n', n'' \in N : n \neq n' \neq n''} \frac{sp_n(n', n'', G)}{sp(n', n'', G)}$$

5.1.2.4. Training and Model Building Phase

Using the above described QDFG-specific graph metrics we can now establish generic statistical behavior profiles for the discrimination between benign and potentially malicious process nodes in a graph. As sketched before, the training procedure consists of four activities: i) event log generation; ii) graph generation; iii) feature extraction; iv) classifier training. As the first data retrieval and processing steps exactly resemble those presented in Section 5.1.2 we omit a detailed description here. We instead directly jump to the point right after generation of the baseline QDFGs that capture the full system behavior of a monitored system.

5.1.2.5. Reachability Graph Generation

As explained in the introduction, our metric-based detection approaches demand a more controlled generation of the training and detection data. We thus first introduce a pre-processing step before continuing to describe the actual training phase of our approach.

In order to reduce noise and instead of directly using the complete system behavior QDFG for training, we generate so-called reachability graphs for all process nodes in the base QDFGs. Such reachability graphs contain all nodes and edges that are directly or indirectly reachable form the starting process node.

By this means we ensure, that the training graphs only contain activities that are actually triggered by a certain process or of processes that it directly or indirectly influenced, ignoring all activities that are conducted by non-related processes.

With the so obtained process-centric reachability graphs we then populate the training graph repository to be used by the subsequent feature extraction step.

5.1.2.6. Feature Extraction

The *feature extractor* correspondingly computes all features, i.e. the graph metrics described in Section 5.1.2.1 for all process nodes of the reachability graphs in the training graph repository. By this we for each process node obtain a set of feature values that we obtain when evaluating the respective graph metrics on the reachability graph of that process node. Recall that we labeled the known malicious process and thus also the corresponding graph nodes. We are hence able to label the resulting process node features as belonging to a known malicious/benign process, which is a necessary precondition for later training a supervised machine learning algorithm.

5.1.2.7. Classifier Training

After the feature extraction phase, we are now set to feed the obtained features into a supervised machine learning algorithm for training.

To this end, we construct a feature vector for each process node of the training set. Each element of this vector is one of the considered QDFG metrics from Section 5.1.2.1, plus one label element representing the known classification (benign or malign) of the respective process. Each feature vector is thus of size $|\Phi| + 1$.

We can thus write the resulting training data sets as matrices, where the columns represent the different used graph features $\phi_\psi^i \in \Phi$ together with a distinct process class label column $\mathcal{L} := \{malicious, benign\}$ that represents whether the respective feature set corresponds to a malicious or benign node. The labeling function $l : \overline{N} \to \mathcal{L}$ maps such a label to a node. Each row represents the feature values of one specific process node sample $\lambda(n_j, \phi_\psi^i)$ with $n_j \in N_{train}$, with $h = |N_{train}|$, the set of training process node samples. Correspondingly, the training data generation function $gen : 2^N \to T$ generates a training data matrix $t \in T$ for a given set of training process nodes:

$$gen(N_{train}) := \begin{bmatrix} \lambda(n_1, \phi_\psi^1) & \cdots & \lambda(n_1, \phi_\psi^k) & l(n_1) \\ \vdots & \ddots & \vdots & \vdots \\ \lambda(n_h, \phi_\psi^1) & \cdots & \lambda(n_h, \phi_\psi^k) & l(n_h) \end{bmatrix}$$

Note that we only compute feature vectors for process nodes as we are solely interested in determining whether a specific process that originated from a executed binary is malicious or not; we thus do not classify a binary itself, but its runtime representation, i.e. the respective process or its children. By examining the reachability graph associated with a node, it is possible for analysts to investigate the root cause of infection, as discussed in more detail in Appendix A.

Before coming to the training procedure we need to perform some over-sampling steps to balance the ratio between malicious and benign samples in the training data set. Unless the training data roughly contains the same amount of malware and goodware samples, an over-sampling is necessary to prevent a training bias due to imbalanced training sets. Such a training bias often happens for imbalanced training data sets that do not contain roughly equal amounts of samples for all to-be learned classes, as many machine learning classification algorithms tend to put overly much emphasis on the majority class as they try to reduce classification error while widely ignoring the distribution of classes among all samples. Such a bias can easily lead to overfit models that perform well in classifying majority classes but perform bad with respect to accurately classifying minority classes [31]. This imbalance in class distribution, often referred to as base-rate fallacy [2], is a typical problem for machine learning based intrusion and malware detection approaches and limits transferability of evaluation results obtained in lab experiments to reality. By introducing a dedicated over-sampling step into our training procedure we try to at least partially tackle these problems.

In order to prevent these biases, we apply the Synthetic Minority Oversampling Technique (SMOTE) [31] to all training feature sets. The basic working principle of SMOTE is an over-sampling of the minority class (malicious process nodes) by generating synthetic minority class samples. The over-sampling of the minority class is performed by performing a nearest-neighbor search to obtain close neighbors of minority class members, randomly selecting a subset of those neighbors, and finally randomly generating a defined number of synthetic minority samples on this basis.

Simplistic over-sampling tends to modify training data in a way that leads to over-fitting of generated classification models as consequence of altering their statistic properties. SMOTE in contrast does not alter these statistic properties due to the applied randomization and balancing steps [31]. After synthetically over-sampling the minority class data, which in our case is are the goodware samples, we feed the obtained feature data into a machine learning classification algorithm for training. We can express this training phase as a function $\tau : T \times \mathcal{M} \rightarrow \mathcal{C}$ that takes a training data matrix $t \in T$ and applies a supervised machine learning algorithm $m \in \mathcal{M}$ on that data to generate a classification model $c \in \mathcal{C} : \mathbb{R}^k \rightarrow \mathcal{L}$ that is modeled as a function that returns a process node label for a given set of $k = |\Phi|$ feature values.

Considering the high number and diversity of the value space of the selected training features we need a machine learning algorithm that is robust towards training set diversity and scales well with respect to the number or training features. After initial attempts to use simplistic classifiers like naive Bayes with poor performance in terms of detection precision, we explored more complex algorithms like support vector machines and meta-learners. Particularly good results were achieved with the Random Forest algorithm [18]. Random Forest is a meta- or ensemble-learner, which means that it uses several distinct, potentially imprecise, classification models and merges their decisions to form a more precise combined decision. More specifically, the Random Forest algorithm constructs a multitude of individual decision trees, called decision forest, based on random selection of limited feature subsets of the overall feature space.

Due to this randomized feature selection, the Random Forest algorithm is able to generate a huge variety of distinctly built decision trees. Depending on the algorithm's parameters and used feature sets, the generated decision trees in consequence are generated based on completely different feature sets. To infer stable and precise prediction from these decision trees, the Random Forest then performs a majority vote on the prediction results of all trees. Due to the resulting inherent distribution of learning and classification error, the Random Forest algorithm is particular suited for classification problems with high feature diversity. We thus identified Random Forest as ideal candidate for the classification problem behind our detection approach. For this reason the last step of the training procedure is to feed the sanitized and over-sampled feature set as training data into a Random Forest learner to generate respective prediction models. These generated models are then persistently stored into a classifier model repository as basis for subsequent detection steps.

5.1.2.8. Detection Phase

We now have a classifier that can predict the class (malicious or benign) of an unknown process node based on its characteristic local and global graph features. In a nutshell, for the detection phase we thus only need to build the graph of a potentially infected system at runtime based on captured events, compute the characteristic features for each process node in the graph, and match the resulting feature set against the classifier.

Like for the training phase, we intercept relevant system events at runtime, interpret them in terms of their data flow semantics, and then build the corresponding (reachability) QDFGs for each process. We then compute the characteristic feature sets for the process nodes of the generated reachability graphs and match them against the classifier, using the classification model that was generated as result of the training phase.

This detection step is modeled as a function $\delta : (N \times \mathcal{C}) \rightarrow \mathcal{L}$ that determines the process label for a graph node $n \in N$, using a given classifier $c \in \mathcal{C}$: $\delta(n, c) := c(\overrightarrow{s})$ where $\overrightarrow{s} = s_1, \ldots, s_k$, and each s_j corresponds to a feature value computed by one variant of $\lambda(n, \phi^i_\psi)$ described in Section 5.1.2.1 for some i and $\phi^i_\psi \in \Phi$.

Consequently, after doing this, all process nodes of these reachability graph are classified into benign or potentially malicious ones.

5.1.3. Evaluation

For evaluating our deductive metric-based detection approach we implemented the detection framework depicted in Figure 5.2 and executed it on our standard evaluation data set, using the same evaluation setup as described in Section 4.1.3.

For the training phase we used a the RandomForest which we configured to build a forest of 10 distinct decision trees, each using a different randomly determined subset of the features described in Section 5.1.2.1.

Just like for our previously introduced detection approaches we were mainly interested in evaluating the general effectiveness (*RQ1*) and in particular the potential accuracy boost that we get from using quantitative data flow properties (*RQ2*). Also we aimed at evaluating the efficiency of our approach; in particular with respect to the expected performance increase due to the, in comparison to sub-graph isomorphism, inexpensive computation of graph metrics.

Differently to the evaluations of our pattern-based detection approaches we were furthermore interested in assessing our initial hypothesis of our approximate similarity checks through graph metrics leading to an significant improvement in robustness towards behavior obfuscation.

5.1.3.1. Effectiveness

To evaluate the detection performance of our approach on the obtained feature set we thus first performed ten times a 10-fold cross validation test. For this tests we split the entire feature set into two parts, using 90% of the set for training and the remaining 10% for testing. The sets were randomly generated and the splitting repeated 10 times for each test to limit bias from specific set compositions. For each run we built a classification model on basis of the training data and used it for classifying the remaining test set.

Table 5.1 (a) depicts the average *effectiveness quality metrics* of the cross validation experiments. As we can see, the deductive metric-based approach at average can correctly detect 98 % of the provided malware set with a low false positive rate of only about 0.5%. The low standard deviations furthermore indicates a good stability of the results.

	a) Real	b) Fixed	c) Random
Avg. Det. Rate (Std. Dev.)	0.980 ($\sigma = 0.005$)	0.980 ($\sigma = 0.006$)	0.952 ($\sigma = 0.009$)
Avg. FP Rate (Std. Dev.)	0.005 ($\sigma = 0.003$)	0.009 ($\sigma = 0.004$)	0.011 ($\sigma = 0.004$)

Table 5.1.: Effectiveness Quality Metrics

The Receiver Operator Characteristics Curves generated on the basis of the performance of the trained classifiers furthermore revealed an average AUC of 0.984.

This result already answers research question *RQ1* for the deductive metric-based detection approach in that sense as we could show that our concept of using approximate metric-based profiling on QDFGs indeed leads to highly accurate detection results. Given these encouraging results we then next wanted to investigate the concrete influence of the quantitative data flow aspects encoded within our QDFGs on the overall detection accuracy to answer research question *RQ2*.

To evaluate our hypothesis, that the consideration of the actual *quantitative* data flow information within a QDFG has a significant impact on the effectiveness of the classification, we performed two more tests. For the first test we replaced the real quantities associated to the edges of the QDFGs with a globally fixed value of 1. For the second test we performed the edge quantity replacement by associating varying random quantities to the edges. With this we effectively destroyed the inherent quantitative information of the QDFGs. For both experiments we again performed 10-fold cross validation tests to ensure stability of the results. Table 5.1 (b) and (c) depicts the average detection and false positive rates for both settings.

To calculate the relative impact of quantities on the detection effectiveness we divided the false positive and false negative rate (which is the dual of the detection rate) for the fixed and randomized quantities experiment by the respective rates of the experiment with the real quantities.

As we can see, fixing the quantities to a constant value increases the false positives by a factor of 1.8 ($\frac{.009}{.005}$). For the randomized quantities experiment we could observe an even bigger loss of effectiveness. Here the false positives increased by a factor of 2.3 ($\frac{.011}{.005}$), while also the false negatives increased by a factor of 2.4 ($\frac{1-.952}{1-.980}$) with respect to the experiments with the actual quantities.

These observations thus support the hypothesis about the utility of quantitative information for malware detection and answer our research question *RQ2* in that considering quantitative data flow information for detection indeed improves classification accuracy. To further fortify this conclusion, we verify the statistical significance of these finding with a standard two-tailed t-test on the detection and false positive rates of the different experiments.

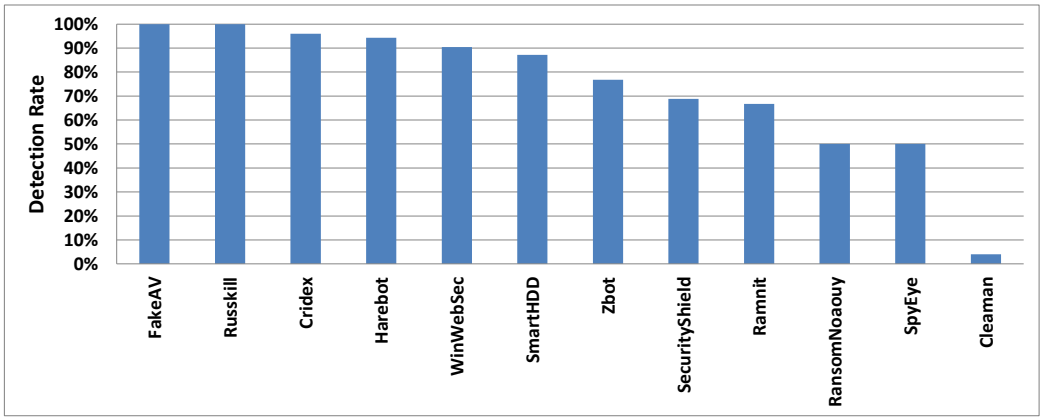


Figure 5.3.: Detectability of new malware

The resulting p-values were all far below 0.01%, which indicates a high statistical significance of our observation.

For evaluating our hypothesis that our idea of approximating graph similarity with graph metrics indeed leads to detection models that cater well to variations in malware behavior, i.e. that we are able to detect new malware, we performed an additional classification experiment.

For each experiment run we split our data set into two parts where one. The first part, which we used for training, contained all goodware samples and the samples of all malware families except for one. The second set contained all samples from the remaining family and was used as test set. To avoid categorization bias we further only considered those families whose samples were all executable in our evaluation environment. This strategy ensured that the training set did not contain any samples from the same family that was used for testing. In consequence the classifier could not gain any knowledge about the to be classified malware family.

With this test procedure we simulated the real-world scenario that our approach faced a sample from a new malware type that was never seen before and thus could not be used for training the detection model.

Figure 5.3 depicts the detection rates that could be achieved for the different malware families. Each bar shows the percentage of all malware samples of a specific family that could be detected using a classifier that was trained on all samples from the remaining malware families.

As we can see, our approach in all cases was able to detect samples from unknown malware families. On average our approach was able to correctly identify 73.68% of the new malware samples; some malware families could even be classified with 100% correctness.

These results for this experiment supports our hypothesis that our approach is capable of detecting new unknown malware and goodware. We account this effect to the fact that most modern malware shares common distribution and initial infection functionality, e.g. by using droppers and connecting to command-and-control servers. Our approach then is likely to identify these common generic behavior while abstracting from malware family specific noise. This assumption is also reflected in the fact that for a few malware families we were not able to detect any sample without prior knowledge of the behavioral characteristics of that family. In this cases the behavior of the malware samples was too different to those used for training, e.g. in consequence of entirely different infection and persistence functionality.

Nevertheless, with these experiments we indicated a certain robustness and generalizability of the obtained detection models towards noise in the QDFGs. In the following we will extent this investigation to also assess the prediction stability of our approach on malware that adds noise and thus tampers with QDFGs by intentionally obfuscating and thus randomizing its behavior and by this hopefully answer our research question *RQ4*.

5.1.3.2. Obfuscation Resilience

Approaches that obfuscate the binary image of malware through build-time code encryption and run-time decryption or code diversification barely have any influence on the detectability through behavior-based detection approaches as such code transformations typically do not alter the externally observable program behavior. In consequence, by construction our approach, like almost all other behavior-based detection approaches, is likely to be widely robust to such used code obfuscation. This assumption is further supported by the ability of our approach to detect variants of malware families that were obfuscated through code transformations. Our evaluations for instance show that we were able to detect 96 of 101 variants of the Harebot trojan from our data set, which is known to employ different forms of code obfuscation.

On the other hand, if malware e.g. non-deterministically executes bogus non-malicious activities or randomly alters between semantically equivalent system calls to achieve the same behavior, it is intuitive to think that it can effectively trick common behavioral approaches that base on n-grams profiling and re-identification as consequence on the unpredictable diverse resulting n-grams. The same holds for most call-graph based approaches as call-graphs can be easily obfuscated by altering or reordering system calls.

Our approach is by construction more robust against call reordering or substitution approaches. This is because reordering of system calls does not alter the corresponding QDFGs, and because semantic substitutions typically exhibit similar data flow properties that result in similar QDFG updates. Moreover, the injection of bogus calls can change QDFGs, in particular if new edges are created in consequence of e.g. previously untouched system entities being read or written to, or if certain operations are repeated such that the edge weights are altered.

To empirically evaluate the absolute effects of different types of behavioral obfuscation techniques on the effectiveness of our approach we thus set up a series of additional experiments.

First, we picked a set of 100 malware and 100 goodware samples as baseline for our experiments. To reason about the obfuscation resilience of our approach and related behavioral detection approaches like n-gram based ones we then step-wise applied behavioral obfuscation transformations on the call traces of these samples to achieve two typical types of behavior obfuscation, namely re-ordering of calls and injection of bogus calls.

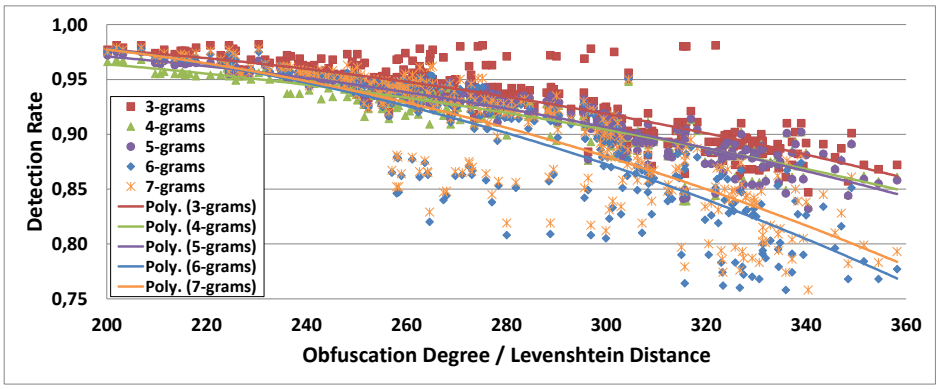
We did so by developing a behavior obfuscation tool [6] and applying it on the malware samples of our evaluation set. This tools obfuscates commodity malware by dynamically instrumenting its behavior at runtime to force it to randomly reorder its issued system calls or injecting new bogus ones that are not part of the core behavior.

By this we could effectively resemble the behavior of malware that actually implements dedicated functionality to obfuscate its behavior.

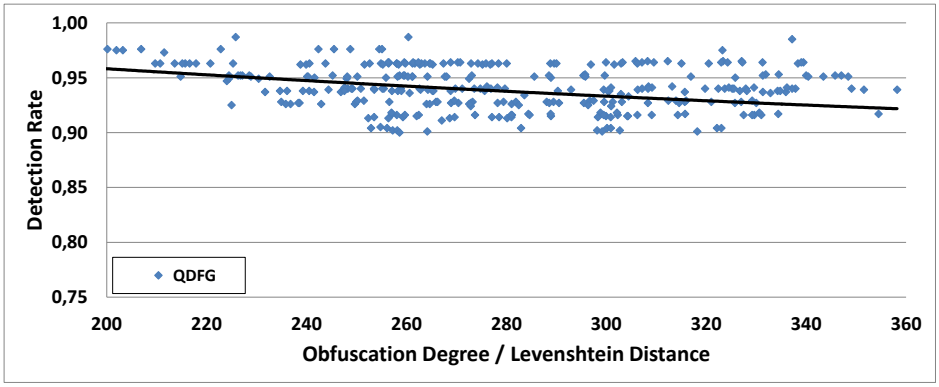
To investigate the effect of increasing degree of behavior obfuscation, we repeated these obfuscation steps 360 times with different configurations for call reordering and injection probabilities, as well as different upper bounds for the amount of to be reordered or injected calls. We represented the degree of obfuscation as Levenshtein Distance between the obfuscated and the non-obfuscated baseline call traces. More specifically, we computed the average number of insertions, deletions, and substitutions needed to transform the non-obfuscated call traces into the respectively obfuscated ones. To also get a relative comparison with other behavioral, raw system call based detection approaches, we conducted these experiments with measuring both, the detection effectiveness of our approach and the effectiveness of a typical behavioral detection approach based on n-grams of unordered system calls without arguments with varying n-gram sizes as e.g. discussed in [24].

As we can see in Figure 5.4, the ability of the n-gram based approach to correctly identify malicious processes significantly dropped with increasing obfuscation degree. In particular, higher-order n-grams seem to be more sensitive to behavioral obfuscation than lower-order ones. In contrast to n-gram based approaches, our approach seems to be considerably more robust and stable towards behavioral obfuscation. While the effectiveness of the n-gram approaches quickly dropped quadratically and lost prediction stability when obfuscation transformations where applied, the effectiveness of our approach at the same time remained rather stable and only slowly dropped linearly.

In sum, our evaluation indicates that we are rather robust with respect to realistic behavior obfuscation such as random bogus call injection or reordering, whereas we could show that common n-gram based approaches are considerably challenged by such obfuscation techniques. This to some extent answers our research question *RQ4* in that, at least with respect to the analyzed obfuscation operations, our approach is indeed widely robust towards behavior obfuscation. In a more recent publication, co-authored by the author of this thesis, we could further extend this evaluation to a broader variation of detection approaches for all of which we could reveal the sensitivity towards behavior obfuscation [6].



(a) n-gram classifier



(b) QDFG metric classifier

Figure 5.4.: Obfuscation experiments

5.1.3.3. Efficiency

The final set of experiments to evaluate our deductive metric-based detection approach aimed at assessing its computational efficiency, i.e. answering research question *RQ3*. Besides the absolute efficiency of the approach we furthermore wanted to asses, whether or not our initial assumption holds in that the metric-based similarity approximation leads to a faster classification than when using expensive sub-graph isomorphism checks.

To this end we individually measured the computational effort that was spend for computing our features, i.e. graph metrics, on the different-sized QDFGs in our evaluation data set, as well as the overall effort to train a classifier on them. Furthermore, we also measured the absolute computation effort for matching a set of QDFG features against a trained classifier to get a classification prediction.

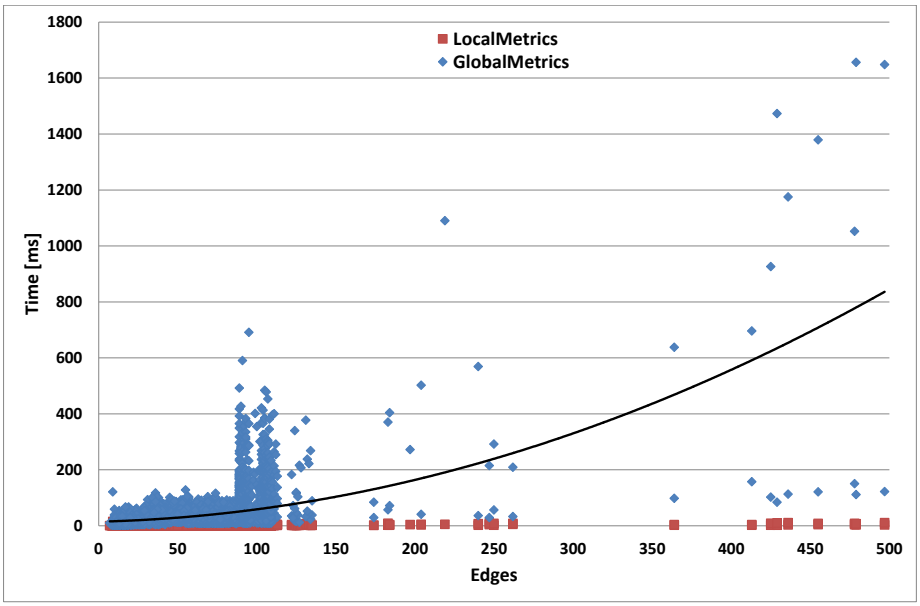


Figure 5.5.: Computation time vs. graph complexity

First of all, we measured the time needed to create the actual detection classifier which consists of the time needed to calculate the features for all training graphs ($454314ms$) and the time needed to train the RandomForest classifier on them ($288ms$). Note that the resulting total training time of about 7.6 minutes only needs to be invested once and does not contribute to the overhead during the detection phase.

As we can deduce from Figure 5.5 the overall detection time of the deductive metric-based approach at average remained below 39ms. The fact hat the biggest overhead is induced by global metric computation is not surprising, as most graph algorithms such the used centrality metrics have a theoretical complexity of $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$, with n being the number of nodes in the graph [16]. On the other hand, the overhead for computing local features remained at an constantly low level as those features only consider a one-hop neighborhood. Finally, the overhead induced by matching the generated graph features against the classification model was below the evaluation precision threshold of 1 ms and thus ignored for this analysis as it has no noticeable impact on the overall overhead.

5.1.3.4. Discussion and Threats to Validity

Naturally, the results and insights gained from our experiments need to be put in the context of our study. First of all the evaluation results were obtained on the basis of a proof-of-concept prototype with event logs generated in a controlled lab environment. Therefore, as for all other machine learning based malware detection approaches, the obtained insights might not generalize to the application of our approach in real-world settings.

We tried to limit the risk of over-fitting the model to specific sub-sets of the training data by pro-actively diversifying the training set by selecting a wide and diverse range of popular malware and goodware samples for our training set. Furthermore, the fact that our cross-validation results show a very low standard deviation supports the assumption that we did not over-fit our models.

Furthermore, our deductive metric-based approach also suffers from the same limitations of generalizability as all other approaches that are based on data obtained from executing malware in controlled sandbox environments. It is known that modern malware often includes functionality to detect virtualization. There hence is a risk that we train our classification models on unrealistic malware behavior and thus are not able to detect their behavior in real-world scenarios.

Because current malware still mainly focuses on avoiding detection by signature-based approaches, it rarely employs advanced behavior obfuscation techniques. Although we applied as much randomization as possible for our obfuscation it can of course not be excluded that our artificially provoked obfuscations do not adequately reflect real-world obfuscation techniques, or that adversaries might come up with more complex behavioral obfuscation operators and advanced mimicry attacks (for instance by learning benign flows and trying to reproduce them). Also, for this study we logged malware behavior for a fix period of time, which might invalidate our result for purposefully stalling malware.

5.1.4. Related Work

Besides the obvious differences between our work and related behavior-based detection approaches in that we are the first and so far only to use quantitative data flow analysis for detection purposes there also exist few other approaches that also proposed to use graph metrics for malware detection.

The work of Jang et al. [68] also leverages graph metrics to discriminate malware from goodware. In contrast to our work, they base the computation of those metrics on system call dependency graphs, while our model is based on quantitative data flow graphs. As we could show, this abstraction improves detection accuracy and increases the robustness towards behavioral obfuscation which gives us better resilience than approaches that directly base on raw system calls.

The work of Mao et al. [92], also published after the conceptual and implementation work on our approach was done, also leverage graph metrics on system entity dependency graphs for malware detection. The main difference between our approach and theirs again is that in contrast to us they do not incorporate any quantitative flow information and consider metrics from a integrity and confidentiality perspective rather than using them to approximate graph similarity.

In sum, the main technical difference between our work and related contributions is that we leverage QDFG features rather than raw system calls or system entity dependency graphs. This makes our approach fast and robust against common types of behavioral obfuscations, and, due to the additional quantitative dimension, we achieve a good detection precision.

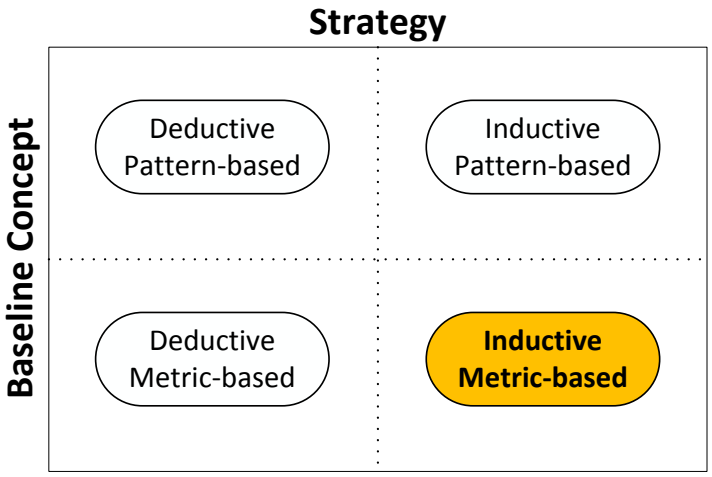
5.1.5. Discussion and Conclusion

Summarizing the main contributions and findings of this section we have presented a novel approach to perform graph metric based malware detection on the basis of quantitative data flow analysis. By profiling quantitative data flow graphs with semantically justified graph metrics we propose to establish an approximate notion of graph similarity. Based on this notion of approximate similarity and a machine learning algorithm to accordingly learn similarities and differences between metric profiles of known malicious and benign QDFGs we are then able to accurately classify unknown samples.

In conclusion, we again showed the general utility of quantitative data flow analysis for malware detection, but this time using a more advanced metric-based instead of a isomorphic pattern-based detection technique, which answers our research question *RQ1*. Furthermore, we showed that our deductive metric-based approach is more than two times faster than our fastest pattern-(isomorphism)-based approach (*RQ3*). Also we could again show that using quantitative data flow information for computing our graph metrics significantly boosts detection precision (*RQ2*).

Our evaluation furthermore showed that the proposed approach is robust to certain classes of behavioral obfuscation: by construction the *order* of system calls is irrelevant, since they produce the same QDFGs, and more interestingly, random injection of system calls that potentially modify both the structure and the original quantities does not significantly alter the detection effectiveness either. This answers our research question *RQ4* as we could show the superior robustness of our approach with respect to other common behavior-based detection models.

5.2. Inductive Metric-based Detection



5.2.1. Introduction

In the last section we showed the general utility of QDFG profiling and approximate similarity checks using graph metrics for behavior-based malware detection. In particular we showed that using a supervised machine learning classifier, trained on features obtained from computing a small set of generic graph metrics on QDFGs of known benign and malicious samples, allows us to classify unknown malware with high accuracy. Furthermore, our evaluations indicated that metric-based detection on QDFGs is faster than using pattern-based detection and that the deductive metric-based detection approach is robust towards common types of behavior obfuscation.

While the deductive metric-based detection approach in comparison to the pattern-based ones already improved upon detection accuracy, it still suffers from a couple of limitations that, as we will later see, hinder it from exploiting the full potential of metric-based QDFG profiling.

The first limitation is that the deductive metric-based approach leverages only slightly modified general-purpose graph metrics for profiling. This is, although we found semantic justifications for them, they originally were not designed for being used in the context of malware detection and mainly borrowed from related fields like social network analysis.

Considering the strong influence of the selection of graph metrics on the overall detection process we thus argue that it might be more reasonable to, instead of using generic metrics for profiling, use metrics that were specifically devised to be used in the context of malware detection. By doing so we expect to achieve higher detection accuracy.

Another limitation of the deductive metric-based detection approach is that it uses fixed sets of metrics for detection. In principle it is possible for a malware to infer the basic functionality of a classifier by iteratively probing it and then devise targeted countermeasures to confuse it or circumvent detection [61, 11]. The more predictable and stable a classifier is, the easier it also is to conduct such counter-analysis techniques. Intuitively, we deem it to be feasible for an attacker or a malware to probe a classifier that operates on data that was generated from applying a fixed number graph metrics on QDFGs and thus infer or at least approximate its decision function with sufficiently high precision to hamper detection. Especially when considering that the form and structural properties of the profiled QDFGs are largely under control of an attacker, i.e. a malware with built-in behavior obfuscation and detection evasion mechanisms, or even anticipating that malware developers would get hands on the exact specification of the used graph metrics, circumventing fixed-metric based detection approaches to us at least theoretically seems feasible.

To tackle these limitations we again follow our basic research methodology of first assessing the general feasibility of a detection concept before exploring its conceptual boundaries. This means, in this section we propose an inductive extension to the deductive metric-based detection approach that instead of using a fixed set of generic graph metrics automatically generates highly characteristic and arbitrarily complex QDFG metrics that a-priori are designed to be useful in a malware detection context. With this we mainly pursue the goal of further improving upon detection accuracy, robustness towards behavior obfuscation, and in particular counter-detection techniques based on machine learning poisoning.

The main motivation behind this inductive extension to the metric-based detection approach is an analogy that we, like others, see between computer viruses and viruses in biology in that both try to evade detection by continuously evolving and adapting its behavior and physical representation [138]. In biology a commonly employed and well-proven successful solution to thwart this situation is to use continuously evolving antibodies to fight continuously mutating viruses. More precisely, through continuous exposure to viruses biological organisms also continuously evolve and adapt antibodies to fight them and in a sense remember which antibody worked well against which virus. The basic idea for our inductive metric-based approach in essence is thus to mimic this concept and adapt it to our malware detection context.

We thus propose to also use a concept that malware often employs to stay below the radar, i.e. evolution, for the large-scale generation of sets of diverse targeted QDFG metrics, which we call *FrankenMods*. By continuously generating and evolving new *FrankenMods* we can adapt our metric-based detection concept to better cater to changes in the malware landscape and find locally optimal performing sets of detection model, i.e. sets of graph metrics that optimally describe behavior specificities of given training sets of malware.

The evolutionary and dynamic aspect comes into play when we, in contrast to our deductive metric-based approach and related metric-based approaches [68, 92] that rely on fixed sets of graph metrics, automatically generate large and diverse sets of graph metrics that through evolution and continuous effectiveness assessment by design are ensured to yield high detection effectiveness.

The rationale for this approach is that, while the metrics used in these approaches usually come with some intuition, their choice essentially is arbitrary. Usually, it is not known if there are metrics that yield even better discrimination power. The identification, derivation, and evolution of "better" specialized metrics is then subject of this section, i.e. main goal of the proposed inductive metric-based approach. In a nutshell, we obtain these specialized metrics by a genetic programming scheme that first randomly generates sets of metrics, the initial FrankenMods; then evaluates their fitness w.r.t. their ability to discriminate known malware from goodware; and then continuously evolves the best candidates. We see reason to assume that the combination or mutation of already well performing (Franken)models, i.e. sets of – in terms of malware detection – already well-performing graph metrics, potentially yields even better models. The genetic programming scheme hence evolves the best-performing models of one evolution round by applying various mutation and combination operations. In the end, this produces a (at least on the training set) optimally performing diverse set of detection models. For sake of brevity, we will use the terms "FrankenMod" and "model" interchangeably in this section.

Malware detection, i.e. finding detection models with good effectiveness, can be interpreted as an instance of the more general optimization problem of finding a function (i.e. detection model) that optimizes certain quality characteristics (i.e. detection effectiveness). From this perspective we consider genetic programming useful to automatically define and optimize effective malware detection models. In sum, by replacing the fixed detection metrics from the deductive metric-based approach by automatically generate and by-design accurate and diverse ones we aim at further improving the overall accuracy of our metric-based detection concept and substantially improve upon robustness towards behaviorally obfuscated malware.

In sum, the contributions of this approach thus are as follows:

- To the best of our knowledge this approach is the first to use a genetic programming scheme in the context of behavior-based malware detection.
- The inductive metric-based approach is able to automatically generate detection metrics that in absolute terms are up to 6.5% more effective than the generic ones used by our deductive metric-based approach.
- Automatically generated sets of diverse graph metrics, i.e. FrankenMods, are up to 9 times more robust towards certain types of behavioral obfuscation than our deductive metric-based approach.

5.2.2. Preliminaries

Before diving in the description of the actual functionality and architecture of the inductive metric-based approach we first need to briefly introduce some preliminaries that are fundamental for the understanding of the subsequent parts of this section. More precisely, in the following we briefly introduce the concept of genetic programming and discuss its utility in the area of malware detection.

Genetic programming leverages concepts from genetic evolution theory for automatically generating algorithms that solve a defined task as well as possible [7]. Genetic programming specializes genetic algorithms in that it also builds on the concept of a population item that represents a solution candidate to solve a defined problem, and a fitness function to measure the quality of this solution, i.e. how well a given population item solves the problem.

Similar to genetic algorithms, genetic programming also uses basic evolutionary concepts like elite selection, mutation, and mating of population items. The underlying baseline assumption is that mutation and mating of already well-performing items likely yield better performance. However, in contrast to standard genetic algorithms where population items usually are inputs for a given function, typically in the context of an optimization problem, for genetic programming the population items are the functions, or algorithms themselves. Thus, while standard genetic algorithms usually define and evolve solution candidates in form of inputs to a defined function (fixed functions and variable inputs), genetic programming typically does the inverse in that they define various functions as population items which, given specific inputs, solve a defined problem (variable functions and fixed inputs). Accordingly, in genetic programming, it is common to describe population items, i.e. functions, as expression trees. In contrast, for genetic algorithms, population items are usually represented by simple vectors of function inputs. Both concepts leverage ideas from biological evolution.

They mutate and mate different population items, i.e. solution candidates, with the goal of continuously improving their ability to solve the given problem. The oracle to assess the quality of a solution candidate is called *fitness function*, typically yielding one real number. By mutating and mating only the best-performing population items in certain settings, such a scheme then can yield better and better performing solutions to a problem without explicitly specifying how such a solution should look like. In that sense both concepts follow the same credo, i.e. *"I do not know how a good solution looks like, but I will recognize one when I see it"*.

Genetic programming is an interesting concept to be used in the context of malware detection for the following main reason. Malware detection approaches are nothing but complex functions that take certain characteristics of a potentially malign sample as input, e.g. system call traces for dynamic detection approaches or PE header properties for static approaches. They output a classification of the sample being more likely malign or benign. The classical malware detection problem can then be seen as an optimization problem where false predictions are to be minimized and detection accuracy is to be maximized. With this observation the relationship between the malware detection problem and genetic programming is straight-forward: detection algorithms are the population items, and their detection effectiveness in terms of correct and incorrect classification decisions yields the fitness function.

5.2.3. Approach

As our inductive metric-based detection approach is an extension to the deductive metric-based approach, its initial and final processing steps are very similar to that of the deductive metric-based approach. In a nutshell, with the inductive metric-based approach we also use sets of graph metrics to profile QDFGs of known malicious and benign samples to establish a fuzzy notion of malicious behavior in term of characterisitc metric profiles. In contrast to our deductive metric-based and other metric-based approaches from literature [92, 68] that built upon fixed sets of manually defined generic graph metrics we aim at *automatically* generating detection models, i.e. sets of highly specialized graph metrics, that by design are effective and efficient for malware detection purposes.

To generate such models we use a genetic programming scheme that randomly generates FrankenMods, i.e. sets of graph metrics; evaluates their discriminatory power; and then mutates and combines the best-performing models to create better-performing ones until a defined termination criteria is fulfilled. Just like the generic graph metric of the deductive metric-based approach, FrankenMods are then used to profile processes through characteristic feature vectors by evaluating the metrics on the corresponding QDFGs.

With these vectors we then train a supervised machine learning classifier that allows us to classify vectors of unknown process samples as benign or malign. In essence our inductive metric-based approach thus consists of the following steps:

1) *Base Data Generation*: Just like for all previously discussed detection approaches, we first generate a set of full system behavior QDFGs that we build from the system call traces that we get from executing known malicious and benign samples in our modified malware sandbox. Like for the deductive metric-based approach we then generate process-centric reachability graphs for each baseline QDFG to restrain the training data set to only encompass behavior that is directly or indirectly related the the executed sample.

2) *Model Candidate Generation*: In contrast to the deductive metric-based approach we then do not use a fixed set of generic graph metrics for profiling the reachability QDFGs but instead first create sets of randomly generated metrics, i.e. the model candidates, as simple functions over basic graph properties. These metric sets, i.e. FrankenMods, are then applied on the evaluation QDFGs to yield numeric feature vectors that then in turn are used to train a supervised machine learning classifier. Applying a standard cross-validation scheme where we continuously split the evaluation data set into non-overlapping training and test sets we then evaluate the individual detection effectiveness and efficiency of each FrankenMod. The effectiveness and efficiency results on the evaluation data set are then what makes up the fitness function that we use for the subsequent evolution steps.

3) *Model Candidate Evolution*: Following the basic assumption of genetic programming that mutating and mating population items, in our case FrankenMods, that already perform well with a high likelihood yield descendants of better performance we then continuously evolve the initial model candidates. The iterative evolution and fitness evaluation process then stops once a certain termination criterion is reached and in the end outputs a set of locally optimal performing detection FrankenMods, i.e. sets of highly specific and complex graph metrics, that then are used for the final detection process.

4) *Detection*: The detection process is very similar to the one of the deductive metric-based approach in that, to classify a unknown QDFG, we first evaluate a FrankenMod, i.e. its graph metrics, on it to yield a numeric feature vector. To get a classification for the analyzed QDFG we then match the obtained feature vector against the classifier that corresponds to the FrankenMod used for generating is.

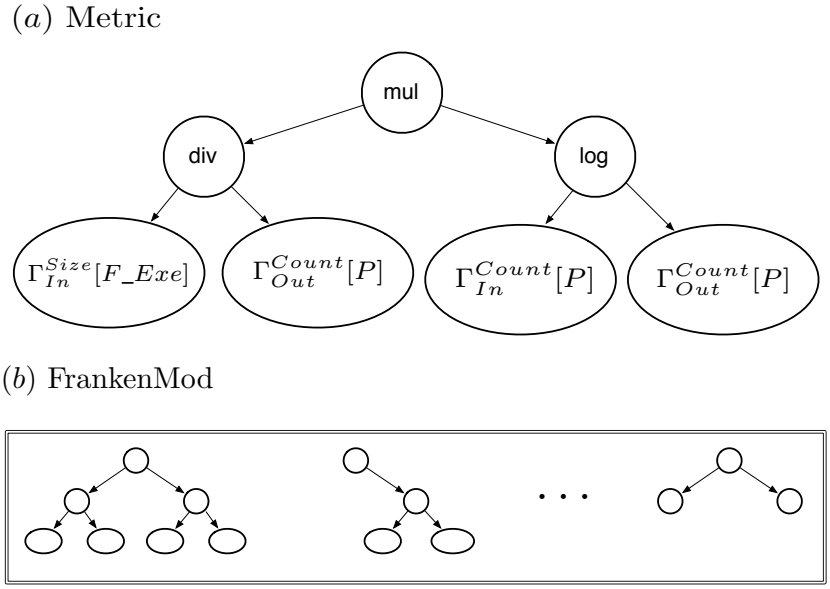


Figure 5.6.: (a) Metric as binary tree and (b) FrankenMod.

After a more elaborate discussion of the concept FrankenMods and the idea of representing graph metrics as complex functions over graph properties we in the following will elaborate on this training and detection procedure in more detail.

5.2.3.1. FrankenMods

As we have already seen in the discussion and evaluation of our deductive metric-based approach, graph metrics are a useful means to approximate the similarity of graphs and thus suitable to establish characteristic behavior profiles if a graph, like our QDFG, model system or process behavior. As one metric in itself is usually not sufficiently expressive to distinctively separate malign from benign QDFGs, we introduce *FrankenMods* (*FMOD*) as core concept of this approach to group multiple metrics in form of ordered sets. We call these sets of metrics *FrankenMods* as they are – just like Frankenstein's monster – created by "gluing together" different metrics by means of an evolutionary scheme and then used as *models* to create the actual feature vectors. These are used to train a machine learning classifier for malware detection.

Listing 5.1: Abstract Syntax in EBNF.

```
FMOD = { ℕ , METRIC };

METRIC = PROP
       | METRIC , OP , METRIC
       | ℤ⁺ , OP , METRIC
       | METRIC , OP , ℤ⁺;

OP = "add" | "sub" | "mul" | "div"| "log" | "pow";

PROP = PROP_LOC | PROP_GLOB;
IO = "In" | "Out";
CS = "Count"| "Size";
GT = "Avg"|"Sum"|"Prod";
PROP_LOC = " Γ_", IO, "^", CS, "[",NODE_TYPE,"]";
PROP_GLOB = PROP_LOC , "[", ℤ⁺ , GT, "]";
NODE_TYPE = "*"|"S"|"P"|"F"|"F_EXE"|"F_SYS"|"F_DLL"|"F_INI";
```

We define metrics as structures consisting of simple graph properties and constants, composed by different algebraic *operators (OP)*. Graph properties in this sense capture specific information about the contextual relation between process nodes within a QDFG and their direct or indirect "neighborhood", that is, they are functions $\Gamma(G, n)$ taking as parameters a graph G and a node $n \in G$. Just like for the deductive metric-based approach we also differentiate between *local properties (PROP_LOC)* that capture the relationship of one node with its direct one-hop neighborhood, and *global properties (PROP_GLOB)* that capture such information for a group of related process nodes, i.e. a process and its parent or child nodes. We will discuss the syntax and formal semantics of such properties in Section 5.2.3.2 in more detail.

Listing 5.1 provides the abstract syntax of a metric structure. Metrics can be nested, thus yielding a binary tree with the nodes being constants, properties, or operators. In order to have a wide range of different properties to define complex metrics, we further extend them by having subscript and superscript parameters, and to restrict them to only consider edges connected to certain node types.

Note that global properties are extensions of local ones, and take the distance to the original node as a parameter. Figure 5.6 (a) shows the tree-representation of an exemplary metric $m = \Gamma_{In}^{Size}[F\_Exe] \text{ div } \Gamma_{Out}^{Count}[P] \text{ mul } \Gamma_{In}^{Count}[P] \log \Gamma_{Out}^{Count}[P]$ which, as we will see later, captures the relation of the amount of data read from .exe files by a specific process and its child processes.

Figure 5.6 (b) shows one FrankenMod which is a set of metrics of possibly different complexity and structure.

We now need to give semantics to the defined structures to apply FrankenMods, i.e. sets of metrics, to nodes of QDFGs in order to obtain vectors of values that can be used as input for a machine learning classifier. To this end, we introduce the function $eval : (\mathit{FMOD} \times \mathcal{G} \times \overline{N}) \rightarrow \mathbb{R}^k$ that, for a given node of a given QDFG, outputs a vector of real numbers by evaluating the metrics $m_i \in M$ of the given FrankenMod M on this pair. As it might happen that a particular metric is not defined for the provided node-graph pair (like division by zero etc.), we introduce the dummy value $\perp$. Formally, $eval(M, G, n) := (s_1, \ldots, s_k)$, with

$$s_i = \begin{cases} v & \text{if } v = eval(m_i(G, n)) \text{ is defined} \\ \perp & \text{otherwise.} \end{cases}$$

Function $eval : (\mathit{METRIC} \times \mathcal{G} \times \overline{N}) \rightarrow \mathbb{R}$ evaluates a single metric on a given node-graph pair by recursively evaluating the nested combinations of basic graph properties that themselves return a real number for a given node-graph pair:

$$eval(m, G, n) := \begin{cases} p(G, n) & \text{if } m = p \text{ and } p \in \mathit{PROP} \\ eval(m_1, G, n) + eval(m_2, G, n) & \text{if } m = m_1 \; add \; m_2 \\ eval(m_1, G, n) - eval(m_2, G, n) & \text{if } m = m_1 \; sub \; m_2 \\ eval(m_1, G, n) * eval(m_2, G, n) & \text{if } m = m_1 \; mul \; m_2 \\ eval(m_1, G, n) / eval(m_2, G, n) & \text{if } m = m_1 \; div \; m_2 \\ log_{eval(m_2, G, n)}(eval(m_1, G, n)) & \text{if } m = m_1 \; log \; m_2 \\ eval(m_1, G, n)^{eval(m_2, G, n)} & \text{if } m = m_1 \; pow \; m_2 \end{cases}$$

with $m_1, m_2 \in \mathit{METRIC}$.

5.2.3.2. Graph Properties

Graph *properties (PROP)* are the basic facts that we can capture in terms of the relationship between a node and its direct or indirect neighborhood. In a sense, together with the introduced basic algebraic operators, properties can be seen as atomic elements needed to define arbitrarily complex detection metrics. Facts in this sense are variations of in-degree or out-degree of a process node, as well as sums of data flows received by or sent to certain types of nodes. Properties map node-graph pairs to real numbers: $p \in \mathit{PROP} : (\mathcal{G} \times \overline{N}) \rightarrow \mathbb{R}$.

Local properties (PROP_LOC) capture facts about the interaction of a process with its direct, i.e. 1-hop, neighborhood. *Global properties (PROP_GLOB)* express the same kind of facts but consider a multi-hop neighborhood of the respective node.

This distinction is necessary as we are interested in both, the direct context in which a process is operating, but also – considering that malware often establishes chains of clone processes to distribute the actual malign behavior – the indirect context that is given by its parent and child processes.

We can now assign semantics to both types of properties and give rationales why we consider them relevant in the context of quantitative data flow driven malware detection.

5.2.3.3. Local Properties

Local properties capture characteristics of the direct neighborhood of a process node. Examples range from the number of child processes of a specific process, i.e. number of successor process nodes, to data flow related properties like the amount of data received from .exe-files, i.e. the sum of the size of all incoming edges that originate from nodes that refer to .exe-files.

The names of local properties follow the schema in Listing 5.1 that reflects their semantics. The structure determines whether the property expresses a fact concerning the incoming (In) or outgoing edges (Out) of the to-be-profiled node. It also determines whether the property expresses a data flow related fact (Flow) or only a cardinal aspect (Count). Finally, the naming scheme also specifies if all ($*$) incoming or outgoing edges of a node should be considered for computing the property, or if the computation should be constrained to only consider edges that are attached to certain type of nodes (Node_Type). The semantics of a local property $pl \in PROP\_LOC$ is generically defined as

$$pl(G,n) := \begin{cases} \sum_{e \in CE(G,n,t,In)} \lambda(e, size) & \text{if } pl = \Gamma_{In}^{Size}[t] \\ \sum_{e \in CE(G,n,t,Out)} \lambda(e, size) & \text{if } pl = \Gamma_{Out}^{Size}[t] \\ |CE(G,n,t,In)| & \text{if } prop\_loc = \Gamma_{In}^{Count}[t] \\ |CE(G,n,t,Out)| & \text{if } prop\_loc = \Gamma_{Out}^{Count}[t] \end{cases}$$

where CE represents the adjacient edges of n connecting to nodes of type $t \in$ NODE_TYPE and direction dir:

$$CE(G,n,t,dir) := \begin{cases} \{(src,n) \in in(n,G) | \lambda(src, type) = t\} & \text{if } dir = In \\ \{(n,dst) \in out(n,G) | \lambda(dst, type) = t\} & \text{if } dir = Out \end{cases}$$

An example of a local property is $\Gamma_{In}^{Size}[S]$ that returns the amount of data that a given process node received from connected sockets, expressed as the sum of the sizes of the respective incoming edges.

The choice for this selection of local properties was driven by the insights from a manual analysis of a wide range of malware QDFGs. We identified a strong correlation between malicious process nodes being highly interconnected with .exe-file nodes and, at the same time, having many connections to other process nodes (e.g., Parite virus). We are aware that this property is shared by certain installer processes and thus in itself is not discriminative enough. In consequence, we needed to extend these properties to allow for a more fine-grained, differentiated profiling and thus reduce false-positives.

5.2.3.4. Global Properties

Global properties consider a variable k-hop distance around a node. The rationale is that malware often creates process chains to distribute the malign behavior among multiple children and grand-children processes. Local properties cannot capture such distributed behavior. We define global properties as higher-order functions that take a local property function and a distance bound as argument but, instead of evaluating it on just one node-graph pair, apply it to the entire process neighborhood of a given process node.

Regardless of the input, property functions always need to map to one single real number to ensure feature vectors of constant dimension. Considering that a naive application of the respective function on an entire process neighborhood would yield a set of real numbers, we need to map the obtained value set to one singe real number. We hence first evaluate the function on a defined neighborhood, i.e. on a set of process node-graph pairs, and aggregate the obtained value sets by either summing (Sum) or multiplying (Prod) them, or by computing their average (Avg). While this aggregation certainly leads to information loss when compared to the entire value set, the aggregation is necessary to make global properties syntactically compatible to local ones. The semantics of a global property $pg \in PROP\_GLOB$ is defined as:

$$pg(G,n) := \begin{cases} \sum\limits_{n' \in NBH(G,n,k)} pl(G,n') & \text{if } pg = pl[k, Sum] \\ \prod\limits_{n' \in NBH(G,n,k)} pl(G,n') & \text{if } pg = pl[k, Prod] \\ \frac{\sum\limits_{n' \in NBH(G,n,k)} pl(G,n')}{|NBH(G,n,k)|} & \text{if } pg = pl[k, Avg] \end{cases}$$

with $pl \in PROP\_LOC$ and

$$NBH(G,n,k) := \begin{cases} \{n\} & \text{for } k = 0 \\ \bigcup\limits_{n' \in (pre(n) \cup suc(n))} NBH(G,n',k-1) \cup n' & \text{for } k > 0 \end{cases}$$

where $\lambda(n, type) = \lambda(n', type) = Process$.

An example is $\Gamma_{Out}^{Count}[F][2, Sum]$ that computes the total number of files that were read by a process node, its parent, its great-parent, its children, or its great-children.

Now that we have set the conceptual foundations for our concept of FrankenMods, we can begin to describe how we instantiate and evolve them in order to continuously evaluate and evolve them with the goal of in the end producing locally optimal performing detection models.

5.2.3.5. Evolution Process

The complexity of a metric is determined by the depth of its expression tree. The most simple metric that combines two atoms with one operator is thus of complexity 1. As it is infeasible to generate and evaluate all possible metrics that can be constructed on the basis of the introduced operators, we cut down the set of relevant metrics by applying a genetic evolution scheme to randomly determined subsets of all possible metrics. Evolving them creates increasingly complex descendants with the goal of determining subsets of metrics with close to optimal detection capabilities. This evolution is repeated until a specific termination criterion is reached.

5.2.3.6. Initialization

In the initialization phase an initial population of k FrankenMods is generated where each FrankenMod consists of $l = 3$ randomly generated metrics of complexity $n = 1$. A metric is generated through random creation of nodes in the respective expression tree with the node types being uniformly chosen from the introduced operator and property sets. By starting the evolutionary process with such simplistic FrankenMods, we can easily investigate the correlation between FrankenMod complexity and performance, as well as the effect of evolution, by looking at the performance of the children of the respective simple FrankenMods in subsequent evolution steps.

5.2.3.7. Fitness Evaluation

After the definition of an initial population we assess the performance of the individual members, i.e. FrankenMods, following a similar process as we did use for evaluating the deductive metric-based approach (see Section 5.1.3.1). In genetic programming terms, in this evaluation step we determine the fitness of each FrankenMod in terms of discriminating known benign from malign samples. To this extent we evaluate the generated FrankenMods on a large and diverse labeled evaluation data set, consisting of the captured activity of multiple known benign and malign samples. We do so by computing all metrics contained in each FrankenMod for each process-centric reachability graph that we can extract from the sample's baseline QDFG. For each FrankenMod we thus obtain a list of labeled vectors, where each vector corresponds to the results of evaluating each metric included in the FrankenMod on one specific process node of a QDFG. For the evaluation data set, we know the identity of the respective processes, and we can hence label the respective vectors accordingly as *malign* or *benign*.

We subsequently assess the fitness of a FrankenMod by a standard cross-validation experiment on the generated labeled vectors by using them as features for training and testing a supervised machine learning classifier. We perform a 10-fold cross-validation experiment for each FrankenMod, where we repeatedly take 9/10 of the generated process vectors for training a Support Vector Machine (SVM) with Gaussian kernel; and then use the obtained classifier on the instances of the remaining 1/10 of the generated process vectors that was not used for training. As all vectors sets are labeled, we can then assess the fitness of the used FrankenMod. We used the AUC of the classifier to measure the fitness of a FrankenMod as it nicely captures a FrankenMod's ability to discriminate goodware from malware with good accuracy. Applying this notion of fitness, we then determine the fitness of all population members as prerequisite for determining an elite of best-performing FrankenMods to be considered for the subsequent evolution step.

In addition to determining the fitness of a FrankenMod in terms of its classification effectiveness we also evaluate the absolute efficiency of that FrankenMod. As we can expect a correlation between FrankenMod complexity and model computation time, we also measure the average time it took to evaluate the respective FrankenMod, i.e. compute it on the training and testing QDFGs. As this efficiency directly relates to the computational effort that is needed to later use a FrankenMod for malware detection, this allows us to reason about the relationship between FrankenMod efficiency and effectiveness, and potentially also to find an optimal trade-off between these two quality aspects.

After conducting the evaluation step, each population item (FrankenMod) has a pair of values associated to it that represent its absolute effectiveness (AUC) and efficiency (Average Detection Time).

Finally, to prevent excessive growth of the population with FrankenMods with little effectiveness, we sort the population with respect to the AUC of the respective members and prune it by only keeping a fixed number of k' best-performing FrankenMods for the next evolution step.

5.2.3.8. Evolution

At this point, we can potentially stop the evolutionary process depending on whether a defined evolution termination criterion is met. To this extent we evaluate a set of pre-defined termination criteria on the created and evaluated population. A more detailed description of potential termination criteria will follow in Section 5.2.3.10.

Following our hypothesis that mutants or combinations of already well-performing FrankenMods have a high likelihood of performing better than the underlying base models, we now describe the actual evolution phase. The idea is to apply a set of evolution operators to an elite subset of the initial population to create mutants or child metrics from this elite population. To this end we introduce two different classes of evolution operators: *mutation operators* that take one metric as input and return a slightly mutated metric, and *cross-over operators*, that take two metrics as input and return a derived metric obtained from mating the two input metrics. While mutation operators only slightly vary a metric through targeted structural modifications, crossover operators create entirely new metrics.

To mutate one metric we propose a node- and a tree mutation operator. The node mutation operator (†) randomly replaces an operator or a property with another randomly determined one. We do so by randomly determining a node of the metric tree and then, if it is a operator, replace it with a different randomly chosen one, or, if it is a property, replace it with a different property. Figure 5.7 depicts the application of the node mutation operator † on a metric m, where one randomly chosen operator node of m (*div*) is substituted by a different operator (*mul*).

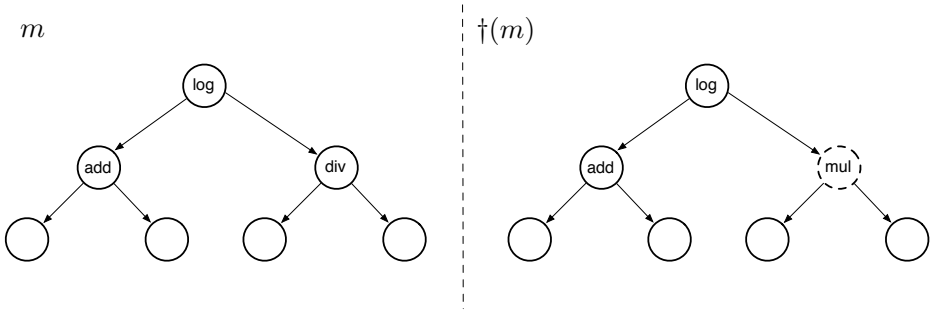


Figure 5.7.: Example of *Node Mutation* operator

The second mutation operator, called tree mutation operator (↕) not only flips single property or operator nodes but rather entire chunks of a metric's formula, i.e. branches of the respective metric tree. More specifically, the tree mutation operator randomly picks a node and then flips this node's child branches. Depending on the metric complexity, i.e. the size of the respective expression tree, and the position of the picked target node, this can yield a significant structural change of the underlying metric. An example is depicted in Figure 5.8.

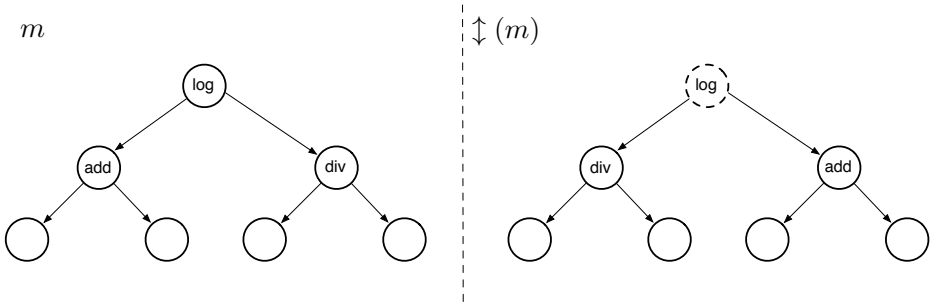


Figure 5.8.: Example of *Tree Mutation* operator

The second group of evolution parameters, the *cross-over operators*, is used to mate two metrics to yield a new, hopefully better-performing, metric and consists of two distinct operators. The *combine cross-over operator* (⋈) combines two distinct parent metrics to form a new child metric that consists of a new operator node of random type as root and connecting the parent metric trees as left and right child branches of the new root node.

Figure 5.9 depicts an example were the parent metrics m_1 and m_2 are combined by attaching them as child branches to a newly created *log* operator.

The set of cross-over operators also contains an add operator (⊎) that operates on FrankenMods instead of concrete metrics. The add cross-over operator takes a randomly determined metric m_i from a randomly determined FrankenMod M' from the elite population subset and adds it to the FrankenMod M upon which the add cross-over operator was applied to. Formally, ⊎(M) =$M \cup \{m_i\}$ where $m_i \in M'$ for a randomly chosen $M' \neq M$.

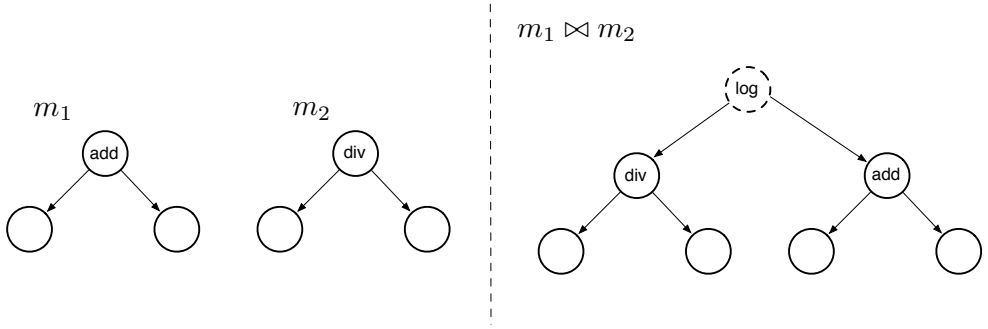


Figure 5.9.: Example of the *Combine Cross-Over* operator

Now that we have introduced a set of basic evolution operators, we perform the evolution steps by iterating over all FrankenMods of a defined elite population. For each metric of a FrankenMod with a defined probability we pick and apply one of the previously introduced evolution operators (†,↕,⋈,⊎) or a so-called *nop-operator* that does not perform any transformation of the target metric. The to-be-evolved elite portion of the population is determined by ordering the respective FrankenMods according to their fitness and then taking the $k \geq 2$ best FrankenMods as elite population.

By assigning different probabilities of picking a certain evolution (or nop-) operator we can drive the likelihood of a certain metric to get mated with another metric, to just get mutated, or to not get altered at all. Finally, we feed all generated mutants and FrankenMod children back to the population and with this conclude the first evolution round.

5.2.3.9. Iterative Repetition

We now repeat the previous steps by jumping back to the evaluation step 2 to evaluate the performance of the new generation of FrankenMods. The process is repeated as long as none of the pre-defined termination criteria are met. Such termination criteria can either define a certain necessary minimum quality of the FrankenMods of the elite part of the current population generation, e.g. a defined minimum AUC of the respective FrankenMods, or specify some constrains on the overall evolution process like a maximum number of evolution steps, i.e. process iterations, until the evolutionary process has to be terminated. By means of such termination criteria we can control the evolutionary process in a targeted way and e.g. prevent uneconomic repetitions of the expensive evolution process if it is likely that this will not produce significantly better results.

5.2.3.10. Termination

After the evolutionary process stops, we pick an arbitrary number of FrankenMods from the last population generation to be used for the subsequent detection phase in an IDS. We do so by sorting the respective FrankenMods first on the basis of their effectiveness (AUC) and then again on basis of their efficiency (FrankenMod Computation Time) for all FrankenMods with equal effectiveness. As the FrankenMods themselves can not directly be leveraged for classifying unknown samples, we instead output a set of classifiers that we obtain by training a Support Vector Machine (SVM) with Gaussian kernel on the labeled feature vectors obtained from applying the respective FrankenMods on the entire QDFG data set that was also used for the evolution step. Finally, we export those classifiers along with the respective FrankenMods to be used by our detection component in an IDS. Note that the metrics used by the generated FrankenMods before exporting them always get optimized by the employed .NET expression tree compiler by means of various compiler optimization techniques.

To summarize the core ideas of the evolutionary process, Figure 5.10 depicts an excerpt of two rounds of a possible run. In round n+1, a new FrankenMod $FM3$ is created by mating two FrankenMods $FM1$ and $FM2$ using the $\uplus$-operator, i.e. merging the metrics of $FM1$ with one metric $m5$ of $FM2$. In the next round, n+2, the $\updownarrow$-operator is applied to $FM3$, yielding the mutant $FM3'$ by inverting the main branch of its metric $m5$. We further see how the FrankenMods are evaluated on the known malign (M) or benign (B) process nodes ($p1$-$p7$) of some QDFGs ($QG1$-$QG3$) to yield feature matrices as input for training and evaluating a classifier and thus determining the FrankenMods's fitness.

5.2.3.11. Deployment and Detection Phase

After generating and exporting effective and efficient FrankenMods, their actual operationalization is straight-forward and follows the same basic detection principle as already discussed for the deductive metric-based approach (see Section 5.1.2.8). For classifying an unknown QDFG we evaluate one or more of the previously obtained FrankenMods on all process-centric reachability graphs of the respective baseline QDFG. This process, just as described in Section 5.2.3.5, yields a feature vector for each process node of the baseline QDFG that is compatible with the machine learning classifier associated with the respective FrankenMod. By matching the obtained feature vector against this classifier we then can determine if the process (node) is more likely benign or malign based on the similarity of the respective metric values with the training data.

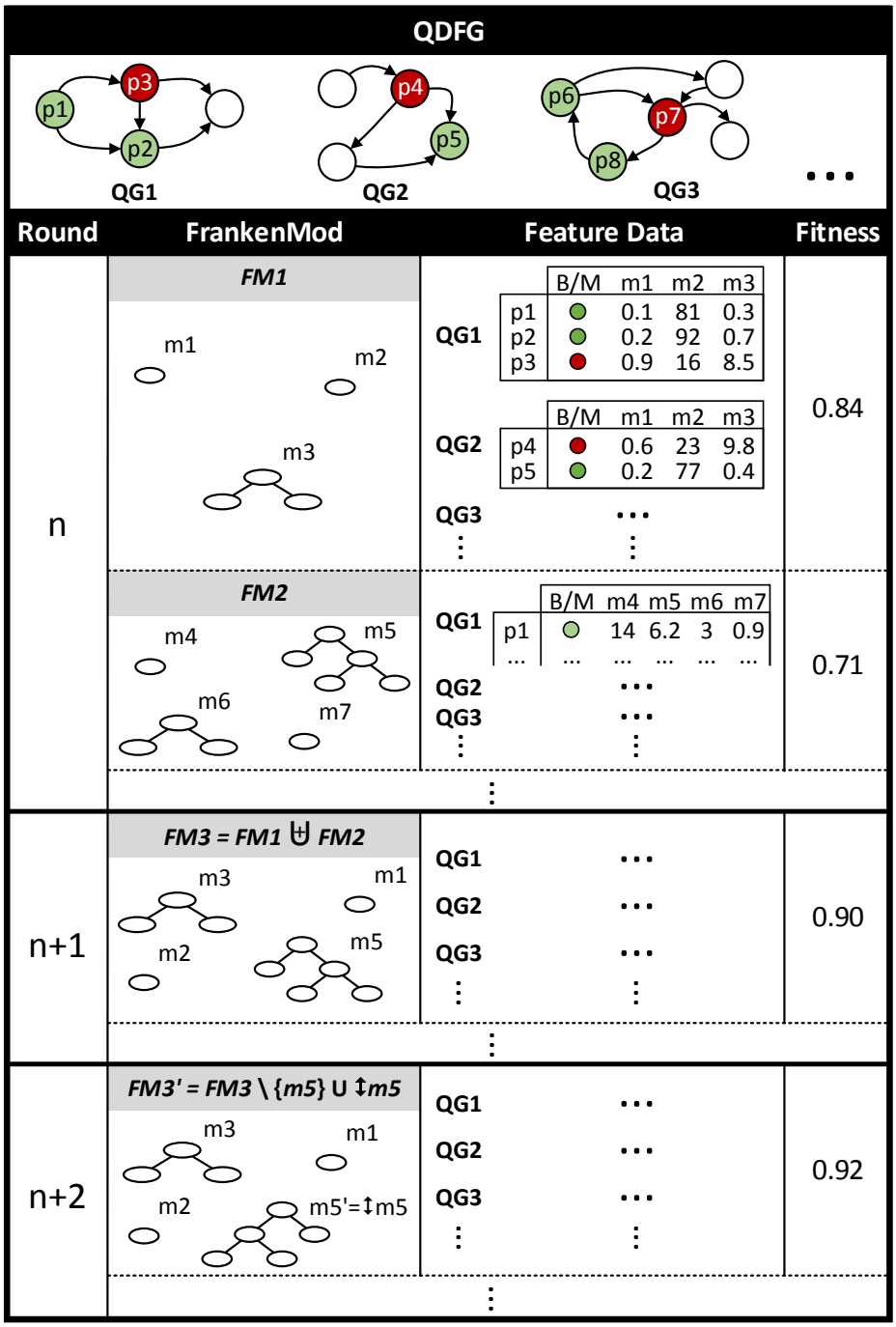


Figure 5.10.: Overview and example evolution.

5.2.4. Evaluation

Considering that the inductive metric-based approach is an extension to our deductive metric-based approach with the goal of improving its effectiveness and robustness, we in the following mainly focus on assessing the actually achieved effectiveness and robustness improvement. We furthermore wanted to investigate, whether the targeted generation of metrics has a positive impact in the induced computational cost for detection. The used evaluation setup and data set resembled the one discussed in Section 4.1.3.

5.2.4.1. Effectiveness

With respect to effectiveness, we were primarily interested in investigating the three following research hypothesis:

(H3) Generated metrics, i.e. FrankenMods, yield higher effectiveness than the generic graph metrics used by the deductive metric-based approach.

(H4) Complex graph metrics in form of polynomials over multiple atomic graph properties yield higher effectiveness than single atomic graph properties.

(H5) The evolution of well-performing FrankenMods, i.e. their mutation and combination, is likely to yield even better-performing FrankenMods.

To investigate the first research hypothesis (H3) we assessed the maximum achievable detection effectiveness of FrankenMods and fixed detection models. For this we first again evaluated the effectiveness of the detection model of the deductive metric-based approach, which we call *generic* as it consists of a set of generic graph metrics (see Section 5.1.2.1), on our evaluation data set. To avoid comparison bias due to using different machine learning algorithms for both detection approaches, we replaced the RandomForest classifier that originally was used by the deductive metric-based approach with the same type of SVM that we also used for the FrankenMod evolution scheme (see Section 5.2.3.7).

Furthermore, to assess H4, we created another simplistic model of generic graph metrics, which we call *complete*, that simply outputs all graph properties described in Section 5.2.3.2 to be directly used for classification without algebraic composition. In a sense that can be seen as creating a FrankenMod that consist of metrics that only consist of atomic graph properties without further composition with algebraic operators. With this we wanted to emphasize the utility of using complex metrics, as subsets of graph properties combined with algebraic operators, versus directly using the atomic graph properties for detection.

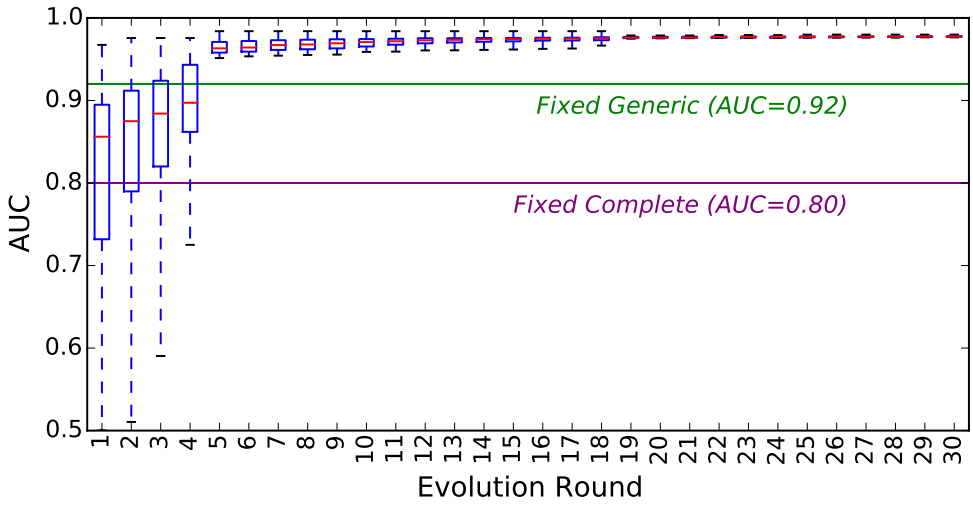


Figure 5.11.: Effectiveness FrankenMods vs. fixed Models

To compare the effectiveness of the generated FrankenMods with the complete and generic ones, we also applied our inductive evolution-based approach on our evaluation data set, starting with an initial population of 100 randomly generated FrankenMods of complexity 1, configured to choose the to be applied evolution operators uniformly, and set the termination criteria to stop after 30 evolution rounds. Figure 5.11 depicts the AUC distributions of the respectively generated populations for all evolution rounds. To directly compare the effectiveness of the generated FrankenMods against the effectiveness of the fixed detection model and the complete model we also depict the AUCs of these models in the same graph.

As we can see, the evolution scheme converges after around 19 evolution rounds, i.e. the standard deviation around the median AUC becomes almost 0. Also we can see that after round 5, at least one FrankenMod of the population reached the local optimum of the entire evolution experiment. From the Receiver Operator Characteristics Curve (ROC) of our best-performing FrankenMod we can furthermore deduce that the best-base detection rate when accepting a maximum of 0.5% false positives was at 97.1%, with an AUC of 0.986.

Although the median AUC of the generated FrankenMods for the first evolution steps remains below the AUC that can be achieved with the fixed generic detection model (0.92), after only 5 evolution steps the AUCs of the generated FrankenMods already exceed that of the fixed generic model.

In comparison to the fixed complete model we can see that right from the beginning the performance, i.e. the median AUC, of the generated FrankenMods, i.e. combinations of randomly selected graph properties with randomly determined algebraic operators, resides significantly above the AUC of the fixed complete model (0.80). Looking at the AUC of the best-performing FrankenMod, i.e. the local optimum of this experiment, we can furthermore deduce a maximum improvement of effectiveness of about 6.5% ($= \frac{AUC\_generated - AUC\_fixed\_generic}{AUC\_fixed\_generic} = \frac{0.98-0.92}{0.92}$) in relation to the AUC of the fixed generic model. In comparison to the fixed complete model we could achieve an even higher improvement of detection effectiveness with the evolutionary generated FrankenMods of about 22.5% ($= \frac{AUC\_generated - AUC\_fixed\_complete}{AUC\_fixed\_complete} = \frac{0.98-0.80}{0.80}$).

In sum, these observations confirm H3 positively, since we could show that FrankenMods can significantly outperform fixed ones. In particular we could show that indeed FrankenMods produced by the inductive metric-based approach outperform the generic metrics that are used by our deductive metric-based approach. Furthermore, the significant difference of effectiveness between the FrankenMods produced by the evolutionary algorithm and the complete model of atomic graph properties also answered H4 since we could show that complex graph metrics outperform models built on simple graph properties.

Besides confirming our approach-specific research hypotheses, this also again positively answers our main research question *RQ1* in that with this approach we could further improve the effectiveness of QDFG-based malware detection.

Finally, to also assess our research question *RQ2* for the inductive extension to our metric-based detection approach, i.e. to show that the consideration of quantitative data flow aspects indeed positively influences detection effectiveness we performed another set of experiments. For these experiments, just like we did for the evaluation of the deductive metric-based approach (see Section 5.1.3.1), we either a) artificially fixed the size of all data flows in the evaluation QDFGs by setting the respective edge weights to 1, or b) artificially obscured the data flows by replacing the real edge weights with random numbers.

With this we effectively destroyed all quantitative data flow information in the evaluation graphs. On both sets we then applied our normal evolution process and, just like for the initial real quantity experiments, set the termination criteria to stop after 30 evaluation rounds. The best-performing generated FrankenMod then for the fixed quantities case (a) yielded an AUC of 0.978 and a BDR (detection rate at 0.5% false positive rate) of 61.3% and for the random quantities case (b) an AUC of 0.977 and a BDR of 56.3%. In both cases the absolute (AUC) and the operational (BDR) effectiveness of the FrankenMods trained and evaluated on QDFGs with fixed or random obscured quantities were significantly lower than the best-performing ones that were generated on QDFGs with real quantities.

Although the effectiveness improvements when quantities for the inductive metric-based approach were less substantial than for the deductive metric-based one, the results also for our inductive metric-based approach positively answer *RQ2*.

Concerning the utility of our evolution concept, Figure 5.11 already gives us a preliminary answer to hypothesis H5 as we can see that the evolution scheme produces FrankenMods with increasing effectiveness with every evolution round. To investigate this observation in more detail we plotted the lineage graph to depict the interdependency of the different generated FrankenMods over multiple evolution rounds. Figure 5.12 shows the lineage graph of all FrankenMods of the above described evolution experiment.

The nodes in this graph represent the different generated FrankenMods whereas the edges between them denote their inheritance relation, i.e. whether a FrankenMod is the result of mating two other FrankenMods, or if it is just a mutant of another one. To quickly see a model's effectiveness within the lineage graph we use the color of the lineage graph nodes to represent the AUC of the respective FrankenMods – the darker the node, the lower the effectiveness, i.e. AUC, of the respective FrankenMod. The layout of the lineage graph correspondingly reflects the *"age"* of the FrankenMod as distance from the FrankenMods from the first round. With this the layout algorithm positions FrankenMods that were created in the first evaluation rounds more closely to the center of the lineage graph, whereas the perimeter is populated by nodes that represent FrankenMods that were created in the last evolution rounds. Note that for presentation reasons we only considered the models generated within the first 23 evolution rounds.

Looking at the actual lineage graph depicted in Figure 5.12 we can see that the average model effectiveness improves with increasing age. In particular, looking at the highlighted lineage of one of the best-performing nodes reveals that indeed the mutation and mating of models from earlier stages of the evolution process mainly yields better-performing ones, thus confirming our hypothesis H5.

5.2.4.2. Efficiency

For investigating the efficiency of our approach we compared the relation between the detection efficiency of using the generic detection metrics employed by the deductive metric-based approach and the FrankenMods generated by the inductive metric-based one. For this we first evaluated the *detection* efficiency of the inductive metric-based approach, i.e., how expensive it is to employ the generated models for detection; and then the *generation* efficiency, i.e. how much computational effort is needed to conclude one evolution round to output a set of models with a target effectiveness.

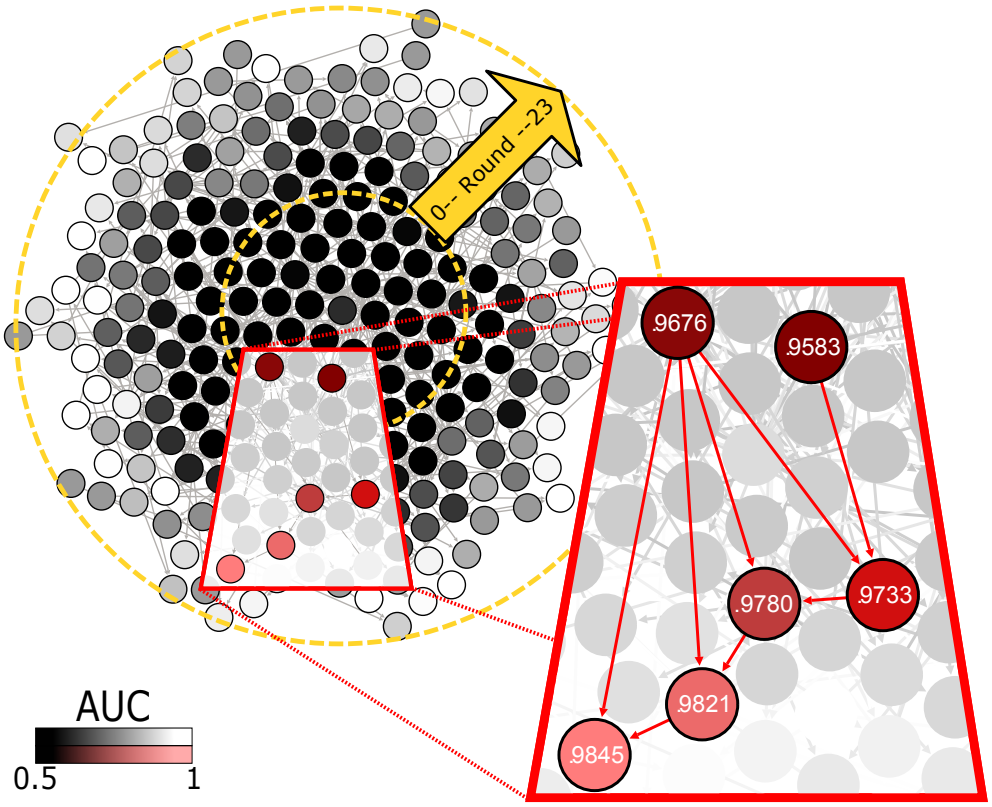


Figure 5.12.: FrankenMod lineage graph

As we already discussed in Section 5.1.3.3, the generic graph metrics used by the deductive metric-based approach are of either quadratic or cubic complexity w.r.t. the number of edges of the reachability QDFG of the to-be-classified process node, rendering the theoretical worst-case complexity of the respective fixed generic model to be in $\mathcal{O}(e^3)$ with e being the edge count of the respective QDFG.

Looking more closely at the algorithmic structure of the graph properties used by the inductive metric-based approach reveals that the local properties are of linear complexity w.r.t. the number of incoming and outgoing edges of a to-be-classified process node and the global properties are of linear complexity w.r.t. the transitively reachable edges of a node determined by the (fixed) depth of the respective recursive algorithm.

Note that for assessing the worst-case complexity of an algorithm, or in our case of a detection model, constant factors like the number or type of operators used to combine multiple graph properties in form of metrics, or the number of used graph properties itself do not play a role.

Therefore, the worst-case computational complexity of a model generated by our approach is bounded by the complexity of the most expensive graph property and thus in the worst case still is linear in the number of QDFG edges, i.e. remains within $\mathcal{O}(e)$.

Recall that in Section 5.2.4.1 we already defined a (complete) model that uses all atomic graph properties for detection. For assessing the actual performance of a maximum complex model on real-world data we thus could simply measure the computational effort needed for applying this fixed complete model on real-world QDFGs. Intuitively, we would thus assume the actual computational effort needed to evaluate evolutionary generated FrankenMods to reside below the cost of evaluating the fixed complete model.

To verify this assumption we thus measured the time it took to evaluate evolutionary generated models of different complexity, i.e. from different evolution rounds, as well as for applying the considered complete and generic models on our evaluation data set.

Figure 5.13 depicts the time needed to evaluate the different model types on all reachability QDFGs (of different size) that originated from the entire evaluation set. While the small picture in the upper left part of the figure depicts the full range of average detection times of all approaches, the compete figure only shows the part of the scale that was relevant for the generic metric and the FrankenMod experiments to emphasize the difference between them.

As we can see, the complete metrics are significantly more expensive than the generic metrics or the generated ones, i.e. Frankenmods, and quickly go in the order of hundreds of milliseconds even for small graphs. The time difference between the generic and the generated metrics in comparison is rather small with a slight efficiency advantage of generic metrics over generated FrankenMods for graphs with less than 250 edges. Only for graphs bigger than this, the generated FrankenMod metrics outperform the generic ones.

In sum, the FrankenMod metrics at average took 54 ms to evaluate and thus were about 1.4 times ($= \frac{time_{FrankenMod}}{time_{generic}} = \frac{54ms}{39ms}$) slower than the generic metrics.

By looking at the slope of the polynomials that we fit on the detection time distributions of the different approaches we can further reason about their scalability. Here the detection time of the generic metrics grows about 1.8 times ($= \frac{slope_{generic}}{slope_{FrankenMod}} = \frac{0.38}{0.21}$) faster than the detection time of our generated FrankenMods. This reflects in significantly different detection times for the biggest graphs in our evaluation set that took more than 2 times ($= \frac{max(time_{generic})}{max(time_{FrankenMod})} = \frac{340ms}{167ms}$) longer to classify with generic metrics than with our FrankenMods.

From these experiments we can deduce that, although at average performing slightly worse than the generic ones, the generated metrics are substantially faster on bigger-sized graphs.

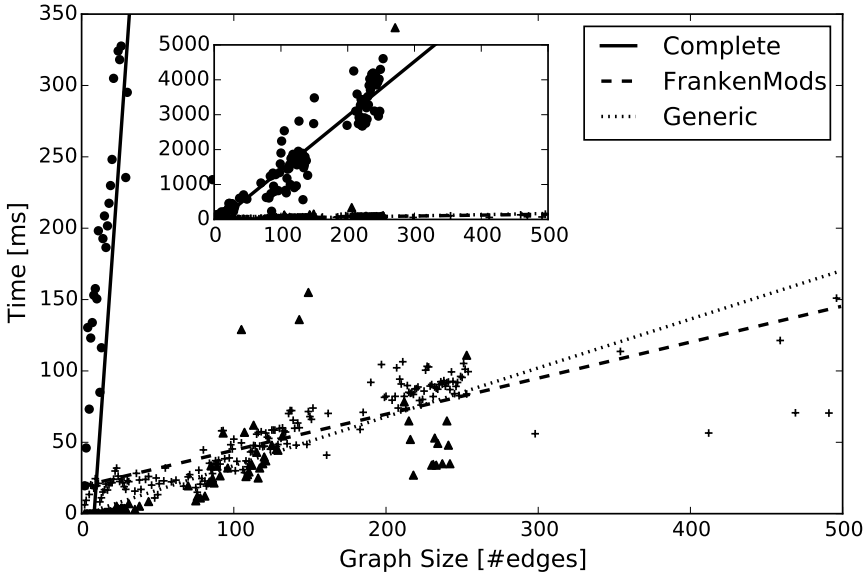


Figure 5.13.: Detection time vs. QDFG size

We account this effect to the fact that the quadratic or cubic complexity of the generic centrality metrics only kicks in for complex graphs, whereas for smaller graphs they are comparably cheap to compute. The evolutionary process, i.e. the fitness evaluation to assess the generated FrankenMod metrics in contrast takes their global performance on all graph sizes into account and thus likely favors metrics that also perform well on bigger-sized graphs.

Finally, for evaluating the *generation* time efficiency of our approach, i.e. to assess how long it takes to generate a stable population of a defined target effectiveness, we measured the average computation time a evolution round took to conclude. The absolute computational effort depends on various factors like model complexity, population size, or size of the evaluation data set and can thus not be generalized to all possible configurations. Nevertheless, investigating the development of computational demands in-between different evolution steps gives us insights into the overall scalability of the approach.

Figure 5.14 shows a stacked bar chart for the corresponding measurements within our evaluation setting. The bars indicate the amount of time spent to generate and evaluate all models of one population generation, i.e. the time needed to complete one evolution round. The gray part of the bars represent the time needed to compute all models of one generation on the entire evaluation data set, whereas the

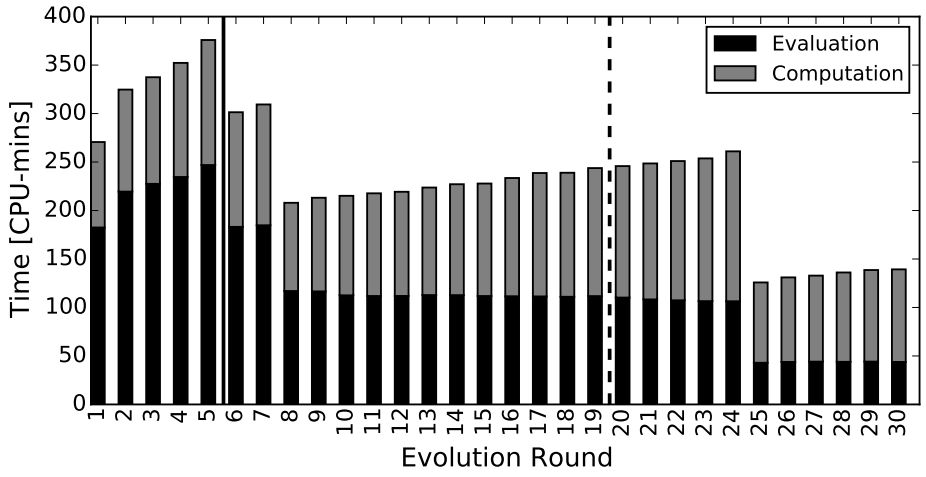


Figure 5.14.: Duration of evolution rounds

black parts refer to the time spent for evaluating the respective models, i.e. determining the model fitness with cross-validation on the data obtained from the previous model computation step. Note that with time we refer to CPU time. Given that the generation and evaluation of models can be parallelized we can almost linearly cut down the execution time with the number of available CPU cores.

Generating and evaluating the initial population took about 4 CPU hours with 1/3 of the time for model computation and 2/3 for the model fitness evaluation; generating the features for all evaluation graphs ($404878ms$) and training an SVN on them ($14547ms$) at average took about 7 minutes. The following evolution rounds then increasingly take longer to conclude with a proportional shift of model computation vs. evaluation time. This can be explained by the fact that more complex models resulting from the proceeding evolutionary process take more time to compute and evaluate than more simple models from earlier rounds. Interestingly though, we can see a drop of overall time needed to conclude a evolution round after the 5th evolution round. This drop is explained in that after 5 rounds the evolutionary algorithm produced enough more simple and effective models to be able kick out the worst-perfroming ones from the previous rounds. This tooth-chain pattern is then repeated in irregular intervals. Looking at the generated models, we could always verify the aforementioned reasoning.

If we correlate these insights with the development of population effectiveness over the different evolution rounds we can furthermore deduce that the first neuralgic point, i.e. the point where for the first time the median population effec-

tiveness exceeded that of the fixed generic model (solid vertical line), is reached after about 28 hours of computation time. The second interesting point, where for the first time all of a generation exhibit almost the maximum achievable effectiveness (dashed vertical line), is reached after about 83 hours of computation time after which the accumulated population effectiveness does not significantly increase anymore.

In sum, we can conclude that it takes the inductive metric-based approach only about 28 hours of CPU time to generate 100 distinct targeted detection models that all outperform the generic metrics used by our deductive metric-based approach in terms of effectiveness. Note again that the entire evolution process can be parallelized and completely conducted offline on powerful machines or even grids to almost arbitrarily cut down the needed model generation (real) time. These insights again give answers to our main research question *RQ3* in that using generated FrankenMods on QDFGs yields a comparably high detection efficiency.

5.2.4.3. Robustness

As final step of our evaluation we investigated the robustness of our generated FrankenMods with respect to behavior obfuscation, i.e. give answers to our main research question *RQ4*. More precisely we wanted to know how hard it is to confuse FrankenMods by applying targeted obfuscation transformations on the structure of the respective QDFGs and the quantitative information on the edges.

We envision that one way of of evading detection by our approaches would be to tamper with the structure of QDFGs, i.e. forcing the creation of edges by pseudo-randomly interacting with bogus system resources that are not needed to conduct the actual malign activities. Such a behavior would make the QDFG of a malware look substantially different with every new execution and thus would make it hard for the training phase of a classifier to learn a characteristic profile.

Furthermore, considering that QDFG-based approaches also make heavy use of the quantitative information on the edges, a malware with targeted compression actions or with unnecessary repeated interaction with the same system entities could effectively tamper the respective quantitative information on the edges and thus likely influence the detection effectiveness of respective models. We hypothesize that *FrankenMods are less sensitive to behavioral obfuscation that aims at confusing the profile building and training phases than the generic graph metric used by the deductive metric-based approach (H6)*.

To investigate this issue we first need to obtain a set of QDFGs that reflect the application of such behavioral obfuscation attempts. Unfortunately we are not aware of malware that already employs such highly targeted behavioral obfuscation mechanism in the sense of intentionally randomizing issued system call traces with the defined goal of confusing behavioral detection approaches like ours.

Hence, we instead use simulation results where we directly applied transformations on unobfuscated QDFGs to resemble such obfuscation techniques. For this we applied two type of graph transformations on a new data set consisting QDFGs from 929 malware samples and 42 goodware samples that we obtained from extracting known benign and malign email attachments sent to an email server under our control, which we labeled using the www.virustotal.com database. As a side-effect of using a data set different to our baseline evaluation data set for this experiment we were able to validate the effectiveness of the generated FrankenMods on a data set as we would find it in a real-world operational settings.

For the actual obfuscation experiments we then took these QDFGs and stepwise obfuscated them by either randomly creating new edges to simulate issued bogus system calls (*structural obfuscation*) or by tampering the quantitative information on the edges by multiplying the edge weights with a factor randomly picked from the interval $(0, 2]$ to resemble compression or inflation. For every obfuscation round we thus either randomly added a new edge to each QDFG of the previous obfuscation round or randomly modified the weight of a randomly determined edge. This procedure was then repeated 100 times. For investigating the effects of such obfuscation operations on the effectiveness of our approach for each obfuscation round we then performed the same kind of cross-validation experiments that we used for evaluating the effectiveness of the FrankenMods on unobfuscated QDFGs (see Section 5.2.4.1), i.e. training and testing on the obfuscated data.

To confirm H6 we performed this experiment for the deductive metric model, the complete atomic property model, and one of the best-performing generated FrankenMods obtained during the effectiveness evaluation. The right part of Figure 5.15 depicts the results of the structural obfuscation experiments, the left part shows the respective results for quantitative obfuscation, with obfuscation degree referring to the number of tampered or added edges.

Recall that the obfuscation experiment was done on a data set different to our normal evaluation data set. This is why the respective baseline effectiveness for unobfuscated graphs (obfuscation degree 0) differs as well. Moreover, note that with this experiment we tested **and** trained on obfuscated graphs, whereas for the robustness experiments of the deductive metric-based approach we trained on unobfuscated data and only classified obfuscated graphs. With this changed setting we want to anticipate the problem that in real-world settings we probably would not be able to get a baseline set of unobfuscated malware samples for training and thus would need to deal with obfuscated malware samples during training phase.

As we can deduce from Figure 5.15, the FrankenMod is significantly less affected by both, structural and quantitative obfuscation transformations, than the fixed models. This first of all reflects in a generally better effectiveness of the FrankenMod in comparison to the metric from the deductive metric-based approach, as well as a significantly higher prediction stability.

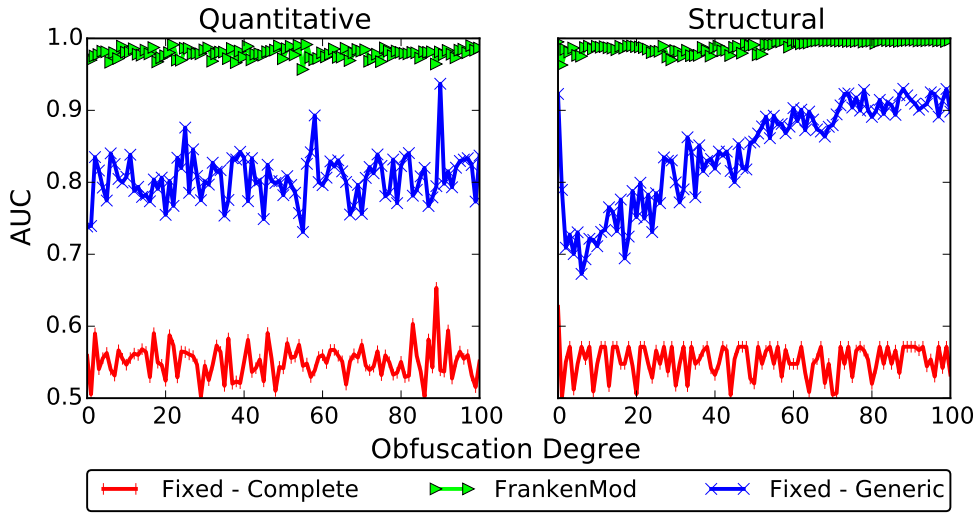


Figure 5.15.: Effectiveness vs. obfuscation degree

While the effectiveness of the FrankenMod remains rather stable, widely independent of the applied obfuscation degree, the effectiveness of the fixed generic model becomes quite unstable when training on behaviorally obfuscated data.

Comparing the average standard deviation of the model's effectiveness, which we consider a useful metric to measure its prediction stability, i.e. ability to deliver stably good effectiveness even on obfuscated malware, we can see that for the quantitative obfuscation experiments the FrankenMod is almost 5 times ($= \frac{\sigma(AUC\_fixed\_generic)}{\sigma(AUC\_generated)} = \frac{0.032}{0.007}$) more stable than the deductive metric-based model; for structural obfuscation the difference in stability between the two type of models becomes even more apparent with the FrankenMod being here almost 9 times more stable than the deductive metric-based metrics ($= \frac{\sigma(AUC\_fixed\_generic)}{\sigma(AUC\_generated)} = \frac{0.070}{0.008}$).

Interestingly, the structural obfuscation experiment conveys a steady increase of detection effectiveness of the fixed generic model with growing obfuscation degree. We explain this effect with the fact that the generic graph centrality metrics leveraged by the deductive metric-based approach by construction perform better for complex graphs than for sparsely connected ones. This caused the obfuscated malware, whose QDFGs we artificially made more complex with the applied structural obfuscation transformations, to look more distinct from goodware.

What we can furthermore see is that the effectiveness of the complete model of atomic properties tremendously suffers from obfuscation and performs only slightly better than random guessing with comparably bad prediction stability.

This shows that the combination of several graph properties with complex metrics not only outperforms the generic metrics from our deductive metric-based approach but also works significantly better than simplistic models that directly use all atomic graph properties for detection.

Finally, from the results of the initial obfuscation round where no obfuscation transformations were applied yet, we can also see that the generated FrankenMod shows similar or even better effectiveness on an entirely different data set as when evaluated on our main evaluation data set. This indicates a certain generalizability and stability of our findings in the sense that FrankenMods stably yield good detection effectiveness on completely disjunctive evaluation data sets. In sum, with these experiments we could confirm hypothesis H6 since we showed that FrankenMods are less affected by behavioral obfuscation than fixed generic ones. We also once more could positively answer our main research question *RQ4* in that we could show that our enhanced QDFG-metric based detection concept is also robust towards behavior obfuscation during training time.

5.2.4.4. Discussion and Threats to Validity

While with our evaluation we were able to confirm all approach-specific research hypothesis, the insights gained, as for all our approaches, must be interpreted in the context of the experiment setting. For instance it is a well-known fact that the effectiveness of machine-learning based malware detection approaches is highly influenced by the quality of underlying training and testing data set [86, 121]. If the evaluation set is too small or does not adequately mirror the real-world malware landscape then the conducted experiments might not well reflect the real-world detection performance of an approach.

Although we use machine learning, we tried to weed out most of these issues by using a diversely composed publicly available data set for the baseline evaluation which ensures reproducibility of our experiments. Furthermore, we also evaluated the effectiveness of the on a data set completely distinct to the baseline evaluation set that we obtained by extracting up-to-date malware and goodware samples from real-world email traffic which to some extent shows that the inductive metric-based approach generalizes to malware samples found in the wild.

Nevertheless there always remains a risk of an approach only working effectively on the evaluated malware families and not on specific families that e.g. employ advanced targeted obfuscation or evasion techniques. While we evaluated the inductive metric-based approach on a selection of such obfuscation threats that we consider realistic in our context, it is possible that we did not anticipate certain attacks or obfuscation techniques that could effectively confuse our FrankenMods.

On the other hand, genetic algorithms are known to have a risk of optimizing towards local optima or to over-fit on the training data.

Although we tried to counter this threat by constantly employing cross-randomly-seeded validation experiments on a big data set and later using a second disjunctive data set for validation this risk can not be ruled out.

Furthermore, one could argue that instead of using a comparably complex genetic programming scheme we could as well have used a leaner evolutionary concept like hill climbing to generate models completely at random until a certain amount of models with a minimum effectiveness has been generated. While this argument seems to be intuitively compelling, we have to recall that in our context we lack the necessary preconditions to use more simple evolutionary concepts since our population members are not interrelated through an obvious topology. In other words, it is not clear how to determine successors for a given initial model, i.e. how to find a way of extending them in a semantically justifiable way. Genetic programming in contrast provides a justifiable methodology to stepwise evolve and combine models.

Generating models in an unguided randomized manner further has the disadvantage that, by construction, such a process has no asymptotic behavior, i.e. we would not know if and when we would reach an (at least) local optimum. As we could show in our evaluation, the employment of a genetic programming scheme in contrast leads to such an asymptotic behavior as it is likely to generate better performing models through evolution of already well-performing ones and thus intuitively at average is likely to produce sets of fixed size of effective models with a defined minimum effectiveness faster than with unguided random generation.

Our selection of operators reflects the ones that are typically used for genetic programming schemes [7]. Although we did not perform a thorough sensitivity analysis we could not identify any significant impact of the applied operators on the overall effectiveness of the generated FrankenMods but only on the convergence behavior of the evolutionary process. We thus believe that our operators are meaningful in that they are at least able to produce FrankenMods that outperform the generic metrics used by the deductive metric-based approach.

5.2.5. Related Work

As for all our approaches, the main difference to related work is that we use quantitative data flow graphs instead of control flow-, system-call-dependency-, or non-quantitative data flow graphs. This, as we now showed several times, has a positive effect on detection accuracy.

Furthermore, the main conceptual difference between to our deductive metric-based approach and other work that uses graph metrics [68, 92] is that our inductive metric-based approach does not rely on a fixed set of generic graph metrics, whose effect could be learned and counteracted by malware, but rather generates arbitrarily large and diverse sets of targeted detection models, i.e. sets of custom metrics, that, as we could show, are far harder to be confused by typical behavioral obfuscation.

Besides work that employs evolutionary concepts for generating malware [106, 105, 25] there also exists some work that uses genetic algorithms for malware detection. Such approaches mainly leverage genetic algorithms for feature selection or in general for optimizing detection classifiers [128, 153, 25, 72], whereas we use *genetic programming* to directly create and evolve targeted detection models.

Probably closest to our inductive metric-based approach is the work of Kim et al. [73] who employ a genetic algorithm to mutate and evolve detection patterns in form of variable dependency graphs to cut-down isomorphism check search space. Their work differs to ours in multiple ways. Firstly, their approach is static whereas ours is dynamic in that they use variable dependency graphs based at binary level whereas we operate on QDFGs obtained from system call traces. Recalling that static approaches have been shown to be more sensitive to obfuscation than dynamic ones we see reasonable arguments to expect our approach to be more robust than theirs. Furthermore, they use a standard genetic algorithm scheme with graph nodes being the to be evolved chromosomes. We instead employ an advanced genetic programming scheme with a defined semantic model to directly create semantically meaningful and justifiable detection model in form of sets of graph metrics that allows credibility checks of issued detection decisions.

The work of Blount et al. [13] is similar to ours in that they also use a evolutionary concept to generate dynamic detection signatures. However, they differ to us in that they mutate and evolve Boolean expressions on resources typically tackled by malware whereas we employ a more complex *genetic programming* scheme to generate detection models in form of QDFG metrics. While it is fairly simple for a malware to avoid profiling and detection by such simple resource name based signatures, e.g. by randomizing the names of written resources, we showed that it is far harder to confuse QDFG metric based detection models.

In sum, our inductive metric-based approach mainly differs from related work in that we are to our best knowledge the first ones to use *genetic programming* for the targeted generation of graph metrics for dynamic malware detection. Furthermore, in contrast to related graph metric based approaches, we do not rely on fixed sets of generic metrics for detection but rather generate, mutate, and deploy arbitrary large and diverse sets of optimized custom graph metrics that we showed to be more robust towards certain types of behavioral obfuscation and at average yield better effectiveness and efficiency than using fixed sets of generic metrics.

5.2.6. Discussion and Conclusion

In sum, with our inductive metric-based detection approach we further pushed the effectiveness, robustness, and efficiency that can be achieved using QDFGs as malware behavior model and thus again positively answered our initial research questions *RQ1-4*. For this we proposed a novel concept for the targeted and automated generation of highly specialized graph metrics through genetic programming to be used for highly accurate, robust, and efficient malware detection. We achieve this by representing metrics as complex functions that combine simple graph properties with basic algebraic operators and mutate them. As fitness function for defining the effectiveness of the generated sets of metrics we evaluate their discrimination capabilities on a diverse labeled data set via repeated cross-validation experiments. The evolutionary process is repeated until a defined termination criterion, e.g. a targeted average performance, is reached.

Our experiments furthermore showed that our inductively generated metrics are up to 6.5% more effective than the generic graph metrics used by our deductive metric-based approach and perform up to 1.8 times better when considering quantitative data flow information than without. Also, our experiments indicate that FrankenMods are up to 9 times more robust than the generic metrics used by our deductive metric-based approach in terms of achieving highly accurate detection results when being trained on obfuscated data. Unfortunately this gain of effectiveness and robustness comes at the cost of a slightly reduced detection efficiency on smaller-sized graphs. For bigger-sized ones however, the inductivly generated FrankenMod metrics turned out to be more efficient and scalable than the generic ones.

6. Assessment and Operationalization

In this chapter we critically assess the devised detection approaches with respect to common malware detection quality attributes, carve out strengths and weaknesses, and relate them to our initial research questions. We also show how to operationalize our approaches for different detection settings and reason about their real-world utility.

6.1. Assessment

In the last chapters we have introduced four different approaches to operationalize our generic quantitative data flow system model for behavior-based malware detection. For each approach, we have already locally discussed the individual contributions, as well as their relation to our initial research questions and main hypothesis. In the following, we will again discuss and compare the different approaches with respect to common malware detection quality goals from a more general perspective, extend the discussion of their strengths and weaknesses, as well as reflecting on their utility for specific detection purposes and deployment settings. As we have evaluated all approaches within the same evaluation environment, using the same baseline evaluation data set, we at least to some extent can compare their detection effectiveness and efficiency, which we consider the most important malware detection quality criteria. As it is not possible to optimize all quality criteria simultaneously, this assessment and comparison will give us important insights to reason about the suitability of our approaches achieving certain detection goals.

6.1.1. Effectiveness

The most important criterion to assess the quality of a malware detection approach is its effectiveness in accurately separating malicious from benign software. By comparing the effectiveness of the different approaches and assessing them from a more macroscopic perspective, we intend to finally answer our main research questions (see Section 1.1.2). Recall that behavior-based malware detection effectiveness can only be measured with respect to a given set of known benign and malicious samples, analyzed within a certain execution environment.

We thus subsequently summarize the detection effectiveness of our approach on a common evaluation data set, analyzed within the same execution environment (see Section 4.1.3.1).

To this end we focus on investigating the following aspects more deeply:

RQ1: How accurately can the different approaches separate malicious from benign samples based on their respective QDFGs?

RQ2: To which extent is the effectiveness of the different approaches improved when we consider quantitative data flow aspects for training and detection?

RQ4: In how far is the detection effectiveness of the different approaches influenced by typical behavior obfuscation techniques?

6.1.1.1. Absolute

For machine learning based approaches, assessing the detection effectiveness boils down to reasoning about false positive (benign software mistakenly classified as malicious) and false negative (malware mislabeled as goodware) classifications. As for classical malware detection we are in a two-class classification setting, the two measures interrelate and usually cannot be optimized at the same time. If we, for instance, want to optimize the false positive rate of an approach, i.e. reduce the number of falsely classified goodware, we typically need to strengthen our notion of something being malicious to avoid corner-cases to get misclassified. Usually this directly impacts the ability of the approach to correctly classify malware that behaves very similar to goodware. This is because by tightening malware behavior patterns or profiles, respectively trained classifiers are more likely to classify corner-cases as benign then as malicious to avoid false positives.

Conversely, optimizing an approach for high detection rates, i.e. avoiding false negatives as much as possible, typically has a direct impact on the resulting false positive rate, as corner-cases are more likely to be classified as malicious than benign. This usually leads to more frequently occurring cases in which goodware that behaves very similar to malware (e.g. web installers that behave very similar to dropper malware) is wrongly classified as malicious.

For the summarizing discussion of the effectiveness of our detection approaches we thus consider both: a generic effectiveness metric to assess the overall effectiveness of an approach, and a more operational metric to reason about the detection effectiveness in a tradeoff-setting that we consider realistic for most real-world detection purposes. To this end, we again look at the *Area Under the Receiver Operator Characteristics Curve* (*AUC*) which we, as discussed in Section 4.2.4.1, consider an ideal measure to express the overall effectiveness of an approach in one number, generalizing from individual false positive to false negative tradeoffs.

While the AUC thus to some extent allows us to compare the general effectiveness of different approaches, due to the abstraction from individual relationships between false positives and false negatives it does not give us a clear and handy idea of the performance in a specific operational context. Being a non-linear metric, a 10% higher AUC, for instance, does not necessarily also imply a 10% higher detection rate for a given target false positive rate. In fact, two classifiers of which one detection-rate-wise performs very poorly for small false positive rates, but yields very low false negative rates when accepting some false positives, might have the same AUC as an approach that already performs well for comparably small false positive rates but does not significantly improve for higher amounts of false positives. Furthermore, AUC for highly effective classifiers is a very sensitive metric with difference between already very well performing classifiers only reflecting in subtle changes to the AUC value. In sum, this means that while AUC is a useful measure to compare absolute effectiveness, AUC in itself does not tell us a lot about effectiveness in concrete operational settings.

To compensate this, we propose the *Best Detection Rate* (*BDR*) metric which represents one point on the ROC curve, i.e. the detection rate that can be achieved when accepting a maximum false positive of 0.5%. While this threshold in essence is arbitrary and in reality would need to be aligned with concrete security demands and risk profiles, discussions with several domain experts confirmed that this is a reasonably realistic assumption to work with.

Table 6.1 summarizes the best AUCs that we were able to achieve when applying the different approaches to our baseline evaluation data set (see Section 4.1.3.1). Note that the deductive pattern-based approach is not included in this table as it does not make use of any form of machine learning and does not feature any adjustable thresholds, which is essential for reasoning about AUC.

	Deductive	**Inductive**
Pattern-based	–	0.988
Metric-based	0.984	0.986

Table 6.1.: Absolute effectiveness (AUC)

What we can deduce from Table 6.1 is that all our machine learning based approaches perform similarly in terms of absolute AUC effectiveness, with the inductive pattern-based approach performing slightly better and the deductive metric-based approach performing a bit worse than the inductive metric-based one.

Interestingly, this exactly inverses when we look at the operational BDR effectiveness, as the deductive metric-based detection approach outperforms the inductive pattern-based one here. Still, we can observe that for both, AUC and BDR, all machine learning based approaches yield very similar performances.

	Deductive	**Inductive**
Pattern-based	$\ll 77\%$	$\sim 96\%$
Metric-based	$\sim 98\%$	$\sim 97\%$

Table 6.2.: Absolute effectiveness (BDR)

We can further see that all our machine learning based approaches substantially outperform our basic deductive pattern-based approach. Finally, this overview also shows that in terms of absolute effectiveness the inductive extensions always outperform the corresponding basic deductive approaches. At this point, we need to recall that for technical reasons our deductive and inductive metric-based approaches use different machine learning classifiers which significantly impacts their detection effectiveness and thus biases direct comparison. However, in Section 5.2.4.1 we have already evaluated the effectiveness of both approaches when using the same SVM classifier and have shown that in this case the inductively generated metrics significantly outperformed the deductively specified generic ones.

Furthermore, we again need to recall that the genetic programming scheme used by our inductive metric-based approaches optimizes the generated metrics towards achieved AUC. By this, the evolutionary process tries to assure a harmonic compromise between false positive and false negative classifications and thus always favors balanced over extreme classification distributions. This is, the evolutionary scheme will always prefer "conservative" metrics that yield consistently low false positive and low false negative rates over more "radical" metrics that, although at average performing wore than the conservative metrics, for some configurations yield higher detection rate or lower false positive rate peaks. The inductive metric-based approach thus is designed to generate arbitrarily large amounts of metrics with a very high average level of absolute effectiveness, but at the same time by construction very unlikely produces outstandingly well-performing metrics – in particular, metrics that significantly outperform the deductively defined ones.

In sum, these results emphasize the utility of our QDFG-based system model for highly accurate malware detection. Given the consistently high level of detection effectiveness that we were able to achieve using conceptually completely different detection approaches, we are finally able to positively answer our initial research question *RQ1* in that QDFG-based system behavior abstraction is indeed highly suitable for malware detection purposes.

6.1.1.2. Quantitative Improvement

In order to assess research question *RQ2* and to confirm our guiding research hypothesis that the consideration of quantitative data flow aspects improves detection effectiveness, we again revisit all devised approaches and recap to which extent quantities improved their effectiveness.

For each approach, we determined the maximum effectiveness improvement by comparing the detection or false positive rate that could be achieved with and without considering *quantitative* data flow information. To avoid comparison bias for the quantitative and non-quantitative experiments, we furthermore either fixed the detection or false positive rate as invariant comparison baseline and only investigated the effects of quantity on the non-fixed metric.

	Deductive	**Inductive**
Pattern-based	$\sim 3.0x$	$\sim 2.6x$
Metric-based	$\sim 2.2x$	$\sim 1.8x$

Table 6.3.: Effectiveness improvement through quantities

Table 6.3 gives an overview of the respectively achieved effectiveness improvement factors. As we can see, the biggest relative effectiveness improvement when using quantitative data flow information was achieved by the deductive pattern-based approach. By incorporating quantitative data flow properties into the detection patterns, we were able to decrease the false positive rate by more than a factor of 3. We account this, in comparison to the other detection approaches, higher relative improvement of detection effectiveness to the significantly lower absolute effectiveness of the deductive pattern-based approach. For all other approaches, the relative detection effectiveness improvement of using quantities revolved around 1.8 to 2.6 times the effectiveness when not using quantities.

Interestingly, the pattern-based approaches in general seemed to benefit slightly more from the incorporation of quantities into the training and detection processes than the metric-based ones. An explanation for this is the at average slightly lower absolute detection effectiveness of the pattern-based approaches. This is, in relative terms it is significantly easier to improve upon mediocre absolute effectiveness than it is to outperform an approach with already very high effectiveness.

Nevertheless, we were able to substantially improve the detection effectiveness of all approaches when incorporating quantitative data flow aspects into the training or classification process. Besides positively answering our main research question *RQ2*, this also strongly confirms our main hypothesis that *"quantitative data flow analysis at average yields better malware detection accuracy than non-quantitative data flow analysis"*.

We consider this the most valuable contribution of this thesis, as we were able to show the utility of quantitative data flow information for multiple conceptually different detection approaches, which suggests a certain generalizability of the insights gained.

6.1.1.3. Robustness

Besides assessing the absolute effectiveness of using our QDFG-based system model and approaches on normal malware, we were also interested in investigating their robustness. Robustness in this context manifests in the ability of respectively devised detection approaches to maintain a consistently high detection effectiveness even when malware employs simple behavior obfuscation. Recent studies [116, 115, 6] revealed that many state of the art behavior-based approaches that leverage raw system call traces for malware detection are sensitive towards behavior obfuscation that leads to random re-ordering or insertion of system calls. A high obfuscation robustness of our detection approaches would thus not only in absolute terms yield better detection effectiveness but also improve upon the state of the art.

In Section 4.1.1 we have already discussed the robustness of our QDFG-based system model towards slight variations in behavior profiles and simple behavior obfuscation operations from a more conceptual perspective. As we recall from Section 3.1, the order of system calls in a recorded behavior trace, i.e. the respectively induced data flows, does not impact the generation of the respective QDFGs. This by construction gives our QDFG-based model and the respectively devised detection approaches an inherent robustness towards trace permutations, i.e. volitional or accidental system call reordering.

In terms of robustness against obfuscation through randomized injection of bogus system calls in the beginning of Chapter 5 we already discussed the sensitivity of our pattern-based approaches towards resulting non-deterministic creation of QDFG edges from a theoretical point of view. More precisely, we have shown that, at least in theory, mined or manually specified detection patterns could be circumvented by certain manipulations to the structural integrity of QDFGs.

However, while our pattern-based approaches thus can to some extent be considered vulnerable to such forms of behavior obfuscation, this is not true for our metric-based approaches. In Section 5.1.3.1, we have empirically shown that our metric-based detection concept is widely robust against simple injection and reordering obfuscation transformations and maintains a consistently high detection effectiveness, even when having to classify behaviorally obfuscated malware. Moreover, we have shown that in terms of obfuscation robustness our metric-based detection concept even outperforms a widely used detection concept of profiling behavior in form of system call n-grams.

In sum, with these insights we can positively answer research question *RQ4* in that we were even able to show that our most advanced detection approaches are to wide extents robust towards the considered behavior obfuscations, but also that they outperform several other state of the art detection concepts.

6.1.2. Efficiency

In order to reason about the operational efficiency of our detection approaches, we distinguish between two different aspects: the training efficiency refers to the computational effort that needs to be spent for generating a detection model, whereas the detection efficiency refers to the time needed for classifying an unknown QDFG, i.e. obtaining a feature vector and matching it against the trained classifier.

Note that for this comparative assessment we were only interested in investigating the individual differences in the computational effort of training and classifying unknown QDFGs. As the computational effort of executing and capturing the behavior of a sample, as well as generating the corresponding QDFG (which at average took less than 10 ms) was the same for all approaches, we deliberately did not consider the effort of producing the raw data for this evaluation.

6.1.2.1. Training

In general, it is not easy to objectively compare the training or model generation time of the different detection approaches. This is partially because it is not entirely clear how to determine the actual time needed to generate a new set of detection patterns or metrics and train the respective machine learning classifier. In particular, for our inductive approaches we do not have any common termination criteria for the mining or the evolution process, which raises the question of which time span to consider as generation time.

To this end, for this evaluation we decided to define the generation or training time as the time needed to produce the locally best-performing detection model, i.e. the ones used as baseline for the effectiveness assessment (see 6.1.1.1). For our inductive metric-based approach, for instance, it took about 1640 minutes to generate a detection model, i.e. a FrankenMod, with a effectiveness that was not exceeded in subsequent evolution steps. For the inductive pattern-based approach this point was reached after slightly over 2 minutes, which is the time it took the mining component to extract a set of highly characteristic graph patterns and train a classier on the respectively obtained matching results.

What we can deduce from the training efficiency overview in Table 6.4 is that our inductive metric-based approach with more than 27 hours by far takes the longest to generate the first locally optimal performing detection model.

	Deductive	Inductive
Pattern-based	–	2.23
Metric-based	7.62	1640

Table 6.4.: Training efficiency [min]

We account this to the high complexity of the fitness function evaluation which repeatedly needs to be conducted for each generated model, each training graph, and for each evolution round.

Interestingly, the training efficiency of the inductive pattern-based approach, i.e. the time required for mining a locally optimal performing set of detection patterns and training the respective classifier, only took about 2 minutes and was thus more than 3 times faster than training the deductive metric-based approach. We attribute this to the aggressive sub-sampling and pattern candidate pruning of the employed MDC graph mining algorithm. This means, the mining component only needs to analyze a small fraction of the entire training set to extract a set of patterns that best describes the training set while the deductive metric-based approach needs to analyze all QDFGs in the training set.

Finally, our deductive pattern-based approach, at least in terms of training efficiency, is the fastest, as it is not machine learning based and thus does not need a dedicated training procedure.

6.1.2.2. Detection

More important than the training efficiency is the efficiency of the detection or classification process, as it has a direct impact on the scalability and potential application areas of an approach. While the training usually only needs to be conducted once and can be offloaded to powerful servers, the detection phase needs to be conducted for each to be classified sample and typically cannot be offloaded.

While all our machine learning based detection approaches operate on reachability graphs and thus do not need to evaluate the detection patterns or metrics on the full-sized QDFGs, our initial deductive pattern-based approach was evaluated on full QDFGs only. As this would induce an unfair bias to the comparison of the detection efficiency, we first had to also evaluate the deductive pattern-based approach on reachability graphs instead of full QDFGs before we could compare the results. The impact on detection efficiency was tremendous, as classifying a reachability QDFG at average could be done more than one order of magnitude faster than classifying a full-sized QDFG and only took a bit more than 1 instead of 23 seconds.

	Deductive	Inductive
Pattern-based	1340	102
Metric-based	39	54

Table 6.5.: Average detection efficiency [ms]

Nevertheless, as we can see in Table 6.5, the deductive pattern-based detection approach was still more than one order of magnitude slower than all other approaches. This can be explained by the fact that we had to develop an own unoptimized implementation of the VF2 sub-graph isomorphism algorithm [48] to introduce quantitative guard properties into the pattern matching process, which likely induced a substantial computational overhead.

A comparison with the results of the inductive pattern-based approach, where the sub-graph isomorphism checks are independent of flow quantities and conducted using a standard optimized VF2 algorithm, fortifies this assumption as here the detection only took about 10% of time of the deductive approach.

Finally, looking at the performance of the metric-based approaches, we can say that metric-based detection was at least 2 times faster than pattern-based detection. Considering the comparably good or even better effectiveness of the metric-based approaches over the pattern-based ones, we can conclude that we indeed found a cheap approximation of graph similarity that is sufficiently precise to yield high detection accuracy, which in sum positively answers research question *RQ3*.

6.1.3. Summary and Discussion

Summarizing the insights gained we can say that there is no significant difference in absolute detection effectiveness (AUC) between our machine learning based approaches that operate on patterns or metrics on QDFGs. In terms of operational effectiveness (BDR) we could identify a slight advantage of metric-based approaches over pattern-based ones. However, we could observe a substantial improvement of effectiveness when moving from pure rule-based detection, i.e. deductive pattern-based detection, to soft-computing supported detection schemes, as e.g. employed by our inductive pattern-based and our metric-based approaches. This is an interesting insight and to some extent suggests that malware behavior might be too diverse or the differences to closely related benign activities too subtle to be accurately captured with only a small set of generic detection heuristics.

Furthermore, our pattern-based approaches seemed to benefit slightly more from using quantitative data flow information than our metric-based ones. This can partially be explained by the slightly higher operational effectiveness of metric-based approaches, which is relatively harder to improve upon.

Although we only empirically evaluated the actual obfuscation robustness of our metric-based approaches, following the arguments given at the beginning of Section 4.2, we see good reasons to assume that metric-based approaches are more robust towards simple behavior obfuscation transformations than pattern-based ones.

In terms of detection efficiency, we could see a clear advantage of the metric-based approaches over the pattern-based ones. This is not surprising as the baseline problems, i.e. metric computation vs. sub-graph isomorphism, of the detection concepts are in different complexity classes. This suggests that our metric-based approaches scale better with bigger graph sizes than the pattern-based ones.

Furthermore, one important difference between our metric-based and the pattern-based approaches is that they lack a clear semantic interpretation. While at least our deductively defined detection patterns come with a defined semantic justification, the used graph metrics in combination lack such obvious and human-understandable semantics. This has a substantial and non-negligible impact on the understandability of classification decisions issued by the approaches to human analysts. While the pattern-based approaches at least to some extent allow a human analyst to follow the line of reasoning that leads to a classification decision and thus verify and learn from them, understanding and manually verifying the machine learning supported classification decisions of metric-based approaches is tedious. This poses a problem to post-detection root-cause and impact analysis.

To put the effectiveness and efficiency of our detection approaches into a broader context, we finally compare our results with the respective ones of closely related approaches, proposed in literature. We focus this comparison on two approaches from literature that we consider to be the closest to our work and that are within the most cited ones in the field.

In particular we compare our approaches with the taint-tracking based Panorama system of Yin et al. [151] as they, like us, leverage data flow analysis for malware detection. However, in contrast to us they use precise full system emulation based taint-tracking to capture data flows, while we approximate potential data flows on the grounds of system call interpretation and correlation.

Furthermore, we compare our work with that of Kolbitsch et al. [80] who were within the first to use system call based data flow approximation for malware detection. The main difference to their work is that we consider data flows from a quantitative perspective, whereas they only take an possibilistic view.

In general it is hard to objectively compare different malware detection approaches, especially when they were evaluated on entirely different malware and goodware data sets and execution environments. The following comparative discussion is thus more meant to give the reader a qualitative means to assess global effectiveness and efficiency or our work in the light of closely related approaches and not to conduct a exact quantitative comparison and ranking.

The Panorama system of Yin et al. [151] was evaluated on a data set consisting of only 42 malware and goodware samples. On this data set the Panorama system was able to correctly identify all malware samples, i.e. yielded a detection rate of 100%. At the same time Panorama incorrectly labeled 3 out of 56 goodware samples as malware and thus yielded a false positive classification rate of more than 5%. Although on this evaluation data set the Panorama system thus yielded a slightly better detection rate than our best approaches (100% vs. 98%), we thus significantly outperform Panorama in terms of false positive rates (5.4% vs. 0.5%).

Note again that our approaches were evaluated on a data set almost 100 times as big as the one used to evaluate Panorama. This needs to be considered when interpreting the relative differences in detection effectiveness. Moreover, Panorama induced an average computational overhead of more than 800% and hence was substantially slower than our approaches ($\approx 30\%$, see Section B.1.2.2).

In comparing to the closely related non-quantitative data flow approximation approach of Kolbitsch et al. [80], which was evaluated on 300 malware and a not further specified number of goodware samples, we can furthermore say that we perform significantly better in correctly classifying malware samples, i.e. detection rate (64% vs. 98%). This becomes even more apparent if malware samples from families that were not used for training are to be classified. Here we at average correctly classified 74% of the unknown malware samples, whereas they were only able to correctly label about 23% of the samples. In terms of false positives the numbers presented in [80] suggest that they perform slightly better than us and did not yield any false positives on their data set, whereas our approaches at average misclassified 0.5% of the benign samples.

Again, these numbers need to be interpreted with care, considering the different data sets and unclear size and composition of the goodware set that was used to evaluate Kolbitsch's approach. In terms of caused computational overhead, Kolbitsch et al. also perform better than us and at average only induce an overhead of about 18%, whereas we induce an average overhead of 30%.

As none of the related approaches was thoroughly evaluated in terms of obfuscation robustness it is hard to make any sound statements on their resilience and to compare them with our approaches in this respect. However, considering our obfuscation robustness results (see Section 5.1.3.2 and Section 5.2.4.3) we see good reasons to assume that our quantitative approaches are more robust than the related non-quantitative ones.

Considering these global comparison results we can conclude that the efficiency and effectiveness our approaches is at least competitive to the state of the art. In particular we can say that it seems that our quantitative concept yields a good compromise between precise taint-tracking based and approximate system call data flow inference approaches in that we seem to be slightly less precise than taint-based approaches, but significantly more efficient and portable.

At the same time the comparison results suggest that we are slightly less efficient, but more effective than non-quantitative data flow approximation based approaches. As stressed before, this of course depends on a comparability of evaluation data sets and environments which is not necessarily given.

In the next section we will now discuss how the differences between our approaches affect their operationalization and utility for concrete detection settings.

6.2. Operationalization

For malware detection, just like in most IT security settings, a generic 100% effective solution is neither possible nor really needed [122]. Especially when considering inevitable interrelations between different quality attributes such as effectiveness, efficiency, and robustness, it becomes clear that the optimal detection solution depends on the concrete deployment settings, i.e. detection goals and constraints. We differentiate between two orthogonal detection settings: *offline*, where training and detection are both done on data obtained from isolated execution of samples in controlled sandbox environments, and *online*, where training might still be conducted on the basis of sandbox data, but detection is done at runtime on incrementally built graphs of unconstrained size and complexity. While this thesis mainly focuses on showing the general utility of quantitative data flow analysis for malware detection in controlled environments, we subsequently will argue why we consider our detection approaches to also generalize to online detection settings.

6.2.1. Offline Detection

An example of a typical offline detection setting are email attachment anti-virus scanners. Such scanners usually intercept and scan emails and their attachments on the email server by either statically analyzing them on the fly or executing suspicious attachments in malware sandboxes to dynamically assess their threat potential. Usually, the classification of email attachments can almost arbitrarily be parallelized and offloaded to powerful grids, as there is no interdependency between different (attachment) samples that needs to be considered for classification. Detection efficiency thus only plays a subordinate role in such settings as long as intercepted emails are not delayed unreasonably long.

Emails these days are one of the primary means [102] of distributing commodity malware with the main goal of infecting as many victims as possible. An email attachment scanner thus is likely to be exposed to a large and diverse number of the most current malware samples, which raises the need for robust detection mechanisms.

Given these comparably low demands on detection efficiency, high effectiveness and robustness requirements, in such a scenario one would typically strive for deploying a detection approach with the highest possible detection rate and robustness, irrespective of the induced computational costs.

Applied to our approaches, this would suggest that our inductive metric-based approach fits offline detection settings best as it yields the highest detection effectiveness and robustness. Moreover, by combining several detection (Franken)models with an additional arbitration scheme, it would likely be possible to further increase detection effectiveness and robustness at the cost of classification efficiency.

Finally, from a conceptual perspective, it would also be possible to combine all approaches, i.e. metric- and pattern-based ones, through a meta-classification scheme. Such a hybrid combined approach could, for instance, be realized by matching all detection approaches against a set of labeled training samples, turn their individual classification results, i.e. their numeric classification confidence, into feature vectors, on which we then could train a combined supervised machine learning classifier. This classifier then essentially would learn complex dependencies between the individual class predictions of the single detection approaches and the actual class of a training sample and thus merge the predictions of all detection approaches into a combined classification decision. The assumption here would be that a joined classifier, in terms of effectiveness, performs better than the individual ones, at the cost of decreased detection efficiency. While we have already started to investigate the utility of such hybrid detection models with encouraging results, we leave their thorough analysis to future work.

6.2.2. Online Detection

In contrast, in online detection settings, computational resources are comparably sparse and the demands on system reactiveness usually higher. Unlike in offline detection settings, where suspicious samples can be executed and monitored in an isolated and controlled environment for a fixed period of time, this normally is not possible in online detection settings. In typical online detection settings, e.g. for desktop malware detection, we have to continuously monitor system behavior in uncontrolled or semi-controlled environments to be able to timely raise alarms or conduct appropriate countermeasures. Performing behavior-based detection in such settings thus underlies a series of performance constraints and must not significantly compromise user experience. This is, one must find an acceptable trade-off between detection effectiveness, efficiency, and robustness.

From an efficiency perspective this therefore would suggest using one of our metric-based approaches as they offer the best efficiency-effectiveness tradeoff. However, we need to recall a series of constraints and limitations of the metric-based approaches that affect their direct utility for online detection.

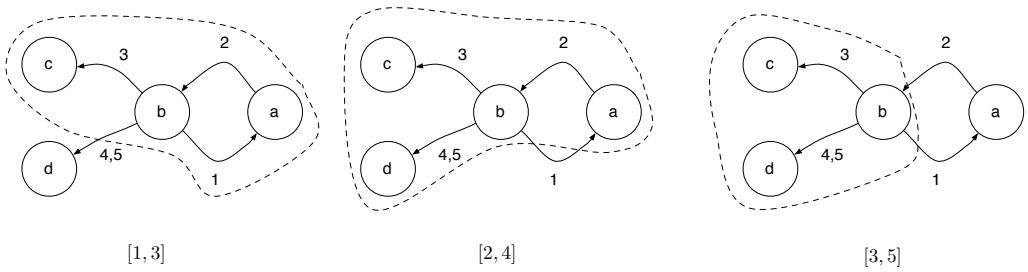


Figure 6.1.: Graph splitting strategy

First of all, our metric-based approaches are trained on features derived from behavior of malware and goodware executed in isolation in controlled sandbox environments, observed for a fixed period of time. Second, we need to recall that the global graph metrics used by the metric-based approaches are sensitive towards changes in baseline graph size and complexity. However, in a continuous monitoring setting in heterogeneous desktop environments, neither the observation time nor the number and types of interacting processes is typically fixed.

Metric-based detection models that were trained on fixed-length behavior of systems, with in essence only one main active process, thus would be likely to perform rather bad on completely open systems with multiple interacting process and non-fixed profiling time spans. This poses certain problems to the direct instantiation of our metric-based detection concept to online detection settings, due to incongruent graph baselines between training and detection time. One way of, at least partially, coping with the problem of incompatible graph baselines would be to artificially establish a consistent baseline by means of graph slicing.

In online detection settings, QDFGs that model observed system behavior grow continuously, in contrast to the fixed-sized training QDFGs. To nevertheless ensure a certain baseline compatibility, we could continuously slice the growing runtime QDFGs into sub-graphs that, just like the training graphs, capture the system's behavior for a fixed time interval. Through a simple sliding window splitting scheme we would then obtain streams of sub-graphs of the runtime QDFGs that share the same observation time baseline as the training graphs and thus could be matched against metric profiles obtained from fixed-sized training traces.

An example of such a graph splitting strategy is depicted in Figure 6.1. For simplicity we use a small time-frame size of three events and a sample rate of one newly generated sub-graph per time-stamp. Each region corresponds to one generated sub-graph of time length three, starting with the time interval $[1, 3]$, then $[2, 4]$, followed by $[3, 5]$.

The main emphasis of this thesis is on showing the general utility of quantitative data flow analysis and thus mainly focuses on controlled sandbox deployment settings. Nevertheless, we see good reasons to assume that with such a graph splitting strategy we could also operationalize our metric-based concept for online detection settings. However, although we had first positive experiences with a simple splitting concept for metric-based online detection (see Appendix B), we leave a thorough analysis to future work.

While our metric-based approach thus, due to dependencies on a common QDFG baseline, can not be directly operationalized for online detection, our pattern-based concepts are considerably easier to instantiate for online detection purposes. The main reason for this is that, unlike our fuzzy metric-based similarity approximation, our pattern-based approaches work with strict sub-graph isomorphism, which by construction is agnostic to the complexity and time baseline of to be classified QDFGs. Hence, although we mine patterns from behavior that was obtained from monitoring malware and goodware executed in a controlled sandbox environment for a fixed period of time, obtained and manually defined detection patterns in principle can also be detected in graphs with a completely different time baseline. As long as a pattern also occurs in an arbitrarily large and complex runtime graph, it is possible to use for classification. However, when using machine learning in addition to pure pattern matching, as our inductive pattern-based approach does, the additional noise in runtime graphs with different time scope than the training graphs can negatively affect classification accuracy.

In addition to that pattern-based detection, as discussed in Section 6.1.2, badly scales with increasing graph complexity. A runtime QDFG that models the behavior of a highly interactive system, despite employed graph simplification and aggregation means, can have several thousand edges which, as we saw in Section 4.1.3.3, can lead to classification times up to a couple of minutes. As this is prohibitive for most online detection purposes, for operationalizing our pattern-based concept we also need to employ some additional detection complexity reduction means to remain sufficiently efficient for online detection. This could either be done by using a similar graph slicing scheme, as proposed for metric-based online detection, or e.g. by marking already evaluated parts of the runtime graph to avoid redundant matching.

Although we have already prototypically implemented a basic graph splitting scheme and successfully shown its utility for online pattern-based detection (see Appendix A), the development of more elaborate online detection concepts is outside the scope of this thesis and thus left for future work.

Summarizing the discussion on operationalizing our detection approaches, we see their main utility for offline sandbox-based detection. While we have sketched the possibility of also operationalizing them for online detection, more research effort will be needed to address expected effectiveness and efficiency issues.

7. Conclusion

As a consequence of the increasing growth and sophistication of the malware underground economy, malware detection remains a very relevant topic and an active area of research, with many still widely unsolved challenges [112, 22, 154, 59].

In this thesis we contribute towards solving some of them by showing how *quantitative data flow information* gathered at system call level can be leveraged (RQ1) for highly accurate (RQ2), robust (RQ4), and efficient (RQ3) malware detection.

To this end, we have introduced a novel model to represent system behavior (see Chapter 3) as so-called quantitative data flow graphs (QDFGs). QDFGs are generated by interpreting observed system calls according to their (quantitative) data flow semantics. This is, we analyze how much data flow takes places between different entities of a system, such as files, processes, sockets, or registry entries, has flown as consequence of an issued system call. This information is inferred from relevant system call arguments like e.g. the size of read or write buffers. Correspondingly, the nodes of a QDFG represent system entities that at least once were involved in a data flow, and edges model aggregated data flows between those entities.

This abstraction from low-level details of the underlying system call traces, such as order or amount of issued calls, yields lean and robust behavior models that, as we have later empirically validated, are widely robust towards noise and intentional behavior obfuscation. Furthermore, the aggregation of semantically related data flows in comparison to maintaining the full trace information reduced the amount of data needed to capture the behavior of a system for a given time span.

In order to show the utility of this model for behavior-based malware detection, we then presented four different QDFG-based detection approaches based on two different classification concepts: *pattern-based* detection, where recurring malicious behavior is codified into patterns that are used to classify unknown samples (see Chapter 4), and *metric-based* detection where we profile QDFGs with sets of graph metrics to establish a flexible notion of behavior similarity (see Chapter 5). Both basic concepts were first instantiated in a deductive way, using manually defined detection patterns and generic graph metrics, and then inductively extended with automatically extracted detection patterns and generated graph metrics.

Finally, we presented a comparative assessment of all devised approaches and discussed their suitability for typical malware detection settings.

7.1. Gained Insights

In sum, with these QDFG-based detection approaches and their evaluation on a big and diverse data set in a representative sandbox environment we were able to answer our initial research questions by showing that:

RQ1: Our QDFG-based system model can be utilized for highly accurate malware detection with an up to 98% detection rate and less than 0.5% false positives.

RQ2: The consideration of quantitative data flow properties for training and detection improves effectiveness by up to 300%.

RQ3: QDFG-based malware detection can be done very efficiently; our metric-based detection approach can classify an unknown QDFG in less than 40ms.

RQ4: QDFG-based detection approaches are widely robust against simple behavior obfuscation transformations and in particular can outperform state of the art detection approaches that rely on raw system call traces.

These results furthermore confirm our main hypothesis in that we, at least for our evaluation setting, have shown that *quantitative data flow analysis **indeed** yields better malware detection accuracy than non-quantitative analysis*.

We need to recall though that a perfect general-purpose malware detection approach cannot exist [40, 34]. In behavior-based detection terms this is because a perfect detection model would essentially need to be functionally equivalent to all possible malware programs. Semantic equivalence is a non-trivial property and, according to Rice's theorem, in general computationally undecidable.

Furthermore, there is no commonly agreed upon consensus on what malware actually is. Taken together that means that behavior-based malware detection can only be done best-effort and example-based. That means, we can only devise detection models that locally optimally separate known malware samples and benign ones. By making them as flexible as possible without compromising accuracy on the training set, we then need to assume that respectively generated detection models to some extent also generalize to unknown malware.

While our cross-validation experiments indicate a certain generalizability of our approaches, we, like all other malware detection approaches, cannot claim that our evaluation results necessarily reflect real-world effectiveness. Furthermore, not only after recently revealed cases of advanced targeted malware like Stuxnet, Duqu, or Regin using highly sophisticated anti-analysis and anti-detection techniques [83, 133, 10, 137], we know that, given enough time and resources, it is always possible for experienced malware developers to hamper detection, especially when the functionality of used detection mechanisms is well-known.

Although we put some effort into making our behavior models and detection approaches robust against typical behavior obfuscation transformations and also anticipated some generic quantitative attacks, we are aware that, given enough knowledge about their functionality, it might also be possible to confuse our classifiers. Still, our evaluations indicate that in comparison to closely related work we significantly raised the bar for malware to evade detection.

On a more abstract level, we have learned that there is an inevitable trade-off between detection quality characteristics that can only be resolved with respect to concrete operational security goals and constraints. This is, there likely is no one-size-fits-all solution that optimally suits all detection purposes. Also, we have learned that static rule-based detection models are probably not suited to coping with the heterogeneity and behavioral diversity of modern malware. In contrast, we strictly believe that soft-computing based approaches that can be re-trained on new malware families once starting to become ineffective, or even dynamically adapt to new threat landscapes, are better suited to deal with ever more sophisticated malware.

7.2. Future Work

Although with the presented detection approaches we were able to show the general utility of quantitative data flow analysis for malware detection and thus met the set research goals of this thesis, there is still room for improvement and follow-up work. We envision possible future work in terms of further *generalizing* our detection concept, improving effectiveness through *combined* or *hybrid detection* strategies, operationalizing the proposed approaches for *online detection*, and improving the detection *infrastructure and process*, which we in the following will discuss in more detail.

Generalization As MS Windows operating systems are still prevalent in the business and consumer desktop market segment [130], they are also the primary targets of commodity malware [112]. As the primary aim of this thesis was to show the general utility of quantitative data flow analysis for malware detection, it was a logical choice to also focus our approaches and evaluations on Windows malware.

Nevertheless, in the past years we could observe a significant rise of mobile malware, i.e. malware that runs on and targets Android operating systems. As our QDFG-based system model is not bound to one specific operating system and in principle can also be instantiated for systems other than Windows, we see the instantiation of our generic model for Android operating systems and the according adaptation of our detection approaches as logical next steps.

Besides the non-trivial effort of modeling Android operating system specificities, profiling and detecting the behavior of Android malware with QDFGs poses some new challenges as Android malware is significantly more reactive than Windows malware and thus demands more sophisticated stimulation strategies to trigger meaningful behavior. Moreover, Android malware often comes as re-packaged goodware and thus blends into seemingly benign behavior, which makes it significantly harder to separate benign from malign behavior.

Although we have with some success adapted our deductive metric-based approach for Android [27], the effectiveness was significantly below that of the Windows-based versions. To achieve similar effectiveness, we would need to develop more elaborate graph splitting and training procedures to reduce training data noise.

Combined and Hybrid Detection Approaches Besides generalizing our system model and detection approaches to other operating systems, we in Section 6.2.1 already sketched the idea of combining our approaches to improve classification effectiveness and robustness. We see good reasons to assume that the combination of several detection approaches, i.e. our deductive metric and deductive pattern-based approaches, might yield improved detection effectiveness.

This, for instance, could be done with a simple arbiter scheme on the different approaches' classification predictions and then, depending on the detection goals, optimizing detection or false positive rates by either disjunctive or conjunctive combinations of the results. Instead, we could also combine different detection approaches by means of a meta-learning scheme that is trained on the individual predictions of different approaches to infer joint classification models. First tests on combining different metric-based detection approaches already indicated that the combination of different approaches or detection models indeed can improve effectiveness at the cost of decreased efficiency. Finally, we would expect hybrid combination of pattern- and metric-based approaches to even further improve detection effectiveness and robustness, under the assumption that graph metrics and patterns can be defined in a non-overlapping way.

Online Detection As already briefly discussed in Section 6.2.2, the detection approaches proposed in this thesis were mainly developed and evaluated for typical offline sandbox detection settings. While this was sufficient for showing the principle utility of quantitative data flow analysis for malware detection, the utility of the presented approaches for online detection purposes was not evaluated in this thesis and likely needs some more adaptation effort. Although, in Appendix A and Appendix B we have prototypically shown that our basic concepts, at least for specific detection use cases, can be extended for online detection, we leave a comprehensive extension to future work.

Furthermore, it would be interesting to investigate the possibility of at least partially shifting the training phase to runtime rather than design time. This is, by continuously re-training the devised detection approaches on previously classified data with high classification consensus among different detection approaches, we would enable a certain self-adaptivity of the detection mechanisms to changes in the threat landscape. This, for instance, could be done by incorporating some sort of feedback loop into hybrid approaches that trigger the re-training of all individual approaches once the classification consensus among them falls below a defined minimum confidence threshold. Furthermore, our inductive metric-based model generation procedure could probably be adapted to continuously creating new detection models at runtime and evaluating their effectiveness on live samples while using the other detection approaches as ground truth oracles.

Infrastructure and Process Finally, as the emphasis of this thesis was on the conceptual contribution of showing the general utility of quantitative data flow analysis and less on the technical infrastructure to retrieve the raw data, we see room for follow-up work in this matter. Malware these days is often environment sensitive and adapts its behavior when it realizes that it is being executed and analyzed in artificial sandbox settings. Although we have already introduced basic countermeasures to tackle this issue, e.g. by using simplistic user behavior simulation scripts, replacing the standard sandbox monitor with our custom developed and more stealthy one, and preparing the virtual execution environment to obscure virtualization indicators that are well-known to be analyzed by malware, we expect more advanced stimulation strategies and more elaborate execution environments to further improve detection effectiveness.

Moreover, we are convinced that a comprehensive malware mitigation strategy should always encompass detection **and** analysis. Although automated detection mechanisms can filter potentially harmful samples, there is always a chance that some of them slip through. Hence, infections can never be ruled out with absolute certainty. To still be able to react to potentially unwanted consequences, ex-post analysis can help to understand infection entry points and estimate induced damages to contain current and prevent future infections. Our graph-based behavior model is an ideal basis for extending pure detection with further-reaching analysis and visualization means, as reasoning about root causes and impact of infections essentially boils down to graph reachability analysis. This is, our detection approaches can flag suspicious process nodes from which we can perform backward or forward reachability analysis to reveal potential infection sources, i.e. malicious network locations or files that might have caused the infection, or estimate potentially compromised resources, e.g. dropped or infected files.

Although we have shown that our detection concepts can in principle be extended by additional means to allow visual root-cause and infection analysis by human experts (see Appendix A), and discussed how to link our basic metric-based detection concept to risk assessment and incident response strategies (see Appendix B), devising a holistic process to detect, analyze, and mitigate potentially malicious behavior is out of the scope of this thesis and thus left to future work. Also, a thorough analysis and comparison of the semantic dimension of deductively defined and inductively mined detection patterns is left to future work.

Bibliography

[1] Uwe Aickelin, Julie Greensmith, and Jamie Twycross. Immune system approaches to intrusion detection–a review. In *Artificial Immune Systems*, pages 316–329. Springer, 2004.

[2] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205, 2000.

[3] Domagoj Babić, Daniel Reynaud, and Dawn Song. Malware analysis with tree automata inference. In *Computer Aided Verification*, pages 116–131. Springer, 2011.

[4] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Recent advances in intrusion detection*, pages 178–197. Springer, 2007.

[5] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[6] Sebastian Banescu, Tobias Wüchner, Alei Salem, Marius Guggenmos, Martín Ochoa, and Alexander Pretschner. An empirical evaluation framework for malware behavior obfuscation. In *International Conference on Malicious and Unwanted Software (MALCON)*, 2015.

[7] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

[8] Ulrich Bayer. *TTAnalyze: A tool for analyzing malware*.

[9] Ulrich Bayer. *Large-Scale Dynamic Malware Analysis*. PhD thesis, Technische Universität Wien, 2009.

[10] Boldizsar Bencsath, Gabor Pek, Levente Buttyan, and Mark Felegyhazi. Duqu: A stuxnet-like malware found in the wild. www.crysys.hu/publications/files/bencsathPBF11duqu.pdf, December 2015.

[11] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases*, pages 387–402. Springer, 2013.

[12] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.

[13] Jonathan J Blount, Daniel R Tauritz, Samuel Mulder, et al. Adaptive rule-based malware detection employing learning classifier systems: A proof of concept. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 110–115. IEEE, 2011.

[14] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow graphs as malware signatures. In *International workshop on the Theory of Computer Viruses*, 2007.

[15] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[16] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[17] Matthew Braverman. Win32/blaster: a case study from microsoft's perspective. In *Proceedings of the Virus Bulletin International Conference*, 2005.

[18] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[19] Ismael Briones and Aitor Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Virus bulletin conference*, pages 1–12. Citeseer, 2008.

[20] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[21] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 129–143. Springer, 2006.

[22] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: the commoditization of malware distribution. In *Proceedings of the 20th USENIX conference on Security*, pages 13–13. USENIX Association, 2011.

[23] Juan Caballero, Min Gyung Kang, Shobha Venkataraman, Dawn Song, Pongsin Poosankam, and Avrim Blum. Fig: Automatic fingerprint generation. 2007.

[24] 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 *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 122–132. ACM, 2012.

[25] Andrea Cani, Marco Gaudesi, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. Towards automated malware creation: code generation and code integration. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 157–160. ACM, 2014.

[26] Stephen Cass. Anatomy of malice [computer viruses]. *Spectrum, IEEE*, 38(11):56–60, 2001.

[27] John Henry Castellanos, Tobias Wüchner, Martín Ochoa, and Sandra Rueda. Q-floid: Android malware detection with quantitative data flow graphs. In *SG-CRC*, 2016.

[28] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pages 61–70. Australian Computer Society, Inc., 2010.

[29] Silvio Cesare and Yang Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 181–189. IEEE, 2011.

[30] Duen Horng Chau, Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. In *SIAM International Conference on Data Mining*, 2011.

[31] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *J. of Artificial Intelligence Research*, 16(1):321–357, 2002.

[32] Chen Chen, Cindy X Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. Mining graph patterns efficiently via randomized summaries. *Proceedings of the VLDB Endowment*, 2(1):742–753, 2009.

[33] Thomas M. Chen and Jean marc Robert. The evolution of viruses and worms. In *Statistical Methods in Computer*, 2004.

[34] David M Chess and Steve R White. An undetectable computer virus. In *Proceedings of Virus Bulletin Conference*, volume 5, 2000.

[35] Mihai Christodorescu and Somesh Jha. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[36] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[37] Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, Randal E Bryant, et al. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[38] Fred Cohen. Computer viruses: theory and experiments. 1984.

[39] Fred Cohen. *Computer Viruses*. PhD thesis, University of southern california, 1986.

[40] Fred Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):297–298, 1989.

[41] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms Applications*, 2007.

[42] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1994.

[43] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Transactions on Pattern Analysis and Machine Intelligence*, pages 1367–1372, 2004.

[44] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[45] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312. IEEE, 2011.

[46] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, page 6, 2012.

[47] Karim O Elish, D Yao, and Barbara G Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Workshop on Mobile Security Technologies*, 2012.

[48] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA*, volume 95, pages 632–640. World Scientific, 1995.

[49] Mojtaba Eskandari and Sattar Hashemi. Metamorphic malware detection using control flow graph mining. *International Journal of Computer Science and Network Security*, 11(12):1–6, 2011.

[50] Parvez Faruki, Vijay Laxmi, Manoj Singh Gaur, and P Vinod. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks*, pages 130–137. ACM, 2012.

[51] Stephanie Forrest, Steven Hofmeyr, Aniln Somayaji, Thomas Longstaff, et al. A sense of self for unix processes. *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128, 1996.

[52] Thomas Fox-Brewster. Netflix is dumping anti-virus, presages death of an industry. http://www.forbes.com/sites/thomasbrewster/2015/08/26/netflix-and-death-of-anti-virus/, December 2015.

[53] Matt Fredrikson, Mihai Christodorescu, Jonathon Giffin, and Somesh Jhas. A declarative framework for intrusion analysis. In *Cyber Situational Awareness*, pages 179–200. 2010.

[54] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 45–60. IEEE, 2010.

[55] Matthew Fredrikson, Mihai Christodorescu, and Somesh Jha. Dynamic behavior matching: A complexity analysis and new approximation algorithms. *Automated Deduction-CADE-23*, pages 252–267, 2011.

[56] Gehad M Galal, Diane J Cook, and Lawrence B Holder. Exploiting parallelism in a structural scientific discovery system to improve scalability. *Journal of the Association for Information Science and Technology*, 50(1):65, 1999.

[57] Michael Garey and David Johnson. Computers and intractability: a guide to the theory of np- completeness. *San Francisco, LA: Freeman*, 1979.

[58] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[59] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *Computer Security–ESORICS 2009*, pages 1–18. Springer, 2009.

[60] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[61] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[62] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, 48, 2007.

[63] Lastline Inc. Lastline analyst. www.lastline.com/documents/Lastline-Analyst-Datasheet.pdf, January 2016.

[64] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*. 2000.

[65] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4(3):251–266, 2008.

[66] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: picking command and control connections from bot traffic. In *Proceedings of the 20th USENIX conference on Security*, pages 29–29. USENIX Association, 2011.

[67] Dae-il Jang, Minsoo Kim, Hyun-chul Jung, and Bong-Nam Noh. Analysis of http2p botnet: case study waledac. In *Communications (MICC), 2009 IEEE 9th Malaysia International Conference on*, pages 409–412. IEEE, 2009.

[68] Jae-wook Jang, Jiyoung Woo, Jaesung Yun, and Huy Kang Kim. Malnetminer: malware classification based on social network analysis of call graph. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 731–734. International World Wide Web Conferences Steering Committee, 2014.

[69] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.

[70] Fatemeh Karbalaie, Ashkan Sami, and Mansour Ahmadi. Semantic malware detection by deploying graph mining. *International Journal of Computer Science Issues*, 9(1):1694–0814, 2012.

[71] Nikhil S Ketkar, Lawrence B Holder, and Diane J Cook. Subdue: Compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, 2005.

[72] Jinhyun Kim and Byung-Ro Moon. Disguised malware script detection system using hybrid genetic algorithm. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 182–187. ACM, 2013.

[73] Keehyung Kim and Byung-Ro Moon. Malware detection based on dependency graph using hybrid genetic algorithm. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1211–1218. ACM, 2010.

[74] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[75] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Detection of intrusions and malware, and vulnerability assessment*, pages 174–187. Springer, 2005.

[76] Samuel T King and Peter M Chen. Backtracking intrusions. In *ACM SIGOPS Operating Systems Review*, pages 223–236, 2003.

[77] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.

[78] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[79] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *USENIX Security Symposium*, volume 6, 2006.

[80] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.

[81] Marc Kührer, Christian Rossow, and Thorsten Holz. Paint it black: Evaluating the effectiveness of malware blacklists. In *Research in Attacks, Intrusions and Defenses*, pages 1–21. Springer, 2014.

[82] Michihiro Kuramochi and George Karypis. Discovering frequent geometric subgraphs. *Information Systems*, 32(8):1101–1120, 2007.

[83] David Kushner. The real story of stuxnet. spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet, December 2015.

[84] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412. ACM, 2010.

[85] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1970–1977. ACM, 2010.

[86] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. On challenges in evaluating malware clustering. In *Recent Advances in Intrusion Detection*, pages 238–255. Springer, 2010.

[87] Wei-Jen Li, Ke Wang, Salvatore J Stolfo, and Benjamin Herzog. Fileprints: Identifying file types by n-gram analysis. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 64–71. IEEE, 2005.

[88] Alexander Long, Joshua Saxe, and Robert Gove. Detecting malware samples with similar image sets. In *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, pages 88–95. ACM, 2014.

[89] Enrico Lovat, Johan Oudinet, and Alexander Pretschner. On quantitative dynamic data flow tracking. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 211–222. ACM, 2014.

[90] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, (2):40–45, 2007.

[91] Hugo Daniel Macedo and Tayssir Touili. Mining malware specifications through static reachability analysis. In *Computer Security–ESORICS 2013*, pages 517–535. Springer, 2013.

[92] Weixuan Mao, Zhongmin Cai, Xiaohong Guan, and Don Towsley. Centrality metrics of importance in access behaviors and malware detections. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 376–385. ACM, 2014.

[93] Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Classifying android malware through subgraph mining. In *Data Privacy Management and Autonomous Spontaneous Security*. 2014.

[94] Takashi Matsuda, Hiroshi Motoda, Tetsuya Yoshida, and Takashi Washio. Mining patterns from structured data by beam-wise graph-based induction. In *Discovery Science*, 2002.

[95] McAfee. The economic impact of cybercrime and cyber espionage. http://www.mcafee.com/mx/resources/reports/rp-economic-impact-cybercrime.pdf, 2013.

[96] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 112–131. Springer, 2014.

[97] Aziz Mohaisen, Omar Alrawi, Matt Larson, and Danny McPherson. Towards a methodical evaluation of antivirus scans and labels. In *Information Security Applications*, pages 231–241. Springer, 2014.

[98] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[99] Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. Unknown malcode detection using opcode representation. In *Intelligence and Security Informatics*, pages 204–215. Springer, 2008.

[100] Antonio Nappa, M Zubair Rafique, and Juan Caballero. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer, 2013.

[101] Antonio Nappa, Zhaoyan Xu, M Zubair Rafique, Juan Caballero, and Guofei Gu. Cyberprobe: Towards internet-scale active detection of malicious servers. In *Network and Distributed System Security Symposium*, 2014.

[102] Mark EJ Newman, Stephanie Forrest, and Justin Balthrop. Email networks and the spread of computer viruses. *Physical Review E*, 66(3):035101, 2002.

[103] Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 441–450. IEEE, 2009.

[104] NIST. Guide to malware incident prevention and handling. November 2005.

[105] Sadia Noreen, Shafaq Murtaza, M Shafiq, and Muddassar Farooq. Using formal grammar and genetic operators to evolve malware. In *Recent Advances in Intrusion Detection*, pages 374–375. Springer, 2009.

[106] Sadia Noreen, Shafaq Murtaza, M Zubair Shafiq, and Muddassar Farooq. Evolvable malware. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1569–1576. ACM, 2009.

[107] Gavin O'Gorman and Geoff McDonald. *Ransomware: a growing menace*. Symantec Corporation, 2012.

[108] Kazuya Okamoto, Wei Chen, and Xiang-Yang Li. Ranking of closeness centrality for large-scale social networks. In *Frontiers in Algorithmics*, pages 186–195. Springer, 2008.

[109] Philip O'Kane, Sakir Sezer, and Keiran McLaughlin. Obfuscation: The hidden malware. *Security & Privacy, IEEE*, 9(5):41–47, 2011.

[110] Digit Oktavianto and Iqbal Muhardianto. *Cuckoo Malware Analysis*. Packt Pbl. Ltd, 2013.

[111] Sirinda Palahan, Domagoj Babić, Swarat Chaudhuri, and Daniel Kifer. Extraction of statistically significant malware behaviors. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 69–78. ACM, 2013.

[112] PandaLabs. Pandalabs annual report 2014. http://www.pandasecurity.com/mediacenter/src/uploads/2015/02/Pandalabs2014-DEF2-en.pdf, February 2015.

[113] Younghee Park and Douglas Reeves. Deriving common malware behavior through graph clustering. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 497–502. ACM, 2011.

[114] Younghee Park, Douglas S Reeves, and Mark Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 39:419–430, 2013.

[115] Romain Péchoux and Thanh Dinh Ta. A categorical treatment of malicious behavioral obfuscation. In *Theory and Applications of Models of Computation*, pages 280–299. Springer, 2014.

[116] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, 42(1):377–388, 2007.

[117] Marco Ramilli, Matt Bishop, and Shining Sun. Multiprocess malware. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 8–13. IEEE, 2011.

[118] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[119] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[120] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 1978.

[121] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 65–79. IEEE, 2012.

[122] Ravi Sandhu. Good-enough security: Toward a pragmatic business-driven discipline. *IEEE Internet Computing*, (1):66–68, 2003.

[123] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[124] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and*

Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on, pages 38–49. IEEE, 2001.

[125] David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178. ACM, 2010.

[126] Payload Security. Vxstream sandbox. http://www.payload-security.com/products/vxstream-sandbox, January 2016.

[127] M Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107. Springer, 2008.

[128] M Zubair Shafiq, S Momina Tabish, and Muddassar Farooq. On the appropriateness of evolutionary rule learning algorithms for malware detection. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2609–2616. ACM, 2009.

[129] Eugene H Spafford. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[130] Statista. Os market share. http://www.statista.com/statistics/218089/global-market-share-of-windows-7, September 2015.

[131] Salvatore J Stolfo, Ke Wang, and Wei-Jen Li. Towards stealthy malware detection. In *Malware Detection*, pages 231–249. Springer, 2007.

[132] Symantec. Malware database. http://www.symantec.com/security_response, November 2013.

[133] Symantec. Regin: Top-tier espionage tool enables stealthy surveillance. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/regin-analysis.pdf, August 2015.

[134] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[135] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *RAID*. 2014.

[136] TheGuardian. Antivirus software is dead. http://www.theguardian.com/technology/2014/may/06/antivirus-software-fails-catch-attacks-security-expert-symantec, October 2014.

[137] Nikos Virvilis and Dimitris Gritzalis. The big four-what we did wrong in advanced persistent threat detection? In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 248–254. IEEE, 2013.

[138] Vasileios Vlachos, Diomidis Spinellis, and Stefanos Androutsellis-Theotokis. Biological aspects of computer virology. In *Next Generation Society. Technological and Legal Issues*. 2010.

[139] VMRay. Vmray analyzer. www.vmray.com/wp-content/uploads/2016/01/VMRay_Analyzer.pdf, January 2016.

[140] John Von Neumann, Arthur W Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1966.

[141] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *ACM SigKDD Explorations Newsletter*, 2003.

[142] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, (2):32–39, 2007.

[143] Tobias Wüchner, Martín Ochoa, Mojdeh Golagha, Gaurav Srivastava, Thomas Schreck, and Alexander Pretschner. Malflow: Identification of c&c servers through host-based data flow profiling. In *Proceedings of the 2016 ACM symposium on applied computing*, 2016.

[144] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. Malware detection with quantitative data flow graphs. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 271–282. ACM, 2014.

[145] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 9148 of *Lecture Notes in Computer Science*, pages 98–118. Springer International Publishing, 2015.

[146] Tobias Wüchner and Alexander Pretschner. Data loss prevention based on data-driven usage control. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 151–160. IEEE, 2012.

[147] Tobias Wüchner, Alexander Pretschner, and Martín Ochoa. Davast: data-centric system level activity visualization. In *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, pages 25–32. ACM, 2014.

[148] James Wyke. The zeroaccess botnet–mining and fraud for massive financial gain. *Sophos Technical Paper*, 2012.

[149] Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 179–190. ACM, 2014.

[150] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Data Mining*, 2002.

[151] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[152] 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, 2010.

[153] Mohd Najwadi Yusoff and Aman Jantan. Optimizing decision tree in malware classification system by using genetic algorithm. *Journal. of New Computer Architectures and their Appl.*, 2011.

[154] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying malicious websites and the underground economy on the chinese web. In *Managing Information Risk and the Economics of Security*, 2009.

Appendix

A. Analysis and Visualization

A.1. Pattern-based Analysis and Visualization

In this section we show how our basic pattern-based detection concept can be extended by additional visualization means to support ex-post analysis of detected potentially malicious activities by human security experts. To this end, we propose *DAVAST*, a novel analysis system that builds on top of our pattern-based detection approach and represents detected suspicious activities at different levels of abstraction. By linking those representations we allow for a seamless purpose-driven analysis, starting from a high-level overview on system activities and step-wise going to a more fine-grained representation of associated low-level data flows. Besides extending our pattern-based detection concept with analysis and visualization means we with this also show, how pattern-based detection can be done online within a live system, while in Chapter 4 and Chapter 5 we focused on malware detection in offline sandbox settings.

Most parts of this section are based on a previous publication of the author [147].

A.1.1. Approach

To support human analysis we do not only want to model low-level interactions as data flows, but also give more high-level semantics to sequences of flows and thus to sub-graphs. We do so by either reusing the already semantically annotated patterns from our deductive pattern-based approach (see Section 4.1.2.1), or by applying our inductive pattern-based approach (see Section 4.2) on samples from similar malware or goodware families to mine family-specific behavior patterns.

This results in a repository with patterns for benign and malign activities that are interesting from a analysis perspective. Just like for our pattern-based detection approaches we then make use of a modified version of the VF2 algorithm [43] to annotate sub-graphs of new QDFGs with the respective high-level activity semantics of the matching pattern. In this way, we can trace activities through different layers of abstraction as a basis for all subsequent visualization steps.

Our approach is based on two pillars: i) the runtime interception of system calls, data flow interpretation, and QDFG generation, and the re-identification of known benign and malign data flow patterns in these graphs; ii) a multi-view representation of the captured system activities, enriched and annotated with context infor-

mation and inferred high-level interpretations of low-level system call activities on the basis of defined data flow patterns.

We see good arguments that our data centric system-wide view on system interactions is more intuitive to human analysts than approaches that focus on raw uninterpreted system calls without context information. By abstracting from concrete system calls we can achieve a significant reduction of data complexity and are able to filter activities that are more likely to pertain to relevant system activities from less relevant ones. The system-wide data flow perspective furthermore not only allows for identifying known (benign or malign) activity patterns or activity anomalies. It also enables analysts to perform more far-reaching ex-post analysis by tracking the flow of data from or to suspicious processes to e.g. assess the extent potential damages or estimate the amount of potentially leaked data.

Finally, our multi-view visualization concept aims at representing all relevant information that can be obtained from the inference and analysis steps in a purpose-specific way. This allows us to provide the right level of abstraction for different analysis use cases. Our tool can present a coarse-grained timeline view of all captured activities for a quick overview on the system state; or a detailed data flow graph based visualization of all activities during a defined time interval for an in-depth analysis of detected malign and benign activities.

A.1.1.1. Architecture

The DAVAST system consists of two subsystem, an event monitoring component deployed at a to be monitored system, and the actual DAVAST analysis and visualization system running on the analyst's side (Figure A.1). The event monitor component intercepts all relevant Windows API calls issued by any process in the monitored system and then either forwards these events to the DAVAST system in real-time, or stores them locally for offline processing in form of event trace logs. In consequence, DAVAST supports two operation modes: In the online mode, the DAVAST system continuously receives and processes events sent from a monitored system; and analyzes and visualizes them on the fly. In the offline mode, DAVAST allows to load previously recorded event traces for ex-post analysis.

We implemented the two distinct operation modes to anticipate the requirements of several typical security analysis tasks. The online mode is well-suited for online forensic purposes like analyzing the activities on a infected system, in order to assess whether it has been infected by a malware or compromised by an attacker and to analyze attacker or malware behavior.

The offline mode is better suited for use cases where permanent human observation and analysis of system activities is not necessary. Examples include settings where the monitor component is permanently installed on workstations that are prone to be attacked or infected by malware, servers that offer important services

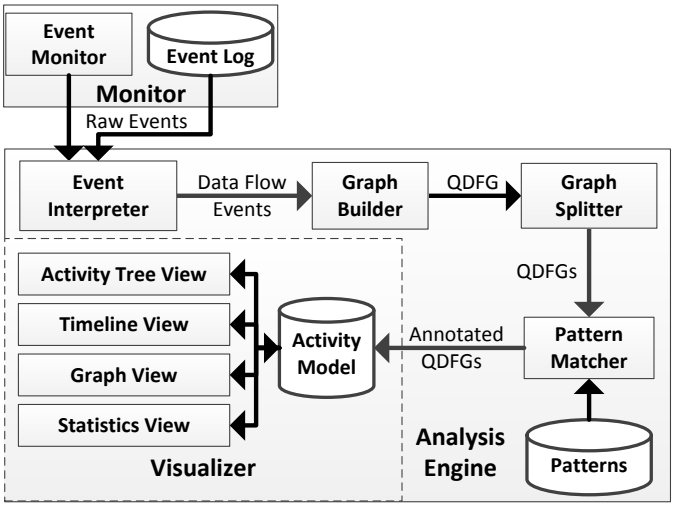


Figure A.1.: Architecture

or host sensitive data, or dedicated honeypots. In such cases, the permanently installed monitor can be configured to continuously store all intercepted system calls of the monitored systems at a secured location. Later on, they can be batch-processed by DAVAST's analysis engine in defined time intervals. If the engine then detects suspicious data flow patterns in the graphs built from these logs, they are flagged for further analysis by a human analyst. The analyst can then use the DAVAST visualizer; in order to manually verify the reported suspicion, to eliminate false positives, or to perform further-reaching analysis steps. In this way, the involvement of expensive human analysts can be reduced. At the same time, it is possible to refine the automated pattern-detection-based analysis with a more intelligent and flexible manual analysis by an experienced human operator.

The architecture of the DAVAST system is depicted in Figure A.1. Depending on the operation mode, raw system call events are received from the *Event Monitor* component, or they are loaded from a recorded *Event Log*. These events are then interpreted by the *Event Interpreter* with respect to their data flow semantics and forwarded to the *Graph Builder* that aggregates them into a QDFG.

Despite event aggregation and graph simplification steps, real-world QDFGs quickly grow large with several thousand nodes end edges. For efficient process-

ing and visualization that is comprehensible to human operators, DAVAST uses a *Graph Splitter* component. It splits the full graph into approximately equal-sized graph slices that capture the activities of a configurable time interval.

After splitting, the QDFG slices are forwarded to the *Pattern Matcher* component. This component tries to match each slice against a list of loaded predefined data flow *Patterns* that pertain to known benign or malign activities. This step is at the core our approach, as it annotates matching sub-graphs of the QDFG slices with inferred high-level activities. Enriching QDFGs with high-level semantics allows to trace between different levels of abstraction for detected activities, from a high-level activity description down to corresponding low-level sub-graphs.

The annotated slices are then pushed to the *Activity Model*, along with additional context information like corresponding time intervals, hierarchical relations between different (sub-)activities, level of confidence of the predicted activity, and a flag whether it is considered malign or benign. Finally, the different views of the *Visualizer* component visualize projections of the data stored in the *Activity Model*.

A.1.1.2. Visualization and Interaction

The purpose of DAVAST's visualization concept is to offer the human operator just the right level of granularity and abstraction that he needs for a specific security analysis task. To achieve this, the *Visualizer* of DAVAST is implemented as a plug-in system. Plug-ins can access the *Activity Model* and even manipulate it to some extent. The current DAVAST prototype comprises four distinct views that aim at different typical security analysis tasks and goals. Because of defined interfaces and a clear Model-View-Controller architecture, it is comparably easy to extend DAVAST with additional views.

The views themselves are projections of the *Activity Model* data as they usually only load specific types and dimensions of information from the model. The *Activity-Tree View* for example uses the inferred high-level activity semantics and the information about respective activity relations and hierarchies and association of activities to time intervals from the model. On the other hand, it ignores the structure of the respective QDFG slices. In contrast, the *Timeline View* uses information about top-level activities, and ignores hierarchical relations.

Due to the shared data model, all views are connected to each other. This allows traceability through different representations of activities. For instance, if a user clicks on one top-level activity in the *Timeline View*, the corresponding top-level event in the *Activity-Tree View* is highlighted to visually connect multiple representations of the same concept. Correspondingly, if the user double-clicks on an activity in the *Activity-Tree View*, the system opens a *Graph View* window that visualizes the graph slice that corresponds to the selected activity.

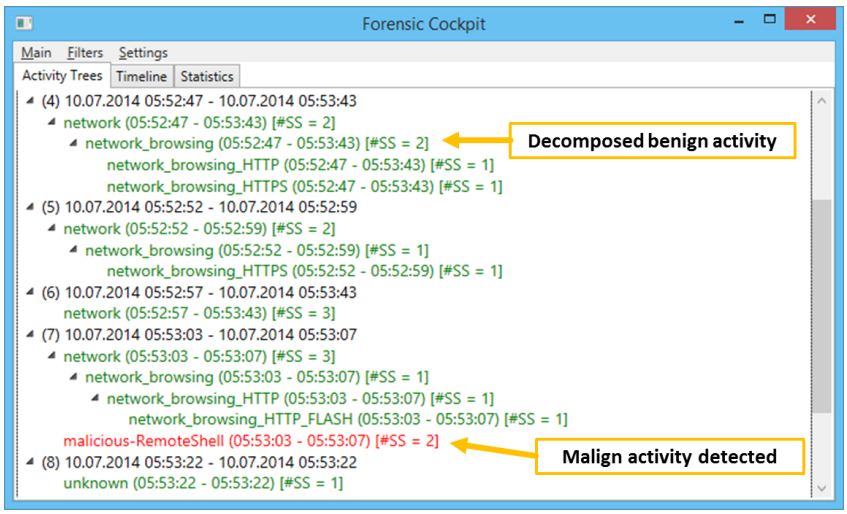


Figure A.2.: Activity-Tree View

A.1.1.3. Activity Tree View

The purpose of the *Activity-Tree View* is to visualize the hierarchical relation between different activities. It depicts the list of time intervals, stored in the *Activity Model*, each denoted by a time interval number, a start, and an end time, and populates each time interval item with the activities associated with this interval. Instead of representing all contained activities in a flat way, the *Activity-Tree View* hierarchically nests related activities according to the activity hierarchy information stored in the *Activity Model*.

Figure A.2 shows an example of such a decomposition where DAVAST detected a *network* activity within time interval 4. The corresponding more specific sub-activities indicate that this *network* activity in fact was a *browsing* activity, further refined into *HTTP* and *HTTPS* activities.

DAVAST colors activities in green if they pertain to benign activities, and in red if they relate to known malicious ones. For each activity, we show a prediction confidence level that represents the number of occurrences of the corresponding activity data flow pattern in the associated QDFG slice. If DAVAST identified multiple distinct activities during one time interval, this confidence number allows the human analyst to reason about the dominance and temporal proportion of one specific activity with respect to the other activities within the same time frame.

The *Activity-Tree View* concept is useful in situations where a human analysts wants to get a coarse-grained overview of all captured activities, with the possibility to get more detailed information about specific activities on demand.

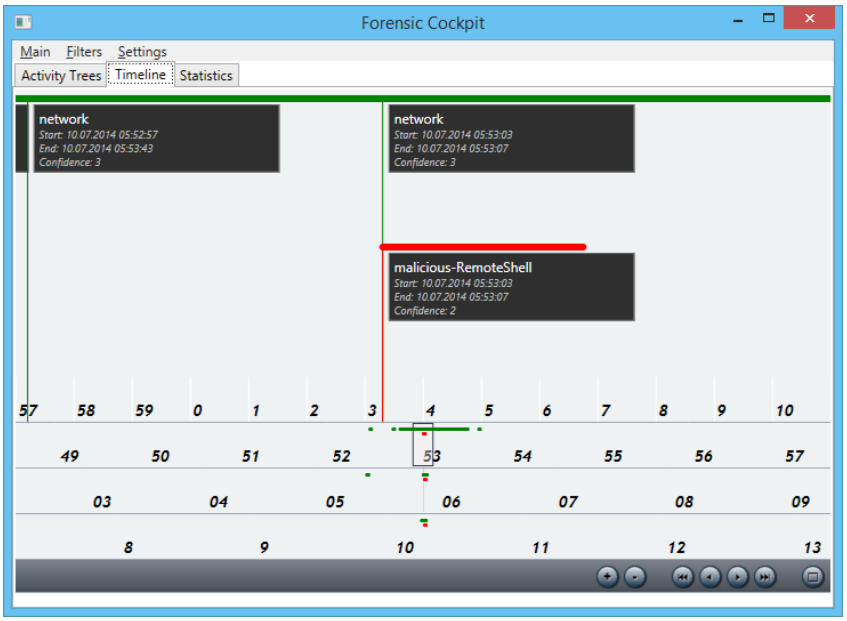


Figure A.3.: Timeline View

By default, DAVAST initially only shows the top-level activities within each time frame and hides all sub-activities. These can be decomposed step-by-step along the hierarchy by expanding the respective sub-activities. Moreover, as mentioned before, the analyst can always open the respective *Graph View* to analyze the corresponding QDFG slice and thus the lowest level of interaction.

An example for such a setting is a scenario where the human analysts wants to analyze the system behavior of a system that he suspects to be compromised, but does not exactly know how and when it was attacked. Even if no known malign pattern matched one of the QDFG slices, the analyst can step through the time intervals and decompose interesting activities to isolate and then further analyze potential infection or attack entry points.

A.1.1.4. Timeline View

The purpose of the *Timeline View* is to give a human analyst a quick overview on the "healthiness" of a system.

In contrast to the *Activity-Tree View*, the *Timeline View* does not show hierarchical relationships between detected activities. However, it allows a more convenient way to browse through the timeline and get an overview of the proportion of potentially benign and malign activities over time. To that end, the *Timeline View* con-

tains a view where top-level events stored in the *Activity Model* are visualized in chronological order. Four timeline band controls allow the user to browse through the timeline with various degrees of precision. The top-most timeline band depicts and controls the *seconds* of the timeline, the one below depicts the *minutes*, followed by the bands relating to *hours* and *days*. Similar to the *Activity-Tree View*, the system highlights potentially malicious activities in red, and benign ones in green. A colored bar visualizes the absolute event duration, independent of time intervals, whereas the *Activity-Tree View* only depicts the relative duration of an activity within the considered slice of the graph. Multiple activities during the same period of time are stacked on each other. The activity with the highest confidence level or dominance is placed on top. Activities with lower dominance are placed below, in a descending order. A zooming function allows the user to control how many activities are displayed on the view panel at one time to e.g. get a coarse birds-eye view on the overall system health.

Figure A.3 shows a *Timeline View* of a potentially infected system. In this example we can see that both a benign *network* and a malign *RemoteShell* activity started on 10.07.2014 05:53:03, and that the malign activity lasted for about 7 seconds, whereas the benign *network* activity was ongoing.

A.1.1.5. Statistics View

The intention of the *Statistics View* is to provide aggregated statistical information about all activities detected within the monitored period of time. This allows analysts to quickly identify anomalies in the activity profile that are likely to correlate with unwanted system behavior, as a consequence of attacks or infections.

In a nutshell, this view provides information to analysts to perform manual visual anomaly detection. This extends DAVAST's "hard" pattern matching functionality, based on fixed data flow patterns, with "soft" human anomaly detection capabilities, primarily based on intuition.

To this end, the *Statistics View* visualizes the proportion between different activities over time (usually, more than one activity is taking place at a time). The x-axis represents the time interval number as defined in the *Activity Model*. The y-axis depicts the relative proportion of one event type in comparison to all other since the beginning of the monitoring period. More precisely, the proportion is defined as the number of occurrences of events assigned to a given activity type divided by the absolute number of occurrences of all events in the observation period.

Figure A.4 depicts the statistics of a typical browsing session. The browsing session starts with an activity peak that reflects a database reading access of the browser. This is the browser initialization phase where configuration data is loaded from a local file-based database. Subsequently, we can see that the browsing session constantly consists of about 45% retrieval of pictures via HTTP, of about 30%

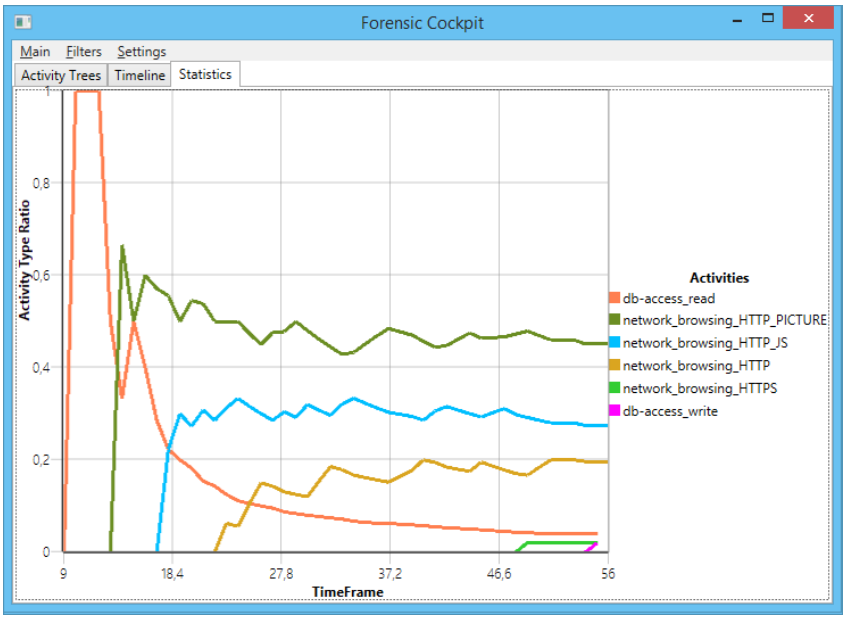


Figure A.4.: Statistics View

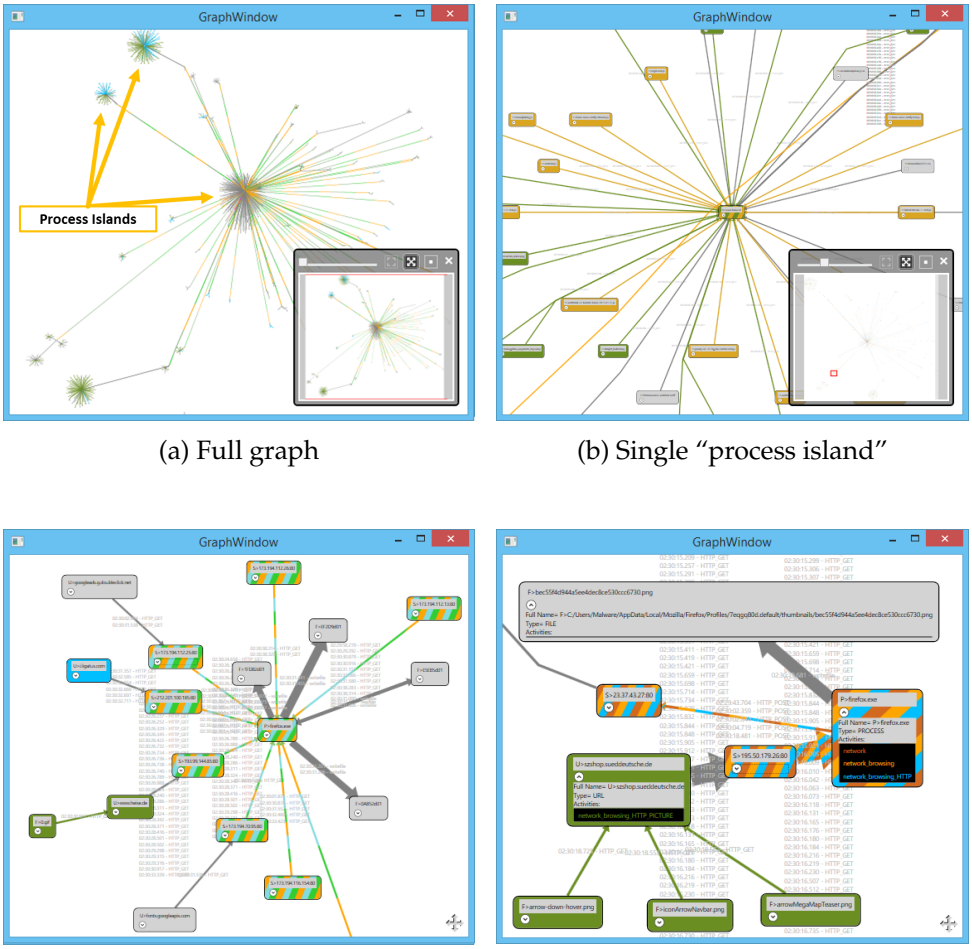
retrieval of JavaScript code, and of less than 5% HTTPS communication and interaction with the local browser database.

Due to the highly aggregated way of representing activity information, this views is a good starting point for forensic investigations. In a forensic scenario, a security analyst can use the *Statistics View* to spot anomalies in the operation of a monitored system with respect to isolated time intervals that potentially contain attacker or malware activity. Examples of such anomalies include irregularly dominant HTTPS activity of an enterprise workstation during non-working lunch time or other non-working hours. During these hours, it can with some confidence be ruled out that the activities are caused by legitimate usage.

Even if no explicitly defined malicious data flow pattern matched the underlying QDFGs, such an initial suspicion, as consequence of a visually identified behavioral anomaly, can still be the starting point for more detailed investigations using the *Activity-Tree View* or the *Graph View*.

A.1.1.6. Graph View

The *Graph View* is DAVAST's most central view as it provides a visualization of the core QDFG-based model that captures all system interactions during the monitored period of time. The *Graph View* provides the highest level of detail and



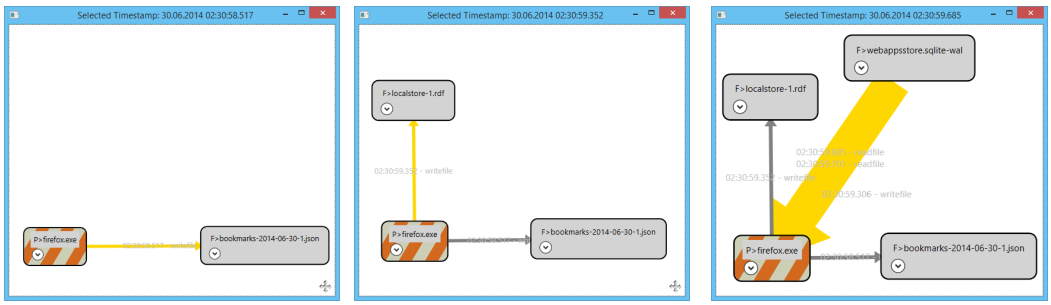
(a) Full graph

(b) Single "process island"

(c) Graph slice of one time interval

(d) Graph slice showing node details

Figure A.5.: Graph View



(a) Timestep *t* (b) Timestep *t+1* (c) Timestep *t+2*

Figure A.6.: Graph View - Chronologically browsing through interactions

most-fine grained visualization to the human analyst.

Its goal is to provide information about all low-level interactions in a detailed way, but thanks to the data flow abstraction, in a concise and comprehensible manner. To access the *Graph View*, the user can either double-click on a (sub-) activity in the *Activity-Tree View* to only visualize the graph slice that pertains to a specific time interval (Figure A.5c), or open the full graph view that visualizes the QDFG of the entire monitored time span (Figure A.5a).

The reason for this distinction is that QDFGs can become huge. In many cases their sheer size renders them useless to be analyzed by humans if they are visualized in entirety. While the full graph view is good to get an overview on the entire system interaction landscape and to quickly track interactions across time interval boundaries, the time interval graph view is more suited to analyze specific interactions within a bounded period of time. These time interval boundaries can be enlarged and shrank at runtime. This gives users full control on the amount of visualized data flow interactions within one window.

For the same reason, the *Graph View* features a special zooming control in the lower right corner of the graph view window that allows to conveniently zoom in and out within a full graph or a graph slice (Figure A.5a). For higher zooming degrees an embedded mini-map that always shows the entire graph context with the currently zoomed part of the graph highlighted limits the risk of the user from getting lost while browsing through the graph (Figure A.5b).

To give the graph visualization an intuitive appearance, we use a modified *linlog* layouting algorithm that arranges the nodes and edges in a way that the graph builds so-called *process islands*. These refer to small circle-shaped sub-graphs of closely connected nodes that pertain to the interactions of one specific process. In these sub-graphs the node in the center of the circle represents the currently acting process. Passive resources that this process interacts with, including files, sockets, or registry keys, are located on the periphery of the circle.

We visualize the amount of data by the edge's thickness. This gives the human analyst additional means to assess the importance and consequences of a specific sequence of interactions on the basis of transferred data amounts. In forensic use-cases, for instance, this helps the analyst assess the dimension of a potential data leakage. Edges are also labeled with the time stamps of the underlying data flow events that were aggregated into that edge.

As can be seen in Figure A.5a, the different *process islands* are sparsely interconnected. This is a result from processes comparably rarely exchanging data with each other, while they more often interact with distinct sets of passive system entities. If processes exchange data with each other, this rarely happens through direct memory access between communication partners, but much more often via shared files, or socket communication. In the graph, this is represented by intermediate socket or file nodes between pairs of communicating processes.

To relate high-level activities to the corresponding sub-graphs in the QDFG visualization, DAVAST assigns a unique color to each activity and paints the respective nodes and edges with this color, if they are part of the corresponding data flow pattern (Figure A.5c). In cases where an edge or node relates to multiple activity patterns, it is painted with a stripe pattern containing all relevant activity colors.

For the special case where one or more nodes and edges are detected to be associated with a known malign activity pattern, these edges and nodes are colored in red. All other nodes and edges are visually faded by ignoring their associated activity colors, and painted in a light gray instead. This directly draws the attention of the human operator to the interactions that are considered malicious (Figure A.8). For presentation purposes, all node labels are kept as short as possible to prevent overly large nodes. This means that for file nodes, instead of directly showing the fully qualified name of the file, only the file name without its path is depicted within the node. When interested in all details of a specific node, users can expand a details panel (Figure A.5d), which contains all available context information associated with the node, including a colored activity legend to clarify the color-activity mapping.

To further increase the usability and comprehensibility of the *Graph View* the visualization in a purpose-driven way, DAVAST features extensive filtering capabilities to e.g. filter certain node or edge types from the visualization, or to assign visual tags to single nodes that are persistently maintained throughout all graph slices to help to track and quickly identify nodes in-between different slices.

As additional means to track flows of data across individual graph slices, DAVAST includes a reachability analysis feature. For each node in a graph slice, a user can trigger a reachability analysis, which generates a new graph that only contains nodes and edges that are directly or indirectly reachable by the selected source node, under consideration of the inherent temporal information associated with the graph edges. This analysis can also be conducted in backward mode where

the new graph contains all nodes and edges that can directly or indirectly reach the selected target node. The reachability analysis usually reduces graph complexity quite a lot as it filters all nodes that are irrelevant for a specific sequence of flows and thus eases comprehensibility of the visualization. We consider this to be a useful tool for forensic purposes whenever one e.g. wants to asses the worst-case impact of a potentially malicious process (forward analysis), or wants to isolate potential sources of a suspected infected file (backward analysis).

Finally, DAVAST's *Graph View* also includes a so-called *TimeMachine* mode where users can browse through the temporal dimension of the graph in a step-by-step manner. This is done via a keyboard-controlled step-by-step construction of the currently active graph slice. If the *TimeMachine* mode is active, the *Graph View* initially only shows the edge and connected nodes with the smallest time stamp in the corresponding time interval. By repeatedly pushing the arrow keys, the user can add or remove additional edges at the granularity of single time stamps. The current edge is colored gold, which allows to conveniently follow the sequence of interactions over time. Figure A.6 shows an slice in time machine mode that visualizes the interaction of a Firefox browser process with other system entities.

A.1.2. Application

In the following we present three scenarios where our approach can be useful to understand malicious behavior. In particular we show, how the visualization supports manual root-cause analysis or verification of reported alarms.

Drive-by Malware Infection An increasing threat is the proliferation of so-called drive-by malware infection attacks. To evaluate the usefulness of our approach, we locally installed a malicious web application exploiting a Flash vulnerability, resulting in a drive-by infection. The matched pattern corresponds to the process `iexplorer.exe` binding a shell (`cmd.exe`) to it, which is considered as malicious. This is detected by our approach, which allows analysts to understand the infection process as depicted in Figure A.7.

Email Worm Infection In this scenario we open a malicious compressed attachment with Thunderbird. We then decompress the file and execute it, resulting in an infection. As depicted in Figure A.8, we see that the malicious process replicates itself, which is detected by our library of malicious behaviour patterns. The replication pattern checks if process creates child processes or executable files of roughly the same size of the originator. A human analyst can then understand the root cause and the steps before the infection, since the pattern highlights the flow

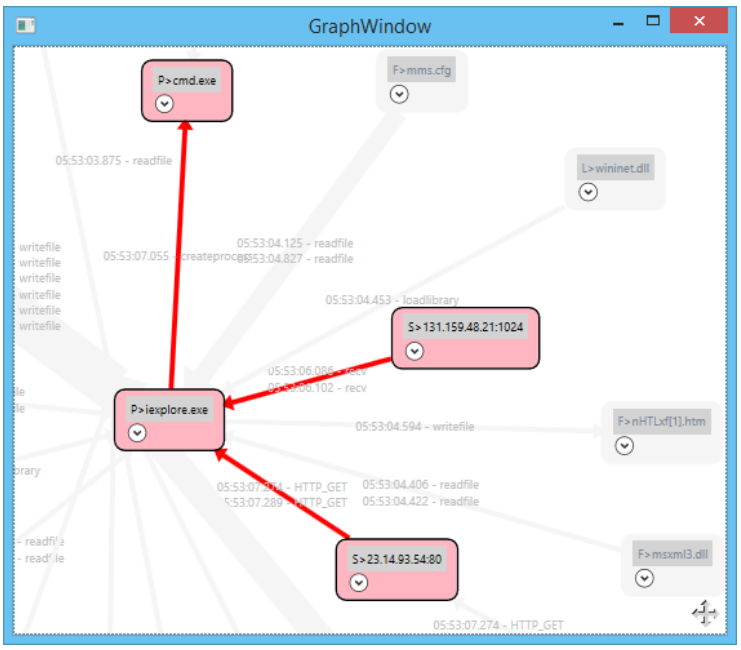


Figure A.7.: Drive-by malware infection

of data from Thunderbird until the self-replication. In the example, process node `invoice.exe` creates two new processes named `cpsdv.exe`.

Data Leakage For forensics purposes, we consider the following scenario: an insider or attacker tries to copy big amounts of sensitive data to a remote server. If such transfers only rarely happen, which likely is the case for cases of data theft, a typical *Statistics View* for such a scenario would likely look as depicted in Figure A.9. While the ratios of most benign activities are almost constant, we can clearly see two spikes for the *HTTP_Upload* activity pattern. We consider such a profile typical for cases of irregular data leakage and easy to spot by human analysts. Such an initial suspicion based on the *Statistics View* can then be a starting point for further investigations, for instance by exploring the related *Graph View*.

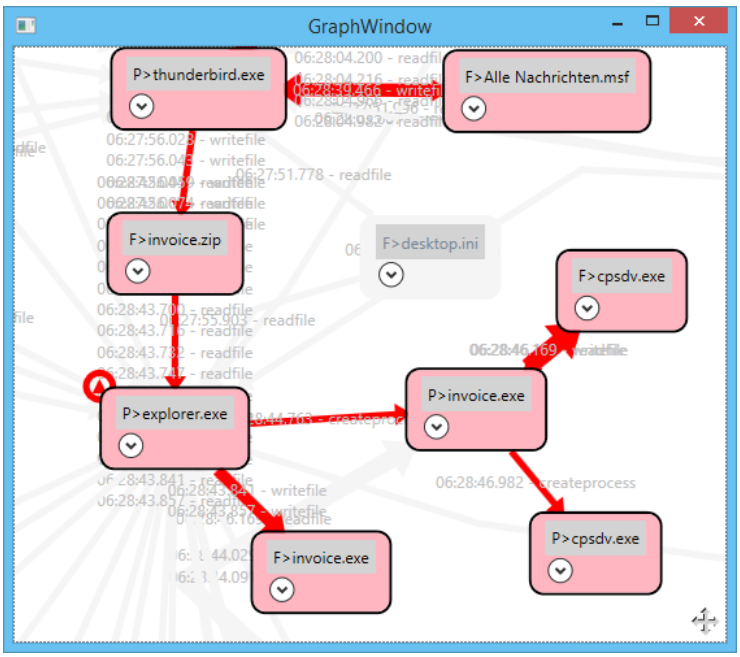


Figure A.8.: Email worm infection

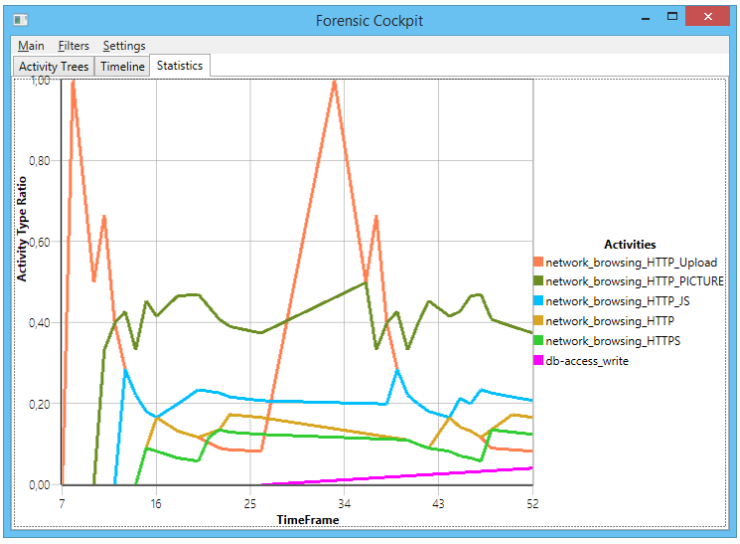


Figure A.9.: Data leakage

B. From Detection to Mitigation

B.1. Metric-based Risk Assessment and Mitigation

In this section we discuss an approach, which we call MalFlow, to extend our basic metric-based detection concept with mechanisms to quantitatively reason about the security risk that is posed by detected potentially malicious behavior. Although, the approach proposed in this section slightly changes the detection setting as if focuses on the detection and risk classification of command-and-control servers rather than malware, it still uses the same baseline idea of profiling malicious activities as quantitative data flows at host level. Besides showing that our basic metric-based detection concept can be enriched by additional means for quantitative risk assessment, we also discuss how we can use this risk information to propose appropriate mitigation action, i.e. incident response strategies. Finally, with this approach we again show that our originally offline detection focused metric-based profiling concept can also be internationalized for online detection.

An extension of parts of this work already has been presented in a publication [143], co-authored as first author by the writer of this thesis.

B.1.1. Approach

Adopting our basic QDFG-based behavior modeling concept (see Chapter 3), we use QDFG metrics that capture the network communication activity between processes of a system and remote network locations, build a clustering scheme on labeled metric sets, and then obtain clusters of domains with similar data flow characteristics, i.e. similar network-centric metric profiles. These clusters get assigned a numeric threat level, roughly capturing the ratio and threat potential of different kinds of known malign and benign domains associated to the clusters. With this technique we can assign risk levels to unknown domains that then can be used to decide upon triggering fine-grained risk-dependent incident response actions. The risk is calculated based on the distance of the feature vectors to the centroids of the different clusters and their respective threat levels.

In contrast to related work [23, 101, 149] this approach is entirely based on quantitatively profiling network-related activity captured at the *host-/system-call level*. This allows us to analyze the interactivity of individual processes with potentially malicious domains and remote resources independent of specific communication

protocols or network topologies and can be seamlessly integrated into standard dynamic malware analysis sandbox environments. Finally, our approach allows to conduct fine-grained incident response strategies based on assessed risk of identified potentially malign domains, independent of concrete communication protocols. We consider our approach to be more precise and targeted than approaches that use network-level monitors, since it allows us to inhibit communication at a per-processes, or even per-socket level if malign behavior is detected. In contrast, network-based approaches reason about entire hosts, IP address ranges, or ports in such cases and are considerably less precise and more intrusive.

We now describe the different steps of the in more detail:

1. The initial steps are similar to the raw data generation process described in Section 4.1.2. First we execute malware samples in a sandbox environment that captures network related API calls. These are interpreted as (quantifiable) data flows between the malware process, its established sockets, the contacted domains, and the resources at the remote endpoint that the malware interacts with. All data flows are aggregated as QDFGs.

2. Given a set of such QDFGs, each node of the *domain* type is profiled in terms of temporal characteristic data flow metrics, such as average number of bytes sent and received, or the number of accessed remote resources.

3. Using a set of features from a large body of malware samples we then employ a (unsupervised) clustering algorithm that outputs clusters of communications with domains with similar data flow behavior.

4. To assess the potential risk that arises from communicating with the domains of a cluster, we annotate them according to domain blacklists. To do so, we obtain lists of known malign domains by interfacing with popular blacklist services like the Google SafeBrowsing API.

5. Knowledge about known malign domains is used to estimate the overall threat potential for the computed domain clusters. Following the hypothesis that domains with similar data flow profiles share similar threat levels (e.g. because they are all referring to addresses of C&C servers and thus show typical malware communication behavior), we infer the overall cluster threat potential (severity) based on the threat potential of its known members.

6. Interpreting the distance between unclassified feature sets and the clusters as risk scores, we perform targeted risk mitigation strategies by selecting appropriate risk-based incident response actions based on predefined profiles.

The upper-right part of Figure B.1 shows our high level server-side architecture. Solid lines depict the steps of the training phase, whereas dashed lines represent

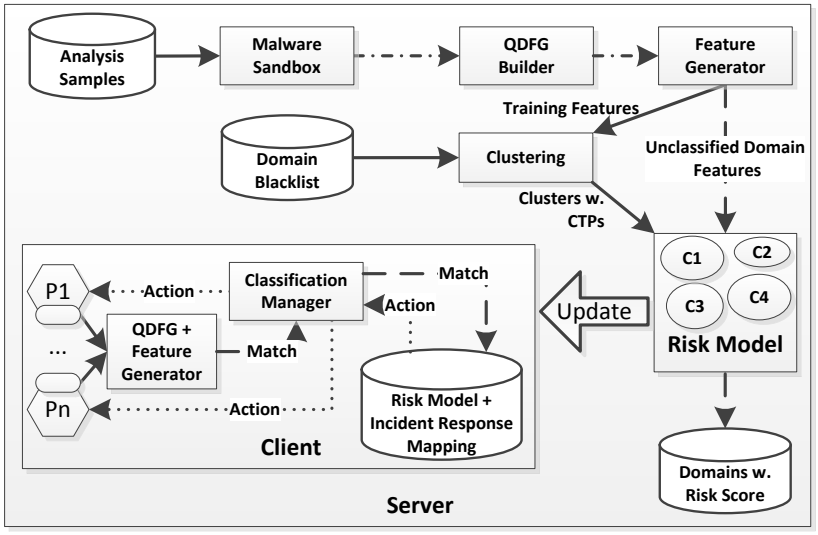


Figure B.1.: Architecture

classification steps. The semi-dashed lines in the upper right part denote processing steps used for both, training and classification. In the following we describe the different steps of this process in more detail.

B.1.1.1. Step 1: Network Behavior Extraction with Dynamic Malware Analysis

To obtain the raw data needed for profiling malware interaction with C&C servers, we follow the same procedure as described in Section 4.1.2.

MalFlow is based on the interpretation of data flows between malware and contacted domains at the system call level. We hence only capture networking-related system and API calls issued by the executed malware samples or processes that interacted with them. We decided to directly profile the malicious samples at the Windows API level for two reasons: first, directly capturing network activity by intercepting system calls allows us to selectively capture network activities of specific processes and thus significantly reduce to-be-processed data and noise. This is hard to achieve with monitors at the network level. Second, if the monitor is deployed at the host level, the interception of network-related system calls for

individual processes allows us to inhibit further malign network activity at a per-process rather than at a per-machine level.

In addition to our standard raw data retrieval, we obtain fine-grained networking information like the domain of a remote endpoint and requested or manipulated resources by having our monitor inspect the buffers of the respective *send* or *recv* WinSock function calls. We apply specifically crafted protocol information inference techniques on the buffers to extract meta-information from the buffers for several standard internet protocols. For instance, we try to identify patterns that correspond to domain or IP strings within the buffer, both in the raw buffer content and in base64-decoded interpretations.

B.1.1.2. Step 2: Network-focused QDFG Features

The network-related system call traces obtained in the previous step give rise to the corresponding QDFGs as basis for data flow profiling by applying the Windows instantiation of our generic system model (see Section 3.2) to the system calls of the captured traces.

The actual profiling of system behavior in terms of features on QDFGs works similar to that of our metric-based approaches Section 5.1.2. In contrast, however, we are interested in the changes of behavior over time instead of looking only at the behavior at one specific point in time. For that reason we capture the behavior over time rather than only at a single point. We thus generate one QDFG every n events, similar to the *temporal* features of [135]. Sampling the activity at only one specific moment in time would likely miss a lot of characteristic behavioral information (e.g. due to dormant functionality). Using multiple sampling points over time and looking at incremental differences solves this problem to some extent.

One way of capturing structural differences of QDFGs over time is to calculate characteristic graph properties like in-degree and out-degree of interesting nodes or edge (weight) distributions. As we are interested in changes of the communication behavior of a process with different network endpoints, we calculate such properties for all domain nodes that we want to classify.

More specifically, for each domain node we compute the number of connected sockets and files, which relates to requested or manipulated remote files, respectively to involved local processes. We consider both nodes connected with outgoing edges and incoming edges. Table B.1 surveys all considered feature types.

Features can be seen as functions on QDFG nodes and edges. As we need to encode the temporal information of all QDFG steps for all features in *one* feature vector, we sequentially store all calculated feature values as elements of a vector $f \in \mathbb{R}^n$ where $n = |Features| \times |Steps|$.

The intuition behind the used features and their temporal scope is that we expect the data flow profiles of interaction with malign endpoints to look substan-

Feature	In	Out
$Degree_{Socket}$	Number of sockets receiving data from remote endpoint.	Number of sockets sending data to remote endpoint.
$Degree_{File}$	Number of files requested from remote endpoint.	Number of files sent to remote endpoint.
$Events$	Number of events that contributed to an in-flow to a remote endpoint.	Number of events that contributed to an out-flow to a remote endpoint.
$Avg.\frac{Flow}{Event}$	Average incoming bytes per event.	Average outgoing bytes per event.

Table B.1.: Overview of used temporal QDFG Features

tially different from communication with benign endpoints. An example for such a behavioral difference is connectivity tests with a benign website like google.com that should be reachable if the infected host is connected to the internet. After a successful connectivity test, the malware then periodically communicates with its command and control server to receive new instructions or upload stolen data.

The respective data flow profiles for the interactivity with the benign website would then likely indicate a steep increase of the transferred data to the respective domains in the beginning, followed by a long period of silence, as the benign domain only needs to be contacted once. In contrast, the malign C&C communication would require periodic interaction and thus would lead to a more "spiky" data flow profile. This situation is depicted in Figure B.2, showing data from the real-world execution traces of a malware sample.

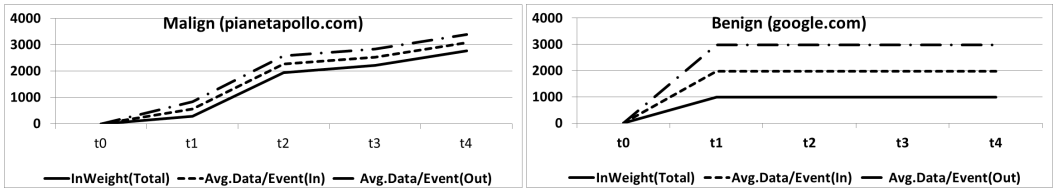


Figure B.2.: Temporal Features for Malign and Benign Domains

B.1.1.3. Step 3: Training Phase

After generating QDFGs and their respective data flow features for captured event traces of a large body of (likely malign) executables, we are now ready to train the machine learning core of our approach. In principle, one could do that with the help of a standard supervised machine learning scheme, fed with features sets for interactivity with certain domains, labeled by matching those domains with blacklists/whitelists of known malign/benign domains.

Unfortunately, such domain lists are inherently incomplete [81], quickly become outdated, and thus would often provide incorrect labels to the feature sets. This is for instance caused by malign domains taken down by authorities, domains not anymore used by malware, or blacklists not being able to keep the pace with malware developers that register hundreds to thousands new websites every day. Trained on such imprecise information for labeling the training feature sets, a classifier would likely incorrectly classify unknown feature samples.

Therefore, our approach instead relies on an unsupervised clustering technique that takes unlabeled feature sets as input and outputs clusters of samples with approximately similar data flow behaviors. More specifically, we use Mini Batch k-means, a faster approximate version of the more expensive k-means clustering [125], to obtain clusters based on aggregation of instances that are located at a similar distance around a cluster centroid. This notion of distance of an instance, i.e. a set of features corresponding to a domain instance, to a certain cluster is later needed as the first component (i.e. likelihood) for calculating individual distance-based risk scores.

B.1.1.4. Step 4: Classification and threat potential

We want to assign a so-called *threat potential* to domain clusters. This cluster threat potential represents the weighted potential security threat that arises from communicating with the members of this cluster and will thus be later used as the second component (i.e. severity) of the per-sample risk score calculation.

We first need to obtain additional information for a cluster. We do so by matching the cluster members' domains against various malware domain blacklists. For our prototype, we use the APIs offered by popular blacklisting services like Google SafeBrowsing and malwaredomains.com. We differentiate between two classes of domains: domains that had a match in a blacklist, denoted by *malign*, and *unknown* domains that we did not find in a blacklist.

Formally, let $\mathcal{D}$ be a set of domains. Domains $d \in \mathcal{D}$ have a type $\gamma \in \Gamma = \{malign, unknown\}$. $\tau : \mathcal{D} \rightarrow \Gamma$ returns the type of a given domain. Let $c \in C$ be one specific cluster drawn from a set of clusters, C. Each cluster is a set of pairs of domains and their respective features $c \in (\mathcal{D} \times \mathbb{R}^n)^m$. The *threat level* of a

domain captures the estimated threat that arises in a specific deployment context (i.e. one company or organization) if communication with a potentially malign or unknown domain is not controlled or blocked, respectively. For known *malign* domains we assign a positive threat level and for *unknown* domains a zero threat level. We thus model the threat level by $tl := \Gamma \rightarrow \mathbb{Z}$. One simple threat level function is $tl(malign) = 1$ and $tl(unknown) = 0$.

The exact threat levels for the different classes manually need to be set once to match the protection goals within a specific context. They can substantially differ for different operational contexts. In some companies, where undetected security incidents are considered significantly worse than the denial of legitimate actions, one might assign a very high threat level to known malign sites with the risk of producing false positives (i.e. blocking legit actions). In contrast, in companies where even slight interference with legitimate workflows is considered prohibitive, reducing false positives would be an important goal which can be achieved by assigning smaller threat levels to malign domains.

We then calculate the threat potential for each cluster $c \in C$ as the average of the cluster members' individual threat levels. We compute the average in order to explicitly address the problem that the individual domain class labels and thus their individual threat levels might not be up-to-date and reliable and thus, when taken alone, do not provide enough reliable training information.

This definition of a cluster's threat potential means that the threat potential gets higher the more known malign and less unknown members it has. To project the cluster threat potential levels to an $[0, 1]$ interval, we divide each individual cluster threat potential by the size of the cluster. The *cluster threat potential*, $ctp := C \rightarrow [0, 1]$, is thus defined as $ctp(c) := \frac{1}{|c|} \cdot \sum_{(d,f)\in c} tl(\tau(d))$.

This way, in contrast to directly using the labels obtained from blacklists for supervised classification, we distribute the risk induced by potentially incorrect labels among the different clusters. This makes the corresponding predictions less sensitive towards noisy and partially incorrect training data.

B.1.1.5. Step 5: Risk Assessment Phase

With the trained classifier and the ctp-annotated clusters we can now assign meaningful risk scores to unknown domain samples. To classify an unknown sample, i.e. calculate its risk level, we determine the cluster with the highest similarity of its member's data flow profiles, i.e. (data flow) feature sets, and the profile of the sample to be classified.

To this end, we compute the distance of each cluster's centroid to the feature set of the sample to be classified. As we use a variant of the k-means algorithm for clustering, centroids in our case refer to the mean of the feature vectors belonging to a cluster. As a consequence, we can use the Euclidean distance between the

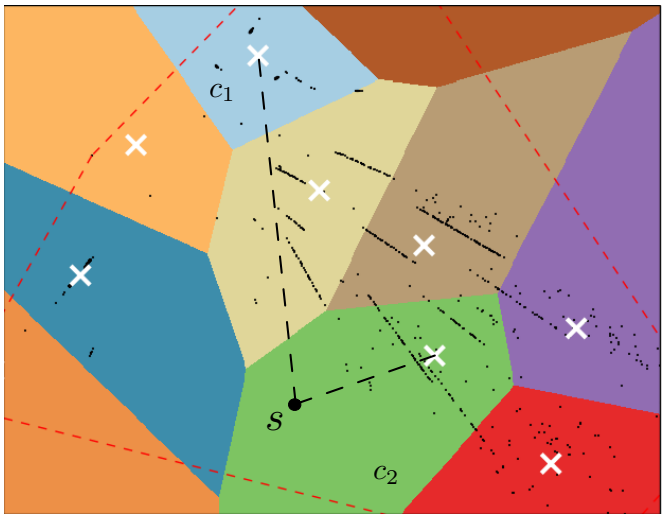


Figure B.3.: 2-dimensional projection of clusters and their centroids.

samples feature vector and the clusters' centroids.

More precisely, we define the distance $\|\cdot\|_c : \mathbb{R}^n \times C \to [0,1]$ of a sample, represented by its feature vector $s \in \mathbb{R}^n$, to any cluster $c \in C$ as follows:

$$\|s\|_c := min\Big(1, \frac{\|s - \text{centroid}(c)\|_2}{N}\Big)$$

where $\text{centroid} : C \to \mathbb{R}^n$ is the centroid vector of a cluster, and $\|\cdot\|_2$ is the Euclidean distance. As for *ctp*, we want to project the distance to an interval between 0 and 1. We do so by normalizing each distance by the diameter N of the smallest sphere containing all clusters. Since there could be samples whose distance to a centroid is greater than N, we set 1 as the maximum possible distance. As an example, the point s in the 2-dimensional projection of Figure B.3 is relatively close to the centroid of c_2 but relatively far from the centroid of c_1.

The actual risk score is then classically computed as "likelihood times severity". Intuitively, the smaller the distance to a high risk cluster, the higher is the overall risk. This leads us to using the complement of the normalized distance to represent the likelihood. It is also intuitive to choose the risk associated with the *closest* cluster to the sample. The risk assessment $risk : \mathbb{R}^n \to [0,1]$ is thus computed as $risk(s) = (1 - \|s\|_{\hat{c}}) \times ctp(\hat{c})$, where $\hat{c}$ is the cluster closest to sample s, i.e., $\forall\, c \in C \;\; \|s\|_{\hat{c}} \leq \|s\|_c$.

By applying this classification strategy we are now able to determine individual

risk scores for unclassified domain communication data flow profiles that we later use for reasoning about appropriate incident response actions.

B.1.1.6. Step 6: Deployment Modes and Incident Response

Our approach can now classify unknown domain samples and annotate them with risk scores. We need to map this risk to specific incident response actions.

Malflow supports the definition of so-called *incident response profiles* (IRP). These map risk score intervals to incident response actions for an identified malign network activity risk. Such actions include: reporting the potential threat to security responsibles, forwarding the traffic associated with a detected malign domain to a sinkholing server, blocking further communication with this domain, or even killing the respective We describe IRPs as partial functions $irp \in IRP : \mathbb{R} \rightarrow A$ that return a specific incident response action $a \in A$ if a provided risk score r is within a certain interval. An example for such a profile $irp_1 \in IRP$ would be: $irp_1(r) =$ Ignore if $0.0 \leq r \leq 0.2$, $irp_1(r) =$ Notify if $0.2 < r \leq 0.5$ and $irp_1(r) =$ Block if $0.5 < r \leq 1.0$.

To operationalize this process, MalFlow supports two distinct deployment modes: *server-only* and *client-server* deployment. For server-only deployment, we perform all previously discussed training and classification steps directly at the server where the malware sandbox is deployed. The obtained classification results are then used to e.g. populate proxy blacklists to a-priori block the identified malign domains or notify security responsibles upon detected connection attempts to these domains. This deployment model bears the benefit of full centralization and easy integration into a typical dynamic malware analysis process. The downside is that it does not allow to conduct precise per-process incident response actions at the client side.

This is overcome by MalFlow's *client-server* deployment model where domain classification and risk mitigation are pushed to the endpoints. Rather than forwarding the identified malign domains to proxy blacklists, the server merely generates and provides the classification and risk models according to the process described above – classification of new domains is done by the clients.

To this end, clients periodically receive updated classification models from the MalFlow server along with the risk mitigation mappings. To monitor the system for potential malign network communication, the clients run the same custom IAT-patching based function call interposition monitor, that w that gets injected into each user-mode process within before its start-up. After injection, this monitor intercepts all network-related calls to the WindowsAPI, i.e. WinSock functions like Socket, recv, or send. Intercepted events are forwarded to the QDFG and feature generation component. The classification manager matches these feature sets against the classification model, which it received from the MalFlow server,

and calculates the respective risk score for the contacted remote endpoint.

Together with the pre-defined IRPs, the risk score determines the incident response action for the identified potential threat. Depending on the IRP, compensating actions can reach from simple notifications to security responsibles, inhibition of subsequent calls to an identified malign domain, modification of the respective calls to partially forward traffic to a sinkhole server, or even terminating the issuing process to contain further damage. A high-level description of the client-side architecture is depicted in the lower-left part of Figure B.1.

Lazy vs. greedy modes. MalFlow clients can be configured in two modes: *lazy*, where the client only captures events for a fixed initial time window after establishing a networking session, and *greedy*, where events are intercepted continuously, employing a sliding window scheme over the captured event traces to ensure an evaluation baseline that is compatible to the one used for training. While the *greedy* mode is more reliable in that it covers more network interaction, it is more expensive as it requires continuous interception of events.

In contrast, the *lazy* mode is leaner and significantly less expensive as the monitors only intercept API calls for a fixed period until automatically unloading the costly interception mechanisms. On the other hand, this mode might miss events that happen after the initial monitored time window and thus fail to detect delayed malign network activity. However, our evaluations on a large set of malware revealed that only few malware families implement some form of time-triggered functionality. The choice of the appropriate detection mode thus depends on the protection goals and performance constraints.

B.1.2. Evaluation

We conducted experiments for both the server and the client components of a prototypical implementation. The server-side experiments, i.e. the training steps, were conducted on an Intel Xeon 12-core 3.5GHz machine with 64GByte of RAM. The client side was evaluated on a machine equipped with two physical 2.8GHz Intel i7 cores and 8GByte of RAM.

In this setting we executed about 11.000 malware samples as evaluation baseline in our sandbox and recorded their behavior for 5 minutes.

For profiling data flows, we captured 5 sample points, one per minute. Calculating 8 features per sample point yielded 40 ($= 8$ Features $\times$ 5 Steps)-dimensional feature vectors. Their threat level was determined by matching the respective domains against a blacklist with about 70.000 entries. In total we obtained a set of 2035 distinct domain-feature vectors out of which the blacklist matching revealed 261 to refer to known (active) compromised servers.

B.1.2.1. Effectiveness

We now want to investigate MalFlow's ability to discriminate known malign from likely benign ones based on the risk values computed from the data flow profiles. To this extent, we performed a 10-fold cross validation evaluation where we repeatedly trained our system with 90% parts of the data set and used the resulting risk model to calculate risk values for the remaining 10% of the data set. Doing so ensured that no knowledge about the testing samples was leaked to the training phase, i.e., disjoint testing and training sets.

The goal for this experiment was to find a partitioning of the risk value space that discriminated best between known malign domains from likely benign ones. This means that we aimed at finding a good risk threshold $\theta \in [0, 1]$ for which in the optimal case the risk values of all known malign samples remained above and the risk values of all likely benign ones remained below.

To investigate the discrimination ability for different thresholds θ we calculated the classification performance in terms of *recall* and *precision*. We denote benign domains incorrectly classified as malign as false positives *fp*, correctly classified malign (benign) domains as true positives *tp* (true negatives *tn*), and benign labels wrongly classified as malign as false negatives *fn*. Precision and recall are defined by $precision = tp/(tp + fp)$ and $recall = tp/(tp + fn)$.

Note that we use domain blacklists to check the correctness of our prediction. More precisely, we evaluate the performance of our classification in terms of congruence of the predictions with the results from matching the domains against popular domain blacklists. Given that such blacklists are incomplete [81], there is a chance that some of the testing domains are in fact malign and mislabeled.

This has some consequences for the interpretation of our evaluation results. As we cannot say with absolute certainty that a domain in fact is malign or benign, a disagreement between predicted class and ground-truth class does not necessarily constitute an incorrect prediction. In particular, cases in which our approach predicted, from the viewpoint of the blacklist matching, a benign domain to be malign, do not necessarily need to be real false positives. As we are thus in the malign vs. likely-benign setting rather than in the malign vs. benign setting, the subsequently presented precision and recall measures need to be interpreted as lower bounds.

Figure B.4 shows the relationship between precision and recall for different cluster counts. As we can see, we can achieve a recall of at least 0.78 when fixing the precision to 0.98. This means that we are able to detect at least 78% of all known malign domains with little risk of benign domains being misclassified as malign. Again note that this is a conservative estimation under the assumption that all domains that were not included in a blacklist were in fact benign.

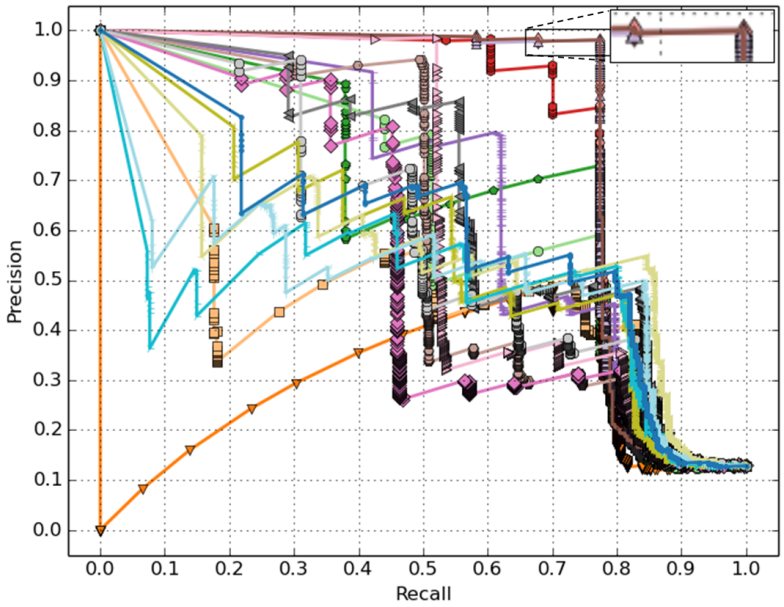


Figure B.4.: MalFlow effectiveness

B.1.2.2. Efficiency

For the efficiency evaluation we analyzed the computational effort for the server and the client side components. We also investigated the impact of running our client-side prototype on the overall system performance.

While less important for server-only deployment settings, the introduced overhead for monitoring network-related events with a system call monitor at host-level potentially has a significant impact on system performance at client side in client-server deployments.

To reason about the absolute and relative performance overhead we conducted a series of performance measurement experiments in a typical client environment (clients are more resource-limited than servers). For these experiments we simulated typical networking scenarios like browsing a website or downloading files from a remote server with varying file sizes. All experiments were conducted both on a standard Windows 7 SP1 installation and in the same system with our client component running.

To minimize noise when observing the effect of the client-side monitoring infrastructure, we used a Windows version of the *wget* command to query a webserver for specific websites and different-sized files. More specifically, we loaded the www.google.com and the www.cnn.com main pages and downloaded different files between 0.1 and 100 MByte of size from a big webspace hosting provider.

Target	T_{norm}[ms]	T_{mon}[ms]	Δ_{abs}	Δ_{rel}
www.google.com	288	2162	1874	7.52
www.cnn.com	504	2632	2128	5.22
0.1MByte	258	2113	1885	8.21
1MByte	414	2320	1906	5.60
10MByte	936	2835	1899	3.03
100MByte	6907	8289	1382	1.20

Table B.2.: Average Monitoring Overhead at the Client Side

To weed out environmental influences we repeated each experiment 100 times.

We measured the average durations for both the unmonitored (T_{norm}) and the monitored (T_{mon}) settings, as well as the absolute ($\Delta_{abs} = T_{mon} - T_{norm}$) and relative ($\Delta_{rel} = \frac{\Delta_{abs}}{T_{norm}}$) overheads, shown in Table B.2.

To get an idea about the bootstrapping overhead we also measured the start-up time for running wget with and without our monitor running. Our client component needed 1917ms in average to hook into a newly created process. Although the relative overhead of up to 7 times the normal execution time seems prohibitive for short-living processes, this overhead is barely noticeable for longer-living processes (the evaluations for such longer-term networking activities only show a relative overhead of 1.2). Considering that by default (in lazy mode) we capture a network connection for about 5 minutes, the 100MByte experiment gives us the closest hint on the actual overhead that would be introduced in an operative setting. By subtracting the duration of the unhooked from the hooked experiment, we get an average absolute overhead of $8289ms - 6907ms = 1382ms$. As the data flow profile features used by MalFlow are properties of the respective QDFGs, the computational effort is directly determined by the size of the respective graphs. Within the considered data set the average graph size was 10 edges (ranging from 3 to 75 with σ=4), resulting in an average total feature effort of about 22ms (ranging from 6 to 95 with σ=11). Looking at the effects of increasing graph sizes on the feature computation time, we could infer that the relative computation time per edge remained roughly constant, i.e., independent of the graph size.

For evaluating the computational effort of our training phase we measured the total time spent on different-sized training set sizes and varying cluster counts. While the cluster threat potential calculation boils down to hash-table lookups with linear complexity, finding an optimal cluster partitioning with vanilla k-means is NP-hard. However, for the employed approximate variant of k-means the actual complexity is within $\mathcal{O}(k * n * s)$, with k being the number of clusters, n the number of instances, and s their dimensionality.

Interestingly, our experiments only revealed a measurable correlation between computation time and cluster count, but not between sample count and overall training time. We explain this with the relatively small number of clustering instances in our setting in comparison to the several orders of magnitude bigger sample sizes that the employed clustering algorithm was originally designed for [125]. Considering various cluster counts, training our approach on the full data set at average took 62ms (ranging from 22 to 90 with σ=20).

As the bulk of the risk calculation consists of calculating the Euclidean distance of a sample to all cluster centroids, the effort is linear with respect to the number of clusters. Considering the small number of clusters and the comparably low dimensionality of the vectors in our setting, the resulting overhead at average was 3ms (ranging from 1.5 to 7.1 with σ=0.8). When compared to the monitoring overhead, this is negligible.

In sum, classifying a network connection at the client side would thus on average take $1382ms + 5 * 22ms + 3ms = 1495ms$. This amounts to a relative overhead of about 30% w.r.t. a monitoring window of 5min ($= \frac{5000ms+1495ms}{5000ms}$), which, of course depending on the operational context, we consider an acceptable cost for being able to perform fine-grained real-time risk-based enforcement.

B.1.2.3. Threats to Validity and Limitations

While we could show that our approach performs well in discriminating domains with reasonable computational overhead, we are aware of some limitations of our evaluation. As usual with machine-learning approaches to malware, we cannot claim that our results generalize to malware found in the wild or to other data sets. An objective evaluation of this generalizability is very hard to achieve.

As discussed before, the usefulness of the effectiveness results highly depends on the reliability of the ground truth database, i.e. the coverage of the employed domain blacklists. A bad coverage possibly leads to effectiveness reported to be lower than it actually is. We tried to counteract this threat by feeding our blacklist with data from diverse sources, including public domain blacklist services like Google's SafeBrowsing service.

Our current prototype profiles malware network interaction with a user-mode hook to intercept calls issued to the WinSock library. Kernel-mode malware, or more generically, malware that does not use the WinSock library can thus not be profiled by our current prototype. We did some experiments using a kernel-driver based on SSDT patching technology instead of user-mode hooking in which we were able to intercept even such malware. However, we deem the necessarily introduced overhead and stability issues not justified considering that the vast majority of commodity malware uses WinSock for networking.

Finally, our current prototype only considers the first 5 minutes of activity of a

malware for training. Although we barely saw time-triggers in the analyzed malware set, there is a risk of malware delaying the actual malign behavior and thus subverting our training scope. One can to some extent counteract this threat by extending the monitoring period for generating the training features at the cost of increased profiling time and thus reduced malware analysis throughput. Also utilizing stimulus-response techniques, e.g. finding and removing sleep-based time triggers [20], would to some extent limit this risk.

C. Evaluation Data Set

C.1. Malware

To foster comparability with other detection approaches we decided to use a publicly available malware collection, named *Malicia* [100], for populating our evaluation data set. These samples were obtained by milking 500 different drive-by download servers for 11 months in 2011, i.e. automatically connecting to them and crawling them for potentially malicious binaries. To label the obtained binaries, i.e. to establish our ground truth, a combination of static analysis (on embedded icons) and dynamic analysis (on captured network traffic) was used [100].

In sum, this left us with 6991 distinct malware samples from 18 different malware families, as listed in the following:

Malware Family	**Samples**
Cleaman	25
Cridex	50
Cutwail	2
DPRN	1
Fakeav	5
Harebot	35
Ramnit	4
RansomNoaouy	4
Reveton	7
Russkill	1
SecurityShield	112
Smarthdd	47
Spyeye	4
Ufasoft-bitcoin	3
Winwebsec	4234
WinRescue	4
Zbot	1572
Zeroaccess	881

C.2. Goodware

To obtain a representative and diverse set of benign Windows applications for our evaluations we followed two data retrieval strategies: 1) we downloaded the 50 most popular free software packages from www.download.com and extracted all executable binaries from the downloaded archives, and 2) we crawled 3 Windows 7 installations of researchers in our working group for all executable binaries.

In total we obtained 481 distinct goodware samples, which we subdivided into five different categories: *Console Applications* are windows programs that often do not require any sort of user input and do not feature any graphical user interface, *GUI Applications* that usually react on user input provided via graphical user interfaces, *Installers* that semi-autonomously install software on a computer, and *Internet Applications* that interact with remote computers via a network connection. While the distinction between the categories is not clear-cut, this categorization is intended to draw a picture of the diversity of evaluated benign software samples.

The distribution of samples over this categories looks as follows:

Malware Family	**Samples**
Console Applications	278
GUI Applications	114
Installers	52
Internet Applications	37

In detail we evaluated the following benign applications:

7zip, 7-Zip, A Note, Abakt, AdapterTroubleshooter, Adaware, AddInUtil, Agent, Aitagent, Analysis, Anote, Append, Appidcertstorecheck, Appidpolicyconverter, Arp, Ascsetup, Aspnet_wp, AstroGrep, Astrogrep, At, Audacity, Audiodg, Autoit, Avast, Avg, AxInstUI, Baaupdate, Bamital, Bckgzm, BdeUISrv, BdeUnlockWizard, Bfsvc, BitLockerWizard, BitLockerWizardElev, Bitsadmin, Bthudtask, CamStudio, Carberp, Cbeplay, Ccsetup, Cdrtfe, Change, Charmap, Chess, Chgport, Chkdsk, Chkntfs, Cideamon, Cipher, Clamav, ClamWin, Cleanmgr, Clickonce_bootstrap, Clip, Cmak, Cmbins, Cmdl32, Cmmon32, Comp, Compact, CompMgmtLauncher, ComputerDefaults, ComSvcConfig, Consent, Control, Convert, Csc, Csrstub, Ctfmon, Cttune, Cvtres, Dcomcnfg, DCplusplus, Debug, Defrag, Delegate_execute, DeviceEject, DfdWiz, Dialer, Diantz, Dinotify, DiskCleaner, Diskperf, Diskraid, DisplaySwitch, Djoin, Doskey, Dpapimig, DpiScaling, Drivermax, Driverquery, DVDMaker, Dvdplay, Dvdupgrd, Dw20, Dwm, Dwwin, Dxdiag, Dxpserver, Edlin, EdmGen, Efsui, Ehexthost, Ehprivjob, Ehrecvr, Ehsched, Ehvid, EjectUSB, Email_-thunderbird, EMule, Eraser-6, Esentutl, Eudcsettings, Eventvwr, Evntwin, Explorer, ExtExport, Extrac32, Fastopen, FaXcooL, Fc, FileZilla, Find, Findstr, Finger, Firefox, FlickLearningWizard, Fontview, Forfiles, FreeFileSync, FreeOTFE, Fsquirt, Fsutil, Fxscover, Gdi, Gimp, GOMPlayer, GoogleChromePortable_34.0.1847.137_online-

.paf, GoogleUpdate, GoogleUpdateSetup, Gpscript, HelpPane, Hfs, Hotspotshield, Httpfileserver, Hwrcomp, Icacls, Idmanager, Ie4uinit, IEExec, Iexplore, Iisrstas, Iissetup, Ilasm, ImagingDevices, Imjpdsvr, Imjppdmg, Imfpuex, Imjpuexc, Imscprop, Imtcprop, Inetinfo, InfDefaultInstall, Infocard, InstallUtil, Ipconfig, Irftp, Iscsicli, Iscsicpl, Isintsup, JkDefrag, JkDefragStarter, Journal, JPEGView, KeePass, Krnl386, Ktmutil, Label, Libreoffice, Lightscreen, Lpq, Lpr, Lpremove, Lsass, MagicMailMonitor, Magnify, Makecab, Man, Manycam, Mcbuilder, Mcspad, Mctadmin, Mcx2Prov, McxTask, MdRes, Mem, MigAutoPlay, MigRegDB, MigSetup, Migwiz, MineSweeper, Mobsync, Mofcomp, Mount, Mountvol, MpSigStub, Mqbkup, Mqtgsvc, Msg, Mshta, Msiexec, MuCommander, Mucommander, MuiUnattend, MxdwGc, Napstat, Narrator, Nbtstat, Ndadmin, Net1, NETFXRepair, NetProj, Newdev, Nfsadmin, Nfsclnt, Ngen, Nlsfunc, Nltest, Nslookup, Ntprint, Odbcad32, Oobeldr, Openfiles, P2phost, Pathping, Pcalua, Pcawrk, Pcwrun, Perfmon, Pidgin, Ping, Pipanel, PkgMgr, Planner, Plasrv, PnPUnattend, PnPutil, PostMig, PowerLoader, PresentationFontCache, PresentationHost, PrintBrm, PrintBrmEngine, Psxss, PurblePlace, PushPrinterConnections, Python, Python_icon, Pythonw, Qappsrv, QBittorrent, Qtorrent, Raserver, Rdpshell, Rdrleakdiag, Re, Recover, Redir, Reg, RegAsm, Regedt32, Regini, RegisterMCEApp, RegSvcs, Regsvr32, Regtlibv12, Rekeywiz, Replace, RMActivate, RMActivate_isv, RMActivate_ssp, RMActivate_ssp_isv, RmClient, Robocopy, Rqc, Rrinstaller, Rssowl, Runas, Rundll32, Rwinsta, Sapisvr, SBEServer, SciTE, Sdbinst, Sdchange, Sdiagnhost, SearchProtocolHost, SecEdit, Secinit, Services, SetupSNK, Setupsqm, Setver, Setx, Sfc, Shrpubw, Shutdown, Shvlzm, Sigverif, SMConfigInstaller, Smi2smir, SndVol, SoundRecorder, SpeechUXTutorial, SpeechUXWiz, SpiderSolitaire, Spinstall, Spoolsv, Sppsvc, StarDict, Startup Manager, StikyNot, SumatraPDF, sumatrapdf, SvcIni, SyncHost, Synkron, Sysedit, Syskey, Sysprep, Systeminfo, SystemPropertiesAdvanced, SystemPropertiesHardware, SystemPropertiesProtection, TabTip, TapiUnattend, Taskhost, Taskkill, Telnet, Thunderbird, Timeout, Tlntsvr, Tracert, Tscon, Tsdiscon, Tskill, TSTheme, Twunk_16, Twunk_32, Tzupd, UI0Detect, Umount, UniExtract, Uniextract, Unregmp2, Upnpcont, User, Userinit, VaultSysUi, VBoxDrvInst, VBoxService, VBoxTray, VBoxWHQLFake, Vds, Vdsldr, VideoLAN, Videolan, Videoservicethief, Vlc, Vssadmin, Wabmig, WatAdminSvc, Wbadmin, Wbengine, WerFaultSecure, Wextract, Wfs, Windeploy, Windirstat, Windirstat, WindowsAnytimeUpgrade, WindowsAnytimeUpgradeui, Winhlp32, Wininst-7.1, Wininst-8.0, WinMgmt, Winon, Winrarx64, Winrs, Winrshost, WinSAT, Winspool, Wksprt, Wlrmdr, WmiApSrv, WMIC, Wmlaunch, Wmpconfig, Wmpdmc, Wmpenc, Wmplayer, Wmpnetwk, WMPSideShowGadget, WOWEXEC, Wrar500, Write, WsatConfig, WSManHTTPConfig, Wsmprovhost, WSOphp, Wuapp, Wusa, X86_microsoft-windows-basic-misc-tools_6.1.7600.16385_none_expand, X86_microsoft-windows-dns-client_6.1.7601.17514_none_dnscacheugc, X86_microsoft-windows-htmlhelp_6.1.7600.16385_none_hh, X86_microsoft-windows-i..i_initiator_service_6.1.7601.17514_none_iscsicli, X86_-

microsoft-windows-international-core_6.1.7601.17514_none_muiunattend, X86_microsoft-windows-msauditevt_6.1.7600.16385_none_auditpol_83c870f4, X86_microsoft-windows-networkbridge_6.1.7600.16385_none_bridgeunattend, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_append, X86_microsoft-windows-ntvdm-system32_-6.1.7601.17514_none_csrstub, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_-none_debug, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_dosx, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_drwatson, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_edlin, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_fastopen, X86_microsoft-windows-ntvdm-system32_-6.1.7601.17514_none_gdi, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_-none_krnl386, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_mem, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_mscdexnt, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_ntvdm, X86_microsoft-windows-ntvdm-system32_6.1.7601.17514_none_setver, X86_microsoft-windows-ntvdm-system32_-6.1.7601.17514_none_user, X86_microsoft-windows-p.installerandprintui_6.1.7601.17514_-none_printui, X86_microsoft-windows-p.unterinfrastructure_6.1.7601.17514_none_-lodctr, X86_microsoft-windows-p.unterinfrastructure_6.1.7601.17514_none_unlodctr, X86_microsoft-windows-recdisc-main_6.1.7601.17514_none_recdisc, X86_microsoft-windows-session0viewer_6.1.7600.16385_none_ui0detect, X86_microsoft-windows-t.localsessionmanager_6.1.7601.17514_lsm, X86_microsoft-windows-tcpip_6.1.7601.17514_-none_netiougc, X86_microsoft-windows-winon-tools_6.1.7600.16385_none_mpnotify, X86_microsoft-windows-winon-tools_6.1.7600.16385_none_wlrmdr, X86_microsoft-windows-wmi-core-svc_6.1.7601.17514_none_winmgmt, Xcopy, Xscite, XVideoServiceThief, Xwizard, Youtubedownloader