Institut für Systemsicherheit

Technische
Universität
Braunschweig

Efficient Machine Learning for Attack Detection

Von der Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig
zur Erlangung des Grades eines Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation
von

Christian Wressnegger

geboren am 29. Juli 1984
in Graz, Österreich

Eingereicht am:	11. September 2018
Disputation am:	16. November 2018
Referent:	Prof. Dr. Konrad Rieck
Korreferent:	Prof. Dr. Thorsten Holz

Betreuungsausschluss: Prof. Dr. Konrad Rieck
Technische Universität Braunschweig
Prof. Dr. Klaus-Robert Müller
Technische Universität Berlin

Promotionsauschuss:
Referent: Prof. Dr. Konrad Rieck
Technische Universität Braunschweig

Korreferent: Prof. Dr. Thorsten Holz
Ruhr-Universität Bochum

Vorsitz: Prof. Dr. Martin Johns
Technische Universität Braunschweig

Tag der Einreichung: 11. September 2018
Tag der Disputation: 16. November 2018

Dedicated to my parents.

Abstract

Detecting and fending off attacks on computer systems is an enduring problem in computer security. In light of a plethora of different threats and the growing automation used by attackers, we are in urgent need of more advanced methods for attack detection. Manually crafting detection rules is by no means feasible at scale, and automatically generated signatures often lack context, such that they fall short in detecting slight variations of known threats.

In this thesis, we address the necessity of advanced attack detection and develop methods to detect attacks using machine learning to establish a higher degree of automation for reactive security. Machine learning is data-driven and not void of bias. For the effective application of machine learning for attack detection, thus, a periodic retraining over time is crucial. However, the training complexity of many learning-based approaches is substantial. We show that with the right data representation, efficient algorithms for mining substring statistics, and implementations based on probabilistic data structures, training the underlying model can be achieved in linear time.

In two different scenarios, we demonstrate the effectiveness of so-called language models that allow to generically portray the content and structure of attacks: On the one hand, we are learning malicious behavior of Flash-based malware using classification, and on the other hand, we detect intrusions by learning normality in industrial control networks using anomaly detection. With a data throughput of up to 580 Mbit/s during training, we do not only meet our expectations with respect to runtime but also outperform related approaches by up to an order of magnitude in detection performance. The same techniques that facilitate learning in the previous scenarios can also be used for revealing malicious content, embedded in passive file formats, such as Microsoft Office documents. As a further showcase, we additionally develop a method based on the efficient mining of substring statistics that is able to break obfuscations irrespective of the used key length, with up to 25 Mbit/s and thus, succeeds where related approaches fail.

These methods significantly improve detection performance and enable operation in linear time. In doing so, we counteract the trend of compensating increasing runtime requirements with resources. While the results are promising and the approaches provide urgently needed automation, they cannot and are not intended to replace human experts or traditional approaches, but are designed to assist and complement them.

Zusammenfassung

Die Erkennung und Abwehr von Angriffen auf Endnutzer und Netzwerke ist seit vielen Jahren ein anhaltendes Problem in der Computersicherheit. Angesichts der hohen Anzahl an unterschiedlichen Angriffsvektoren und der zunehmenden Automatisierung mit der Angriffe durchgeführt werden, bedarf es dringend moderner Methoden zur Angriffserkennung. Das manuelle Erstellen von Regeln zur Erkennung von Angriffen ist in diesem Umfang nicht mehr zu bewerkstelligen und automatische Ansätze zur Signaturgenerierung generalisieren oft nur schlecht.

In dieser Doktorarbeit werden Ansätze entwickelt, um Angriffe mit Hilfe von Methoden des maschinellen Lernens zuverlässig, aber auch effizient zu erkennen. Sie stellen der Automatisierung von Angriffen einen entsprechend hohen Grad an Automatisierung von Verteidigungsmaßnahmen entgegen. Für die zuverlässige, lernbasierte Angriffserkennung müssen die zugrundeliegenden Modelle im Laufe ihres Einsatzes periodisch neu gelernt werden. Das Trainieren solcher Methoden ist allerdings rechnerisch aufwändig und erfolgt auf sehr großen Datenmengen. Laufzeiteffiziente Lernverfahren sind also entscheidend. Wir zeigen, dass durch den Einsatz von effizienten Algorithmen zur statistischen Analyse von Zeichenketten und Implementierung auf Basis von probabilistischen Datenstrukturen, das Lernen von effektiver Angriffserkennung auch in linearer Zeit möglich ist.

Anhand von zwei unterschiedlichen Anwendungsfällen, demonstrieren wir die Effektivität von Modellen, die auf der Extraktion von sogenannten n-Grammen basieren: Zum einen, betrachten wir die Erkennung von Flash-basiertem Schadcode mittels Methoden der Klassifikation, und zum anderen, die Erkennung von Angriffen auf Industrienetzwerke bzw. SCADA-Systeme mit Hilfe von Anomaliedetektion. Dabei erzielen wir während des Trainings dieser Modelle einen Datendurchsatz von bis zu 580 Mbit/s und übertreffen gleichzeitig die Erkennungsleistung von anderen Ansätzen deutlich. Die selben Techniken, um diese lernenden Ansätze zu ermöglichen, können außerdem für die Erkennung von Schadcode verwendet werden, der in anderen (passiven) Dateiformaten eingebettet und mittels einfacher Verschlüsselungen obfuskiert wurde. Hierzu entwickeln wir eine Methode die basierend auf der statistischen Auswertung von Zeichenketten einfache Verschlüsselungen bricht. Der entwickelte Ansatz arbeitet unabhängig von der verwendeten Schlüssellänge, mit einem Datendurchsatz von bis zu 25 Mbit/s und ermöglicht so die erfolgreiche Deobfuskierung in Fällen an denen andere Ansätze scheitern.

Die erzielten Ergebnisse in Hinsicht auf Laufzeiteffizienz und Erkennungsleistung sind vielversprechend. Die vorgestellten Methoden ermöglichen die dringend nötige Automatisierung von Verteidigungsmaßnahmen, sollen den Experten oder etablierte Methoden aber nicht ersetzen, sondern diese unterstützen und ergänzen.

Acknowledgments

Above all, I would like to thank my parents who have supported me on my bumpy journey through school and all the years at university, where I have finally been able to live out my passion for computers. I consider myself very lucky.

Speaking of luck, I would like to especially thank Prof. Dr. Konrad Rieck for not brushing me off when I have first contacted him out of the blue, most naively asking for an internship or, I quote, “something”. No, I did not have funding and obviously I was not thinking this through, but I was looking for a change and I was passionate about it. Luckily, he was patient enough to listen to me and suggested that, if I was serious about it, I should apply as a PhD candidate at the upcoming call in a couple of months—so, I did. Ever since Konrad has been a great advisor and friend to me. Thank you for all the discussions, valuable suggestions and words of advise. Special thanks also goes to Prof. Dr. Klaus-Robert Müller and Dr. Sebastian Mika who enabled me a smooth start into my PhD back in Berlin.

Moreover, I would like to thank Prof. Dr. Thorsten Holz for refereeing the thesis and Prof. Dr. Martin Johns for chairing the defense. In light of your full schedules, I am extremely glad that both of you took the time to participate in the process. Thank you for an interesting discussion and all the kind words.

My most profound thanks, furthermore, goes to all my colleagues and friends who I have met along the way. I would like to emphasize that these two groups of people are not mutually exclusive. I very much appreciate the time we have spent discussing anything and everything, playing in the worst imaginable soccer team, traveling, trolling and ranting. I am not even going to try to name you all, but I am truly thankful.

Finally, I would like to gratefully acknowledge funding from the German Federal Ministry of Education and Research under the projects PROSEC (FKZ 01BY1145) and INDI (FKZ 16KIS0154K).

Publications

In the following, papers and journal articles are listed that have emerged in the course of this thesis. These include works from various fields of computer security such as *malware and intrusion detection* or the *discovery of vulnerabilities*. Publications indicated by a filled square (■) have been authored by the thesis author, those denoted by an empty square (□) originate from collaborations under the lead of other researchers.

Vulnerability Discovery

□ Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery.
F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck.
In *Proc. of the 20th ACM Conference on Computer and Communications Security (CCS)*

□ PULSAR: Stateful Black-box Fuzzing of Proprietary Network Protocols.
H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck.
In *Proc. of the 11th International Conference on Security and Privacy in Communication Networks (SECURECOMM)*

■ Twice the Bits, Twice the Fun: Vulnerabilities caused by Migration to 64-Bit Systems.
C. Wressnegger, F. Yamaguchi, A. Maier, and K. Rieck.
In *Proc. of the 23th ACM Conference on Computer and Communications Security (CCS)*

■ 64-Bit Migration Vulnerabilities.
C. Wressnegger, F. Yamaguchi, A. Maier, and K. Rieck.
In *Information Technology (IT)*, 59 (2)

Malware and Intrusion Detection

■ Deobfuscating Embedded Malware using Probable Plaintexts.
C. Wressnegger, F. Boldewin, and K. Rieck.
In *Proc. of the 15th Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*

■ A Close Look on n-Grams in Intrusion Detection: Anomaly Detection vs. Classification. C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck.
In *Proc. of the 6th ACM Workshop on Artificial Intelligence and Security (AISEC)*

■ Comprehensive Analysis and Detection of Flash-based Malware.
C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck.
In *Proc. of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* **Best-Paper Award**

■ Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks.
C. Wressnegger, K. Freeman, F. Yamaguchi, and K. Rieck.
In *Proc. of the 12th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*

■ Looking Back on Three Years of Flash-based Malware.
C. Wressnegger and K. Rieck.
In *Proc. of the 10th ACM European Workshop on Systems Security (EuroSec)*

■ ZOE: Content-based Anomaly Detection for Industrial Control System.
C. Wressnegger, A. Kellner, and K. Rieck.
In *Proc. of the 48th Conference on Dependable Systems and Networks (DSN)*

Adversarial Learning

☐ Poisoning Behavioral Malware Clustering.
B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Roli.
In *Proc. of the 7th ACM Workshop on Artificial Intelligence and Security (AISEC)*

String Processing for Machine Learning

☐ Sally: A Tool for Embedding Strings in Vector Spaces.
K. Rieck, C. Wressnegger, and A. Bikadorov.
Journal of Machine Learning Research (JMLR)

☐ Harry: A Tool for Measuring String Similarity.
K. Rieck and C. Wressnegger.
Journal of Machine Learning Research (JMLR)

Privacy

☐ Privacy Threats through Ultrasonic Side Channels on Mobile Devices.
D. Arp, E. Quiring, C. Wressnegger, and K. Rieck.
In *Proc. of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*

List of Figures

1.1 Schematic depiction of two learning schemes for attack detection. 3
1.2 Exemplary ROC curve with the bounded AUC (area under the curve). 6
1.3 Overview depiction of approaches for attack detection. 8

2.1 Static malware signatures derived from anti-virus products. 20
2.2 Statistics for the derived byte-pattern signatures. 22

3.1 Examples for n-grams based on bytes and tokens. 26
3.2 The prefix tree and radix tree representing n-grams of the word bananas. . 29
3.3 The suffix tree and the corresponding suffix array for the string banana . . 30
3.4 Schematic depiction of a hash table. 31
3.5 Schematic depiction of a Bloom filter. 32
3.6 Schematic depiction of a Count-Min sketch. 33
3.7 Data suitability criteria on example of various datasets. 35
3.8 Data suitability criteria applied to the binary-based protocols 35
3.9 Influence of the saturation of a hash function's output range. 39
3.10 Histograms of 3-gram occurrences in network traffic for exact and probabilistic counting, and the difference between both approaches. 40

4.1 Schematic depiction of the analysis and detection process of Gordon. . . 43
4.2 Excerpt of the Gordon's structural report for a LadyBoyle malware sample. 45
4.3 Illustration of the path-selection strategy 47
4.4 An ActionScript 3 code snippet, the corresponding control flow graph (CFG) and its acyclic transformation (ACFG) 48
4.5 Excerpt of Gordon's behavioral report. 49
4.6 Detection performance of Gordon-SVM as ROC curve. 55
4.7 Detection performance of Gordon-SVM and Gordon-TCA as ROC curves. 56
4.8 Throughput and runtime of Gordon-TCA and Gordon-SVM during training. 57
4.9 Gordon's performance over 12 consecutive weeks 58

5.1 Schematic depiction of Zoe. 65
5.2 Rule inference for a set of simple messages. 70
5.3 Schematic depiction of the generation of network attacks. 71
5.4 Detection performance of Zoe for the protocols P1 to P6. 74
5.5 Detection performance of Zoe$(1,t)$ for varying thresholds t. 75
5.6 Detection performance of Zoe$(1,35)$ and Zoe$(4,50)$. 75
5.7 Throughput of Zoe during training. 76

5.8 Detection performance of Zoe and Anagram side-by-side as ROC curves. . . 77
5.9 Polymorphic blending attacks against Zoe. 78

6.1 Examples for Vigenère-based obfuscation. 84
6.2 Schematic depiction of Kandi. 86
6.3 Byte frequency distributions of English text and Windows PE files. 89
6.4 Deobfuscation performance of Kandi on obfuscated PE files. 95
6.5 Runtime performance of Kandi in comparison to a brute-force attack. . . 95
6.6 Deobfuscation performance of Kandi on Word documents containing obfuscated PE files. 96
6.7 Distribution of key lengths detected by Kandi, and the number of anti-virus scanners detecting the extracted malware binaries. 99

List of Tables

2.1 Assignments using basic types on Win32/Linux32, Win64 and Linux64. . . 15
2.2 Comparisons using basic integer types on Win32/Linux32, Win64 and Linux64. 15
2.3 Number of implicit type conversions per package on 64 bit. 17
2.4 Number of specific patterns for 64-bit migration issues in source packages of Debian stable and 200 popular C/C++ projects hosted on GitHub. 17
2.5 Results of signature derivation. 21
2.6 Detection results on pattern-based signatures implanted in benign PE files. 23

3.1 Description of the datasets used for evaluating our data suitability criteria. 36

4.1 Overview of the evaluation dataset. 54
4.2 Detection rates of FlashDetect, Gordon-TCA, and Gordon-SVM. 55

5.1 Dataset and protocols used for the evaluation of Zoe. 73
5.2 Detection performance of Zoe and Anagram for protocol P3. 77

6.1 Operators of Vigenère ciphers used for obfuscation. 85
6.2 Depiction of LCP delimited arrays based on the example of the string banana. 88
6.3 Overview of the four datasets of malicious documents and images. 97
6.4 Deobfuscation performance of Kandi on real-world malware 98

A.1 Widths of basic integer types in bytes for different data models and exemplary operating systems using them (Lauer, 1996; Mashey, 1996). 110

Contents

1 Introduction 1
- 1.1 Attack Detection using Machine Learning . . . 2
- 1.2 Efficient String Processing . . . 6
- 1.3 Thesis Contributions . . . 8
- 1.4 Thesis Overview . . . 10

2 Necessity of Advanced Attack Detection 11
- 2.1 Software Vulnerabilities . . . 11
- 2.2 Insufficiency of Traditional Methods . . . 19

3 Data Representation 25
- 3.1 Language Models . . . 25
- 3.2 Suitability for Learning . . . 27
- 3.3 Time and Space-efficient Storage . . . 28
- 3.4 Evaluation . . . 34

4 Learning Malicious Behavior 41
- 4.1 Use Case: Detecting Flash-based Malware . . . 41
- 4.2 System Overview . . . 43
- 4.3 Data Analysis . . . 44
- 4.4 Classification . . . 50
- 4.5 Evaluation . . . 53
- 4.6 Related Work . . . 59

5 Learning Normality 63
- 5.1 Use case: Detecting Intrusions in SCADA Systems . . . 63
- 5.2 System Overview . . . 65
- 5.3 Building Prototype Models . . . 66
- 5.4 Anomaly Detection . . . 68
- 5.5 Evaluation . . . 71
- 5.6 Related Work . . . 79

6 Revealing Malicious Content **83**
6.1 Use Case: Detecting Malware in Office Documents 83
6.2 System Overview 86
6.3 Determining Frequent Substrings 87
6.4 Deriving the Key Length 88
6.5 Breaking the Obfuscation 91
6.6 Evaluation 93
6.7 Related Work 99

7 Conclusion and Outlook **103**
7.1 Summary of the Results 104
7.2 Future Work 106

A Supplementary Information **109**
A.1 Data models 109

Bibliography **111**

Index **131**

Chapter 1

Introduction

Computer security faces a constant and daily growth of new threats on the Internet. The attack vectors, types and ramifications of these threats are manifold. On the client side, malware infections pose a risk to the security of individual hosts and connected networks. Numerous types of malware are used for infecting computers at a large scale, for instance, to distribute spam messages or steal personal data (see Franklin et al., 2007; Caballero et al., 2011), blackmail users (Young and Yung, 1996, 2017), or even hijack resources for mining crypto-currencies (Huang et al., 2014; Konoth et al., 2018). On the server side, a plethora of attacks target network services, which range from web-based injection attacks, such as cross-site scripting (Stock et al., 2015) and cross-site request forgery (Schreiber, 2004; Pellegrino et al., 2017), to classic exploits against vulnerable implementations. In recent years, also critical infrastructures and industrial control systems have increasingly become target of such attacks (e.g., Karnouskos, 2011; Kaspersky Lab, 2015; Cherepanov, 2017).

In the past, it may have been sufficient to manually craft detection rules for such threats. Nowadays, however, manual analysis fails to keep pace with attack development such that more automation is needed to handle the overwhelming amount of novel threats in a timely manner. Moreover, traditional methods for attack detection, such as static signature matching, often do not provide sufficient protection. Unknown threats for which no signatures exist remain undetected. This problem is further aggravated by the frequent use of obfuscation that obstructs static signature matching (Linn and Debray, 2003; Moser et al., 2007a) and the fact that such signatures often match artifacts, which are unrelated to the semantics of the considered attack or malware.

In response, alternative lines of research have been explored for attack detection. Most notably, machine learning has been used for predicting (Shen et al., 2018) and preventing network intrusions (Wang et al., 2006a; Du et al., 2017), reverse-engineering malware (Chua et al., 2017; Katz et al., 2018), and detecting malicious codes (Arp et al., 2014; Huang and Stokes, 2016; Jordaney et al., 2017). In an attempt to further push the limits in terms of detection performance and the number of false alarms, more and more complex learning methods have been employed. At testing time, that is, during operation once an adequate model has been trained, many methods boil down to a simple vector multiplication and thus are comparable fast. Training complexity, on the other hand, often is substantial. Many fields of application in attack detection however require periodic retraining over time, such that the runtime performance of the training phase is equally crucial in practice.

In this thesis, we explore the use of efficient machine learning for attack detection and strive for approaches that are both, effective and runtime efficient. We particularly focus on the training complexity of learning-based detectors and introduce methods, that are able to keep the underlying model up-to-date in linear time. In doing so, we counteract the trend of compensating increasing runtime requirements with resources. We introduce the use of efficient string processing to extract features for machine learning and develop implementations based on probabilistic data structures for storing them. This forms the basis for linear-time learning algorithms to identify malware or detect network intrusions. At this, we look at two learning schemes, and discuss how and when these are applicable for our means. In particular, we design methods for detecting Flash-based malware using classification, and for detecting intrusion in SCADA systems using anomaly detection. Moreover, we demonstrate that even substring statistics alone can be used to effectively uncover and detect attacks in practice. As an example, we additionally develop an approach for detecting malware in office documents in linear time.

In the remainder of this chapter, we provide a brief overview of attack detection using machine learning with a focus on the two prevalent learning schemes: classification and anomaly detection. Moreover, we address our evaluation methodology and look upon fundamentals of efficient string processing in view of attack detection in general and for using it to efficiently gather features for machine learning in particular. Finally, we highlight our contributions and provide an overview of the thesis.

1.1 Attack Detection using Machine Learning

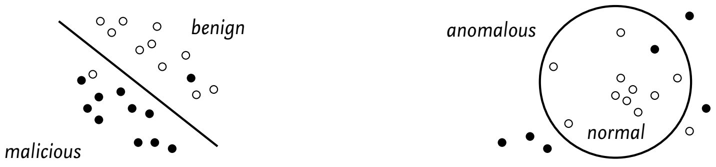
Machine learning has a long-standing history in computer security and in intrusion detection in particular, that dates back to the 1990s (Forrest et al., 1996; Lane and Brodley, 1998; Lee and Stolfo, 1998; Lane and Brodley, 1999; Sinclair et al., 1999; Lee et al., 1999; Warrender et al., 1999). Over the last two decades these approaches have steadily improved and have been applied in different contexts, including SCADA systems and industrial networks (Feng et al., 2017). Ever since the advent and growing dissemination of malware, machine learning has also been used to automate detection on desktop systems (Bayer et al., 2009; Huang and Stokes, 2016), mobile devices (Arp et al., 2014; Olejnik et al., 2017), and web pages (Canali et al., 2011; Kapravelos et al., 2013).

A number of different learning methods have emerged in this scope, all of which have unique advantages and disadvantages for specific fields of application. For instance, clustering methods (see Hastie et al., 2009, Chp. 14) may be employed for malware triage to establish a preselection and grouping of data, whereas methods for dimensionality reduction, such as Principal Component Analysis (Jolliffe, 1986), may be used to automatically learn protocols used in malware communication. For attack detection, however, two learning schemes are prevalent: classification and anomaly detection. In Chapters 4 and 5, we use these to develop methods for detecting malware and network intrusions, respectively.

As depicted in Figure 1.1, classification allows to discriminate one class from another, while anomaly detection models a single class and makes a distinction to everything else. At first sight, the difference between both appears to be marginal. From a machine learning point of view these however are fundamentally different. The first corresponds to the field of supervised learning, meaning that the method trains on labeled samples from all classes, that are taken into account for classification. Anomaly detection in turn is commonly considered an unsupervised method as no labels or only labels for one class exist. Moreover, both learning schemes offer their very own advantages, if used in the right setting. As a rule of thumb one may formulate the following prerequisites for learning-based attack detection:

- **Classification.** If enough representative, labeled data is available for both classes, this scheme allows to learn a model for discriminating one class from another. Depending on the type of attacks, this may generalize to unknown attacks but is not guaranteed to do so.
- **Anomaly Detection.** If only one class is available for training, anomaly detection may allow for detecting unknown attacks as deviations from a notion of normality. However, a careful design of the detection system is crucial, as anomalies do not per se correspond to attacks—this is commonly known as the semantic gap.

Due to the importance of these learning schemes for the methods developed later on, subsequently, we describe both in more detail in Sections 1.1.1 and 1.1.2. In Section 1.1.3, we then additionally outline the evaluation methodology used in this thesis for learning under these two schemes.



(a) Classification (b) Anomaly detection

Figure 1.1: Schematic depiction of the two prevalent learning schemes for attack detection. Figure taken from Wressnegger et al. (2013b).

1.1.1 Classification

In computer security often very strict definitions are in demand for deciding about something being benign or malicious, which immediately suggests a classification task. The identification of the two classes is achieved by learning a discrimination as illustrated in Figure 1.1a. Several learning methods, such as decision trees, (deep) neural networks and boosting can be used for learning such a classification (Duda et al., 2000). An intuitive example is the two-class Support Vector Machine (SVM), that learns a hyperplane separating two classes with maximum margin in a feature space (Schölkopf and Smola, 2002). This requires to solve a dual optimization problem which in the general (non-linear) case is rather costly in terms of runtime. A linear SVM may however achieve comparable results in significantly less time (Fan et al., 2008).

Learning such a classification requires enough data of both classes in order to be able to generalize to unseen samples. If one class is represented by a few instances only, it is likely that the model overfits and thereby impedes detection of unknown attacks. In this regard a lack of data for one class is already a crucial factor for abstaining from using classification. In some cases of attack detection, sufficient data for both classes can be acquired automatically. As an example, for learning a client-side detection of web-based attacks, it is possible to actively visit benign and malicious web pages using honeyclients (Wang et al., 2006b) and specialized crawlers (Invernizzi et al., 2012). Such crawling enables assembling a recent collection of training data for both classes. VirusTotal (Google Inc., 2004–2018), for instance, maintains a large collection of malware as well as benign software originating distributed acquisitions of data, which we have used in Chapter 4 for classifying Flash-based malware. In other settings, such as server-side detection of web-based attacks or network intrusion, one is restricted to passively observing attacks as they happen and record them, for instance, using network honeypots (Provos and Holz, 2007). As a consequence, it is disproportionately more difficult to put together a representative set of server-side attacks in a timely manner and classification methods should not be employed.

1.1.2 Anomaly Detection

Detecting unknown attacks is of critical importance in computer security, as these may relate to zero-day exploits or new instances of known malware. Fortunately, it is possible to take this scenario into account using anomaly detection—even if no attacks are available for training the detector at all. By focusing on the prominent class and learning its structure, it is possible to differentiate that class from everything else, as illustrated in Figure 1.1b. In literature this frequently is also called outlier detection (Aggarwal, 2013). Several methods are suitable for learning such a model of normality, for example, by analyzing the density, probability or boundary of the given class (Duda et al., 2000). A common approach for anomaly detection are so-called Support Vector Data Descriptions (SVDDs), that determine a hypersphere enclosing the data with minimum volume (Shawe-Taylor and Cristianini, 2004). These include spherical one-class SVMs (Schölkopf et al., 2001) as well as simpler models, as used by PayL (Wang and Stolfo, 2004) and Anagram (Wang et al., 2006a). In Chapter 5 we demonstrate how an extension of this concept can be used to detect attacks in proprietary binary protocols.

At this point it is important to stress that methods using anomaly detection do not explicitly learn to discriminate benign traffic from attacks, but instead normality from anomalies. This semantic gap requires designing features and detection systems with special care, as otherwise identified anomalies may not reflect malicious activity (Gates and Taylor, 2006; Sommer and Paxson, 2010). Two things need to be particularly taken in consideration here: First, the notion of normality may change over time, such that anomaly detectors need to be periodically updated to close the semantic gap between anomalies and attacks (Maxion and Tan, 2000). Second, to avoid incorporating attacks in the model of normality, it is necessary to sanitize the training data before learning the model (Cretu et al., 2008). The latter complicates the use of methods for online-learning, that is, gradually updating the currently best model with new data over time, and forces these to revert to updates in large batches. Runtime efficient training thus is essential for anomaly detection in practice. Nonetheless, anomaly detection is the learning scheme of choice, if little or no data from the attack class is available, as for example, when learning a server-side detection of attacks.

1.1.3 Evaluation Methodology

As mentioned before, carefully adjusting learning-based detectors is essential for effective attack detection. In the course of this thesis, we evaluate the performance of the developed detectors with the aid of *receiver operator characteristics (ROC)* and corresponding ROC curves. These curves plot the true-positive rate (TPR) over the false-positive rate (FPR) of individual detectors or a single detector with different classification thresholds (Bradley, 1997; Fawcett, 2006). The number of samples correctly identified as malicious are referred to as *true positives* and the ratio of them to the overall number of malicious samples as true-positive rate. *False positives* are benign samples that are wrongly detected as malicious—in other words, false alarms. The number of true positives, either given as percentage or rate in the interval $[0, 1]$, is always specified with reference to a specific false-positive rate. Without this relation the true-positive rate does not carry any weight at all: A detector that flags everything as malicious by definition yields a true-positive rate of 100 %. However, it also provokes false alarms for every single benign sample.

Moreover, we use the *area under the ROC curve (AUC)* as an additional continuous measure for the overall detection performance of our detectors, that yields a minimal and maximal value of 0 and 1, respectively. The AUC can hence be interpreted as a measure of how steep the curve increases towards higher true-positive rates. Figure 1.2 illustrates a ROC curve that shows the full scale of false positives as rate in the interval $[0, 1]$ on the x-axis or 0 % to 100 %, respectively. As measuring the AUC for the full range is of little expressiveness (performances for low false positives are poorly represented) we use the *bounded AUC*, that is, the area under the ROC curve up to a threshold b of false positives and normalized to that value: $\text{AUC}(b)$. For the field of attack detection it is particularly important to push forward detection with few false positives.

For training and testing individual detectors we create strictly separated datasets that do not overlap. 75 % of the samples are used as *known data* for training and the remaining 25 %

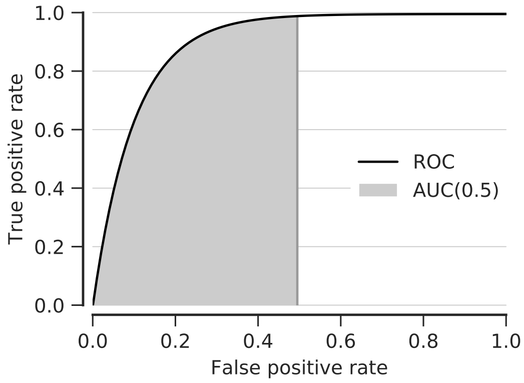


Figure 1.2: Exemplary ROC curve with the bounded AUC (area under the curve) for a false-positive rate of 0.5 or 50 %, respectively. Setting this bound allows to zoom in on the critical range of low false positives that is particularly important for a detector. Figure taken from Wressnegger et al. (2018).

as *unknown data* for testing. This partitioning is applied to benign and malicious samples likewise, and is repeated for 10 experiments, which are averaged to determine the overall detection performance. The parameters for the detector are chosen solely based on the results obtained on training data, where one third of it (25 % of the overall data) is used as so-called validation dataset to compute the expected detection performance. Only then, the configuration that performed best during training is used to evaluate the detection performance on the testing dataset. For use cases where data is gathered with respect to a specific timeline, the datasets are split in strict chronological order. For example, in Chapter 4 we evaluate our detectors based on data collected during 12 consecutive weeks, where *weeks 1–6* are used for training, *weeks 7–9* for validation and *weeks 10–12* for testing the detection performance. If there is no such relation to the point in time the data has been recorded, as for instance in Chapter 5, we choose the samples for training and testing datasets randomly, while still ensuring disjoint subsets.

1.2 Efficient String Processing

Efficiently processing large amounts of string-based data is essential for attack detection, both with and without machine learning. Here, strings generally refer to byte strings of arbitrary data. Conventional approaches for intrusion detection, for instance, require highly specialized algorithms for simultaneously matching multiple signatures against a constant stream of network traffic. This ranges from plain string matching (Aho and Corasick, 1975; Karp and Rabin, 1987) to full blown processing of regular-expressions (Thompson, 1968; Navarro and Raffinot, 1999) and is equally applied for malware detection. For machine learning, by contrast, the efficient processing of strings is crucial for translating input data into a representation that learning algorithms can work with.

Many algorithms for efficiently processing string build on specific data structures to facilitate the execution. As an example, Aho and Corasick (1975) use an annotated trie or prefix tree as automaton for multi-pattern matching. Others incorporate probabilistic data structures, such as Bloom filters (Bloom, 1970), to speed up exact string searching using the Rabin-Karp algorithm (Moraru and Andersen, 2012). Bloom filters have been very popular in various database and networking applications (Broder and Mitzenmacher, 2005). These can be thought as hash sets that use multiple hash functions per item and accept a certain probability of error for membership queries. Moreover, an item may falsely be reported as part of the set, but an actually inserted item is never overlooked or missed. In Chapter 3 we look at this data structure in more detail as a foundation for Chapter 4, where we use Bloom filters for classifying Flash-based malware. Efficiently storing sets of strings or even the number of their occurrences is essential for many applications. The latter however is not possible with Bloom filters, such that extensions have been developed, at first, to make up for the lacking possibility of deleting items by additionally tracking a limited range of counts (Fan et al., 2000), and then to associate arbitrary numeric values to probabilistically stored strings in so called Count-Min sketches (Cormode and Muthukrishnan, 2005, 2012). We employ these in Chapter 5 to mine large amounts of substrings for detecting attacks in proprietary protocols of industrial control systems and use the counts for pruning rare features of the detection model.

From a computational point of view, finding suitable substrings that can be used as signatures for traditional detection methods is equally challenging as it requires to find and count all common substrings across a large number of documents, such as a collection of malware samples. Fortunately, such statistics can be efficiently collected using suffix trees (Weiner, 1973) or suffix arrays (Manber and Myers, 1990) in linear or quasi-linear time. Yamamoto and Church (2001) show how to compute the *term frequency*, that is, the number of occurrences of a specific substring in the entire corpus, as well as the *document frequency*, that is, the number of documents, that contain a particular substring, in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. These statistics cannot only be used for identifying signatures that cover a large number of malware samples, but also for collecting evidence for breaking simple encryption schemes. In Chapter 6 we show how to statically deobfuscate malware that uses the Vigenère cipher to evade detection by traditional methods.

In machine learning, broadly speaking, many algorithms compute distances between data points or try to establish a decision boundary that maximizes the distance of data points of opposing classes. These distances or similarities can be computed on strings directly, or on strings that have been embedded in vector spaces. The edit distance, for instance, operates on byte strings and defines how dissimilar two instances are by measuring the number of deleted, inserted, and substituted bytes in either string (Navarro, 2001). While many implementations of this class of distances exist, such as the Hamming distance that only allows substitution, or the longest common subsequence distance that by contrast does not allow substitution, the most generic and probably most common variant is the Levenshtein distance (Levenshtein, 1965, 1966). It defines penalty costs for all three operations, where these often are set to 1 for the sake of simplicity. Alternatively, one may use similarity measures such as the Jaccard coefficient that generically operates on arbitrary sets (Levandowsky and Winter, 1971), in this particular context, on sets of characters or bytes. This already is very close to string-based embeddings that map strings to vector spaces, for instance using language models, which we use in this thesis. We discuss this particular representation in Chapter 3 in more detail.

Efficient implementations of a variety of distances, similarity measures and embeddings are provided by open-source tools available since Harry (Rieck and Wressnegger, 2016) met Sally (Rieck et al., 2012).

1.3 Thesis Contributions

In this thesis, we strive for improving attack detection using efficient methods from machine learning. In principle there exist two basic approaches for detecting attacks, which are complementary to each other: On the one hand, proactive protection by means of developing secure systems, and hardening them by finding and eliminating software vulnerabilities. On the other hand, reactive approaches that detect attacks as they happen and perform analysis in reaction to these. Examples are the detection of malware or intrusions on the network, which we focus on in this thesis. In contrast to traditional approaches, such as virus scanners or signature based intrusion detection systems, we use techniques from machine learning (classification and anomaly detection) and efficient string processing to push forward attack detection. Figure 1.3 depicts the relations between these fields of application. White boxes illustrate approaches contributed in this thesis.

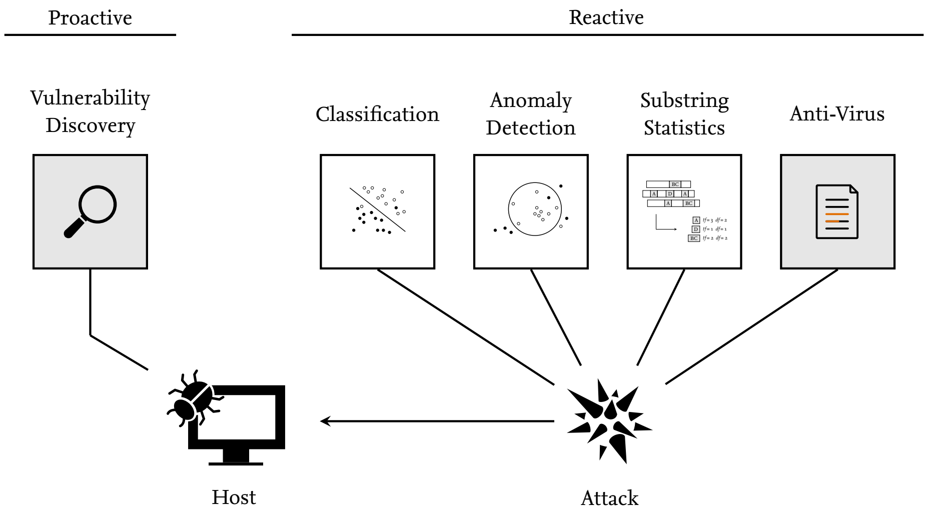


Figure 1.3: Overview depiction of approaches for attack detection.

In summary we make the following contributions:

- **Necessity of Advanced Attack Detection.** We study two key issues in today's computer security landscape that call for the use of advanced malware and attack detection in practice: First, *software vulnerabilities* as a stepping stone for attackers, and second, the lack of timely protection against such attacks. As an example, we review 64-bit migration vulnerabilities and quantify their prevalence. Moreover, we evaluate the current state of consumer malware detection and the effectiveness of signature-based malware detection as frequently deployed by *anti-virus scanners* (Chapter 2).

- **Data Suitability for Machine Learning.** We introduce representations of attack and malware data, that are suitable for the efficient use with machine learning algorithms. We present *language models* for embedding data into vector spaces, before we study the suitability of these for a detection task at hand. In particular, we develop criteria that allow to gauge whether a specific representation is more suitable for either classification or anomaly detection. Additionally, we present time and space-efficient data structures for efficiently processing string-based data, such as features from language models, to facilitate linear-time attack detection (Chapter 3).

- **Learning Malware Detection.** We demonstrate the feasibility of the timely detection of malware in practice. To this end, we combine an in-depth analysis of malware using static and dynamic analysis with efficient machine learning methods. While previous research has proven that two orthogonal analyses are beneficial from an evasion point of view, we show its effectiveness for detection tasks where neither one nor the other is sufficient on its own. We present novel learning-based *classification* approaches using Bloom filters and SVMs, and further discuss how the latter can additionally be facilitated with Count-Min sketches in practice (Chapter 4).

- **Learning Proprietary Binary Protocols.** We show that, in contrast to prior belief, language models can very well be used for detecting attacks in arbitrary binary network protocols. For this, we employ *anomaly detection* to learn a model of normality that can be used to uncover attacks. We propose the use of space-efficient counting of string-based features using Count-Min sketches to cluster data in linear time and, in further follow, to prune out noise from these models. In combination this outperforms related work by an order of magnitude in detection performance as well as the number of false positives (Chapter 5).

- **Deobfuscating Embedded Malware.** We present a technique to locate, extract, and deobfuscate malware binaries hidden in third-party containers. Without the need for identifying the container's format or involved parsing, we statically remove malware obfuscation based on simple encryption schemes. This is made possible by efficiently computing *substring statistics* and the use of probabilistic data structures to store evidence collected in the process. With that, we deobfuscate malware in linear time—irrespective of the length of the used encryption key (Chapter 6).

1.4 Thesis Overview

In the remainder of this thesis, we discuss specific aspects of reactive attack detection using efficient string processing, machine learning, and the combination thereof. In the two subsequent chapters, we first motivate the necessity of this line of research, before we lay the technical foundations for the methods described in this thesis. The latter is particularly important for understanding the implementation aspects of our work. The following three chapters connected by increasing sophistication of the used data structures and algorithms for string processing as well as the underlying motivation. The last chapter concludes and suggests future work.

Chapter 2 motivates the necessity of new, more advanced approaches for the task of malware detection based on two central issues. First, we look upon a prevalent, long-lasting form of software flaws that may be a building block for attacks and second, we inspect the effectiveness and robustness of traditional methods as employed in today's security solutions.

Chapter 3 introduces language models, which we use throughout the thesis for attack detection, and develops suitability criteria for this specific representation of data. We provide practitioners with concrete measures for choosing a classification setting over anomaly detection or the other way around. Moreover, in this chapter we discuss probabilistic data structures that can be used to efficiently store and process string-based data.

Chapter 4 describes a malware detector based on classification using language models. We discuss structural and behavioral representations of Flash-based malware up to their embedding in vector spaces. Finally, we look at two approaches for learning a classification: One based on differences of substring statistics and another, more complex one, using Support Vector Machines (SVMs). Both are compared to related approaches.

Chapter 5 discusses the use of anomaly detection for reliably detecting attacks in industrial networks, where we particularly focus on binary-based protocols. We present a method based on probabilistic data structures that first, clusters network data and second, enables to on-the-fly prune rare features and generate robust models of normality. Finally, we evaluate our approach and show that we improve the state-of-the art.

Chapter 6 demonstrates methods to extract obfuscated malware hidden in third-party containers. While dynamic approaches may achieve the same result, we show that by narrowing down the field of application it is possible to reveal embedded malware statically and agnostic to the container format. We discuss the efficient implementation of the approach using substring statistics and make an comparison to related approaches.

Chapter 7 concludes the thesis and provides an outlook to future work. Supplementary information can be found in **Appendix A.**

Chapter 2

Necessity of Advanced Attack Detection

Two factors are crucial for the success of malware and targeted attacks: First, the availability of an attack vector that serves as gateway to a victim's system. For this, a malware or an attacker may gain unprivileged access to a computer system in a number of ways. This ranges from social engineering over software vulnerabilities to hardware flaws that effect the entire system. Second, traditional methods for attack detection often fall short in protecting these systems. With the growing number of attacks reacting in a timely manner has become a significant issue. Addressing both factors is key for effectively and efficiently circumventing attacks in practice.

In this chapter, we thus look into two instances of these issues. We begin with a systematic study on a specific kind of software vulnerabilities in Section 2.1, in order to convey a feeling for the prevalence of programming flaws in popular software and the potential of them being used as attack vector. In particular, we focus on 64-bit migration vulnerabilities, that is, flaws introduced when software is ported from 32-bit to 64-bit platforms (Wressnegger et al., 2016b, 2017b). These defects are introduced indirectly and are difficult to spot by the developer without anticipating a later migration. In Section 2.2, we then inspect the effectivity of static signature matching as—even nowadays—frequently employed by many anti-virus scanners (Wressnegger et al., 2017a). By analyzing this particular detection scheme and its implementation, we unveil its restrictions and insufficiency for modern malware detection.

These two aspects highlight separate strains of research for protecting computer systems that are complementary to each other: (1) Vulnerability discovery in software, in order to proactively get down to the root of the problem, and (2) advanced attack detection, to contain and control the effects of attacks and malware reactively.

2.1 Software Vulnerabilities

While targeting the human element has been shown to be effective in the past (e.g., Krombholz et al., 2015; Nelms et al., 2016), a certain level automatization of an attack can only be achieved by leaving external dependencies out to a large extent. To this end, adversaries make recourse to software vulnerabilities to unobservedly enter a system, gain privileges, and/or execute malicious codes. In practice, complex campaigns and targeted attacks, as for instance the *Stuxnet* attack that has targeted industrial control systems (Falliere et al., 2011), use a combination of software vulnerabilities and the unintentional aid of people to access air-gapped systems.

In this section, we focus on vulnerabilities that occur in system software in particular. One of the most prominent classes of vulnerabilities in this scope are integer-based vulnerabilities. Subtle flaws in integer computations may in further consequence enable to alter the program flow, cause the denial of service (DoS) of an entire system, or even allow remote code-execution. Researching causes and countermeasures for this fundamental kind of vulnerabilities has a long standing history in computer security research (Ashcraft and Engler, 2002; Brumley et al., 2007; Dietz et al., 2012) so that one may assume that such flaws are very well contained and accordingly rare nowadays. Unfortunately, this is not the case. The difference in the widths of data types and the rules for converting from one type to another are still gotten wrong by many developers. With the growing dissemination of 64-bit computing in consumer systems, integer-based vulnerabilities have taken on even greater significance again. The change in data models and the developer's unaltered expectations with respect to the width of data types give rise to previously dormant vulnerabilities that become exploitable on the new architecture.

Let us, for instance, consider the following simplified excerpt of a flaw that we have discovered during our study in *zlib* version 1.2.8:

```
1   int len = attacker_controlled();
2   char *buffer = malloc((unsigned) len);
3   memcpy(buffer, src, len);
```

This code is perfectly secure on all 32-bit platforms, as the variable `len` is implicitly cast to `size_t` in line 2 and 3 when passed to `malloc` and `memcpy`. However, if the code is compiled using 64-bit Linux, line 2 and 3 produce different results, where a 64-bit sign extension is performed in line 3. An attacker controlling the variable `len` can thus overflow the buffer by providing a negative number. For instance, -1 is converted to `0x00000000ffffffff` in line 2 and `0xffffffffffffffff` in line 3, resulting in a buffer overflow.

While modern compilers are able to raise warnings in many situations, various corner cases still remain difficult to detect automatically. This is further aggravated by the fact that, for performance reasons, system software often relies on implicit conversion rules, which elsewhere are considered as potential or even likely programming flaws. In order to avoid excessive amounts of false alarms in such situations, compilers often adopt a conservative position with respect to warnings (López-Ibáñez and Taylor, 2006). As a result, 64-bit issues are rather the rule than the exception in migrated code and there exist several examples of vulnerabilities solely induced by migration. Subsequently, we study the prevalence of this particular kind of vulnerabilities in order to convey a feeling for the severity of software flaws as an attack vector.

2.1.1 64-bit Migration Vulnerabilities

All types of integers available on 32-bit platforms also exist in 64-bit platforms, however, their width may differ. This discrepancy gives rise to new vulnerabilities or enables to trigger vulnerabilities that lie dormant in existing code. The width and signedness of integer types is defined by the data model which in turn is specific to the respective platform. The 32-bit variants of Windows and Linux, for instance, both use the ILP32 data model, where type `int`, `long` and pointers are 32-bit wide. The 64-bit versions of these operating systems differ, though: Windows uses LLP64 (type `long long` and pointers are 64-bit wide), while Linux, BSD, and MacOS use LP64. Further details on the particular sizes of integers under different data models and platforms can be found in Appendix A.1.

In the following, we briefly characterize five different types of vulnerabilities that emerge when compiling code for a 64-bit data model that securely runs on 32-bit platforms. These vulnerabilities can be categorized by two generic sources of defects: changes in the width of integers and the larger address space available on 64-bit systems.

Effects of Integer Width Changes. The change in data models introduces truncations and sign extensions in assignments, that previously have not existed. Surprisingly, the migration to 64 bit may even flip the signedness of comparisons and render checks for buffer overflows ineffective:

1. New Truncations A truncation occurs when an expression is assigned to a type narrower than that of the expression itself. Table 2.1 provides an overview of integer issues caused by assignments, broken down by basic integer types. For each of the three prevalent data models, truncations are marked by a filled circle (●). Particularly noteworthy are those assignments that behave differently between the platforms, such as conversions from `size_t` to `unsigned int` or `long` to `int`. In these cases, new truncations occur that are specific to the migration process from 32-bit to 64-bit data models.

2. New Signedness Issues Additionally, two types of integer signedness issues arise as code is ported from 32-bit to 64-bit platforms. First, sign extensions may occur as signed integers are converted to unsigned types that have become wider than their ILP32 equivalents. Table 2.1 indicates signedness errors in assignments as gray circles (●), where sign extensions are additionally marked (Ⓔ). For the LLP64 data model, new sign extensions occur for conversions from `int` and `long` to `size_t`, and for LP64 from `int` to `unsigned long` and `size_t`. From a security perspective, conversions to the `size_t` type appear to be especially fruitful when looking for vulnerabilities. The example of the *zlib* vulnerability presented in the introduction illustrates this issue in a real-world scenario.

Second, the signedness of comparisons potentially changes, so that checks to protect from buffer overflows may become ineffective. Such checks are only effective if they correctly account for the signedness of the integers involved. Typically, this means that all integers should be converted to unsigned types prior to comparison. In many cases, explicit conversions can be omitted on 32-bit systems as integer conversion rules ensure that the comparisons will be performed unsigned. This, however, is not guaranteed on 64-bit platforms anymore, bringing forth comparisons that change their signedness when being ported. Table 2.2 provides an overview of the signedness of comparisons for basic integer types and different data models. Unsigned comparisons are marked as filled circles (●) while signed comparisons are indicated by empty circles (○). Of particular interest are those cases where the indicators change between 32-bit and 64-bit data models. For instance, a comparison involving `long` and `unsigned int` is unsigned on both, ILP32 and LLP64, but signed on the LP64 data model.

Effects of a Larger Address Space. In addition to flaws that result from changes in integer widths, code running on 64-bit platforms has to be able to deal with larger amounts of memory as the size of the address space has increased from 4 Gigabytes to several hundreds of Terabytes. In effect, the developer can no longer assume that buffers larger than 4 Gigabytes *cannot* exist in memory. As a result, additional integer truncations and overflows emerge, which do exist on 32-bit data model, but cannot be triggered on the corresponding platforms in practice.

3. Dormant Integer Overflows A security-relevant integer overflow cannot be detected by reasoning about the types of variables alone. Instead, the range in which these variables operate needs to also be considered. A larger address space allows (a) larger objects to be created and (b) a larger number of objects to be used. Thus, code that performs arithmetic operations on the sizes or number of objects with variables narrower than that of pointers become candidates for integer overflows on 64-bit platforms.

4. Dormant Signedness Issues In addition to truncations, signedness issues may also lie dormant in existing code and become exploitable as the size of the address space grows. A common occurrence of such dormant signedness issues is the practice of assigning the return value of `strlen` to a variable of type `int`. For strings longer than `INT_MAX`, this results in a negative length. However, on 32-bit platforms, exploiting this type of flaw is deemed unrealistic due to the restricted amount of memory available (Koziol et al., 2004, Chp. 18 pp. 494). On 64-bit platforms, however, strings of this size can be easily allocated by a single process, making it possible to trigger these dormant signedness issues.

5. Unexpected Behavior of Library Functions Several standard C library functions have been originally designed with 32-bit data models in mind and thus become vulnerable to truncations, overflows or signedness issues. Although some of these functions have been adapted to 64-bit data models, developers are often not aware of the changed functionality.

For instance, functions for printing strings, such as `fprintf`, `snprintf` and `vsnprintf` have been designed with the assumption that strings cannot be longer than `INT_MAX`. While this assumption is reasonable on 32-bit platforms, it does not hold true for 64-bit data models. If the string to be formatted grows too large, the C99 standard demands that `snprintf` returns a fixed value of -1 (ISO, 1999, Sec. 7.19.6). In practice, this can result in vulnerabilities when programmers directly make use of the return value to shift pointers.

source type	`int`	`unsigned int`	`long`	`unsigned long`	`ssize_t`	`size_t/pointer`	`long long`
dest type	4		4 (→ 8)		4 → 8		8
`int`	○○○	◍◍◍	○○●	◍◍●	○●●	◍●●	●●●
`unsigned int`	◍◍◍	○○○	◍◍●	○○●	◍●●	○●●	●●●
`long`	○○○	◍◍○	○○○	◍◍◍	○●○	◍●◍	●●○
`unsigned long`	◍◍Ⓔ	○○○	◍◍◍	○○○	◍●◍	○●○	●●◍
`ssize_t`	○○○	◍○○	○○○	◍○◍	○○○	◍◍◍	●○○
`size_t/pointer`	◍ⒺⒺ	○○○	◍Ⓔ◍	○○○	◍◍◍	○○○	●◍◍
`long long`	○○○	○○○	○○○	○○◍	○○○	○◍◍	○○○

Table 2.1: Assignments using basic types on Win32/Linux32 (left circle), Win64 (middle circle) and Linux64 (right circle): ○ denotes no conversion problem, ◍ a change in signedness, possibly with sign extension (E) and ● marks an integer truncation (Wressnegger et al., 2016b).

right operand	`int`	`unsigned int`	`long`	`unsigned long`	`ssize_t`	`size_t/pointer`	`long long`
left operand	4		4 (→ 8)		4 → 8		8
`int`	○○○	●●●	○○○	●●●	○○○	●●●	○○○
`unsigned int`	●●●	●●●	●●○	●●●	●○○	●●●	○○○
`long`	○○○	●●○	○○○	●●●	○○○	●●●	○○○
`unsigned long`	●●●	●●●	●●●	●●●	●●●	●●●	○○●
`ssize_t`	○○○	●○○	○○○	●●●	○○○	●●●	○○○
`size_t/pointer`	●●●	●●●	●●●	●●●	●●●	●●●	○●●
`long long`	○○○	○○○	○○○	○○●	○○○	○●●	○○○

Table 2.2: Comparisons using basic integer types on Win32/Linux32 (left circle), Win64 (middle circle) and Linux64 (right circle): ○ denotes a *signed* and ● an *unsigned* comparison of integers (Wressnegger et al., 2016b).

2.1.2 Implicit (Integer) Type Conversions

We proceed to analyze how wide-spread 64-bit migration issues are in today's software. In this experiment, we study how often type conversions potentially go wrong. To this end, we inspect all 198 source packages from Debian stable ("Jessie", release 8.2) that are tagged as either *Required*, *Important* or *Standard* and are written in the C or C++ programming languages. We compile each package on Debian 32-bit and Debian 64-bit in order to inspect all warnings raised.

Most compilers enable to emit warnings when an assignment, arithmetic operation or a comparison is applied to operands of incompatible integer types and an implicit conversion is required. Frequently, these compiler flags are however omitted due to the sheer amount of warnings potentially raised in practice (López-Ibáñez and Taylor, 2006). As a matter of fact, we find that none of the 198 Debian packages uses one of these flags. We thus explicitly add the following compiler flags to separately report flaws related to type conversions: `-Wconversion` for width conversions, `-Wsign-conversion` for changes in signedness, `-Wsign-compare` for comparisons of signed and unsigned types and `-Wfloat-conversion` for conversions that involve a loss in floating point precision.

Table 2.3 summarizes the results. For each conversion type we list the total count of warnings raised by the compiler on the 64-bit system per package and especially highlight warnings that have emerged from the migration process. We find that the vast majority of warnings are width and sign conversions with 442 and 259 warnings per package, respectively. Especially, the conversion from `size_t` to `int` and vice versa appears to be problematic in practice, spawning 21,527 warnings in core packages of Debian stable. All these warnings are exclusive to 64-bit and do not occur on 32-bit platforms. By contrast, sign comparisons only slightly increase due to the 64-bit migration. However, migration vulnerabilities often occur on 64-bit platforms due to comparisons that remain signed rather than being implicitly converted to unsigned. Hence, the amount of warnings resolved in comparison to a 32-bit platform has to be taken into account as well, such that in total 15 % of the warnings can be considered critical.

2.1.3 Patterns of 64-bit Migration Issues

Of course, not all implicit conversions indicate a bug or even a vulnerability that in further follow may be used for an attack. We hence narrow this vast amount of suspect locations down by specifically looking for code patterns that may cause unintended operations on 64-bit platforms. To this end, we make use of techniques from control-flow and data-flow analysis to model specific patterns of 64-bit migration issues. In particular, we characterize patterns from the five categories presented earlier and count the occurrences of these in two code bases: We again consider the packages from Debian stable described in the previous section and additionally examine the 200 (at the time) most popular C/C++ projects on GitHub.

Debian stable		**Average per package**			
Category	# packages	`-Wconversion`	`-Wsign-conversion`	`-Wsign-compare`	`-Wfloat-conversion`
Required	53	576 (334)	1,009 (216)	18 (2)	5 (1)
Important	56	738 (437)	976 (269)	33 (1)	10 (0)
Standard	89	913 (510)	993 (279)	28 (1)	3 (1)
*	198	773 (442)	993 (259)	27 (1)	5 (1)

Table 2.3: Number of implicit type conversions per package on 64-bit platforms. The first value denotes all warnings raised, the value in brackets the amount that is *exclusive* to 64-bit and that does not occur on 32-bit systems (Wressnegger et al., 2016b).

Code-base	**P1: `atol`**	**P2: `memcpy`**	**P3: `loops`**	**P4: `strlen`**	**P5a: `snprintf`**	**P5b: `ftell`**
Debian Jessie	21.49 % (133)	7.76 % (2,536)	8.47 % (1,264)	13.85 % (7,595)	27.55 % (762)	64.74 % (628)
GitHub	18.66 % (25)	15.19 % (2,918)	12.56 % (658)	22.54 % (3,572)	34.79 % (502)	85.05 % (182)
Average	20.98 % (158)	10.51 % (5,454)	9.53 % (1,922)	15.80 % (11,167)	30.03 % (1,264)	68.41 % (810)

Table 2.4: Number of specific patterns for 64-bit migration issues in source packages of Debian stable (“Jessie”, release 8.2) and 200 popular C/C++ projects hosted on GitHub, relative to their absolute usage (Wressnegger et al., 2016b).

P1. *New truncations.* As an example for a truncation that exclusively happens on 64-bit systems we consider the faulty use of the standard library function `atol`. We count all occurrences of `atol` and relate these to those invocations that assign the return value to a variable of type `int` rather than `long`.

P2. *New signedness issues.* When used in the context of memory operations, signedness issues may cause severe security flaws. For this class, we focus on unexpected sign-extensions in combination with the memory copy operation `memcpy`. We count all invocations of `memcpy` that use a signed variable of type `int` to specify the amount of data to copy and compare these to the overall number of calls to `memcpy`.

P3. *Dormant integer overflows.* Integers may under or overflow in various situations. For this pattern we choose a rather strict scenario, in which a loop iterates over code based on 64-bit related data. In particular, we count `for` loops that use a loop-variable of type `size_t` and relate these to the subset of loops that additionally increment or decrement a variable of type (`unsigned`) `int` in their loop body.

P4. *Dormant signedness issues.* For this class, we consider the incorrect use of the `strlen` function. We count the calls to `strlen` that do not use a string literal as parameter and put these in relation to occurrences that assign the return value of `strlen` to (`unsigned`) `int` rather than `size_t`. Left values of other types are not considered as problematic in this example and count for the reference quantity.

P5. *Unexpected behavior of library functions.* Finally, we inspect two examples of library functions that behave differently than (a) developers might expect or (b) the C99 standard specification. First, we count occurrences of `snprintf` that make use of their return value and put these in relation to occurrences that use the return value but *do not* check its validity. Second, we count the calls to `ftell` in relation to the absolute usage of `ftell*` functions (`ftell`, `ftello`, `ftell64` and `_ftelli64`).

The presented patterns do not provide a complete list of 64-bit migration flaws, but should convey a feeling for the prevalence of such flaws by example. Surprisingly, these trivial patterns already point out a large number of potential issues. Table 2.4 summarizes our findings. 21 % of all calls to function `atol` are assigned to a variable of type `int` instead of `long`, causing a truncation on 64-bit systems (P1). Also, developers frequently pass signed integers of type `int` to function parameters defined as `size_t`. In case of the `memcpy` function and its parameter for specifying the number of bytes to copy, roughly 10 % of the calls are used incorrectly, allowing for the malicious use of implicit sign-extensions (P2). Our pattern modeling integer overflows induced by simple `for` loops reveals that in 9.5 % an `int` variable is incremented although the loop-counter is specified as `size_t` (P3). 15 % of all calls to `strlen` are falsely assigned to a variable of type `int` rather than `size_t` (P4). Finally, the `snprintf` and `ftell` functions are incorrectly used and their results insufficiently checked in 30 % and 70 % of all cases, respectively (P5a & P5b).

In summary, we observe that projects included in Debian appear to exhibit less such patterns of 64-bit migration flaws than the projects retrieved from GitHub—the absolute number however suggests a significant potential for misuse.

2.1.4 Discussion

Ideally, software vulnerabilities are addressed by thorough code audits performed by human experts. In case of flaws induced by migrating from 32-bit to 64-bit platforms, such audits need to specifically focus on problematic type conversions and related code patterns. This however is a time-consuming and complex task. Our study suggests that, on a large scale, this currently is not put into practice and shows that vulnerabilities resulting from the migration process are still a major issue.

The prevalence of this particular type of flaw underlines the enormous attack surface that complex software projects exhibit as of today. An adversary may use vulnerabilities to attack a system or specially craft malware to do so. By further improving tools that assist the detection of software vulnerabilities, but also the development process itself, it is possible to contain a large portion of weak points in software. Eliminating the risk and fundamentally solving the issue, however, is unlikely. In this thesis, we thus strive for reactive approaches for detecting attacks.

2.2 Insufficiency of Traditional Methods

Next, we study the effectivity of traditional methods for fending off malware using the example of commercial anti-virus software. Virus scanners are one of the most common defenses against security threats. Millions of end hosts run them on a regular basis to check files for infections and to detect malicious code. The success and prevalence of these products largely build on their simple yet reliable functionality: Files are matched against a database of known detection patterns (signatures) which is regularly updated by the vendor to account for novel threats. Such pattern matching can be implemented very efficiently and is able to spot all sorts of threats if appropriate and up-to-date signatures are available (see Aycock, 2006; Ször, 2005). Signature-based detection however suffers from a well-known drawback: Unknown threats for which no signatures exist can easily bypass the detection. This problem is further aggravated by the frequent use of obfuscation in malicious code that obstructs static signature matching (Linn and Debray, 2003; Moser et al., 2007a; Ferrie, 2008). Over the last decade, anti-virus vendors have increasingly adopted more advanced detection mechanisms from computer security research to keep up with malware evolution. Still, signatures based on byte patterns remain an integral part of security products and complement more sophisticated detection mechanisms.

```
 1  83 c7 43  ; add  edi, 0x43
 2  89 fa     ; mov  edx, edi
 3  83 ea 2e  ; sub  edx, 0x2e
 4  {2}       ; 2-byte gap
 5  00 00
 6  33 c0     ; xor  eax, eax
 7  8a 1f     ; mov  bl, byte ptr [edi]
 8  32 1C 10  ; xor  bl, byte ptr [eax + edx]
 9  88 1F     ; mov  byte ptr [edi], bl
10  40        ; inc  eax
11  83 f8 05  ; cmp  eax, 5
12  7c 02     ; jl   +2
13  33 c0     ; xor  eax, eax
14  47        ; inc  edi
```

(a) Derived signature for the *Swizzor* malware.

```
 1  49 64 00 00    ; "Id"
 2  37 36 34 38    ; "76487-337-8429955-22614"
 3  37 2d 33 33
 4  37 2d 38 34
 5  32 39 39 35
 6  35 2d 32 32
 7  36 31 34 00
 8  50                ; push eax
 9  81 c4 f4 fe ff ff ; add esp, 0xfffffef4
10  33 c0             ; xor eax, eax
11  54                ; push esp
12  6a 01             ; push 1
13  6a 00             ; push 0
```

(b) Derived signature for a generic backdoor.

Figure 2.1: Static malware signatures derived from anti-virus products (Wressnegger et al., 2017a).

In order to assess the sufficiency of wide-spread attack detection solutions, such as anti-virus software, we subsequently study the prevalence and specificity of static signatures. To this end, we make use of a method that derives signatures based on static byte patterns from anti-virus products by conducting a series of black-box tests (Wressnegger et al., 2017a). In particular, a virus scanner is provided with a number of slightly modified versions of the same malware sample. By combining the individual scan results it becomes possible to reason about the underlying (static) malware signature and approximate its byte pattern. Figure 2.1a and 2.1b illustrate the byte patterns of two malware signatures derived by the method. Figure 2.1a shows a signature for the malware family *Swizzor*. Clearly, an effort has been made to pinpoint the decryption loop of the malware (starting at line 7) that unveils the malicious payload at runtime. In particular a 5-byte long key is used to decrypt data using the XOR operator (line 8). By contrast, the signature shown in Figure 2.1b contains an ASCII string (line 1–7) that corresponds to the Windows product key of the Anubis sandbox (Bayer et al., 2006). The instructions at line 8–13 are part of a call to the `RegOpenKeyExA` API function. This signature captures a simple environment check used by several backdoors that compares the product key of the execution environment against known sandbox systems.

While these signatures describe essential parts of the malware, they are very specific and thus can easily be circumvented by modest changes of the malicious code. In the first example, for instance, swapping the independent instructions at line 6 and 7 effectively bypasses detection. The same principle can be applied to the second example. Moreover, the second signature matches static data (the product key) that can be trivially relocated or reformatted in future variations of the malware. Obviously more advanced detection schemes that take context and semantics of the malicious code in consideration are needed. This can, for instance, be achieved by detection based on a combination of static and dynamic analysis as employed by the method presented in Chapter 4 of the thesis.

Subsequently, we first inspect the prevalence and complexity of such static signatures in today's anti-virus solutions, before we further reason about their quality by examining their context and semantics.

2.2.1 Prevalence and Complexity of Static Anti-Virus Signatures

In the first part of our study, we investigate how wide-spread static pattern matching still is in comparison to more advanced techniques. For this experiment, we derive signatures for the open-source scanner *ClamAV* (*AV1*) and four commercial anti-virus products (*AV2*-*AV5*). For the purpose of comparability, we operate on malware samples that are detected by all five scanners. In total we have collected 9,969 malware samples, from which the vast majority are applications or dynamic libraries in the Portable Executable format. The remaining files correspond to archives, Windows shortcuts and other carriers of malicious code. Depending on the applied virus scanner, the files in the dataset are assigned to roughly 250 malware families, where the concrete number of different signature labels assigned by the scanners ranges from 277 up to 1,327 (see the first column in Table 2.5).

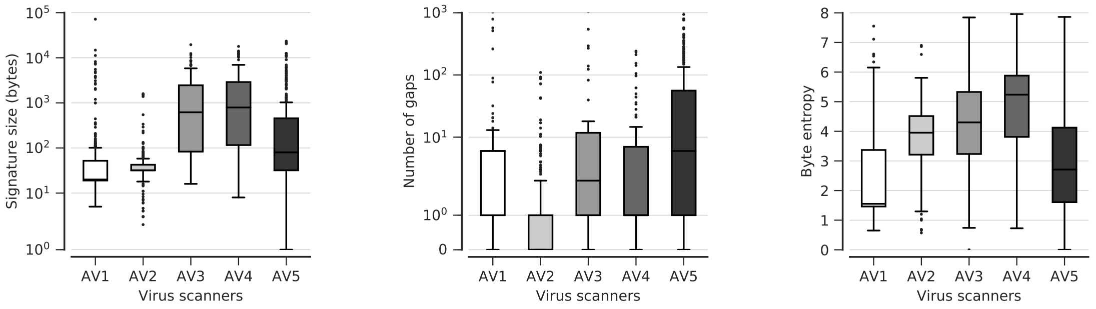
Using black-box tests (Wressnegger et al., 2017a) we derive signatures for each of these malware samples in order to statistically assess the quality and content of the used signatures. Table 2.5 provides a first overview of the derived signatures. The scanners considerably differ in type and amount of signatures that can be derived. On average, 38 % pattern-based signatures and 54 % hash-based signatures are inferred, whereas only the remaining 8 % cannot be uncovered.

AV2 especially stands out with 92 % byte patterns and only a single hash-based signature, while for the other scanners at least one third of all signatures corresponds to hash sums. On the other end of the scale, for *AV3* one fourth of all signatures could not be extracted. This may indicate the use of advanced heuristics or that multiple pattern-based signatures are assigned to the same label, implicitly constructing a signature with long disjoint patterns.

		Derived Signatures		
	# Labels	Hash sums	Byte patterns	Unknown
AV1	1,327	940 (71 %)	377 (28 %)	10 (1 %)
AV2	277	1 (0 %)	255 (92 %)	21 (8 %)
AV3	522	323 (62 %)	69 (13 %)	130 (25 %)
AV4	282	178 (63 %)	93 (33 %)	11 (4 %)
AV5	586	177 (30 %)	353 (60 %)	56 (10 %)
Average	598	323 (54 %)	229 (38 %)	45 (8 %)

Table 2.5: Results of signature derivation.

However, not all signatures the method derives can be interpreted as clearly as the examples from the previous section (Figure 2.1). Many signatures, for instance, match resources, wrongly aligned code, import tables or even compressed data. To get a better understanding of the quality of these signatures, we statistically analyze the characteristics of the corresponding byte patterns. For each of the five scanners Figures 2.2a to 2.2c show the distribution of the signature size, the number of gaps and the entropy of the patterns. With respect to the size, *AV2* again draws attention: It appears that this scanner makes use of very compact and equally sized patterns that contain no or only a few gaps. While short signatures generalize well, they may also induce more false positives which presumably is the reason why other scanners such as *AV3*–*AV5* use longer signatures—partly also with more gaps between the byte patterns.



(a) Size of signatures

(b) Gaps in signatures

(c) Entropy of byte patterns

Figure 2.2: Statistics for the derived byte-pattern signatures.

Moreover, the entropy of signatures can be an indicator for the data used to characterize a malware sample. An entropy of 0.0 indicates uniform byte patterns, such as padding, while values close to 8.0 are reached by random, encrypted or compressed data. Matched import tables reside on the lower end of the scale due to the identical significant bytes of addresses and x86 code is mostly located in the middle and upper half of the scale. Three of the scanners cover almost the full range of entropy, indicating the diversity of the byte patterns in the signatures.

2.2.2 Context and Semantics of Anti-Virus Signatures

As final experiment of our study, we analyze the quality of the derived signatures. We are interested in determining whether the signatures are bound to a specific context within the file and if they model semantics of the targeted malicious code. To this end, we examine how well the derived signatures can be implanted in benign files that have nothing in common with the original malware. Since our dataset is mainly comprised of PE executables, we consider a set of benign applications of the same format for this task, which we have verified not to be flagged as malicious by any of the scanners. In particular, we choose the following applications from the Windows system directory: *Calculator*, *Windows Command Prompt*, *Internet Explorer*, *Microsoft Management Console*, the *System Configuration Application* and *Microsoft Paint*. We proceed to implant the derived signatures into these applications, irrespective of the relative position to PE sections or existing code, and apply the virus scanners to the resulting files. A breakdown of the detection results for this experiment is provided in Table 2.6. On average 68 % of the implants are successful. Apparently, several of the pattern-based signatures are applied without checking the context or semantics of the matching region for plausibility, allowing the direct transfer of signatures from one file to another. The situation is especially troublesome for *AV1*, *AV2* and *AV3* that simply flag the majority of implants as malicious. *AV5* appears to be the only one to use semantics-aware matching in most of the cases. Still, the overall quantity of working implants is higher than for *AV3* or *AV4*.

	Byte patterns	**Implantable**
AV1	377	358 (95 %)
AV2	255	227 (89 %)
AV3	69	65 (94 %)
AV4	93	54 (58 %)
AV5	353	80 (23 %)
Average	229	157 (68 %)

Table 2.6: Detection results on pattern-based signatures implanted in benign PE files.

2.2.3 Discussion

Despite several advanced detection techniques, such as behavioral blocking and packer-agnostic unpacking, most anti-virus products still rely on static signature matching as a fall-back mechanism. This signature-based detection is an effective and efficient tool, if appropriate and up-to-date signatures are available. Our study shows that in practice, a considerable amount of the deployed signatures correspond to simple byte patterns that are not bound to particular file types or contexts.

To be effective in fending off attacks at the end host, malware signatures however need to incorporate context and semantics of the matched program code to avoid side-effects. Dynamic approaches and heuristics, for instance, implicitly include these relations. In the remainder of this thesis, we investigate methods for advanced malware detection that prove effective in this particular scenario. With the aid of machine learning it is possible to automatically learn complex relations of the malware code and its context.

Chapter 3

Data Representation

In this chapter, we investigate how data can be represented to optimally support learning attack detection. This includes the embedding into an appropriate feature space, but also to check whether the chosen representation is suitable for a particular learning task and how the data is stored to optimally facilitate the algorithms to be used for learning.

We begin by discussing language models that generically map arbitrary data into feature spaces in Section 3.1. This high-dimensional embedding enables us to learn a separation of individual classes, such as goodware from malware, or benign network traffic from attacks and intrusions. Whether or not these models are suitable for a particular learning task is however by no means certain. In Section 3.2, we thus develop suitability criteria that can be used to decide when to use the one over the other based on the data at hand. Moreover, the high-dimensionality of language models implies an enormous amount of unique features. Processing this kind of data efficiently, requires specialized data structures, which we describe in Section 3.3. We particularly highlight data structures that (a) allow to efficiently access and search for stored items, and (b) store data probabilistically in order to increase data density on a limited amount of memory.

3.1 Language Models

Most learning methods operate on numeric vectors rather than on raw data. Therefore, it often is necessary to construct a map to a vector space for interfacing with learning methods. In some settings, this can be achieved by defining numeric measures describing the data, such as the length or the entropy of packets. A more generic map can be developed by representing strings or substrings, which is applicable to a large variety of data and its representations (Salton et al., 1975). One particular prominent example are *n-gram models*, which in turn are a special case of bag-of-words models. Initially proposed for natural language processing (Suen, 1979; Cavnar and Trenkle, 1994; Damashek, 1995), n-grams have become the representation of choice in many detection systems (e.g., Kolter and Maloof, 2006; Wang et al., 2006a; Rieck and Laskov, 2006; Perdisci et al., 2009; Rieck et al., 2010; Laskov and Šrndić, 2011; Hadžiosmanović et al., 2012; Wressnegger et al., 2013b, 2016a, 2018).

To describe data in terms of n-grams , each data object x from an input set $\mathcal{X}$ first needs to be represented as a string of symbols from an alphabet $\mathcal{A}$, where $\mathcal{A}$ often is defined as bytes or tokens. As an example, for analyzing network packets we simply consider the data in each packet as a string of bytes. Similarly, we model the structure of Flash animations as hierarchical tokens of container identifiers. By moving a window of n symbols over each object x, we can then extract all substrings of length n. Figure 3.1 shows examples of byte and token n-grams as used in Chapters 4 and 5. While byte n-grams of course are not limited to human-readable strings, token n-grams are not necessarily words either, but are extracted from bytes based on arbitrary delimiters—in this case blanks, such that numbers as well as brackets result in individual string tokens.

```
\x00\x1aGetVal  →  \x00\x1aGe         69 77 9 2 [ 39 26 ] 86  →  69 77 9 2
                       \x1aGet                                      77 9 2 [
                           GetV                                        9 2 [ 39
                            etVa                                         2 [ 39 26
                             tVal                                          [ 39 26 ]
                                                                             39 26 ] 86
```

(a) Byte n-grams

(b) Token n-grams

Figure 3.1: Examples for n-grams based on bytes and tokens with $n = 4$, extracted from (a) network data and (b) analysis reports for Flash animation, respectively.

These substrings (n-grams) give rise to a map to a high-dimensional vector space, where each dimension is associated with the occurrences of one n-gram. Formally, this map ϕ can be constructed from an input set $\mathcal{X}$ using the set S of all possible n-grams as

$$\begin{aligned} \phi\colon \mathcal{X} &\to \mathbb{R}^{|S|}, \\ x &\mapsto \Big(\operatorname{occ}(s, x)\Big)_{s \in S} \end{aligned}$$

where the function $\operatorname{occ}(s, x)$ returns the frequency, the probability or a binary flag for the occurrences of the n-gram s in the data object x.

Several methods for the detection of attacks and malicious software indirectly make use of this map. For example, the methods PAYL (Wang and Stolfo, 2004), McPAD (Perdisci et al., 2009) and Anagram (Wang et al., 2006a) analyze byte n-grams for detecting server-side attacks, where the first two consider frequencies and the latter binary flags for the occurrences of n-grams. Similarly, the methods Cujo (Rieck et al., 2010) and PJScan (Laskov and Šrndić, 2011) use token n-grams with a binary map for identifying malicious JavaScript code in web pages and PDF documents, respectively. We continue this line of research and use binary embeddings of n-grams for the detection of Flash-based malware on top of static and dynamic analysis in Chapter 4. In Chapter 5, we then revise the use of n-grams for high-entropy binary protocols by combining count and binary embeddings for effective attack detection on network level.

Although n-grams provide generic and effective means for modeling data, the exponential growth of the resulting vector space often impedes efficient operation. However, the number of n-grams in a particular object x is linear in the object's size and thus efficient data structures can be used for processing the extracted n-grams. Adequate data structures to optimally support attack detection are discussed in Section 3.3. Moreover, in the next section, we show that this sparsity is not only beneficial for processing high-dimensional vectors but also provides a good indicator for the feasibility of anomaly detection.

3.2 Suitability for Learning

When it comes to the application of learning-based methods there are several prerequisites to be clarified beforehand. For instance, how much training data is needed in order to learn an expressive model? Obviously the more data is used for training, the better the outcome might be. To detail this rather general statement Wang et al. (2006a) for instance specify the "*likelihood of seeing new n-grams as training time increases*" in order to indicate the completeness of their model in terms of the amount of training data. If this measure converges, the training set can be considered as sufficiently large.

Looking at an even earlier stage of learning, other very important considerations need to be made. For instance, is the underlying problem, the data to be used and the constructed feature space suitable for learning? Unfortunately previous works often fall short of explaining the reasons for or against particular methods and features. We have worked out three criteria for a problem's suitability for being learned in an anomaly detection setting: (1) The *perturbation* within a dataset's main class, (2) the *density* of the used feature space and (3) the *variability* of the n-grams in the individual classes (Wressnegger et al., 2013b).

Criteria 1 (Perturbation)**.** *The perturbation is the expected ratio of n-grams in a benign object that are not part of the training data.*

A perturbation value of 0 means that all possible n-grams in the dataset have been observed in the training phase, whereas a high value indicates that despite training, a large number of unseen n-grams is still to be expected in benign objects during testing. This measure is closely related to the likelihood of unseen n-grams as discussed by Wang et al. (2006a). However, while the convergence of the likelihood only shows that the training does not improve anymore with an increasing amount of data, it does not allow to draw any conclusions about the data's suitability for anomaly detection as such. To do so we consider the likelihood of the last objects observed during the training phase as the expected perturbation during testing. Consequently, the lower the perturbation, the better it is for learning. In Chapter 4, for instance, we abstract string literals of malware reports to cap the used alphabet and reduce the perturbation.

Criteria 2 (Density)**.** *The density is the ratio of the number of unique n-grams in a dataset to the total number of all possible n-grams induced by the underlying alphabet.*

As second criteria we use the density of a training dataset when mapped into the feature space induced by n-grams. A value close to 0 indicates low density, meaning that the feature space is sparse, whereas a value of 1 indicates that the feature space is maximally dense. This is directly related to the overall size of the feature space and can provide some indication of how well a class can be learned for anomaly detection. For instance, if a dataset has been oversimplified in the course of lowering the perturbation, the remaining symbols may become too general to reflect the characteristics that differentiate one class from another: The feature space is too densely populated. In feature space both classes occupy an identical, dense region. This of course assumes a certain homogeneity of the benign and malicious data. The method presented in Chapter 5 actively counteracts a too large density by pruning seldom occurring features.

Criteria 3 (Variability). *The variability is the ratio of the data's (information) entropy to its maximal value as induced by the alphabet size (normalized entropy).*

The third suitability criteria is the variability of a dataset to be used for training. A value of close to 0 means that the variability of the data is very low, whereas a value of 1 indicates maximal uncertainty about the next element in a sequence. As the variability is normalized to its maximal value, consequently it again cannot be determined for data that has a virtually infinite set of unique n-grams. Using this measure it is possible to characterize the structure of a dataset. A moderate variability is equally important for anomaly detection as well as classification that the data possesses noticeable structure, which can be learned and used for detection. Random data, that does not present any kind of visible structure, but largest possible variability, would appear as a flat line at the maximum of 1.0. Together with the density, the variability also is crucial for the resistance against polymorphic blending attacks as discussed in Chapter 5.

3.3 Time and Space-efficient Storage

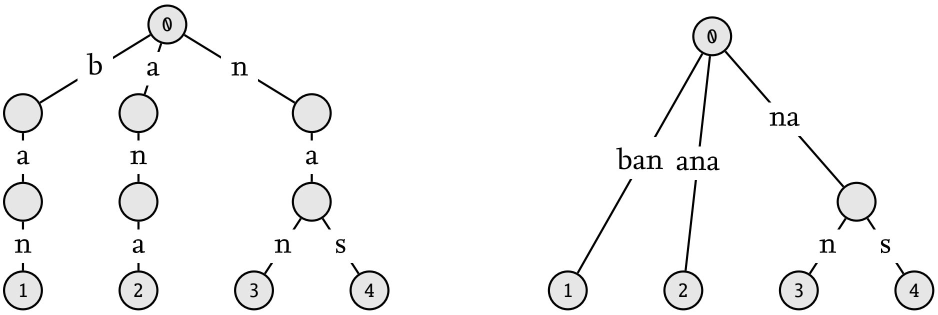
For efficiently learning attack patterns and modeling attacks with language models two aspects are key when it comes to data structures. First, they need to facilitate runtime-efficient algorithms and provide fast access. Second, the high number of features induced by language models requires a considerable amount of memory when stored explicitly. Space-efficiency thus is essential for realizing efficient machine learning on large amounts of data.

Subsequently, we discuss data structures that fulfil these criteria on different levels. We start with tree representations, such as radix trees and suffix trees in Section 3.3.1, that allow to efficiently search for data and calculate substring statistic as used in Chapter 6 for deobfuscating malware. In Section 3.3.2, we then proceed with hash tables as a representative of data structures for sparse representations of data, before we discuss probabilistic extensions thereof such as Bloom filters and Count-Min sketches in Section 3.3.3. The latter are central components of the methods presented in Chapters 4 and 5.

3.3.1 Trees and Tries

String-based data, such as static malware signatures or n-grams used for self-learning attack detection, frequently are stored in tree-like structures. Tries, for instance, are employed to reduce redundancy in the data and facilitate efficient string matching algorithms, such as the well known Aho-Corasick algorithm (Aho and Corasick, 1975).

A trie, or prefix tree, is a search tree that stores items so that each edge corresponds to one element of the input—in case of strings, one character of the string. At this, inputs with common prefixes overlap from the root of the trie towards its leaves. Figure 3.2a, for instance, shows a trie for all 3-grams of the word bananas. Neither the n-gram ban nor ana share a prefix with any of the other n-grams and thus these constitute separate branches in the trie. The substrings nas and nan, on the other hand, have a common prefix such that their paths are merged and only branch for the last element. Nodes themselves do not necessarily carry any information regarding the values stored in the trie, except for terminal nodes, that indicate the end of an item. Since the n-grams used as example in Figure 3.2 by definition are of the same length, here this only applies to leaves and the root node, which represents the empty string.



(a) Trie/ Prefix tree

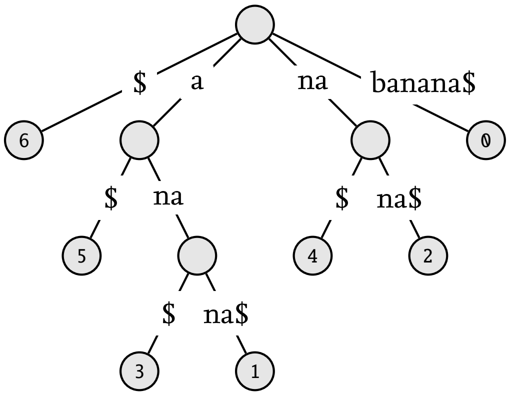
(b) Radix tree or compact prefix tree

Figure 3.2: The (a) prefix tree and (b) radix tree representing n-grams of the word bananas. The numbered nodes indicate the id of the substring matched by the path. The root node represents the empty string.

Radix Trees. These trees are a compact representation of a trie that collapses path segments with no branches. All remaining nodes thus are either branching points or terminal nodes, that indicate the end of an item. In consequence, edges may be labeled with sequences of elements. Formally, radix trees are defined over bits, such that the radix r of the tree specifies the number of bits per element. A branching node thus has at most 2^r children. Figure 3.2b shows the radix tree for the n-grams of the word bananas. In contrast to the prefix tree, here, direct edges connect the root node with the corresponding leaves for ban and ana. Also, the prefix of nan and nas is collapsed to a common edge, while the branch remains. As the example operates on characters of strings the radix r depends on the encoding and may be 7, 8 or 32 for ASCII, ISO-8859-1, and UTF-32, respectively.

Suffix Trees. These are another variant of tries that similar to radix trees condense edges without branching or terminal nodes. The data structure not only stores inputs as is, but all its suffixes (Weiner, 1973). In doing so, it can be used to efficiently search for substrings, and collect various statistics in linear or quasilinear time, that are useful to attack detection. For instance, finding the longest common substrings for signature generation, or counting term and document frequencies for statistically deobfuscating malware (see Chapter 6).

Figure 3.3a shows the suffix tree for the word banana. Note that the input ends with an additional character $. This terminal marker may be any character that is not part of the input alphabet and ensures that each suffix ends in a leaf node. This would not be the case for suffixes a, ana, and na otherwise. In practice, this of course may also be realized with alternative means rather than extending the input alphabet.



(a) Suffix tree

Input	b	a	n	a	n	a	$
Offset	0	1	2	3	4	5	6
SA	6	5	3	1	0	4	2
	$	a	a	a	b	n	n
		$	n	n	a	a	a
			a	a	n	$	n
			$	n	a		a
				a	n		$
				$	a		
					$		

(b) Suffix array

Figure 3.3: The (a) suffix tree and (b) the corresponding suffix array for the string banana. In order to mark the end of the string an additional terminal character $ is introduced. Leaf nodes are numbered with the starting position of the suffix within the input string.

Such a tree can be built in linear time, $\mathcal{O}(n)$. Each input of length n consists of n suffixes, such that a suffix tree contains just as many leaves. Since edges carry these suffixes as labels, the overall space complexity is quadratic in the input's length, $\mathcal{O}(n^2)$, which however can be reduced to $\mathcal{O}(n)$, if the offsets of the suffixes in the input strings are stored instead. Nevertheless, the space requirements of such a tree quickly pose a bottleneck for large applications.

Suffix arrays (Manber and Myers, 1990) offer a space-efficient alternative that in combination with various auxiliary data, such as the Longest Common Prefix (LCP) Array, can be used to replace any algorithm using suffix trees and solve the same problem within the same time complexity (Abouelhoda et al., 2004). The suffix array for the word banana including the terminal character $ is shown in Figure 3.3b. It contains the starting positions of the lexicographically ordered suffixes of the input and thus consumes $\mathcal{O}(n)$ space. For instance, at offset 2 the array holds the value 3 that references the suffix `ana$`, while offset 3 refers to suffix `anana$`. As the suffixes are ordered, searching for as well as counting all occurrences of a substring straight-forward.

3.3.2 Hash Tables

Efficiently storing and processing high-dimensional models relies on a sparse representation of vectors. This commonly is realized as associative array, that in contrast to conventional arrays, maps keys of arbitrary type and order to values. A feature vector thus only stores dimensions that are non-zero and assumes all missing dimensions to be zero, which amounts to huge savings in storage space in practice. Such a map can be implemented with trees and hash tables likewise (Cormen et al., 2009). On average the latter however offers constant search, insertion, and deletion time in $\mathcal{O}(1)$.

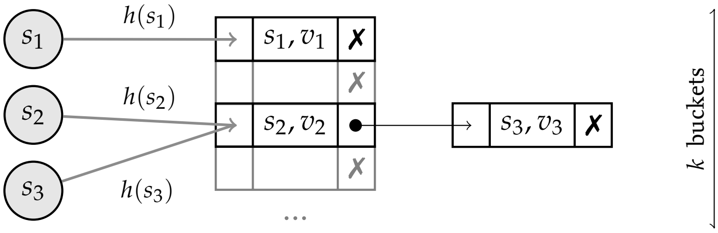


Figure 3.4: Schematic depiction of a hash table using "separate chaining" for collision resolution.

Hash tables are defined over an one-dimensional array of k buckets (slots that may contain one or more inputs) and a hashing function h that maps input strings to numeric values in the interval $[0, k-1]$. To insert a key-value pair (s_1, v_1) in the data structure, first, the hashing function is applied to the key. The resulting value is then used to select the bucket to store the key alongside with its value as depicted in Figure 3.4. If the hashes of two different keys collide, as in the case of s_2 and s_3, the key-value pairs are stored in the same bucket, connected as linked list, and identified based on string comparison of the key itself. In the worst case, that is, hashes of all input keys collide, hash tables thus fall back to (linked) lists which cannot be traversed faster than in the number of items, $\mathcal{O}(n)$. However, how exactly collisions are resolved and how the colliding items are stored is subject to the respective implementation.

3.3.3 Probabilistic Data Structures

Over the last decades a number of different probabilistic data structures have been proposed. The objective of their design vary but may roughly be categorized into two strains: Some facilitate certain algorithms and store data without the loss of information, such as Skip Lists (Pugh, 1990) or Random Trees (e.g., Seidel and Aragon, 1996; Martínez and Roura, 1998). Others put up with the potential loss of data and accuracy to increase the density of the stored data and to use up significantly less physical memory. In this section, we look upon the latter and discuss two particular popular variants that have been used in the scope of this thesis: (1) Bloom filters for classifying malware in Chapter 4, and (2) Count-Min sketches for robust anomaly detection in Chapter 5.

Bloom Filters. This data structure allows to probabilistically store membership information of sets (Bloom, 1970). Unlike with hash tables or hash sets inputs are not stored explicitly, but only marked as included in the filter. Retrieving all stored items, thus is not possible. Moreover, this data structure may falsely report inclusions (false positives), whilst its design rules out that included items are not reported (false negatives). For attack detection this property is particularly beneficial when the severity of wrongly flagged benign programs or network traffic outweighs false alarms.

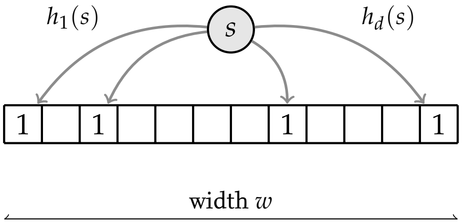


Figure 3.5: Schematic depiction of a Bloom filter (Bloom, 1970).

A Bloom filter is defined as an one-dimensional array of w bits, and d hash functions h_i that map input strings to numeric values in the interval $[0, w-1]$. Initially all cells of that array are set to 0. For storing an item in the Bloom filter, each hash function is applied to that item, for instance a string s appearing in a malware sample. As visualized in Figure 3.5, the resulting value is then used as position in the bit array. At these offsets the bit value is set to 1, indicating the presence of the item. Checking for the existence of an item follows the same procedure: The hash functions are applied to the item to determine the offsets in the bit array. The item has been stored in the Bloom filter (within a certain probability) only if *all* bits at these positions are set to 1. The time needed for inserting as well as querying items thus is fixed by the number of hash functions applied, $\mathcal{O}(d)$, and therefore comparable to hash tables. False positives may however occur due to hash collisions of the input string. While it is rather unlikely that all d hashes collide on the same input, overlaps between multiple strings happen more easily depending on the size of the Bloom filter. Overall the probability for false- positives can be quantified as

$$\left(1 - \left[-\frac{1}{w}\right]^{dn}\right)^{d} \approx \left(1 - e^{-dn/w}\right)^{d}$$

with n being the number of elements inserted in the Bloom filter (Mitzenmacher and Upfal, 2005, pp. 110). The expected number of false positives can thus be adjusted by the size of the bit array and the number of hash functions for the expected number of items.

Count-Min Sketches. This type of probabilistic data structure is closely related to Bloom filters but additionally allow for counting occurrences rather than answering membership queries only (Cormode and Muthukrishnan, 2005, 2012).

A Count-Min sketch is defined as a $w \times d$ two-dimensional array of numeric items of arbitrary size and precision, and d hash functions h_i that map input strings to numeric values in the interval $[0, w-1]$. To store a particular key-value pair or increment the value for a key in the sketch, each hash function is at first applied to the key, for instance a string s. As depicted in Figure 3.6, the resulting value is then used as position in the corresponding row. At these offsets the stored numeric value is incremented by the provided value v. Retrieving the value for a key works analogous: The hash functions are applied to the key in order to determine the numeric values associated to it. The minimum of these values then recites the approximated, true value—in this example the approximate count of string s.

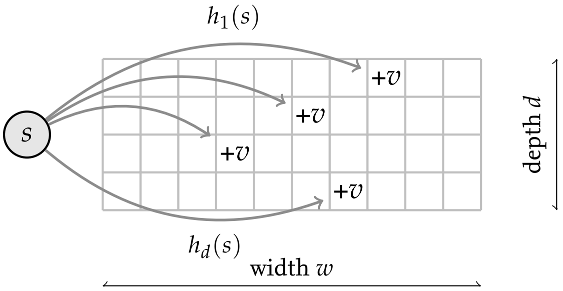


Figure 3.6: Schematic depiction of a Count-Min sketch (Cormode and Muthukrishnan, 2005, 2012). The substring to be added is denoted as s and is processed by d (the depth of the sketch) hash functions h_i to determine the position $p_i = h_i(s)$ with $i \in [1, d], p \in [0, w)$ at which value v is added. Figure taken from Wressnegger et al. (2018).

By construction, the approximated count $\hat{c}$ retrieved from a Count-Min sketch is always larger or equal to the true count c, meaning that the data structure will never underestimate a stored value:

$$c \leq \hat{c} \quad \forall c \in \mathbf{c}$$

where $\mathbf{c}$ is the vector of all values stored in the Count-Min sketch. Furthermore, for a width $w = \lceil \frac{e}{\varepsilon} \rceil$ and a depth $d = \lceil \ln \frac{1}{\delta} \rceil$ it is guaranteed that the difference between the true and approximated value is at most $\varepsilon \|\mathbf{c}\|_1$ with a probability p of at least $1 - \delta$ (Cormode and Muthukrishnan, 2005, 2012)

$$\hat{c}_i \leq c_i + \varepsilon \|\mathbf{c}\|_1$$

In other words, the estimate of the Count-Min sketch is correct within ε times the number of items stored in the data structure with a probability of p.

3.4 Evaluation

We proceed with an empirical evaluation of the suitability criteria and central aspects of the data structures presented in this chapter. To this end, we have gathered data from client and server domains covering a wide range of applications that have been discussed in related works. The composition and nature of these datasets are detailed in Section 3.4.1, before Section 3.4.2 inspects the suitability of them for learning-based attack detection. Finally, we look at probabilistic data structures and empirically assess the influence of hash collisions in Section 3.4.3.

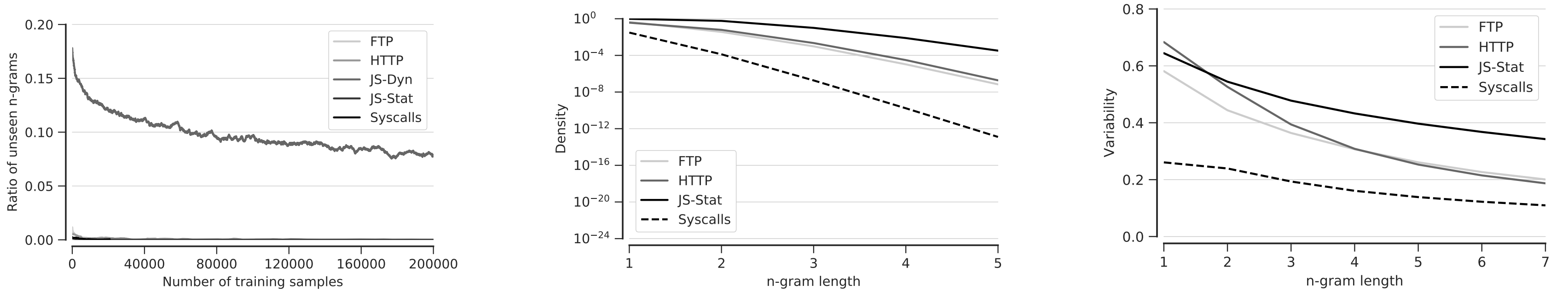
3.4.1 Datasets

In order to cover a large variety of data from different fields of applications, we consider two scenarios: First, data for client-side intrusion detection, and second, data for server-side intrusion detection. The first covers system call traces of programs and JavaScript code of web pages, while the latter involves binary and text-based network protocols.

Client-side Datasets. For the JavaScript dataset we randomly choose 230,000 URLs from the 1 million most popular web pages at the time (Alexa Internet Inc., 1996–2018) and additionally scan all URLs using the *Google Safe Browsing* service (Google Inc., 2008–2018). We then crawl the web pages using the client-side honeypot ADSandbox (Dewald et al., 2010), extract all JavaScript code and dynamically analyze it. The executed instructions and the monitored events constitute the *JS-Dyn* dataset. Additionally, we process the plain JavaScript code similarly to Rieck et al. (2010) by extracting lexical tokens, such as identifiers and numerical constants, for the *JS-Stat* dataset.

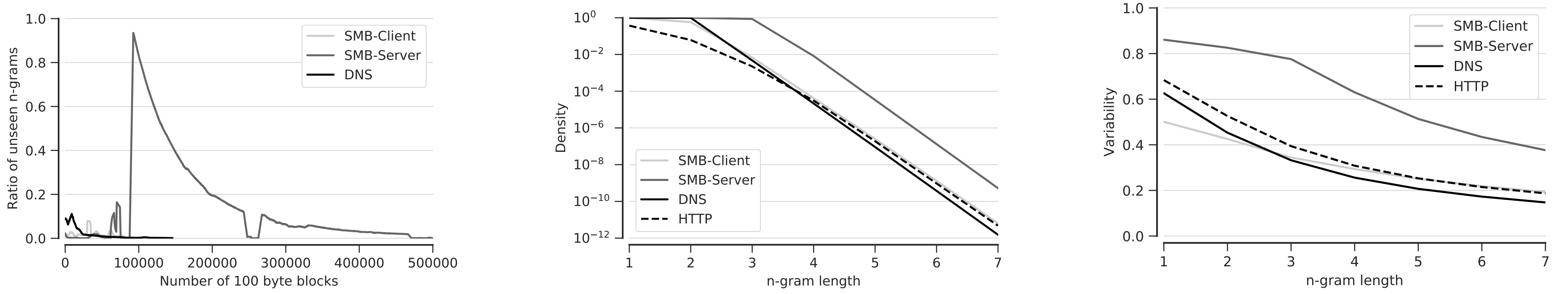
The dataset of system call traces (*Syscalls*) is extracted from the original DARPA IDS evaluation datasets (Lippmann et al., 1999) and corresponds to audit trails of the Solaris BSM module. Although the dataset is rather outdated and has been criticized in the past (McHugh, 2000; Mahoney and Chan, 2004), we additionally study its characteristics for the sake of completeness.

Server-side Datasets. The application layer of the Internet protocol suite offers a variety of different protocols, which we may use for our study. In particular, we consider DNS and SMB as representatives for binary protocols, and HTTP and FTP for text-based protocols. The DNS dataset has been recorded in a period of one week and contains a total of 294,431 requests. In case of SMB an overall amount of 22.6 GiB of data in 154,460 messages has been collected on a single day. SMB often encapsulates files and consequently involves far more data than other protocols. The HTTP data has been recorded from a content management system in a period of one week and incorporates 237,080 HTTP requests. For the FTP communication we make use of data recorded during a period of 10 days with more than 22,000 FTP sessions (Pang and Paxson, 2003).



(a) Perturbation (b) Density (c) Variability

Figure 3.7: Data suitability criteria on example of the *FTP*, *HTTP*, *JS-Dyn*, *JS-Stat* and the *Syscalls* datasets. (a) The perturbation within a dataset (averaged on a sliding window of 5,000 samples), (b) the density of the introduced feature space and (c) the variability of the *n*-grams in these datasets.



(a) Perturbation (b) Density (c) Variability

Figure 3.8: Data suitability criteria applied to the binary-based protocols in our datasets: (a) Perturbation, (b) Density and (c) Variability. For the latter two we additionally included the HTTP communication dataset as a baseline.

Dataset	**Size**	***n*-gram type**	
		byte	token
HTTP	237,080 requests	✗	
FTP	22,615 sessions	✗	
JS-Stat & *JS-Dyn*	230,000 URLs		✗
Syscalls	25,797 traces		✗
DNS	294,431 requests	✗	
SMB	154,460 msg blocks	✗	

Table 3.1: Description of the datasets used for evaluating our data suitability criteria. Additionally to the size of the datasets also the used n-gram types are specified.

On top of the distinction between client-side and server-side detection we differentiate the datasets according to the used type of n-grams. For our analyses and experiments we use the types as shown in Table 3.1, which are consistent with previous work (see Hofmeyr et al., 1998; Wang et al., 2006a; Rieck and Laskov, 2006; Rieck et al., 2010). Specifically, we use byte n-grams for the network protocols and token n-grams for the JavaScript and system call datasets, where the tokens correspond to events, lexical tokens and system call identifiers, respectively.

3.4.2 Suitability

For this experiment we measure the three suitability criteria that we have developed in Section 3.2. In the course of this, we differentiate between text-based and binary-based data/protocols, to better understand the influence of the structure of a protocol on the suitability for learning attack detection.

Text-based protocols. Protocols based on human-readable text, like HTTP and FTP, or client-side data such as system call sequences or JavaScript usually maintain a very strict separation of structure and (payload) data, that often is perceivable without in-depth knowledge of the protocol or the composition of the data. Intuitively this appears to simplify the task, which however is not the deciding factor for automated processing such as machine learning.

Figure 3.7a illustrates the perturbation of the five datasets on the example of 3-gram models. Note that one of those clearly stands out, namely the datasets composed out of the dynamic JavaScript reports. The other four quickly converge to zero and do not exhibit any significant level of perturbation after training. Also *JS-Dyn* seems to converge but to a value unequal to zero. Hence, in this particular case benign data constantly exhibits 5–10 % of unseen n-grams. This renders anomaly detection very difficult, as each benign object appears to be anomalous to 5–10 % already. So, where do the perturbations in the

JS-Dyn dataset come from? The dataset covers the behavior of JavaScript code and thus contains variable names and strings. As a result, the alphabet of the n-grams is not fixed, that is, there exists an infinite number of tokens, consisting of variable names and strings. While this impedes learning in an anomaly detection settings, it is unproblematic for classification as shown by Rieck et al. (2010). To be able to still use anomaly detection the data needs to be preprocessed such that the parts causing the perturbation are abstracted. For *JS-Stat* this was done by lexically analyzing the raw program code and introducing dedicated string tokens that hide but describe the raw data (Rieck et al., 2010; Laskov and Šrndić, 2011). That way also the names of functions, parameters and variables are abstracted. A similar approach would be possible to abstract reports from *JS-Dyn*.

For the density we have to restrict our analysis to data with a closed alphabet. Remember, the density is the ratio of the number of unique n-grams in a dataset to the total number of all possible n-grams induced by the underlying alphabet. In consequence, it is not possible to measure the density for an infinite large set of n-grams as induced by the dynamic JavaScript reports. Figure 3.7b hence shows the density of the *HTTP*, *FTP*, *JS-Stat* and *Syscalls* datasets only and leaves the *JS-Dyn* dataset out. At first sight the plotted values seem vanishingly low due to logarithmic scale and the overall size of the feature space as denominator. However, the steepness of the density's decay over increasing n-gram lengths is the crucial indicator here. The steeper the decay the better.

Figure 3.7c shows the variability of our datasets. As the variability is normalized to its maximal value, it thus again cannot be determined for data that has a virtually infinite set of unique n-gram tokens such as the dynamic JavaScript reports (*JS-Dyn*). Therefore, this dataset is yet again spared out in the figure for this criteria. For the other datasets (from *JS-Stat* over *HTTP* and *FTP* down to *Syscalls*) a constant decay of the variability levels can be observed. This directly mirrors the difficulty to learn a model based on these datasets.

Binary-based protocols. Protocols and formats that use complex binary structures are not to be categorically ruled out either, only because they happen not to be human-readable and therefore, less intuitively comprehensible. Depending on the specific dataset it might be necessary to parse and generalize the protocol beforehand, though. However, for this study and in line with previous work we apply byte n-grams on the raw payloads, similarly to HTTP and FTP.

In particular, we examine the SMB/CIFS protocol suite and the DNS protocol. SMB is mainly used for the transfer of data, but includes other functionality such as the Windows RPC (*MSRPC*). Our SMB dataset features the special property that data was mainly retrieved from the server and only in a few exceptions uploaded by the client. This allows us to study two different scenarios in the use of SMB: First, the use of largely pure SMB requests sent by the client to the server and only little additional raw data (*SMB-Client*) and second, SMB commands that are heavily interleaved with the transmitted data (*SMB-Server*). DNS, on the other hand, consists of relatively short requests and in a large part of printable characters—the requested domain name. In both cases the basis for a low perturbation is provided due to the bounded alphabet of 256 bytes. Figure 3.8a illustrates the perturbation levels for *DNS* and the SMB datasets. Especially *DNS* quickly reaches zero perturbation, whereas *SMB-Server* requires much more data and also *SMB-Client* needs more training. Note that *SMB-Server* not only stands out clearly, but its peak is displaced with respect to the beginning of the recording. This indicates that in the beginning similarly to *SMB-Client* mainly pure message blocks are exchanged, before at some point the transmission of large amounts of raw data starts. Such raw data often is compressed and therefore exhibits high entropy. Therefore, the uncertainty about the overall observed n-grams increases and obscures the structure of the protocol.

Figure 3.8c shows the variability and reveals the lack of perceptible structure of the *SMB-Server* dataset. *SMB-Client* on the other hand does not fully share this problem. This suggests that learning n-gram models of SMB traffic would be feasible if the interleaved raw data, which constitutes additional noise for the learning algorithm, is excluded. This is equally true for classification as well as anomaly detection, whereby it is especially critical for the latter. One option to restrict the data to a manageable subset is, for instance, to only look at Windows RPC messages (Hadžiosmanović et al., 2012). The density exposes another interesting property of the considered binary protocols. Figure 3.8b shows that for n-grams of lengths up to $n = 3$ the density is particularly high. This happens due to the use of numeric variables as part of the protocol that may span over the entire range of byte values, such as length specifiers for subsequent fields, headers, etc. With larger values of n this influence decreases. *SMB-Client* and the DNS requests even align with the values for the *HTTP* dataset.

In summary, the developed criteria largely confirm previously expressed concerns regarding the suitability of binary protocols and especially SMB (e.g., Hadžiosmanović et al., 2012) for learning. However, at the same time it shows that learning cannot be ruled out categorically. Constraining the use of raw data in SMB can lower the complexity to a feasible level. In Chapter 5 we demonstrate how this can be achieved for proprietary binary protocols in industrial networks. Also DNS requests appear to be very well manageable. Nevertheless, one needs to note that in the case of DNS the type of attacks usually differ from those seen for HTTP and FTP.

3.4.3 Probabilistic Data Structures

In this section, we address the impact of collisions in probabilistic data structures such as Bloom filters and Count-Min sketches. Whenever hash functions are used for referencing elements in these data structures, it is necessary to take the influence of collisions into account: Two or more objects (e.g., n-grams) may by chance result in the same hash value. Depending on the size of the feature space this is more or less likely. Subsequently, we inspect the influence of such collisions on Bloom filters and the perturbation. Moreover, we experimentally evaluate the deviation of true and estimated counts in Count-Min sketches.

Bloom Filters. As described earlier, Bloom filters use multiple hash functions to scatter item markers across the width of the underlying bit array. While this compensates for collisions of a single hash function, false positives are possible. The probability for these strongly depends on the size of the feature space, but also on the overall number of hashed and stored objects. This can be thought as the *saturation* of the hash function's output range and is related to the density criteria described before (see Section 3.2). We measure the saturation of a Bloom filter as the number of bits in the underlying bit array that are set to 1. In the worst case an n-gram model may describe all possible n-grams (when fully saturated) rather than those representing normal or malicious behavior, thus artificially increasing the density of the feature space.

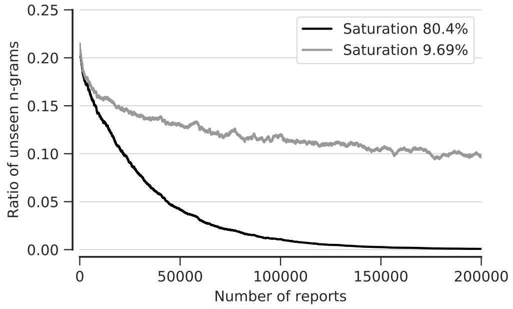
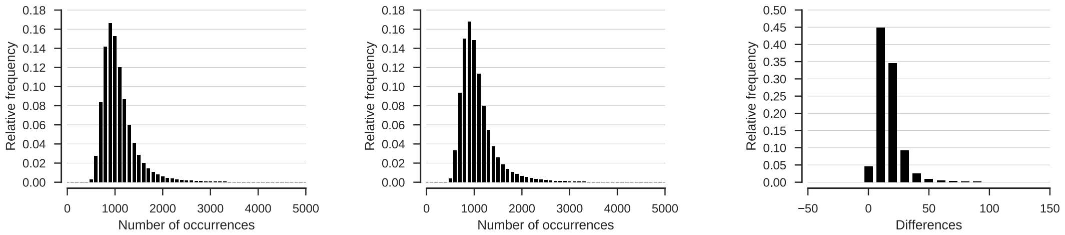


Figure 3.9: Influence of the saturation of a hash function's output range based on the example of 5-grams extracted from the dynamic JavaScript reports (*JS-Dyn*).

Figure 3.9 shows the influence of the saturation of a hash function's output range on the convergence of the dataset's perturbation. The black curve shows a seemingly ideal convergence for *JS-Dyn* (5-grams), whereas the gray one suggests a far from optimal behavior. On closer examination, it however becomes clear that this happens only due to different hash saturation levels of 80.44 % and 9.69 %, respectively. In practice, the size of a Bloom filter hence needs to be chosen so that it does not saturate and collisions are minimized, in order to obtain the most accurate results.

Count-Min Sketches. Similar to Bloom filters, Count-Min sketches also are subject to collisions and the saturation of the data structure is equally critical. In contrast to Bloom filters these however manifest differently. Instead of mere false positives (items wrongly reported as part of the sketch), we are dealing with deviations of the reported value from the true value stored in the sketch. We thus proceed to empirically inspect the differences of exact and probabilistic counting. To this end, we extract all 3-grams from the *SMB* dataset and store them in an hash table, for exact counting, and a Count-Min sketch with probabilistic counting. The latter is parameterized to use $d = 7$ hash functions, with $\varepsilon = 0.0001$ and $\delta = 0.01$, resulting in a width $w = 27{,}183$. Figures 3.10a and 3.10b show the distribution of the relative frequency of occurrences of the n-grams once counted exactly and once probabilistically. Figure 3.10c shows the relative frequency of the difference in value of these counts. Most values differ by roughly 10 to 20 occurrences , which refers to 1 % of the total count for the most prominent input strings. None however are lower than the true value, as guaranteed by design.



(a) Exact counting (b) Probabilistic counting (c) Exact vs. probabilistic counting

Figure 3.10: Histograms of 3-gram occurrences in network traffic for exact and probabilistic counting, and the difference between both approaches.

Chapter 4

Learning Malicious Behavior

As highlighted earlier in Chapter 2, traditional approaches often fall short in detecting threats against end-users as well as computer networks in a timely manner. In this chapter, we thus develop a learning-based detection approach that enables the fully automatic detection of malware. By using language models that generically portray the content as well as the structure of data, no manual intervention is required—neither during training, nor in live operation. We show that with the right representations of the data and the use of probabilistic data structures, learning-based approaches can be both effective and runtime efficient. In order to optimally facilitate detection, however, domain specific analyses and preprocessing are needed.

In particular, we demonstrate the feasibility of effective attack detection of Flash-based malware. We describe the details and peculiarities of this kind of malware in Section 4.1 and outline our method for detecting it in Section 4.2. Due to the complexity of Flash-based malware an extensive analysis for capturing and profiling malicious intent is needed in advance, which we detail in Section 4.3. Our method combines a structural analysis of the container format with guided code-execution of contained program code, that is, a lightweight and pragmatic form of multi-path exploration. While related approaches orient analysis toward normal execution or external triggers, our method actively guides the analyzer towards interesting code regions to maximize its coverage. In Section 4.4, we then show how to learn a classifier using this profiling data and thus spare an analyst from manually constructing detection rules. We present two different approaches: One based on linear Support Vector Machines (SVMs), and another, more simple but faster, approach using Bloom filters . In Section 4.5, we demonstrate the effectiveness of these detectors, before we discuss related work in Section 4.6.

4.1 Use Case: Detecting Flash-based Malware

For years dynamic and multimedia content was equatable to the use of the Adobe Flash platform. This started to change with the upswing and advancement of JavaScript in modern web browsers, and the gradual implementation of the HTML 5 standard (LaForge, 2016). In 2016 about *every fourth* webpage among the Alexa Top 1,000 was still using Adobe Flash for video streaming, gaming, or advertisements (httparchive, 2016). Its unprecedented pervasiveness has naturally attracted adversaries that take advantage of security issues to attack end-users. Unfortunately, the Adobe Flash platform has a long-standing history of severe security vulnerabilities. During the last ten years over 1,000 different vulnerabilities have been discovered in the Adobe Flash Player (Özkan, 2010–2018). In the years 2015 and 2016 alone, the Flash Player has been assigned 595 CVEs, marking an all-time high and a tremendous increase when compared to previous years. In comparison to the 5-year average from 2010 to 2014 the number of publicly documented vulnerabilities has more than quadrupled in both subsequent years. 478 of these vulnerabilities even enable remote code execution and require a user to merely visit a web page to be infected. This growing attack surface provides a perfect ground for miscreants, which is mainly driven by two factors: First, with ActionScript the Adobe Flash platform provides extensive scripting capabilities to handle user input or drive animations, which can equally be used to facilitate exploits. Second, the file format used for Flash animations contains a plethora of special cases that offer a huge attack surface for memory corruption exploits.

Today, two versions of the scripting language are in frequent use, ActionScript 2 and 3, where the latter is a complete redesign based on the ECMAScript standard (ECMA-262) to support object-oriented programming. Alongside with the new language, ActionScript 3 introduces a new virtual machine known as AVM2, whereas ActionScript version 1 and 2 are executed on the initial version of the ActionScript VM (AVM1). The Adobe Flash Player ships both versions to maintain backwards compatibility, such that recent versions may suffer from attacks against the outdated AVM1 to the same extend as earlier versions. The Shockwave Flash (SWF) is common to all versions of ActionScript and Flash, although it has been occasionally extended when new features emerged. Internally, Flash animations are composed of so-called *tags*, containers that are used to store ActionScript code as well as data of various kinds, including audio, video, image, and font data. Current versions of Flash support several dozen different types of tags, some of which occur nested.

The complexity of the SWF file format, a powerful scripting language and the fact that both have evolved over years has led to a large variety of attacks and Flash-based malware in the wild. For instance, several of the reported vulnerabilities have been found to be used for targeted attacks (Zakorzhevsky, 2014) and exploit kits are known to use recently published vulnerabilities as well as zero-day exploits for specially tailored Flash-based malware (Caselden et al., 2015; Trustwave Holdings, 2016).

We can roughly differentiate between three general categories of attacks, that may overlap in practice: First, a piece of malware may be specially crafted to *exploit the Flash Player* during normal processing of its input, which does not necessarily involve the execution of ActionScript. An early example for such a vulnerability is `CVE-2007-0071`, that can be exploited to execute arbitrary code by leveraging an integer overflow caused by a negative "Scene Count" value (Özkan, 2010–2018). Second, by utilizing the rich capabilities of ActionScript, an adversary can further advance the *launching or preparation of exploits* against either the Flash Player or different parts of the browser. `CVE-2015-3113`, for instance, allows to trigger a bug in the Microsoft Internet Explorer using the external interface of ActionScript 3. This can then be used to corrupt the length of a Flash `Vector` object and write outside its initially allocated memory (Caselden et al., 2015). ActionScript may also be used to perform heap spraying (Daniel et al., 2008) or to obfuscate the presence of an attempt to launch an exploit. Third, malware may use ActionScript to fingerprint the execution environment in a first stage of an attack and then, redirect the user to an instance of an exploit kit, serving the actual malware. Adversaries have used this exact scenario in the past, to reach a vast number of victims by distributing malicious advertisements over ad networks (see Ford et al., 2009; Li et al., 2012).

Moreover, malware authors employ different types of obfuscation to hinder analysis by automated systems and human experts (e.g., Linn and Debray, 2003). For malware based on Adobe Flash, three techniques are particularly prominent: First, so-called *staged execution* can be performed. At the first stage the malware only contains functionality to load further code, which in the second stage triggers the payload—a procedure that may be stretched over multiple rounds, potentially using encryption. Second, *source-code obfuscation* by inserting junk code, redirecting the control flow, or the use of rare or invalid instructions is frequently used in practice. Third, malware may *fingerprint the environment* to select a particular exploit for the current version of the platform or withhold its malicious intent if under analysis. In the following, we refer to these techniques were appropriate and provide examples of malware that employs them.

4.2 System Overview

We now present Gordon, our method for the automatic analysis and detection of Flash-based malware (Wressnegger et al., 2016a). The diverse nature of attacks based on Flash requires an analysis method to inspect these animations on different levels. To this end, we implement Gordon by integrating it into different processing stages of Flash interpreters, thereby blending into existing loading and execution processes. Consequently, our detectors takes Flash animations as input in step ① of Figure 4.1. This

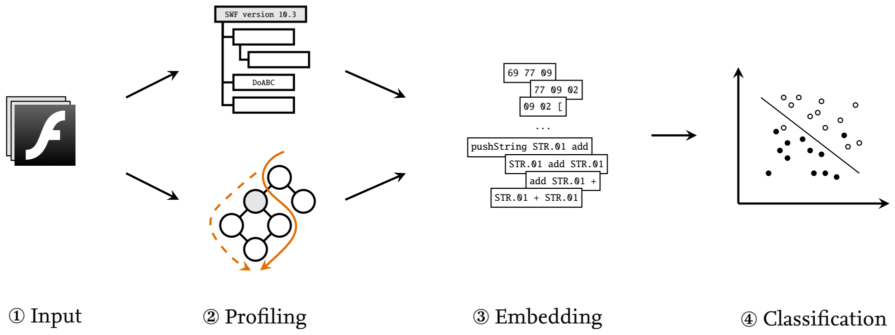


Figure 4.1: Schematic depiction of the analysis and detection process of Gordon with a Flash-based malware as input, the two-step analysis of the profiler and the classification of our method's detectors as output.

tight integration allows us to make use of data generated directly during execution, such as dynamically constructed code or downloaded files. The analysis in step ② is achieved by two complementary profiling components: First, the profiler inspects the hierarchical composition of the Shockwave Flash format. This can be done during the loading phase when the interpreter parses the file for further processing. Second, the control flow of embedded ActionScript code is analyzed in order to determine indicative regions. By strategically changing the control flow at branches in the code, Gordon guides execution along paths covering as much indicative program code as possible. In step ③, we then combine the reports of both analyses into a common malware profile. These profiles can be used by a human expert for analyzing Flash-based malware and decide whether a particular Flash animation is malicious or not. With minor modifications these however are equally appropriate for learning an efficient malware detector. To this end, we strip auxiliary information, that are included to assist the analyst, and transform the malware profile to vectors using language models. Based on these the classifier, indicated in step ④, can then discriminate malicious from benign Flash animations.

Learning Malicious Behavior. The output of the analyses steps provide comprehensive insight into the intent of Flash animations, while language models ensure a suitable representation for machine learning. On top of this, we investigate two different approaches for learning a detector for Flash-based malware and show the feasibility of effectively learning malicious behavior using classification: First, we use Support Vector Machines to establish a baseline using standard learning techniques. Second, we demonstrate how similar results can be achieved using much simpler algorithms based on probabilistic data structures, that we have discussed in Chapter 3, and present a novel two-class classification scheme based on Bloom filters.

4.3 Data Analysis

Gordon's profiler is implemented on the basis of two popular and mature open-source implementations of the Flash platform that are complementary with respect to the versions they support: *Gnash* (Savoye et al., 2005–2018) and *Lightspark* (Pignotti, 2011–2018). While Gnash provides support for Flash up to version 9, Lightspark enables processing version 9 and higher. As a result, Gordon is able to analyze all currently relevant versions and file formats of Adobe Flash animations, including all versions of ActionScript code. The profiling implemented for both interpreters features two kinds of analyses, that in turn make use of data arising during an interpreter's regular loading and execution process (Adobe Systems, 2007): First, a *structural analysis* of the Flash container format which we detail in Section 4.3.1 and second, the *guided execution* of ActionScript code, that is, a lightweight and pragmatic form of multi-path exploration described in Section 4.3.2.

4.3.1 Structural Analysis

We begin our analysis by breaking down Flash animations into *tags*, the primary containers employed by the SWF file format (Adobe Systems, 2013) to store ActionScript code as well as data of various kinds, including audio, video, image and font data. Due to the large number of different types of tags Flash files expose a huge attack surface for memory corruption exploits. As a consequence, many exploits rely on very specific types and arrangements of tags to succeed, and thus, the sequence of tags alone can already serve as a strong indicator for malware.

For the structural analysis, as employed by Gordon, only tag identifiers and structural dependencies are of interest, contained data on the other hand is not considered. Consequently, Gordon does not need to know about the format of individual tags and hence, can also be applied to unknown tags, e.g., tags introduced in future versions. However, to further enhance the overall detection, our method may be combined with approaches to specifically target data formats that can be included in a Flash animation's tags such as the method presented in Chapter 6. Moreover, exploits often rely on corrupt or incomplete tags. To better account for these, we additionally include two specific tag identifiers that mark (a) *incomplete tags*, that is, tags which are known to the interpreter, but could not be correctly parsed and (b) tags that contain *additional data* beyond their specified limits. The latter occurs, for instance, whenever the file contains data at the end that is not fully contained in its last tag.

As a result of this structural analysis, we obtain a sequence of container types including their nesting for each Flash file. Figure 4.2 shows the resulting container listing for a sample[1] of the LadyBoyle malware using `CVE-2015-323`.

```
69   FileAttributes
77   Metadata
 9   SetBackgroundColor
 2   DefineShape
     39   DefineSprite
     26   PlaceObject2
86   DefineSceneAndFrameLabelData
43   FrameLabel
87   DefineBinaryData     // Payload
87   DefineBinaryData     // Payload
82   DoABC                // ActionScript 3 Code
76   SymbolClass
 1   ShowFrame
```

Figure 4.2: Excerpt of the Gordon's structural report for a LadyBoyle malware sample.

In comparison to many other Flash animations, the content of this file is rather short. However, for this specific sample the presence of the `DefineBinaryData` and `DoABC` tags is crucial. The first two contain the malware's payload as binary data, which in turn gets extracted by ActionScript 3 code embedded in the latter. These tags in combination

---

[1] md5: `cac794adea27aa54f2e5ac3151050845`

comprise the malicious functionality of the sample. While in this particular case the structure alone is only an indicator for the malicious behavior, that needs to be backed up by an analysis of the embedded ActionScript code, other types of malware rely on corrupt tags that allow to distinctively distinguish these. Some containers, such as the `DefineShape` tag, allow to enclose an arbitrary number of other containers. We include these in the listing as children of the parent tag. Note that the `DefineShape` tag and its children are not present in the original sample and have been added for illustration purposes only. For convenience, the structural report can also be represented as a sequential list of identifiers, where nested containers are indicated by brackets:

```
69 77 9 2 [ 39 26 ] 86 43 87 87 82 76 1
```

It is important to note, that this representation already encodes the complete hierarchy and relations of the tags to each other. This condensed form is particularly suitable for automated approaches that do not require a textual description of the tags. We revisit this topic when discussing the implementation of GORDON's detectors in Section 4.4.

4.3.2 Guided Code-Execution

When analyzing a sample with GORDON we aim at observing as much indicative behavior of a Flash animation as possible—ideally the analysis covers all possible execution paths and corner cases. However, as extensively discussed in computer security literature in the past (e.g., Moser et al., 2007b; Kolbitsch et al., 2012) this is not feasible due to the potentially exponential number of different paths, making it necessary to revert to approximations and heuristics in practice.

While related approaches orient analysis towards normal execution (Wilhelm and Chiueh, 2007; van Overveldt et al., 2012; Kolbitsch et al., 2012) or external triggers (Crandall et al., 2006; Moser et al., 2007b; Brumley et al., 2008), our method guides execution towards *indicative code regions*: Each branch is chosen such that the execution corresponds to the path that covers the most indicative ActionScript code not observed so far. In particular, we are interested in exploring paths containing security-related objects and functions as well as branches that contain more code than others. Figure 4.3 exemplarily shows the selected paths of two consecutive runs. During the first, GORDON's profiler guides execution towards the `loadMovie` function, which enables Flash animations using ActionScript 2 to dynamically load code in form of another SWF file. The second run then directs the interpreter along the path covering the most bytecode instructions. This strategy can hence be seen as a way to not only maximize code coverage locally (within the sample itself), but globally, including all code that is loaded dynamically.

This is made possible by inspecting the control flow of the ActionScript code contained in a Flash file to learn (a) how much code can be covered along a specific path, and (b) where security-related objects and functions, such as the aforementioned `loadMovie` function, are located. To this end, we first derive the control-flow graph of the ActionScript bytecode and

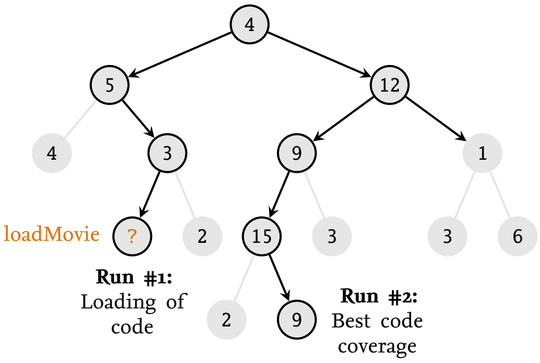


Figure 4.3: Illustration of the path-selection strategy. Node labels correspond to the amount of bytecode instructions in each basic block. Black lines indicate chosen execution paths. Figure taken from Wressnegger et al. (2016a).

remove cycles induced by loops and recursive function calls. Second, the resulting graph is annotated with locations of indicative functionality and the number of instructions contained in each branch, which in turn enables us to efficiently determine the overall code coverage of individual paths. The results of this analysis is then used for the actual execution of the Flash animation, allowing Gordon to navigate through the code in a targeted way.

Control-Flow Analysis. A control-flow graph (CFG) as shown in Figure 4.4 contains basic code blocks as its nodes and directed edges for branches connecting them (see Aho et al., 2006; Allen, 1970). As part of the Adobe Flash Player's verification phase, the ActionScript VM already checks certain control flow properties when bytecode is loaded into the interpreter (Adobe Systems, 2007). Our control-flow analysis can thus be thought of as a natural extension to the examinations conducted by Flash interpreters. We, however, make use of this information only as a starting point for the following analysis.

Upon the generation of a CFG, we are ready to find execution paths that maximize code coverage. To easily determine these paths, the graph needs to first undergo a few modifications. In particular, it is necessary to eliminate cycles that occur due to loop statements in the code. Once these cycles are removed, we obtain an *acyclic control-flow graph (ACFG)* which allows us to efficiently determine the code size of complete paths in the graph. To this end, we rewrite all *back-edges* (edges pointing backwards with respect to the control flow) by linking them to the first code block after the loop. Figure 4.4 demonstrates this for a simple `while` loop. All conventional loops, nested loops and their special cases such as *unnatural loops* can be efficiently resolved using the dominance relations of the individual nodes (see Aho et al., 2006).

Annotating Control-Flow Edges. Once an ACFG has been generated, we annotate each of its edges with the number of bytecode instructions covered by the following code block. We artificially increase the weight of individual instructions, if they correspond to security-related objects and functions. For example, to pinpoint the dynamic loading of code, we set the weighting for calls to the `loadMovie` function (ActionScript 2) and the `Loader`

```
var a:int = 294;
var b:int = 1722;

while (b != 0)
{
  var h:int = a % b;
  a = b; b = h;
}
trace(a);
```

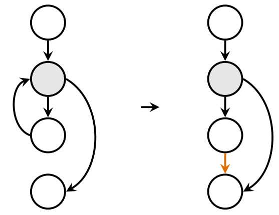


Figure 4.4: An ActionScript 3 code snippet, the corresponding control flow graph (CFG) and its acyclic transformation (ACFG). Dark nodes represent loop headers, while bright nodes depict generic code blocks. Newly inserted edges are shown in orange. Figure taken from (Wressnegger et al., 2016a).

object (ActionScript 3) to the maximum to ensure the analyzer targets these first. Both are frequently used by Flash-based malware to load code downloaded from the Internet or dynamically assembled at runtime. Similarly, it is possible to emphasize other security-related functions and objects in ActionScript, such as `readBytes` and `ByteArray`, which are often used for obfuscated code.

Given the annotated graph, the search for the most indicative code regions can be rephrased as a *longest-path problem*. For arbitrary graphs determining the longest path is NP-hard. Fortunately, for directed acyclic graphs such as the ACFG extracted previously, this perfectly feasible (see Cormen et al., 2009; Sedgewick and Wayne, 2011).

Path Exploration. With the annotated ACFG on hand, we can now guide the interpreter to execute security-related or large code regions by stopping at every conditional jump and choosing the branch corresponding to the path with the highest weight. In order to avoid executing indicative code unnecessarily often, we constantly update visited regions within the ACFG. Moreover, GORDON enables multiple executions based on the coverage analysis of previous runs. Hence, a different path is taken and different code regions are visited in each run, thereby challenging adversarial attempts to hide payload in paths that are not covered initially. As analysis output of the guided execution, we obtain all covered ActionScript instructions across multiple execution runs. Figure 4.5 shows a short excerpt of the instructions executed by a malware to facilitate an exploit[2] in the first run (R1). Instructions at offset 973 to 983 show how the malware obfuscates the usage of the `ByteArray` object at offset 984. This object is frequently used to construct malicious payloads at run-time. The complete listing shows how the encrypted payload is composed out of different parts, decrypted and finally loaded.

Subsequently, we address certain implementation details of GORDON's guided code-execution with a special focus on the characteristics of Flash-based malware and potential adversarial attempts to avoid analysis.

---

[2] md5: `4f293f0bda8f851525f28466882125b7`

```
R1 973: pushString   "fla"
R1 975: pushString   "sh.uti"
R1 977: add          "fla" + "sh.uti"
R1 978: pushString   "ls.Byt"
R1 980: add          "flash.uti" + "ls.Byt"
R1 981: pushString   "eArray"
R1 983: add          "flash.utils.Byt" + "eArray"
R1 984: callProperty [ns:flash.utils] getDefinitionByName 1
R1 >    Looking for definition of [ns:flash.utils] ByteArray
R1 >    Getting definition for [ns:flash.utils] ByteArray
R1 987: getLex: [ns:] Class
```

Figure 4.5: Excerpt of Gordon's behavioral report.

Reducing Branch Candidates Although Gordon is capable of pursuing all branches in ActionScript code, narrowing down the candidates speeds up the process and limits the possibility of breaking the semantics of a sample. Often, web-based attacks are tailored towards specific browser environments and thus, only trigger malicious activity upon checking for the correct target environment (Kolbitsch et al., 2012; van Overveldt et al., 2012). The conditional jumps underlying these checks provide excellent candidates for our guided execution, as they usually lead to a malware sample's payload and are likely to be mutually exclusive, therefore reducing the risk of semantic side-effects.

To restrict our analysis to these conditional jumps, we implement a taint-tracking mechanism that propagates taint from environment-identifying data sources to conditional jumps. In the scope of Flash-based malware, such data typically originates from the `System.capabilities` and `flash.system.Capabilities` data structures available in ActionScript 2 and 3, respectively. To track the data flow across built-in functions, we conservatively taint the result whenever at least one of the input arguments is tainted. Note that for simplicity, we do not consider implicit data-flow and control dependencies in our implementation (see Cavallaro et al., 2008; Nair et al., 2008) but leave this for future work.

Countering Obfuscation To account for dynamically loaded code, we additionally hook the interpreter's loading routines. All such code then passes through the same analysis steps as the host file, allowing to analyze files downloaded from the Internet as well as potentially encrypted code embedded in the Flash animation itself equally thoroughly. This scheme is applied recursively to ensure that all code is covered by our analysis.

Furthermore, Gordon implements an *adaptive timeout* mechanism rather than a fixed period of time as employed in previous works (Ford et al., 2009; Cova et al., 2010; van Overveldt et al., 2012). In particular, we reset a 10 s timer each time the sample attempts to load code, giving the sample time to react to this event. This may increase the analysis duration for certain files but significantly reduces the effort for those that do not load data or do not contain ActionScript code at all. On average a sample is executed for 12.6 s with a maximum duration of 3 min, reducing the analysis time by 93 % compared to a fixed timeout.

We also take precautions for the possibility that an execution path is not present in the statically extracted ACFG. In these rare cases, we switch to determining the size of the branch in an online manner: GORDON looks ahead in order to inspect the instructions right after the branching point and passively skips over instructions to determine the sizes of the branches. This analysis in principle is the same as performed earlier (Section 4.3.2) but applied to the newly discovered pieces of bytecode only.

Lastly, we have observed an increase in the use of event-based programming in recent malware—presumably to circumvent automatic detection—and thus incorporate the automatic execution of such events into GORDON's profiler. Immediately after an event listener is added the specified function gets passed an appropriate dummy event object and is executed without waiting for the actual event to happen.

Updating the ACFG Our method is designed to run a sample multiple times. To this end, we update the edge labels of the ACFG during execution to reflect the visited code and recompute the largest path in an online manner. Consequently, our method implements a lightweight variant of multi-path exploration that executes different code during each run. Since we decide on each condition at runtime, identical code regions (functions) may be referenced multiple times. We hence not only cover the code of the single largest path in the graph, but potentially a combination of a number of paths. This softens the definition of such a path as used in graph theory but makes sense in this scope.

4.4 Classification

In order to demonstrate the expressiveness of our analysis, we implement two learning-based detectors, that are trained on known benign and malicious Flash animations. This approach spares us from manually constructing detection rules, yet it requires a comprehensive dataset for training (see Section 4.5.1). In this section, we propose two approaches: In Section 4.4.1, we first outline a detector using Support Vector Machines, a well-established method from the field of machine learning. Moreover, we discuss how the learned feature weights can be efficiently used with Count-Min sketches in practice. Second, we demonstrate another classifier based on n-gram statistics that seamlessly fits a representation as Bloom filter in Section 4.4.2. We refer to these two variants as GORDON-SVM and GORDON-TCA, respectively. However, before these can be applied, we need to map the analysis output of GORDON to vector space. To this end we quickly recap the description on language models from Chapter 3.

Vector Space Embedding To embed the structural and behavioral reports generated by GORDON in a vector space, we make use of language models. In particular, we extract *token n-grams* from both kinds of analysis outputs. While the compact output representation of GORDON's structural analysis already is in a format that can be used to extract such tokens, the reports generated by the guided code-execution need to be normalized first: We extract all instructions, including their names and parameters. Moreover, in order to contain the perturbation of the data (see Section 3.2), we additionally replace values passed as parameters with their respective type, such as `INT`, `FLOAT` or `STR`. To avoid losing relevant information, we however preserve all names of operations, functions and objects. Finally, we tokenize the behavioral reports using white-space characters. High-order n-grams compactly describe the content, implicitly reflecting the structure of the reports and can be used for establishing a joint map to a vector space. To this end, we embed a Flash animation a using mapping ϕ in a vector space spanned by the set S of all possible n-grams. Each dimension in this vector space is associated with the presence of one n-gram $s \in S$. Analogous to Chapter 3, we define two instances of such a mapping, φ_1 and φ_2, as

$$\varphi_1 : \mathcal{F} \to \{0,1\}^{|S|}, \quad a \mapsto \Big(\mathrm{bin}(s,a)\Big)_{s\in S} \qquad\qquad \varphi_2 : \mathcal{F} \to \mathbb{N}^{|S|}, \quad a \mapsto \Big(\mathrm{cnt}(s,a)\Big)_{s\in S}$$

where $\mathcal{F}$ is the set of all possible Flash animations and the function $\mathrm{bin}(s,a)$ returns 1, if the n-gram s is present in the analysis output of the file a and 0 otherwise. Function $\mathrm{cnt}(s,a)$, on the other hand, returns the number of occurrences of n-gram s in a. These mappings can be efficiently calculated for large amounts of data in linear time (Rieck et al., 2012). In the following, we use φ_1 and φ_2 for classification in Section 4.4.1 and Section 4.4.2, respectively.

4.4.1 Classification using Support Vector Machines

For our first detector we employ a *linear Support Vector Machine* (SVM) that uses the previously defined vector space embedding for learning a classification between benign and malicious Flash animations. While several standard learning algorithms can be applied in this setting, we choose linear SVMs as our baseline for their excellent generalization capability and very low run-time complexity, which is linear in the number of objects and features (Schölkopf and Smola, 2002; Fan et al., 2008).

In short, a linear SVM learns a hyperplane that separates two classes with maximum margin—in our setting these correspond to vectors of benign Flash animations and Flash-based malware. The orientation of the hyperplane is expressed as a normal vector $\mathbf{w}$ in the input space and thus, an unknown sample a can be classified using an inner product as

$$\mathrm{f} : \mathcal{F} \to \mathbb{R}^{|S|}, \quad a \mapsto \langle \mathbf{w}, \phi(a) \rangle - t$$

where t is a threshold and $f(a)$ the orientation of $\phi(a)$ with respect to the hyperplane. That is, $f(a) > 0$ indicates malicious content in a and $f(a) \leq 0$ corresponds to benign content. Additionally, we set $\phi = \varphi_1$ for our detector, Gordon-SVM. Due to the way the mapping of n-grams is defined, the resulting vectors are sparse: Out of millions of possible token n-grams, only a limited subset is present in a particular sample a. These vectors can thus be compactly stored in memory and the inner product to determine the final score can be calculated in linear time in the number of n-grams in a sample:

$$\langle \mathbf{w}, \varphi_1(a) \rangle = \sum_{s \in S} w_s \operatorname{bin}(s, a) = \sum_{s \text{ in } a} w_s.$$

Count-Min sketches are not limited to storing integers, but may equally handle floating point values. We can thus use them to efficiently store a SVM's weight vector $\mathbf{w}$ for its use in practice. Data structures that allow to query values in constant time enable to further optimize the calculation of $f(a)$ by avoiding to explicitly construct and maintain those vectors in memory.

4.4.2 Classification using n-gram Statistics

In the scope of this thesis we additionally explore classification based on n-gram statistics. Prior work on language models for intrusion detection has shown that recording known n-grams during training and counting their number of occurrences during testing is remarkably efficient for detecting anomalies in network traffic (Wang et al., 2006a). We extend this scheme to two-class classification by learning separate models M^+ and M^- for benign and malicious samples, respectively, and combining the resulting similarity scores as weighted difference.

Learning such a model M^* of known n-grams can be expressed as the union of all subsets of n-grams contained in the individual samples a in our training dataset D^*

$$M^* = \bigcup_{a \in D^*} \{s \in S | s \text{ in } a\}.$$

We use superscript $*$ to specify the class (benign $+$ or malicious $-$) and additionally define a weight vector $\boldsymbol{\omega}^*$ that constitutes the contents of model M^* as a binary vector,

$$\boldsymbol{\omega}^* = \Big(\operatorname{contains}(s, M^*)\Big)_{s \in S}$$

where function $\operatorname{contains}(s, M^*)$ returns 1, if n-gram s is contained in the model M^* and 0 otherwise. This definition is very similar to the binary embedding φ_1, but applied to an entire set of Flash animations. Again, these models can conveniently be stored in probabilistic data structures such as Bloom filters. Wang et al. (2006a) propose to use the portion of known n-grams to the total number of n-grams as similarity score. Formally, this can be expressed as the inner product of a model's weight vector $\boldsymbol{\omega}^*$ and the previously defined count embedding φ_2:

$$\begin{aligned} \operatorname{score}_{M^*} \colon \mathcal{F} &\to [0, 1], \\ a &\mapsto \langle \boldsymbol{\omega}^*, \varphi_2(a) \rangle. \end{aligned}$$

For two-class classification, as used in our second detector GORDON-TCA, we now compute the weighted difference of the scores of both classes

$$\begin{aligned} g\colon \mathcal{F} &\to \mathbb{R}, \\ a &\mapsto (1-\gamma)\ \mathrm{score}_{M^+}(a) - \gamma\ \mathrm{score}_{M^-}(a) \end{aligned}$$

where $\gamma \in [0,1]$ is a factor to vary the weights of the individual classes. A factor of $\gamma < 0.5$ gives more weight to the first class (benign samples) and $\gamma > 0.5$ shifts focus to the second class (malicious samples). γ can thus be seen as a parameter for how large the influence of the additional second class should be.

4.5 Evaluation

We proceed to empirically evaluate the capabilities of GORDON in different experiments, based on an extensive dataset of Flash-based malware and benign animations, which we introduce in Section 4.5.1. In particular, in Section 4.5.2 we start by studying the effectiveness of the guided execution in terms of code covered. Next, we compare the detection performance of our method using SVMs as well as the classifier based on n-gram statistics with related approaches in Section 4.5.3 and inspect the runtime performance of both approaches in Section 4.5.4. Finally, we additionally demonstrate the effectivity of GORDON in a temporal evaluation over multiple weeks of operation in Section 4.5.5.

4.5.1 Dataset Composition

The dataset for our evaluation has been collected over a period of 12 consecutive weeks. In particular, we have been given access to files from the VirusTotal service, thereby receiving benign and malicious Flash files likewise. Since many web crawlers are directly tied to VirusTotal, the collected data reflects the current landscape of Flash usage to a large part.

We split our dataset into malicious and benign Flash animations based on the classification results provided by VirusTotal two months later: A sample is marked as malicious, if it is detected by at least three scanners and flagged as benign, if none of the 50 scanners hosted at VirusTotal detects the sample. Samples that do not satisfy one of the conditions are discarded. This procedure enables us to construct a reasonable estimate of the ground truth, since most virus scanners refine their signatures and thus, improve their classification results over time. The resulting dataset comprises 1,923 malicious and 24,671 benign Flash animations, with about half of the samples being of version 8 or below and the other half of more recent versions, therefore handled by the ActionScript VM version 1 and 2 respectively. A summary of the dataset is given in Table 4.1.

To account for the point in time the samples have been observed, we group the samples in buckets according to the week of their submission to VirusTotal. Consequently, we obtain 12 sets containing benign and malicious Flash animations corresponding to the 12-week evaluation period. These temporal sets are used during the evaluation to construct temporarily disjoint datasets for training and testing to conduct our experiments in strict chronological order: For our experiments the performance is determined only on samples that have been submitted to VirusTotal after any sample in the training data. This ensures an experimental setup as close to reality as possible and demonstrates the approach's effectivity of providing timely protection.

Classification	**AVM1**	**AVM2**	**Total**
Malicious	864	1,059	1,923
Benign	12,046	12,625	24,671
Total	12,910	13,684	26,594

Table 4.1: Overview of the evaluation dataset.

4.5.2 Coverage Analysis

In our first experiment, we evaluate the effectiveness of the proposed guided code-execution strategy. To this end, we investigate the code coverage of malware samples in our 12 week dataset. We apply Gordon to the malware and inspect the output of the interpreter. Due to obfuscation techniques employed by malware, the amount of *statically* contained code of a Flash file often is not a reliable measure in this setting. Hence, we compare the number of executed instructions with respect to a regular execution of the samples. With the path-exploration strategy employed by Gordon, we manage to oberserve over 50 % more ActionScript code than during a naive execution, and unveil crucial information not provided otherwise. We mainly credit this leap in coverage to the recursive analysis of dynamically loaded code and code assembled at runtime.

4.5.3 Comparative Evaluation

We continue to evaluate the detection performance of both classifiers of Gordon, showing their ability to correctly classify Flash-based malware and specifically compare them to FlashDetect[3] (van Overveldt et al., 2012). In particular, we evaluate the approaches on the complete set of 12 consecutive weeks, where we use *weeks 1–6* for training and *weeks 7–9* for validation to calibrate the parameters of the detectors. We then combine these two sets for final training and apply the detectors to *weeks 10–12* for testing the detection performance. Table 4.2 summarizes the results as the true-positive rates (TPR) and the corresponding false-positive rates (FPR) of the methods.

---

[3] Versions not supported by FlashDetect (version 8 and below) have been excluded.

Method	FlashDetect[3]	Gordon-TCA		Gordon-SVM	
FPR	1 %	1 %	0.1 %	1 %	0.1 %
TPR	26.5 %	89.7 %	77.7 %	95.2 %	90.0 %

Table 4.2: Detection rates of FlashDetect, Gordon-TCA, and Gordon-SVM.

Each of these methods has a different set of parameters that need to be adjusted during training. In machine learning this commonly is called, hyperparameter optimization and is implemented as simple grid search in our experiments. For instance, for Gordon-SVM we adjust the SVMs penalty parameter $C \in [10^{-3}, 10^{3}]$ and different weightings of it for the individual classes (Fan et al., 2008). Additionally, for preprocessing we validate the length of n-grams and whether or not to scale (L_2-normalize) the feature vector (Rieck et al., 2012). In comparison, Gordon-TCA requires less optimization as only the length of n-grams, the weighting $\gamma \in [0, 1]$ and the details of the Bloom filter can be set. The latter however are fixed to a size of 2^{27} bit and 3 hash functions (see Chapter 3). FlashDetect, in turn, does not require any parameters by design (van Overveldt et al., 2012).

Gordon-SVM. As described in Section 4.4 Gordon's detectors can be applied to either the analysis outputs individually or to the joint representation of both. For the variant using Support Vector Machines, we evaluate the relation thereof in an experiment in which we use 4-grams to map the reports to vector space. Figure 4.6 shows the detection performance for detectors using each representation individually (light/dark gray) and the combination of both (black) as ROC curves, with the true-positive rate on the y-axis over the false-positive rate on the x-axis. At a false-positive rate of 0.1 % the individual representations attain a detection rate of 60–65 %. The combination of both increases the detection performance significantly and enables spotting 90.0 % of the Flash-based attacks. If the false-positive rate is increased to 1 %, our method even detects 95.2 % of the malicious samples in our dataset. Additionally, we break down this results by CVE numbers.

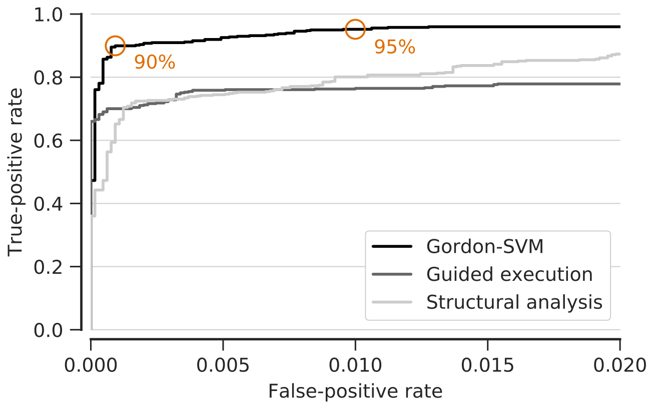


Figure 4.6: Detection performance of Gordon-SVM as ROC curve.

The average performance is slightly below the overall detection rate, indicating that we also detect malware that does not carry exploits itself, but facilitates a different attack or uses obfuscation to obscure the presence of an exploit. This perfectly demonstrates the capabilities of our approach: First, the complementary views on the behavior and structure of Flash animations provide a good basis for analyzing attacks and, second, this expressive representation can be effectively used for detecting malware in the wild.

Gordon-TCA. The detection variant using the weighted difference in n-gram statistics of the two opposing classes is not able to fully reach the performance set by Gordon-SVM. As it operates based on a much simpler concept than SVMs, which in turn solve a complex optimization problem for separating both classes, this is not entirely surprising. However, by detecting 89.7 % of the malicious samples at a false-positive rate of 1 % it performs reasonably well. Figure 4.7 shows a direct comparison of the ROC curves. As weighting factor we have used $\gamma = 0.55$ and hence, a slightly stronger focus has been put on penalizing malicious components. In contrast to Gordon-SVM, we here use 5-grams for embedding the reports. We observe that the detection performance increases with the size of n, meaning that the n-grams become more specific to the data at hand. With $n = 3$ and $n = 4$, for instance, an otherwise identical parametrization detects 81.1 % and 85.4 % of the malware, respectively. This effect has been discussed in the field of intrusion detection before (Tan and Maxion, 2002) and lies in the very nature of methods mining and storing large amounts of features. A possible negative impact can be contained by carefully choosing a large and diverse dataset to ensure the detector generalizes well.

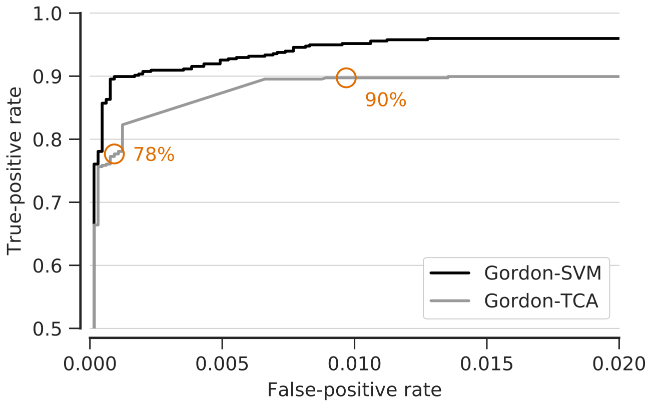


Figure 4.7: Detection performance of Gordon-SVM and Gordon-TCA as ROC curves.

FlashDetect. For the related method FlashDetect we slightly modify the setting and exclude Flash animations of versions below 9 from the evaluation, as this detector is dedicated to the analysis of ActionScript 3 malware only. Nevertheless, FlashDetect only identifies 26.5 % of the malicious Flash samples at a false-positive rate of 1 % and thus cannot keep up with either Gordon-TCA nor Gordon-SVM. Although FlashDetect employs a heuristic for eliciting malicious behavior during the execution of a Flash animation, it misses 3 out of 4 attacks. We attribute this low performance to two issues: First, compared to our method the employed branch selection strategy is less effective and second, the method has been tailored towards specific types of attacks which are not prevalent anymore. GORDON in contrast does not rely on manually selected features, but models the underlying data using n-grams. Therefore, it can better cope with the large diversity of today's malware.

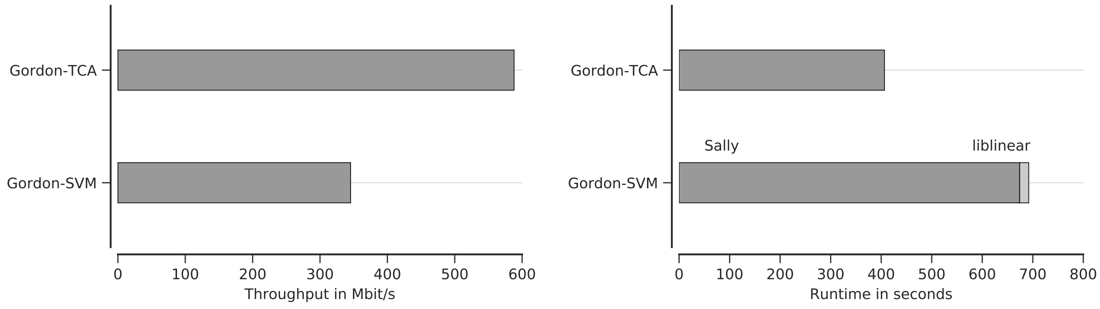
AV Engines. We finally determine the detection performance of 50 virus scanners on the testing dataset. The five best scanners detect 82.3–93.5 % of the malicious samples. However, due to the very nature of signature-based approaches they provide detection with practically no false positives. If we parameterize GORDON-SVM to zero false positives only 47.2 % of the malware is detected. This clearly shows, that GORDON cannot compete with manually crafted signatures in the long run, but provides solid detection of Flash-based malware shortly after its first occurrence in the wild *without* the need for manual analysis.

As a consequence, we consider our method a valuable tool for improving the analysis of Flash-based malware in the short run and a way to provide traditional approaches with a good starting point in day-to-day business to efficiently craft signatures for AV products.

4.5.4 Runtime Evaluation

Next, we compare the two variants of GORDON in terms of runtime. Both approaches use linear time algorithms for learning a classification of malicious and benign Flash animations. The underlying methods however differ fundamentally: While GORDON-TCA pursues an integrated approach that preprocesses the data and simultaneously learns the model on-the-fly, GORDON-SVM performs these two steps separately using two highly optimized tools: *Sally* (Rieck et al., 2012) for embedding the data in vector spaces, and *liblinear* (Fan et al., 2008) for learning the Support Vector Machine.

For this experiment we measure the runtime performance of the full toolchain single-threaded on an Intel Xeon E5-1650 CPU at 3.50 GHz. In order to compensate for disk I/O



(a) Throughput (b) Runtime

Figure 4.8: Throughput and runtime of GORDON-TCA and GORDON-SVM during training.

we however operate entirely in internal memory. Figure 4.8a shows that with 588 Mbit/s Gordon-TCA yields an 1.66× larger throughput than Gordon-SVM. Looking at the runtime for both methods in Figure 4.8b reveals that learning the linear SVM surprisingly is not the limiting factor, but the preprocessing is.

In combination with the lower number of parameters and the therefore reduced cost for hyperparameter optimization, Gordon-TCA, clearly is superior in terms of runtime during training. For testing, however, both methods essentially perform a single vector multiplication and thus are comparable in runtime. In summary, one may thus strike a balance between training performance and the detection performance yield in production.

4.5.5 Temporal Evaluation

To demonstrate the use of Gordon as a fast, complementary detector, we study its performance over several weeks of operation. For this experiment we focus on Gordon-SVM due its superior performance and thus make use of 4-grams. Moreover, we again operate on 12 consecutive weeks of collected Flash data, but this time we apply the detector one week ahead of time, that is, we classify one week after the other, based on the previous weeks.

We start off with *weeks 1* as training, *weeks 2* as validation and *weeks 3* as first test dataset. Over the course of the experiment we shift the time frame forward by one week and gradually extend the training dataset. This can be seen as expanding a window over the experiment's period of time. Hence, Gordon's detector accumulates more and more data for training—just as a system operating in practice would. In order to optimally foster complementary approaches, we choose a liberal false-positive rate of 1 %. Figure 4.9 shows the true-positive rates achieved by our method during 10 weeks of operation.

Gordon starts off below its average performance and takes time until *weeks 5* to perform well, reaching detection rates between 80 % and 99 % for the remaining weeks. As our

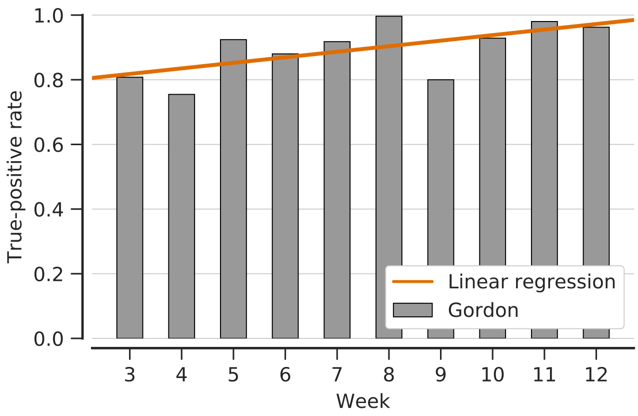


Figure 4.9: Gordon's performance over 12 consecutive weeks. The red line illustrates the detector's progression over time, showing a clear uptrend towards its optimal performance.

method makes use of machine learning techniques, the detector requires a certain amount of training data before it is fully operational and reaches its optimal performance. If parameterized to 0.1 % false positives, Gordon still reaches detection performances of 82 % on average.

Overall, this experiment shows Gordon's potential to improve on the detection performance shortly after a malware's appearance in the wild. We consider the number of false positives—benign samples that need to be additionally analyzed without directly resulting in a malicious signature—as tolerable trade-off for the leap taken in short-term detection performance. In practice, one may start off with a rather strict configuration, accept a lower gain and scale up the interval according to available resources.

4.6 Related Work

A large body of research has dealt with the detection and analysis of web-based attacks, yet Flash-based malware has received only little attention so far. In this section, we discuss work related to Gordon, focusing on two strains of research: (1) Flash-based malware and (2) multi-path exploration.

Note that the implementations of JavaScript and ActionScript interpreters are fundamentally different, making an application of detection approaches for malicious JavaScript *source-code* unlikely to operate on Flash-based malware available in *bytecode*. Consequently, we do not discuss approaches for malicious JavaScript code in this paper and refer the reader to a wide range of research (Ratanaworabhan et al., 2009; Cova et al., 2010; Canali et al., 2011; Kolbitsch et al., 2012; Kapravelos et al., 2013). Nonetheless, combining detection methods for malicious JavaScript and Flash can be of considerable value. Also, the work by Šrndić and Laskov (2013) is of particular interest, since they have been the first to show the practicality of using hierarchical document structure for detection.

4.6.1 Flash-based Attacks and Malware

Only few works have studied means to fend off malware targeting the Adobe Flash platform (Ford et al., 2009; van Overveldt et al., 2012). OdoSwiff (Ford et al., 2009), focuses on detecting malicious ActionScript 2 Flash advertisements based on expert knowledge of prevalent attacks. In contrast to OdoSwiff, our method employs machine learning to automatically produce a classifier based on benign and malicious Flash animations. FlashDetect (van Overveldt et al., 2012), the successor of OdoSwiff also makes use of machine learning techniques and, similar to Gordon, employs an instrumented interpreter to dump dynamically loaded code. However, FlashDetect only pursues one level of staged-execution, focuses solely on ActionScript 3 and employs a simple heuristic for subverting environmental checks that has proven insufficient for modern Flash-based malware. By contrast, Gordon aims at maximizing the coverage of indicative code regions independent of particular attacks, across multiple stages and versions. As a result, our method allows to uncover vitally more code than FlashDetect and thereby attains a better basis for detecting attacks. Furthermore, by not relying on hand-crafted features Gordon can better cope with the large diversity of today's malware.

Industry research has mainly focused on instrumenting Flash interpreters for analysis purposes. Wook Oh (2012), for instance, presents methods to patch ActionScript bytecode to support function hooking, and more recently, Hirvonen (2014) introduces an approach for instrumenting Flash based on the Intel Pin Platform. These systems complement Gordon and may be used to implement our method for other platforms.

Aside from Flash-based malware and therefore, orthogonal to Gordon, several authors have inspected the malicious use of Flash's cross-domain capabilities (Johns and Lekies, 2011), its vulnerability to cross-site scripting attacks (van Acker et al., 2012) and the prevention of such (Louw et al., 2010).

4.6.2 Multi-path Exploration

Ideally an analysis covers all possible paths and corner case, which however is not feasible due to the potentially exponential number of different execution paths. Most notably in this context is the work by Moser et al. (2007b), who propose to narrow down analysis to paths influenced by input data such as network I/O, files or environment information. While this effectively decreases the number of paths to inspect, it still exhaustively enumerates all paths of this subset under investigation. A second strain of research has considered symbolic execution for the analysis of program code and input generation (e.g., Cova et al., 2006; Saxena et al., 2010). Brumley et al. (2008) combine dynamic binary instrumentation with symbolic execution to identify malware behavior triggered by external commands. Similarly, Crandall et al. (2006) use symbolic execution to expose specific points in time where malicious behavior is triggered. Equally to the enumeration of paths, symbolic execution shares the problem of an exponential state space.

With Rozzle, Kolbitsch et al. (2012) also make use of techniques from symbolic execution. However, instead of generating inputs, data in alternative branches is represented symbolically and, upon subsequent execution of both branches, merged. In doing so, Kolbitsch et al. accept to break existing code due to the execution of infeasible paths. Based on the symbolic representation Rozzle likewise is subject to an exponential state space that is dealt with by limiting the depth of the symbolic trees used. Limbo (Wilhelm and Chiueh, 2007) avoids this kind of state explosion and reverts to a more simple strategy of forcing branching conditions to monitor execution. Limbo, however, again exhaustively enumerates paths and thus does not address the underlying problem in the first place.

All these methods are either driven by the original execution path (Wilhelm and Chiueh, 2007; Kolbitsch et al., 2012) or focus on external triggers (Crandall et al., 2006; Moser et al., 2007b; Brumley et al., 2008). Gordon, on the other hand, first identifies indicative code regions and guides the interpreter towards these, enabling a payload-centric analysis. X-Force (Peng et al., 2014) however pursues a very similar idea of forced execution as we do, but focuses on reliably enumerating paths to build comprehensive graph representations of the code. While GORDON records a small number of correctly executed traces, X-Force enumerates as many as necessary to build these graphs and also forces execution beyond errors. With J-Force, Kim et al. (2017) extend this idea to the analysis of JavaScript and attacks facilitated by exploit kits.

Chapter 5

Learning Normality

Learning-based approaches hinge on the availability of sufficient data for training. Suitable and sufficient training data however does not exist across fields of applications alike. While for malware detection, we are able to make recourse to large collections of malware as well as benign software, this is not necessarily the case for network-based attack detection, for instance. On network level benign traffic significantly outweighs recordings of genuine attacks. Learning a classification on such one-sided data hardly is feasible and would result in poor detection performances. Instead, with anomaly detection it is possible to make a virtue out of this limitation and construct a model of normality based on benign data only and identify attacks as deviations thereof.

In this chapter, we show how language models can be used for anomaly detection. To this end, we consider traffic from industrial control systems (ICS) and the detection of attacks therein as a use case, which we detail in Section 5.1. At this, we investigate attack detection in a particular challenging environment that heavily relies on (proprietary) binary protocols with high-entropy data. In Section 5.2, we then provide a brief overview of our method before describing its core components in detail: First, we present a linear-time algorithm for clustering network messages based on Count-Min sketches and introduce the concept of prototype models in Section 5.3. These models are prototypical representations specific to do not only characterize the structure but also the data that is typically contained. Second, in Section 5.4 we then show how prototype models enable the implementation of effective anomaly detection by pruning out irrelevant, noisy features that arise from the intermingling of structure and data in binary protocols. This enables us to further enhance the expressiveness of the models and thus allows for robust anomaly detection in environments with high-entropy data. In Section 5.5 we then evaluate our detector and discuss related work in Section 5.6.

5.1 Use case: Detecting Intrusions in SCADA Systems

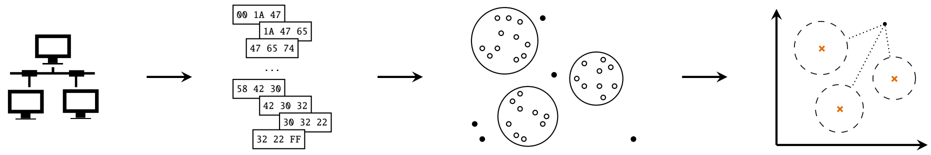
Industrial facilities, such as power stations and water supply systems, are high-value targets for terrorists and nation-state attackers. The progressing automatization of industrial processes and the interconnection between devices, facilities and control centers significantly increase their attack surface and impose new challenges for security solutions. A power plant, for instance, consists of a plethora of proprietary software and hardware components from various manufacturers. Many of these components use non-standardized protocols that are specific to manufactures or even to a particular type of device. Consequently, operators of industrial facilities usually do not know about implementation details of the (computer) systems they run. Moreover, these protocols often rest on compact binary structures for minimizing communication overhead and delay. On the *field level* this frequently is the case due to the limited resources of devices and legacy reasons. Modbus messages, for instance, are limited to 256 bytes as the first implementation has been designed for serial communication over RS485 (Modbus.org, 2012). Interestingly, modern protocols on the *control level* however often rely on similar representations and structures as well. This renders the use of traditional intrusion detection approaches for industrial control systems extremely difficult. As a result, the networks in so-called SCADA systems have been increasingly targeted by attacks in the last years (e.g., Karnouskos, 2011; Falliere et al., 2011; Symantec Security Response, 2011; Kaspersky Lab, 2015; Cherepanov, 2017).

Without the availability of protocol specifications that assist in preprocessing network data, an in-depth analysis of communication content is difficult to accomplish. The research community has thus moved towards approaches that model the appearance of network traffic rather than its content (Koutsandria et al., 2014; Parvania et al., 2014; Schuster et al., 2015; Udd et al., 2016) or even the underlying physical process itself (Hadžiosmanović et al., 2014; Krotofil et al., 2015; Urbina et al., 2016) in order to detect deviations from the expected process states. This more abstract perspective allows for the detection of specific classes of attacks, such as flooding or incremental attacks, but also restricts defense more than necessary. Stuxnet, for instance, has sabotaged the overall production process by making small, infrequent changes to the motor speed of centrifuges (Falliere et al., 2011). While only subtle changes have been made, these operations happened out of the ordinary and posed anomalies among the usual bus communication with respect to used input values. Thus, given a precise model of normality such anomalies can be detected—both on a process- as well as the network-level. Constructing these models however is inherently difficult. Industrial facilities are subject to changes in hardware (e.g., sensors, PLCs, etc.) and adaptations of the process itself. On the one hand, using an expert model of the physical process for detection, implies that this model has to be manually updated at every change in order to prevent divergence from reality that in turn may result in loopholes for an attacker. On the other hand, machine learning has been used to automatically learn and update models of communication contents instead. Previous research has demonstrated the effectiveness of anomaly detection for network-based intrusion detection for various fields of application and protocols (Wang and Stolfo, 2003, 2004; Wang et al., 2005; Rieck et al., 2010; Šrndić and Laskov, 2013; Wressnegger et al., 2013b). However, the prevalent use of proprietary binary protocols in industrial networks significantly complicates the use of content-based approaches for the detection of intrusions and often renders existing approaches ineffective (Hadžiosmanović et al., 2012).

In this chapter, we bridge this gap and present a method that effectively makes use of content-based anomaly detection for proprietary binary protocols. With this we show that, in contrast to prior belief, language models are very well usable in such environments.

5.2 System Overview

We now present Zoe, our method for content-based anomaly detection in industrial control systems (Wressnegger et al., 2018). By focusing on proprietary binary protocols, for which we do not know about the structure or the semantics of the transmitted data, we shift the focus away from the underlying analyses and feature engineering towards methods for accurately modeling the data at hand. In contrast to the method for classifying malware, presented in the previous chapter, here we thus consider raw data as observed on the wire as input in step ①. Subsequently in step ②, we use language models to embed the data in feature space and merely cleanse it in the process, in order to prepare it for learning. An overview of the system is depicted in Figure 5.1.



① Input ② Embedding ③ Prototype Models ④ Anomaly Detection

Figure 5.1: Schematic depiction of Zoe and the individual steps involved: First, input data is recorded at network level, before the raw network traffic is embedded using language models. Then, the data is clustered for identifying prototypical representations of message types and use these *prototype models* for anomaly detection.

A large body of research on content-based intrusion detection has shown that considering the mere presence of features in network traffic, such as n-grams, often is superior over counting their occurrences (Wang et al., 2006a; Rieck et al., 2010; Hadžiosmanović et al., 2012; Šrndić and Laskov, 2013; Wressnegger et al., 2013b). For example, particular strings might already be indicative to spot network attacks at the application layer. However, ignoring the frequency of features prematurely discards valuable information. While this kind of data may not be mandatory for the pure detection of attacks, we show that it plays a key role in modeling normality and constructing corresponding detection models, in particular, in proprietary network environments.

Based on this observation we perform two more steps: In step ③ we automatically partition network traffic into k groups of messages with similar content, thereby approximating states of the underlying protocol. This procedure is designed to not only separate message types but also, to derive one prototype model per type in the process, which makes a classical separation of clustering and subsequent learning of content models unnecessary. In step ④ we then use these for anomaly detection. One of the main obstacles for analyzing unknown protocols and detecting attacks therein is "noise", that is, seemingly random data that hinders inferring suitable content models (see Hadžiosmanović et al., 2012). We address this problem by analyzing the occurrences of features in each prototype model and carefully filtering rare features using a frequency threshold t.

These two key components form the basis of our detector, which thus is parameterized by the number of prototype models k and the frequency threshold t:

$$\textsc{Zoe}(k, t)$$

The usage of one building block without the other can be denoted as $\textsc{Zoe}(1, *)$ for a detector using one global content model but different thresholds for noise-reduction, and $\textsc{Zoe}(*, 0)$ for the use of individual prototype models that do not filter noise. In practice these two steps however seamlessly intertwine and should be used in combination for yielding the best possible performance. Subsequently, we describe both in more detail.

5.3 Building Prototype Models

In the absence of appropriate protocol dissectors and parsers, an analysis of network traffic is only feasible if abstract representations of the exchanged data are developed that are capable of reflecting content and structure in a generic manner. To tackle this problem, we model the communication between two parties in a network as a sequence of incoming and outgoing *binary messages* or more formally application-level data units (ADUs). For stateful transport protocols, such as TCP, these messages can be extracted using regular stream reassembly (Dharmapurikar and Paxson, 2005); for stateless protocols, such as UDP, these messages simply refer to the application-level payloads. While this representation is ideal for conventional signature-based detection, for building protocol models we however require a more structured representation of the data. We thus map each message m monitored in the network to a corresponding feature vector $\mathbf{x} = \varphi_3(m)$ using language models: We extract all n-grams, substrings of length n, from the message and record their occurrences. Each n-grams is associated with one dimension of the feature space, such that a message m can be expressed as a vector of n-gram occurrences. Analogous to Chapter 3 we define this map as

$$\begin{aligned} \varphi_3 \colon \mathcal{T} &\to \{0, 1\}^{|S|}, \\ m &\mapsto \Big(\operatorname{bin}(s, m)\Big)_{s \in S} \end{aligned}$$

where $\mathcal{T}$ is the set of all possible messages in network traffic, the set S denotes all possible substrings of length n and the function $\operatorname{bin}(s, m)$ represents the binary occurrence of the substring s in the input message m. Using this mapping, we translate a set of messages $\{m_1, \ldots, m_N\}$ to a set of vectors

$$X = \{\mathbf{x}_1, \ldots \mathbf{x}_N\} \ \text{ with } \ \mathbf{x}_i = \varphi_3(m_i).$$

Based on this representation, we proceed to learn prototype models as a first cornerstone for coping with high-entropy data in binary protocols. To learn models per message type we build on methods from the field of clustering (see Hastie et al., 2009, Chp. 14).

Unfortunately, clustering is a rather expensive task and many algorithms are not suited for efficiently processing large amounts of data. With ZOE we aim at an integrated solution that breaks up the separation of clustering and subsequent learning of content models. We thus build upon a linear-time algorithm for approximating a clustering (Aggarwal, 2009), that we have tailored to network traffic analysis such that we can build prototype models on-the-fly.

We first transform our network messages m to feature vectors $\mathbf{x}$ as described before, to build the input dataset X. Then, k samples are drawn from the input data to initialize clusters $C_1, \dots, C_k$. For each following sample $\mathbf{x} \in X$ we measure the similarity to each cluster and assign it to the cluster C_j that has the closest proximity

$$j = \underset{i \in [1,k]}{\arg\max}\ \operatorname{prox}_{C_i}(\mathbf{x}).$$

The overall similarity in turn is calculated based on the input sample $\mathbf{x}$ and all samples that belong to a cluster C:

$$\begin{aligned} \operatorname{prox}_C \colon \{0,1\}^{|S|} &\to \mathbb{R}, \\ \mathbf{x} &\mapsto \frac{1}{|C|} \sum_{\mathbf{y} \in C} \operatorname{sim}(\mathbf{x}, \mathbf{y}). \end{aligned}$$

The similarity measure sim for two input samples may be either approximated by the dot product $\tilde{\operatorname{sim}} \colon \mathbf{x}, \mathbf{y} \to \mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n} x_i y_i$ or defined by the cosine similarity based on the l^2-norm $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=0}^{n} x_i^2}$, that is,

$$\begin{aligned} \operatorname{sim} \colon \{0,1\}^{|S|} \times \{0,1\}^{|S|} &\to \mathbb{R}, \\ \mathbf{x}, \mathbf{y} &\mapsto \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2} = \cos(\theta) \end{aligned}$$

where θ denotes the angle between the vectors $\mathbf{x}$ and $\mathbf{y}$. While this requires slightly more effort to realize a linear-time implementation, the normalized angle has the advantage of being a formal distance metric (see Leskovec et al., 2014).

For a sufficiently large input set X the algorithm above approximates a precise clustering with high probability (Aggarwal, 2009). This algorithmic requirement however also demands an especially efficient way of handling the sets of messages as clusters. We hence make the following two optimizations in our implementation: First, we store counts of messages and their substrings rather than the messages themselves in order to save valuable working memory. For each cluster C_i we only maintain the total number of all samples $|C_i|$ contained in the cluster and a vector of cumulative counts as *prototype models*:

$$\mathcal{P}_\mathrm{i} = \sum_{\mathbf{y} \in C_i} \mathbf{y}.$$

This suffices to compute the similarity between messages and clusters as defined before, and can be used for efficient anomaly detection in further follow. Second, due to the large amounts of network traffic we operate on, storing and keeping track of substring counts already poses a considerable challenge. Retaining exact counts simply is not feasible as it requires to store all substrings or at least hashes thereof. We thus revert to probabilistic counting of substrings using Count-Min sketches (see Section 3.3).

5.4 Anomaly Detection

As a second building block for reliably detecting attacks in proprietary network protocols, we propose an extension to content-based anomaly detection made possible by prototype models introduced in the previous section. Language models, such as n-grams (e.g., Wang and Stolfo, 2004; Wang et al., 2006a; Song et al., 2009), have frequently been used for attack detection and have proven impressively effective for *text-based* data (e.g., Wang and Stolfo, 2004; Wang et al., 2005, 2006a; Perdisci et al., 2009; Rieck et al., 2010). For *binary* and *high-entropy* data, however, such models are considered mostly impractical by the research community so far (Hadžiosmanović et al., 2012) and have been widely discarded for the application in industrial networks.

The main reason why language models perform worse in this environment is founded in the density of the message data. The density is defined as the ratio of the number of unique occurrences of a feature to the total number of all possible elements (see Chapter 3). This ratio significantly influences the quality of the resulting model. In case of high entropy, as induced by binary network protocols, a model gets so “packed” that it becomes difficult to differentiate between two classes (benign and malicious messages) and renders mere binary embeddings impractical (Wang et al., 2006a; Hadžiosmanović et al., 2012). Count embeddings, on the other hand, have been shown to be less effective when used as a drop-in replacement in an otherwise identical setting (Wang et al., 2006a; Rieck et al., 2010; Hadžiosmanović et al., 2012; Šrndić and Laskov, 2013; Wressnegger et al., 2013b).

However, rather than discarding this information altogether, in this work, we use the number of occurrences to filter relevant from irrelevant information. Note, that we record the frequency f of samples in which features/substrings $s \in S$ occur rather than the total count of s in the complete dataset. Setting f in relation to the total number of samples N formally yields the well-established *document frequency* measure $df = \frac{f}{N}$. The prototype models $\mathcal{P}_\mathrm{i}$ that we have introduced in the previous section represent exactly these feature frequencies across training samples iff function $\mathrm{bin}(s, m)$ is defined to report the existence of substring s in input message m. Detection schemes based on binary embeddings can thus be directly derived from prototype models $\mathcal{P}_\mathrm{i}$ *without additional training*.

By introducing a threshold t we now prune features that occur in less than t input samples that have been associated with a cluster and thereby effectively discard noise from the training data. Models M_i can hence be interpreted as sets of features that are considered for detection and together form the overall content model used by ZOE:

$$\mathcal{M} = \{M_1, \dots, M_k\} \text{ with } M_i = \{s \in S \mid \mathcal{P}_{i,s} \geq t\}$$

This allows us to revert to detection using the (implicit) binary embedding based on the remaining, most relevant features only. This scheme offers two main advantages: First, it allows to boil down the set of benign features to those that appear more than t times, and thus significantly reduces the size of the model as features associated with a value of 0 are not explicitly stored. Furthermore, this limits an attacker's reach of play when mimicking benign messages. Second, for production use and to improve runtime performance of the final detector this binarized Count-Min sketch can be transformed into a compact Bloom filter that offers a higher accuracy than established methods with the same memory footprint (see Section 5.5).

In order to determine the overall detection using k models ZOE considers the score of the model with the highest resemblance to the message m in question. With a scoring function $\mathrm{d} \colon \mathcal{T} \to \mathbb{R}$ that yields low values for known/benign messages and high values for anomalies, this formally translates to choosing the minimum score of k models:

$$\begin{aligned} \mathrm{score}_{\mathcal{M}} \colon \mathcal{T} &\to \mathbb{R}, \\ m &\mapsto \min_i \mathrm{d}(m, M_i) \end{aligned}$$

In accordance to d and using an overall threshold T, a message is considered malicious for $\mathrm{score}(m, \mathcal{M}) \geq T$ and benign otherwise.

It is important to stress that anomaly detection does not explicitly learn to discriminate benign from malicious data, but instead normality from anomalies. This semantic gap requires one to design features and detection systems carefully, as otherwise identified anomalies may not reflect malicious activity. To further tighten the assumption of anomalies being attacks, it is necessary to periodically retrain the underlying model as the notation of normality might change over time. In practice, a linear-time approach for training as well as testing, as presented in this chapter, thus is highly beneficial.

5.4.1 Adjusting Anomaly Detectors

Effective anomaly detection can only succeed with a carefully chosen parametrization of the detector. This requires benign traffic for building the content model, but also attack samples to validate the chosen settings and estimate the expected detection performance (Sommer and Paxson, 2010). As recordings of such attacks in industrial environments and for proprietary protocols in particular are naturally rare, we have developed a tool for the automatic generation of network attacks against unknown protocols. The tool mimics a protocol based on observed network traffic as close as possible and injects attack strings, that range from program code (e.g., shellcodes, ROP chains, PLC instructions), over scripts fragments (e.g., Perl scripts) to random data, at positions containing variable input. To this end, we first group similar network messages using clustering and then derive generic rules that describe the structure of the messages in each cluster. Second, we generate attack strings with different encodings and obfuscations to populate variable fields of the derived rules.

Inferring Protocol Rules. For industrial facilities we usually do not have the specification of the used network protocols on hand and neither does the operator, as vendors normally do not share product details. Consequently, we are required to automatically infer protocol models based on network traffic only (Leita et al., 2005; Cui et al., 2006; Krueger et al., 2012; Gascon et al., 2015). To this end we build upon work by Krueger et al. (2012) and Gascon et al. (2015) to derive rules for individual messages.

In a first step, we identify messages that have the same structures using off-the-shelf k-means clustering. Similarly to the method described in Section 5.3 network messages cannot be directly used at this point, but are embedded into vector space first. We thus again operate on an input set $X = \{\mathbf{x}_1, \ldots \mathbf{x}_N\}$ yield by the feature map φ_3 with a binary embedding of substrings $s \in S$. However, as inferring protocol models is much more involved than simply deriving states, additional statistical tests are applied to filter relevant from irrelevant substrings/features and to avoid an overly populated vector space that might hinder the clustering process. We hence subdivide the feature space in *constant*, *mutable*, and *volatile* parts by applying a binomial test to each feature (Krueger et al., 2010). A value close to 0 indicates volatile features while a frequency of 1 refers to constant features. Neither volatile nor constant features are particular valuable for discriminating messages in a protocol—think of a protocol's magic values or nonces. We hence reject all features that do not meet a statistical significance level of $\alpha = 0.05$ before clustering the inputs (Holm, 1979).

Subsequently, we derive rules that describe all messages in a cluster which can then be used to generate new messages that comply with the format of that cluster. To do so we consider the original messages in each cluster, rather than the reduced representation in feature space and pair-wise align these using an extended version of the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970). The resulting rules consist of *fixed bytes* that appear at the same position in each message of the cluster and *variable fields* that may contain different byte sequences from message to message. Figure 5.2 shows an example for a set of simple messages consisting out of human readable and binary data.

Message 1	`\x00\x1a GetVal "N2HAJ22CT000 XH24" \xff`
Message 2	`\x00\x1a GetVal "N2HAJ22DT500 XR01" \x00`
⋮	⋮
Message n	`\x00\x1a GetVal "N2HAJ31AA100 XB02" \xff`
Rule	`\x00\x1a GetVal "N2HAJ`▭`00 X`▭`"` ▭

Figure 5.2: Rule inference for a set of simple messages. Figure taken from (Wressnegger et al., 2018).

It is clear to see that our tool is not capable of learning the exact protocol specification but approximates it based on the network traffic on hand. The depicted cluster contains messages that apparently transmit a command string `GetVal` to retrieve sensor values from a particular device. Furthermore, it contains the length of the transmitted string as 16 Bit integer in the front and another unspecified flag (presumably a high and low value) at the end. Within this cluster only individual parts of the device identifier and the binary flag at the end change such that the remaining parts are considered constant by the algorithm. For populating a valid network message with attack payloads this however is sufficient.

Generation of Attack Messages. With the inferred rules as detailed descriptions of protocol messages at our disposal, we can now produce authentic network traffic that contains arbitrary attack payloads. For a wide range of different attacks we query Metasploit (Rapid7 LLC, 2003–2018) for payloads and encoders—simple obfuscations that, for instance, encrypt scripts with a simple xor operation, or map program code to printable characters. To generate attack messages we proceed as follows: For each attack string we randomly choose an existing network message. The corresponding rule then allows us to replace variable fields within this message at will. For our experiments we choose exactly one at random and inject the attack payload. The remaining (constant and variable) fields remain unchanged. Figure 5.3 depicts this simple scheme.

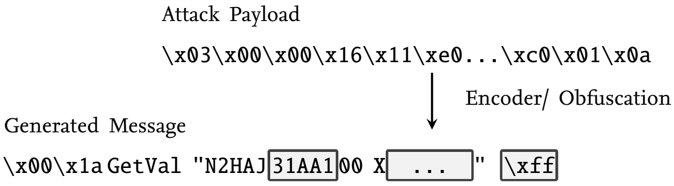

Attack Payload
\x03\x00\x00\x16\x11\xe0...\xc0\x01\x0a
Encoder/ Obfuscation
Generated Message
\x00\x1a GetVal "N2HAJ 31AA1 00 X ... " \xff

Figure 5.3: Schematic depiction of the generation of network attacks.

5.5 Evaluation

We empirically evaluate Zoe based on the six industrial protocols described in Section 5.5.1 and conduct a number of experiments that examine the different aspects of our method: First, in Section 5.5.2 we demonstrate the overall detection performance of our method. Second, the influence of Zoe's de-noising capabilities as well as the impact of message-specific prototype models is studied in Section 5.5.3 and 5.5.4, respectively. Third, we measure the data throughput of our method during training in Section 5.5.5 and compare Zoe with related approaches in Section 5.5.6. Finally, we investigate the feasibility of evasion attacks in the form of polymorphic blending attacks against Zoe, as a final experiment in Section 5.5.7.

In the course of this evaluation we describe detection performances with the aid of ROC curves and the corresponding (bounded) AUC, as described in Section 1.1.3. For training and testing individual detectors we create strictly separated datasets that do not overlap. 75 % randomly chosen samples are used as *known data* for training and the remaining 25 % as *unknown data* for testing. This partitioning is applied to benign and malicious samples likewise, and is repeated for ten experiments which are averaged to determine the overall detection performance. The parameters for the detector are chosen solely based on the results obtained on training data and only then, this configuration is used to evaluate the performance on the testing dataset.

5.5.1 Datasets

For evaluating our approach under realistic conditions, we have partnered with a large European energy producer. In particular, we have collected roughly 210 GiB of raw network data during 6 days at a large power plant (total 1,900 MW), and the operation of a coal mining facility. At the power plant we have recorded network traffic at the *control level* of a power unit producing 500 MW. Our recording period covers a ramp-up phase as well as normal operation of the unit. From this, we have extracted the five most prominent protocols, all of which are proprietary and publicly undocumented[1]. Through manual analysis we however were able to attribute these to a large plant manufacturer. In order to provide a comprehensive study on protocols that occur in industrial networks the data recording at the coal mining facility targets PROFINET IO at the *field level*. In total we have recorded communication between 92,700 unique devices in an authentic production environment. Table 5.1 summarizes the gathered data.

Protocols P1–P3 build on TCP while P4 & P5 use UDP for direct communication. PROFINET IO, on the other hand, is situated somewhat differently: While TCP/IP is used for the parameterization and configuration, real-time messages are exchanged on a separate channel that does not use the Internet protocol at all. Although PROFINET IO traffic can be easily parsed and therefore reliably filtered, there are no protocol parsers publicly available for the remaining traffic from the power plant. For these protocols we hence resort to filtering the network traffic based on IP ports, but explicitly consider relations between communicating entities to sanitize the data. For TCP traffic we additionally reassemble network streams, such that we are able to evaluate ZOE based on complete "messages" (approximated as consecutive, unidirectional traffic) for P1–P3, datagrams for P4 & P5, and PROFINET IO packets. All in all, this gives us 46.7 GiB of raw data for our evaluation.

Furthermore, these different protocols show a highly diverse structure. While protocols P1 and P2 seem to exclusively use either text-based or binary-based data for their communication, protocols P3–P5 use a mixture of both. The latter appear to mainly consist of binary structures that additionally transmit string-based (printable) data. As these strings exhibit strong structure that may be interpreted as another protocol on top of the base protocol we assign these to both sub-groups. PROFINET IO then again is strictly based on binary data. Additionally, server and client communication often vary significantly for these protocols. To account for this difference in composition and structure, we thus split the individual protocol subsets in *incoming* and *outgoing* traffic and analyze these individually in our evaluation. UDP and PROFINET IO traffic is not effected by this preprocessing step.

---

[1] Consequently, publicly available tools such as Wireshark are not capable of parsing these protocol beyond the TCP/UDP packet structure.

	Generation					Mining
Protocol	P1 2000	P2 2069	P3 4241	P4 2010	P5 2070	P6 PROFINET IO
TCP	●	●	●			(●)
UDP				●	●	
Binary		●	●	●	●	●
Text	●		●	●	●	
Size	2,542 MiB	219 MiB	18,506 MiB	2,506 MiB	1.6 MiB	1,878 MiB
Count	7,032,323 msgs	310,205 msgs	13,046,151 msgs	16,024,175 packets	33,329 packets	13,957,589 packets

Table 5.1: Dataset and protocols used for the evaluation of ZOE.

5.5.2 Overall Detection Performance of Zoe

We begin with demonstrating our method's detection performance for the six industrial protocols we have collected. Figure 5.4 shows the results for our detector parameterized with different de-noising thresholds and numbers of prototype models using 5-grams. The y-axis shows the detection performance as AUC bound to different thresholds of false positives, denoted as AUC(b), over the individual protocols on the x-axis. Zoe performs very well across different protocols and detection performance only differs in nuances. The parametrization of the detection however is crucial as we demonstrate in the following sections. For protocol P3 (outgoing), for instance, our experiments show that at least 4 different prototype models are needed to enable good detection. Subsequently, we thus inspect the influence of the individual components of Zoe in detail and use protocol P3 as recurring example.

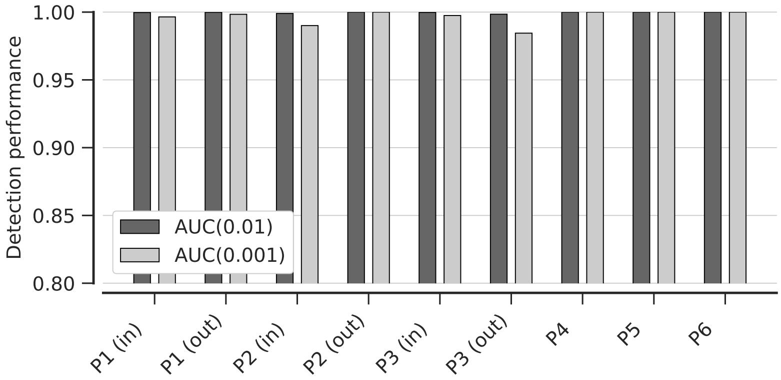


Figure 5.4: Detection performance of Zoe for the protocols P1 to P6.

5.5.3 De-noising Content Models

Next, we evaluate the impact of different thresholds t used with Zoe to demonstrate its de-noising capabilities. We hence parameterize Zoe$(1, t)$ with thresholds $t \in [20, 100]$ with a granularity of 5. Figure 5.5 shows the results for protocol P3. At a threshold $t = 35$ the detector reaches its peak performance with an AUC(0.0001) of 0.8 (light gray line) and a true-positive rate of 0.978, meaning that 97.8 % of the attack patterns are detected with at most 1 false positive out of 10,000 network messages. In comparison to no de-noising $(t = 0)$, we record a tremendous improvement that clearly shows that pre-filtering the features used for detection is of the essence for effective attack detection in binary protocols.

Moreover, it is interesting to see that the detection performance shoots up at thresholds of 20 to 25, peaks at $t = 35$ and flattens out towards higher values. This underlines the importance of a thorough evaluation with the aid of a comprehensive set of attack samples.

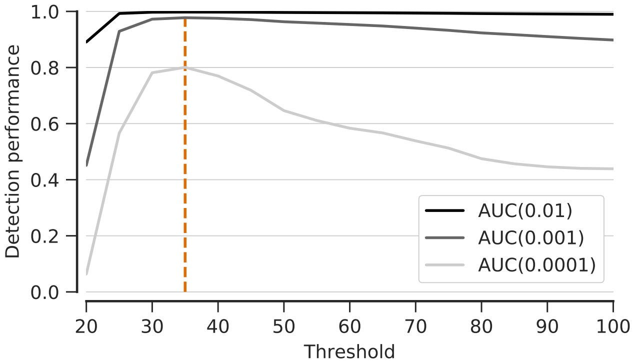


Figure 5.5: Detection performance of Zoe$(1, t)$ for varying thresholds t. The dashed line indicates the threshold yielding the highest detection performance ($t = 35$).

5.5.4 Message-specific Prototype Models

In this section, we bring together the two key components of Zoe and study the influence of using multiple prototype models on the detection performance. Furthermore, we explicitly highlight the additional improvement in the eminently important low range of false positives.

To this end, we train our detector for different numbers of prototype models k and thresholds t in order to calibrate the detector as described in the previous section. The best configuration with both components enabled, Zoe$(4, 50)$, is shown as a ROC curve in Figure 5.6 in relation the best detector without prototype models, Zoe$(1, 35)$. Note, that the x-axis (false-positive rates) shows a logarithmic scale to better emphasize the improvement, that the detection specific to message types entails. Such enhancements in detection performance are of great importance for the application of the detector in production and limits the costs of false alarms.

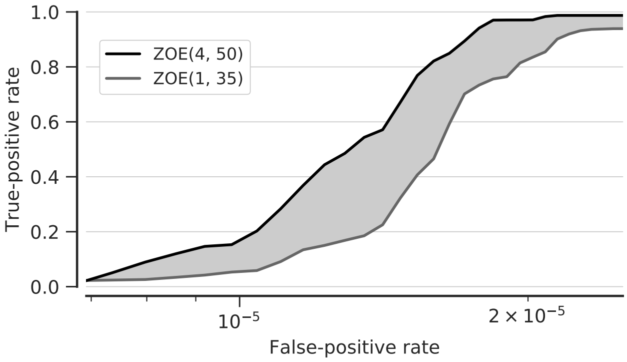


Figure 5.6: Detection performance of two configurations of Zoe: First, *a global content-model* with $(1, 35)$ and second, *multiple prototype models* with $(4, 50)$. The latter clearly improves upon the other in the critical region of low false-positive rates.

5.5.5 Runtime Evaluation

Zoe makes use of a linear-time algorithm for clustering network messages and deriving the final detection models is done in constant time. The latter is achieved by applying a threshold when processing the prototype models. The runtime performance thus is mainly influenced by the number of prototype models used for detection and the size of the n-grams, that is used to process the input data.

In this experiment, we thus evaluate the runtime performance of Zoe with respect to these two parameters. We again measure the throughput of the full toolchain for training our method, single-threaded on an Intel Xeon E5-1650 CPU at 3.50 GHz and operate in internal memory to compensate for disk I/O. Figure 5.7 shows the results in Mbit/s over the number of used prototype models for three different lengths of n-grams.

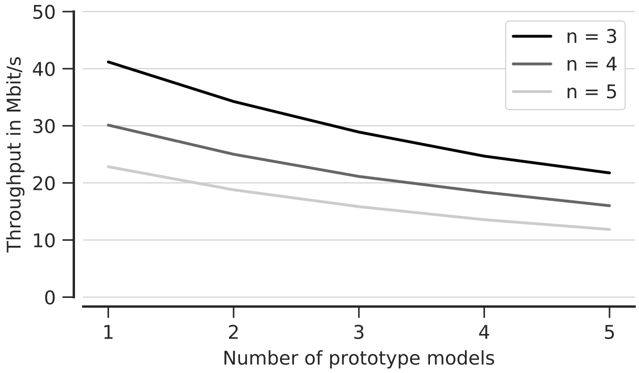


Figure 5.7: Throughput of Zoe during training.

For example, with 28.9 Mbit/s Zoe using three prototype models and 3-grams, performs very well. The throughput however decreases with increasing numbers of prototype models, as each item needs to be compared to each cluster/model before assignment. Moreover, initializing and resetting Count-Min sketches is relatively costly, such that the throughput improves for large data sets. We also observe an impact of the n-gram length, which results from the increasing number of bytes that gets hashed. While the number of prototype models cause an unavoidable decline, hashing costs may be reduced by combining hash functions (Kirsch and Mitzenmacher, 2006), or using rolling hashes (Moraru and Andersen, 2012).

5.5.6 Comparison with Related Approaches

Ultimately, we conduct a comparison of Zoe to Anagram (Wang et al., 2006a), a content-based approach based on Bloom filters using higher-order n-grams ($n \geq 3$). While it originally has been designed for detecting attacks in HTTP traffic it has proven effective for various other protocols as well (Hadžiosmanović et al., 2012; Wressnegger et al., 2013b). Previous research has shown that it even works for certain binary protocols with a limited set of message formats and simple structure such as Modbus (Hadžiosmanović et al., 2012). Anagram is very similar to the most basic configuration of our detector, ZOE$(1, 0)$, that neither make uses of prototype models ($k = 1$) nor de-noising of the content models ($t = 0$). In order to maintain comparability in this experiment we employ filters/sketches that have approximately the same number of items for storing content features. We hence choose $\varepsilon = 6 \times 10^{-7}$ and $p = 0.99$ for Count-Min sketches in order to match the 2^{25} items large Bloom filter we use for Anagram in this experiment. We also make use of d_1 as distance measure for a direct comparison. Table 5.2 shows the detection performance of both as AUC bounded to different thresholds of false positives for protocol P3. While ZOE yields high performance values down to a false-positive rates of 0.0001, Anagram is not able to compete and only achieves an AUC(0.01) of 0.15, which even reduces to 0.04 for AUC(0.001). This is founded in the tight connection of structure and data found in binary protocols, which conventional content-based anomaly detection is not designed to operate on.

	Detection performance		
Method	AUC(0.01)	AUC(0.001)	AUC(0.0001)
ZOE	0.9984	0.9844	0.8463
Anagram	0.1515	0.0408	0.0002

Table 5.2: Detection performance of ZOE and Anagram for protocol P3.

In the ROC curve shown in Figure 5.8 the difference becomes even more apparent. Again, a logarithmic scale has been chosen to highlight the importance of low false-positive rates for intrusion detection. ZOE yields a true-positive rate of 0.971 for as few as 0.002 % false positives, meaning that, 97.1 % of the attacks are correctly detected with only 2 false alarms out of 100,000 messages. Anagram, on the other hand, only detects 16.5 % with a 500× higher false-positive rate of 1 %. These results show that ZOE not only outperforms Anagram by an order of magnitude in sheer detection performance, but also in terms of false-positive rates.

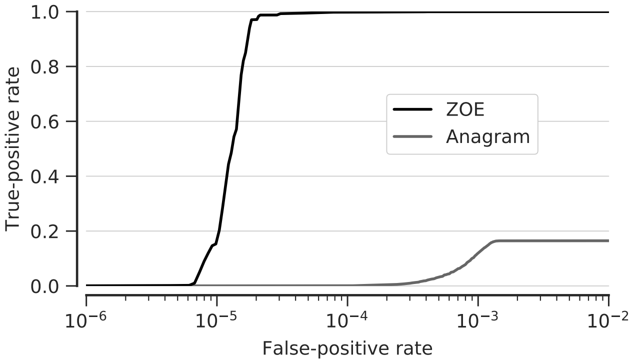


Figure 5.8: Detection performance of ZOE and Anagram side-by-side as ROC curves on a logarithmic scale to better emphasize the extremely small false-positive rates achieved by our method.

5.5.7 Evasion

In order to bypass content-based anomaly detection an adversary might attempt to mimic benign content and slip by attacks. We thus also study the effects of a particular kind of evasion attack—polymorphic blending—on our detector.

Mutation and transformation attacks have a long-standing history for evading intrusion detection systems on different levels (e.g., Vigna et al., 2004; Rubin et al., 2004; Fogla et al., 2006; Fogla and Lee, 2006; Perdisci et al., 2006; Song et al., 2007, 2010). Polymorphic blending attacks are a particular effective tool against content-based intrusion detection (Fogla et al., 2006). Although, generating these has been proven to be NP-hard (Fogla and Lee, 2006), attackers can resort to heuristics to provide good approximations, for instance, using iterative hill climbing. This method from mathematical optimization starts off with an initial solution (the original attack) and iteratively improves it according to a target function (the scoring function of the detector).

Two aspects are key to the success of the attack: First, the application of lower-order n-grams and second, a rich and extensive model containing large amounts of benign features that can be used by the attacker. The latter can also be expressed by the density and variability of the training dataset (see Chapter 3) that directly influence the richness of the model. Note, that using higher-order n-grams ($n \geq 3$) also increases the complexity of the model due to rare but apparently benign features. With Zoe we address both aspects by (a) using substrings of length $n = 5$ and (b) pruning rare features, thus limiting the reach of play of an attacker.

In this final experiment, we generate 32,640 blended attack instances per epoch over a total of 100 epochs. Additionally, after each iteration we select the 10 best performing instances for further mutations. This results in 3,234,624 variations that are tested for each attack. Figure 5.9 shows the results of a single experiment for a number of different thresholds of Zoe. The gray-scale level decreases with higher thresholds from black ($t = 35$) to light gray ($t = 50$).

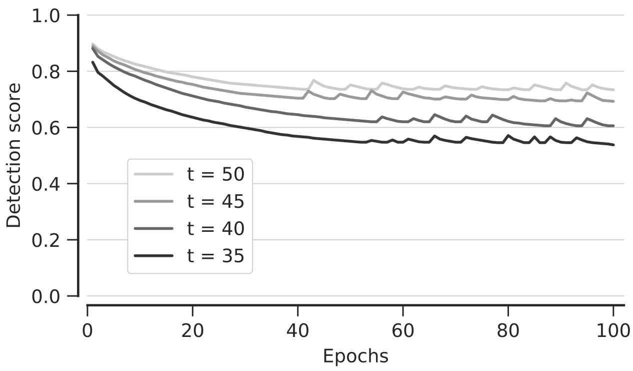


Figure 5.9: Polymorphic blending attacks based on 3,234,624 mutations over 100 epochs against Zoe with different thresholds t.

The ripples observed in the second half of the curves indicate the point in time when the algorithm has detected a potential local minimum and attempts to escape by performing a small number of random mutation at once. The higher the detection score to which the individual curves converge to, the more robust the detector is against polymorphic blending attacks. This clearly shows that narrowing down the set of features that are used in the content model (increasing the threshold) is not only beneficial for the detection performance itself, but also for improving the resistance against mutation attacks.

5.6 Related Work

Anomaly detection has been intensely studied in computer security research in the past and in the scope of network-based intrusion detection in particular (e.g., Ingham and Inoue, 2007; Perdisci et al., 2009; Rieck and Laskov, 2006; Rieck et al., 2010; Wang and Stolfo, 2003, 2004; Wang et al., 2005, 2006a). The uprise of SCADA security has again fostered the development of new methods and has extended the scope such that we differentiate between the following, orthogonal strains of research: (1) content-based attack detection, (2) detection based on network characteristics, and (3) detection by modeling physical processes. Subsequently, we review related work with respect to these different directions with a special focus on industrial control systems. Due to the proximity to our approach, we however also include works on content-based anomaly detection in general-purpose networks.

5.6.1 Attack detection based on Content

A large body of research deals with the in-depth analysis of protocol contents. This includes the identification (Dreger et al., 2006), analysis (Pang et al., 2006; Wondracek et al., 2008) and reverse-engineering (Lin et al., 2008; Comparetti et al., 2009) of protocols, but also the detection of shellcodes in network streams (Polychronakis et al., 2006, 2010; Snow et al., 2011). Zoe in contrast strives for a less specific approximation of protocols tailored to the purpose of attack detection, and aims at a wider range of attacks than shellcodes and abnormal behavior in general.

For this purpose language models such as n-grams have been proven to be very effective (Wang and Stolfo, 2004; Wang et al., 2005, 2006a; Rieck and Laskov, 2006; Perdisci et al., 2009; Song et al., 2009). Early approaches (Kruegel et al., 2002; Wang and Stolfo, 2004) build distributions of byte frequency as a notion of normality to detect attacks as deviation of these histograms. PAYL (Wang et al., 2005) extends these to the use of 2-grams, before Wang et al. (2006a) propose Anagram and establish the use of higher-order n-grams ($n \geq 3$). Similar to our approach Anagram stores a sparse vector representation of content data in a probabilistic data structure. Recently, these data structures have also been used for protocol-specific anomaly detection (in combination with LTSM networks) for SCADA systems (Feng et al., 2017). Rieck and Laskov (2006) address the use of higher-order n-grams with a representation as Trie that allows for the efficient computation of distances of such vector representations. Spectogram (Song et al., 2009), on the other hand, extracts n-grams from HTTP request and models these as a mixture of Markov chains and thus avoids storing observed features altogether. Moreover, it also employs a variant of clustering to control the number of Markov models. While similar in spirit to Zoe, both approaches operate on an entirely different scale.

Wressnegger et al. (2013b) and Hadžiosmanović et al. (2012) provide a general overview of content-based detection based on n-grams. The latter however focuses on industrial control systems and inspects the suitability of various detection methods such as POSEIDON (Bolzoni et al., 2006) and Anagram (Wang et al., 2006a). In concept similar to PAYL, Düssel et al. (2009) presents an anomaly detection system based on n-grams using distance metrics for industrial networks. This certainly shares the motivation with Zoe, but does not pursue any advanced detection strategies to cope with proprietary binary protocols found in large industrial facilities.

5.6.2 Attack Detection based on Network Characteristics

Alternatively to evaluating the content of network packets, researchers have considered sequences of network packets and the relationships of the communicating devices. These approaches operate on a complementary level of network traffic compared to Zoe. Yang et al. (2006) present an intrusion detection system that uses an auto-associative kernel regression model coupled with the statistical probability ratio test. Schuster et al. (2015) apply a one-class SVM on a number of traces from real-world industrial traffic from different industrial control systems. Koutsandria et al. (2014) and Parvania et al. (2014) propose to extend intrusion detection systems for SCADA systems by combining traditional signature-based approaches and communication rules, while considering physical limits of the involved devices. Fovino et al. (2010) propose a new state-based intrusion detection system for SCADA systems that combines traditional, signature-based techniques with a state-analysis technique. Udd et al. (2016) extend the network security platform Bro (Paxson, 1999; The Bro Project, 1994–2018) to support a particular SCADA protocol.

5.6.3 Attack Detection by Modeling Physical Processes

Several researchers attempt to model the physical process of industrial facilities which again is orthogonal to our approach. Teixeira et al. (2010), Alajlouni and Rao (2013), and Vukovic and Dán (2013), for instance, model system state estimators and analyze their security properties and detect emerging anomalies. Luchs and Doerr (2017) have recently presented an anomaly detection scheme that makes use of envelope escalation for sensor readings. Hadžiosmanović et al. (2014) model the semantics of process variables, while Mo et al. (2014) propose a model-based technique to detect integrity attacks on the sensors of cyber-physical system.

Another strain of research formally describes the underlying process. For instance, Pasqualetti et al. (2013) propose a mathematical framework for modeling cyber-physical systems and attacks. Wang et al. (2014) employ a detection scheme based on relation-graphs to detect stealthy false-data injection attacks, while Miao et al. (2014) use linear combinations of coding sensor outputs to detect those attacks. Do et al. (2014) formulate the attack problem as transient changes in stochastic-dynamical systems involving unknown system states. Rocchetto and Tippenhauer (2017), on the other hand, use formal modeling to discover potential attacks on cyber-physical systems.

Other model-based approaches deal with the detection of manipulated sensor data; frequently using clustering techniques: Krotofil et al. (2015) propose a process-aware approach to detect sensor signal manipulations using the correlation entropy in clusters of related sensors. Kiss et al. (2015) detect attacks targeting measurements using a Gaussian mixture model to cluster sensor measurements. Urbina et al. (2016) study the physics-based detection of attacks in control systems and develop an adaptive adversary model as well as a new metric for measuring the impact of stealthy attacks.

Chapter 6

Revealing Malicious Content

Next, we look upon a particular class of malware that hides in third-party containers, so-called embedded malware, that gets executed once another application processes the container format. To avoid detection such malware often is obfuscated by rather simple means, such as the Vigenère cipher. For short keys up to 2 bytes the obfuscation can be trivially broken using brute-force attacks. However, uncovering malware obfuscated with longer keys still necessitates manually reverse engineering the code or dynamically analyzing the malicious document in a sandbox. While in principle it is possible to monitor and observe the execution of embedded malware, similarly to the approach for detecting Flash-based malware presented in Chapter 4, the execution modalities render such an approach ineffective in practice. As the malware is triggered by exploits that are specific to a particular version of the processing application, an analysis environment needs to offer multiple (vulnerable) versions of the target application to observe the malware's execution (e.g., Schreck et al., 2012). This aggravates analysis and multiplies the runtime costs per sample.

In this chapter, we thus propose an alternative solution to the problem that is able to fully statically remove the obfuscation layer of embedded malware. By relying on efficient string processing and probabilistic data structures this is achieved in linear time and thus clearly outperforms related approaches. Similar in motivation to the two learning-based approaches presented before, we show how substring statistics alone can already be enough to effectively fend off malware. In Section 6.1 we elaborate on the use-case of detecting malware embedded in common office documents, which we use as exemplary field of application of our method. An overview of the system is provided in Section 6.2, before we detail our approach of breaking Vigenère cipher for deobfuscating embedded malware in Sections 6.3 to 6.5. In Section 6.6 we then conduct an extensive empirical evaluation and discuss related work in Section 6.7.

6.1 Use Case: Detecting Malware in Office Documents

Documents containing malware have become a popular instrument for targeted attacks. To infiltrate a target system, malicious code is embedded in a benign document and transferred to the victim, where it can—once opened—unnoticeably infiltrate the system. Two factors render this strategy attractive for attackers: First, it is relatively easy to lure even security-aware users into opening an untrustworthy document. Second, the complexity of popular document formats, such as Word and PDF, constantly gives rise to zero-day vulnerabilities in the respective applications, which provide the basis for unnoticed execution of malicious code. Consequently, embedded malware has been used as part of several targeted attack campaigns, such as Taidoor (Trend Micro Threat Research Team, 2012), Duqu (Bencsáth et al., 2012) and MiniDuke (CrySyS Malware Intelligence Team, 2013).

To hinder a detection by common anti-virus scanners, malicious code embedded in document files is usually obfuscated, often in multiple layers with increasing complexity. This obfuscation can be achieved using various techniques, ranging from simple encodings to strong ciphers and emulator-based packing. Implementations of complex techniques, however, often contain characteristic patterns and thus increase the risk of detection by anti-virus scanners (Calvet et al., 2012). As a consequence, simple encodings and weak ciphers are still widely used for obfuscation despite their shortcomings. Subsequently, we investigate a specific type of such basic obfuscation, which is frequently used to hide malware in documents. The substitution of bytes using XOR and ADD/SUB—a variant of so-called *Vigenère ciphers* (Schneier, 1996)—is one of the simplest yet widely used obfuscation techniques. These ciphers are regularly applied for cloaking shellcodes and embedded malware. Figure 6.1 shows two examples of these ciphers in x86 code.

```
start:  mov  al, byte [edx]
        add  al, ADD_KEY
        rol  al, ROL_KEY
        mov  byte [edx], al
        inc  edx
        cmp  edx, LENGTH
        jl   start
```

(a) Obfuscation using ADD and ROL

```
start:  mov  al, byte [PTR + ebx]
        sub  byte [edx], al
        inc  ebx
        and  ebx, 0x0f
        inc  edx
        cmp  edx, LENGTH
        jl   start
```

(b) Obfuscation with 16-byte key

Figure 6.1: Examples for Vigenère-based obfuscation: The data stored at `[edx]` is obfuscated (a) using ADD and ROL of one-byte keys, and (b) using SUB with the 16-byte key at PTR. Figure taken from (Wressnegger et al., 2013a).

Due to the implementation with only a few instructions, Vigenère-based obfuscation keeps a small footprint in the code, thereby complicating the task of extracting reliable signatures for anti-virus scanners. Additionally, this obfuscation is fast, easily understandable and good enough to seemingly protect malicious code in the first layer of obfuscation. Despite these advantages, Vigenère ciphers are far from being cryptographically strong and suffer from several well-known weaknesses.

Subsequently, we introduce some notation and define the family of Vigenère ciphers studied in this chapter. This provides a basis for our method for revealing embedded malware later on.

6.1.1 Definition of Vigenère Ciphers

We consider the original code of a malware binary as a sequence of n bytes $M_1 \ldots M_n$ and similarly represent the resulting obfuscated data by $C_1 \ldots C_n$. When referring to cryptographic concepts, we sometimes denote the original code as plaintext and refer to the obfuscated data as ciphertext. The Vigenère-based obfuscation is controlled using a key $K_1 \ldots K_l$ of l bytes, where l usually is much smaller than n. Moreover, we use $\hat{K}_i = K_{(i \bmod l)}$ to access the individual bytes of the key. Using this notation, we can define a family of Vigenère ciphers, where each byte M_i is encrypted with the key byte $\hat{K}_i$ using the binary operation $\circ$ and decrypted using its inverse operation $\circ^{-1}$, as follows:

$$C_i = M_i \circ \hat{K}_i \quad \text{and} \quad M_i = C_i \circ^{-1} \hat{K}_i.$$

This simple definition covers several variants of the Vigenère cipher, as implementations only differ in the choice of the two operations $\circ$ and $\circ^{-1}$. For example, if we define $\circ$ as addition and $\circ^{-1}$ as subtraction, we obtain the classic form of the Vigenère cipher. Table 6.1 lists binary operations that are frequently used for obfuscating malicious code. Note that a subtraction can be expressed as an addition with a negative element and thus is handled likewise.

Operation	**Encryption $\circ$**	**Decryption $\circ^{-1}$**
Addition (ADD)	$(X+Y) \mod 256$	$(X-Y) \mod 256$
Subtraction (SUB)	$(X-Y) \mod 256$	$(X+Y) \mod 256$
Exclusive-Or (XOR)	$X \oplus Y$	$X \oplus Y$

Table 6.1: Operators of Vigenère ciphers used for obfuscation.

Theoretically, any pair of operations that is inverse to each other can be used to construct a Vigenère cipher. In practice, most implementations build on logic and arithmetic functions that induce a *commutative group* over bytes. That is, the operation $\circ$ is commutative and associative as well as there exists an identity element and inverse elements providing the operation $\circ^{-1}$. These group properties are crucial for different types of efficient attacks as demonstrated in Sections 6.5.

Another important observation is that some bytes are encrypted with the same part of the key. In particular, this holds true for every pair of bytes M_i and M_j whose distance is a multiple of the key length, that is, $i \equiv j \pmod{l}$. This repetition of the key is a critical weakness of Vigenère ciphers and can be exploited to launch further attacks that we discuss in subsequent sections.

6.2 System Overview

We now present our method, KANDI, that combines and extends methods from classic cryptography for deobfuscating embedded malware (Wressnegger et al., 2013a). It consists of two separate components that complement each other: First, we conduct probable-plaintext attacks against common variants of the Vigenère cipher. Second, we additionally test each possible transposition for ROL/ROR instructions to account for this frequent additional layer of obfuscation. In this case we consider brute-forcing a legit compromise since there exist only a few combinations to check. Subsequently, both parts are briefly outlined, before the rest of the chapter mainly focuses on the first, more evolved, component and explains the attacks against Vigenère ciphers in the scope of embedded malware.

Breaking Vigenère ciphers. For deobfuscating embedded malware and breaking the used encryption, we first collect the most frequent substrings from a representative set of executable files in step ① of Figure 6.2, which later on are used as probable plaintexts. These plaintexts can be automatically retrieved in linear runtime and may include fragments of the PE header, library names and common code stubs. The method then approximates the length of possible keys in step ② using the Kasiski Examination and computes so-called *difference streams* of the document and plaintexts in step ③. With these streams at hand it is possible to look for the plaintexts directly in the obfuscated data. Again, with careful algorithmic engineering, this third step—that boils down to the multi-string search of difference streams—can be realized in linear time. If sufficient matches are identified, our method automatically derives the obfuscation key and reveals the full embedded code for further analysis, for example, by an anti-virus scanner or an human expert.

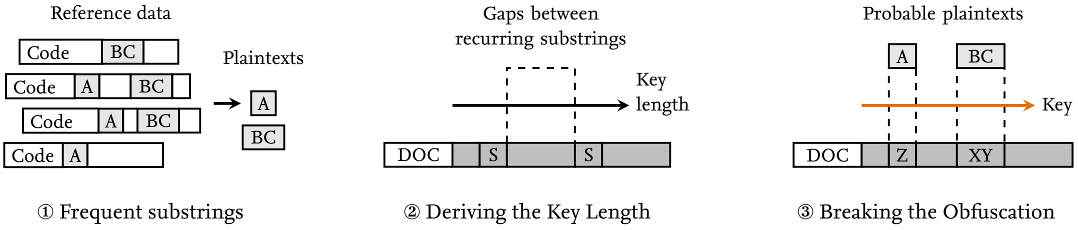


Figure 6.2: Schematic depiction of KANDI and its analysis steps: First, we extract frequent substrings that may be used as plaintexts. Second, we derive the length of the key used for obfuscation, based on which, we then mount a probable-plaintext attack. Figure taken from Wressnegger et al. (2013a).

Incorporating ROL and ROR. Finally, in order to increase the effectiveness of KANDI, we additionally consider transpositions using ROL and ROR instructions. ROL and ROR are each others inverse function, that is, when iterating over all possible shift offsets they generate exactly the same output but in different order. Furthermore, in most implementations these instructions operate on 8 bits only, such that the combined, overall number of transpositions to be tested is very small. Consequently, we simply add a ROL shift as a preprocessing step to KANDI. Although, we attempt to improve over a plain brute-force approach for breaking obfuscation, we consider the seven additional tests as a perfectly legit trade-off from a pragmatic point of view.

6.3 Determining Frequent Substrings

The deobfuscation performance of our method critically depends on a representative set of probable plaintexts. In the scope of this work, we focus on Windows PE files, as these are frequently used as initial step of an attack based on infected documents. However, our method is not restricted to this particular type of data and can also be applied to other representations of code from which probable plaintexts can be easily extracted, such as Flash animations or ELF objects. To determine these substrings efficiently, we process PE files using a suffix array (see Chapter 3) and extract all binary strings that appear in more than 50 % of the files.

Yamamoto and Church (2001) show how to determine the term frequency, tf, as well as the document frequency, df, of all substrings in the input in (quasi) linear time. They observe that a suffix array is partitioned in “LCP-delimited intervals”, that is, an interval of the suffix array $SA = \{SA_1, \dots, SA_n\}$ whose elements have a longest common prefix, LCP, which is larger than the LCP of the outermost elements with its adjoining elements, that are just about not part of the interval. More formally, these intervals are defined as

$$\langle i, j \rangle : \left\{ SA_i, SA_{i+1}, \dots, SA_j \,\middle|\, SIL_{i,j} > LBL_{i,j} \right\},$$

where $SIL_{i,j}$ is the *Shortest Interior LCP* determined by $\mathrm{lcp}(i, \dots, j)$, and $LBL_{i,j}$ refers to the *Longest Bounded LCP* given by $\max\left(\mathrm{lcp}(i-1, i), \mathrm{lcp}(j, j+1)\right)$. These intervals describe a class of substrings that (a) have the same term frequency and (b) the same document frequency (Yamamoto and Church, 2001):

$$class_{i,j} = \left\{ s_i^m \,\middle|\, LBL_{i,j} < m \leq SIL_{i,j} \right\},$$

with s_i^m being the substring of length m of the suffix at position i in the suffix array. The term frequency of a string $s \in class_{i,j}$ then is $tf(s) = j - i + 1$. Table 6.2 revises the example from Chapter 3 showing the non-trivial classes for the string banana: $class_{1,3}$ contains the string a with a term frequency of three and embeds $class_{2,3}$, which in turn consists of an and ana, each of which occur twice in the input string. Finally, $class_{5,6}$ covers substrings n and na both with a frequency of two. Additionally, there exist n trivial classes which interval includes a single line of the suffix array only. For these the term frequency is one and the shortest interior LCP is defined to be ∞, in order to cover all remaining substrings. For instance, $class_{6,6}$ contains nan and nana. Overall there exist n trivial class and at most

$n-1$ non-trivial classes. As LCP-delimited intervals either precede each other, or occur nested, but do never partially overlap, iterating them is rather straight-forward, such that all statistics can be compiled in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. For document frequency additionally an index, that maps positions in the suffix array to documents, is needed.

i	SA	LCP			
0	6		$		
1	5	0	a $		$class_{1,3}$
2	3	1	a n a $	$class_{2,3}$	
3	1	3	a n a n a $		
4	0	0	b a n a n a $		
5	4	0	n a $	$class_{5,6}$	
6	2	2	n a n a $		

Table 6.2: Depiction of LCP delimited arrays that define classes of substrings with the same term and document frequency based on the example of the string banana.

For the first step of our method for deobfuscating embedded malware, we then make use of this algorithm to extract the most common binary strings found in PE files distributed with off-the-shelf Windows XP and Windows 7 installations. Profitable plaintexts are, for instance, the DOS stub and its text, API strings, library names or code patterns such as push-call sequences.

6.4 Deriving the Key Length

In the second step, we determine the length of the key that has been used for obfuscation. For the Vigenère cipher two methods from classical crypt-analysis exist that can be used to retrieve the key length based on statistics over the ciphertext only: First, the "index of coincidence", that operates on rather strict assumptions about the distribution of the input alphabet which is particular useful for analyzing encrypted language. Second, the "Kasiski examination" which does not assume any knowledge of the input, but the fact that it contains substrings that repeat. Subsequently, we discuss both approaches in detail and inspect their suitability for breaking the obfuscation of embedded malware.

6.4.1 Index of Coincidence

A classic approach for determining the key length from ciphertexts is the *index of coincidence*, commonly denoted as κ (Friedman, 1922; Friedman and Callimahos, 1985). Roughly speaking it represents the ratio of how many bytes happen to appear at the same positions if you shift data against itself. Formally, the index of coincidence is defined as

$$\kappa = \frac{\sum_{i=1}^{256} f_i(f_i - 1)}{n(n-1)},$$

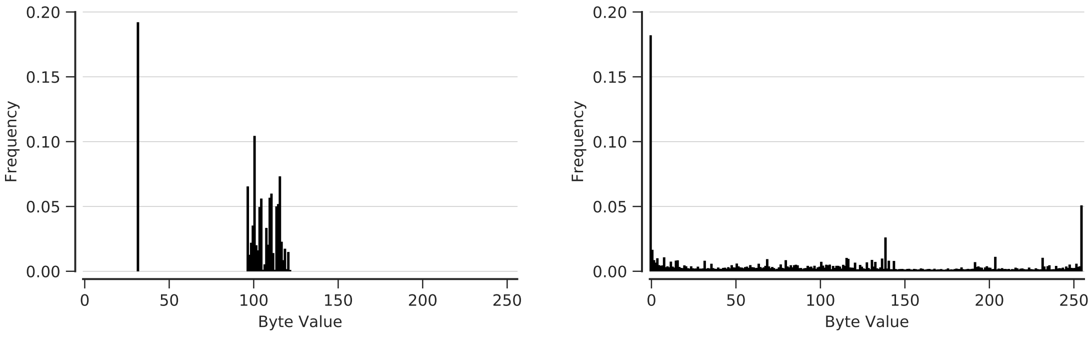
where f_i are the byte frequencies in data of n bytes. Under the condition that we know the index of some plaintext κ_p we are able to infer the key length l of the Vigenère cipher. It is estimated as the ratio of the differences of κ_p to the index of random data κ_r and the ciphertext κ_c:

$$l \approx \frac{\kappa_p - \kappa_r}{\kappa_c - \kappa_r}.$$

Key Recovery using Frequency Analysis. Next to determining the length of the obfuscation key, the index of coincidence may even enable recovering the key itself if certain conditions hold true. Natural languages tend to have a very characteristic frequency distribution of letters. For instance, in the English language the letter e is with more than 12 % the significantly most frequent letter in the alphabet (Lewand, 2000). Only topped by the space character, which is used in written texts in order to separate words.

This frequency distribution can be exploited to derive the key used for the encryption. As one can easily imagine, the actual frequency distribution does not change by simply replacing one character with another as in the case of a key of length $l = 1$. The larger the key length gets, the more the distribution is flattened out because identical letters may be translated differently depending on their position in the text. However, since it is possible to determine the length of the key beforehand, one can perform the very same frequency analysis on all characters that were encrypted with the same single-byte key $\hat{K}_i$.

Although effective in decrypting natural language text, key recovery using frequency analysis is not suitable for deobfuscating embedded malware. If the obfuscated code corresponds to regular PE files, the byte frequencies are almost equally distributed and



(a) Distribution of English text

(b) Distribution of PE files

Figure 6.3: Byte frequency distributions of English text and Windows PE files.

can hardly be discriminated, because executable code, header information and other types of data are mixed in this format. As an example, Figure 6.3 shows the byte frequency distributions of English text and PE files, where except for a peak at `\x00` the distribution of PE files is basically flat. The presented ciphertext-only attacks thus only provide means for determining the key length of Vigenère-based obfuscation, but without further refinements are not appropriate for actually recovering the key.

6.4.2 The Kasiski Examination

Another ciphertext-only attack for determining the key length is the so-called *Kasiski examination* (Kasiski, 1863). The underlying assumption of this method is that the original plaintext contains some identical substrings. Usually these patterns would be destroyed by the key; however, if two instances of such substrings are encrypted with the same portion of the key, the encrypted data contains a pair of identical substrings as well. This implies that the distance between the characters of these substrings is a multiple of the key length. Thus, by gathering identical substrings in the ciphertext, it is possible to support an assumption about the key length.

For our analysis of embedded malware, we use the Kasiski examination (Section 6.4.2) to directly inspect the raw bytes of a document—without any further parsing or processing of the file. The big advantage of this method over the index of coincidence is that we neither need to rely on the byte distribution of the original binary nor have to precisely locate the embedded malware. Furthermore, the Kasiski examination allows us to take multiple candidates of the key length into consideration. Depending on the amount of identical substrings that suggest a particular key length, we construct a ranking of candidates for later analysis. That way, it is possible to compensate for and recover from misinterpretations.

However, finding pairs of identical substrings in large amounts of data needs careful algorithmic engineering in order to work efficiently. We again make use of suffix arrays for determining identical substrings in linear time in the length of the analyzed document. In particular, we are reusing the algorithm used for efficiently extracting substring statistics as presented in Section 6.3. Since the Kasiski examination only states that the distances between identical substrings in the ciphertext refer to *multiples* of the key length, it is necessary to also examine the integer factorization thereof. Fortunately, there exists a shortcut to this factorization step that works very well in practice: If KANDI returns a key that repeats itself, e.g., `13 37 13 37`, this indicates that we correctly derived the key but under an imprecise assumption of the key length ($l = 4$ rather than 2). In such cases we simply collapse the repeating key and correct the key length accordingly.

6.5 Breaking the Obfuscation

In this section we look at two fundamentally different strategies for breaking the obfuscation of embedded malware. Both approaches however are based on first retrieving the encryption key or a set of candidates, with which the obfuscation can then be reversed. We begin with discussing the aspects of *brute-force attacks and heuristics* for determining the encryption key. Once the key is revealed, deobfuscating the malware is trivial. However, depending on the key length the search space is sizeable, such that runtime quickly becomes an issue. We thus then turn to a more evolved approach using *probable-plaintext attacks* that enables us to deobfuscate embedded malware at scale. To this end, both the previously collected most frequent substring in Windows executables as well as an accurate estimate of the key length are of the essence. In the following, we discuss both attack types in detail.

6.5.1 Brute-force Attacks and Heuristics

A straightforward way of approaching malware obfuscations is to brute-force the key used by the malware author. There are two basic implementations for such an attack: First, one encrypts all plaintext patterns that are assumed to be present in the original binary with each and every key and tries to match those. Second, one decrypts the binary or parts of it and looks for the presence of the plaintext as a usual signature engine would do. In both cases a valid key is derived if a certain amount of plaintexts match. For short keys, this approach is both fast and effective. In practice, brute-force attacks prove to be a valuable tool for analyzing malware obfuscated using keys up to 2 bytes (Boldewin, 2009; Stevens, 2014).

Theoretically, an exhaustive search over the complete key space can be used to also derive keys with more than 2 bytes. This however comes at the price of runtime performance. For a key length of only 4 bytes there are more than 4.2 billion combinations that need to be checked in the worst case. This clearly exceeds the limits of what is possible in the scope of the deobfuscation of embedded malware. Even worse, 4-byte and 8-byte keys fit the registers of common CPU architectures and therefore, do not require much different deobfuscation routines. In fact, the underlying logic is identical to the use of single-byte keys and the code size is only marginally larger as illustrated in Figure 6.1.

A more clever way of approaching the problem is by relying on the structure of embedded malware binaries, which are often PE files. In this format `\x00` bytes are used as padding for sections and headers which gives rise to an heuristic. We recall from Section 6.1.1 that the binary operation $\circ$ has an identity element, which simply is 0 for XOR as well as ADD instructions. Therefore, whenever a large block of `\x00` bytes is encrypted, the key is revealed multiple times and can be read off without extra effort. Hence, once a highly repetitive string is spotted in obfuscated data, deobfuscation is a simple task for a malware analyst. According to our tests the very same technique is leveraged in a proprietary system for the analysis of malware called *Cryptam* (Malware Tracker Ltd., 2012-2018). While effective in many cases when a full binary including padding is obfuscated, this heuristic fails when a malware does not encrypt `\x00` bytes. Furthermore, such an approach cannot differ between variants of Vigenère ciphers. Since XOR and ADD have the same identity element, there is no way to decide which one was used for obfuscation in this setting.

6.5.2 Probable-Plaintext Attacks

To effectively determine the key used in a Vigenère-based obfuscation, we consider classic attacks based on known and probable plaintexts. We refer to these attacks as *probable-plaintext attacks*, as we cannot guarantee that a certain plaintext is indeed contained in an obfuscated malware binary. Equipped with an expressive set of probable plaintexts and an estimate of the key length, it is now possible to mount an attack against the obfuscation. The central element of this step is the key elimination that enables us to look for probable plaintexts within the obfuscated data and derive the used key automatically. Again, KANDI directly operates on the raw bytes of a document and thereby avoids parsing the file.

Key Elimination. The idea of this technique is to determine a relation between the plaintext and ciphertext that does not involve the key: Namely, the *difference* of bytes that are encrypted with the same part of the key. Formally, for a key byte $\hat{K}_i$ this difference can be expressed using the inverse operation $\circ^{-1}$ as:

$$C_i \circ^{-1} C_{i+l} = (M_i \circ \hat{K}_i) \circ^{-1} (M_{i+l} \circ \hat{K}_i) = M_i \circ^{-1} M_{i+l}.$$

Note that this relation of differences only applies if the operator used for the Vigenère cipher induces a commutative group. For example, if we plug in the popular instructions XOR and ADD from Table 6.1, the difference of the obfuscated bytes C_i and C_{i+l} allows to reason about the difference of the corresponding plaintext bytes:

$$\begin{aligned} C_i \oplus C_{i+l} &= (M_i \oplus \hat{K}_i) \oplus (M_{i+l} \oplus \hat{K}_i) = M_i \oplus M_{i+l} \\ C_i - C_{i+l} &= (M_i + \hat{K}_i) - (M_{i+l} + \hat{K}_i) = M_i - M_{i+l}. \end{aligned}$$

Based on this observation, we can implement an efficient probable-plaintext attack against Vigenère ciphers. Given a plaintext $P = P_1 \dots P_m$, we introduce the *difference streams* ΔP and ΔC. If the difference streams match at a specific position and the plaintext P is sufficiently large, we have successfully determined the occurrence of a plaintext in the obfuscated data. In particular, we compute the difference stream

$$\Delta P = (P_1 \circ^{-1} P_{1+l}) \dots (P_{m-l} \circ^{-1} P_m)$$

for the plaintext P and compare it against each position i of the ciphertext C using the corresponding stream

$$\Delta C = (C_i \circ^{-1} C_{i+l}) \dots (C_{i+m-l} \circ^{-1} C_{i+m}).$$

Using this technique, we can efficiently search for probable plaintexts in data obfuscated using a Vigenère cipher without knowing the key. This enables us to check for common strings in the obfuscated code, such as header information, API functions and code stubs. Once the position of a probable plaintext is found it is possible to derive the used key by applying the appropriate inverse operation: $K_j = C_{i+j} \circ^{-1} P_{i+j}$ with i being the position where the difference stream of a probable plaintext matches. The more plaintexts match in the obfuscated code, the more reliably the key can finally be determined.

Robust Key Recovery. If a probable plaintext is longer than the estimated key length, the overlapping bytes can be used to reinforce our assumption about the key. To this end, we define the *overlap ratio* r that is used to specify how certain we want to be about a key candidate. The larger r is, the stricter KANDI operates and the more reliable is the key. If we set $r = 0.0$, an usual match of plaintexts is enough to support the evidence of a key candidate. This means that we will end up with a larger amount of possibly less reliable hints. Our experiments show that for the grand total incorrect guesses will average out and in many cases it is possible to reliably deobfuscate embedded malware.

If a more certain decision is desired the overlap ratio r can be increased. However, for larger values of r we require longer probable plaintexts: $r = 0.0$ only requires a minimal overlap, $r = 0.5$ already half of the probable plaintext's length and $r = 1.0$ twice the size. As an example, if the estimated key length is 4 and $r = 0.5$, only plaintexts of at least 6 bytes are used for the attack. Depending on the approach chosen to gather probable plaintexts, it might happen that the length of the available plaintexts ends up being the limiting factor for the deobfuscation. We will evaluate this in the next section.

Implementation. Both, determining the key length using the the Kasiski examination and collecting evidence for key candidates, require to keep track of large amounts of substrings and their counts. Count-Min sketches are especially well suited for this use case. Moreover, the probabilistic nature of the presented attack is able to compensate for minor inaccuracies in counting. With respect to the used memory, Count-Min sketches significantly improve over hash tables, which store substrings explicitly in order to allow exact counting (see Chapter 3). The difference streams, on the other hand, can be efficiently searched for by constructing prefix trees from them. These enable looking for multiple search strings at once in linear time using the Aho-Corasick algorithm.

6.6 Evaluation

We proceed to evaluate the deobfuscation capabilities and runtime performance of KANDI empirically. Since it is difficult to determine whether embedded malware in the wild is actually using Vigenère-based obfuscation or not, we start off with a series of controlled experiments in Section 6.6.1. In Section 6.6.2 we then continue to evaluate KANDI on real-world malware in Word, Powerpoint and RTF documents as well as different image formats. We need to stress that this collection contains malware with unknown obfuscation. Nonetheless, KANDI is able to expose obfuscated malware in every fourth file, thereby empirically proving that (a) Vigenère ciphers are indeed used in the wild and that (b) our method is able to reliably reveal the malicious payload in these cases.

6.6.1 Controlled Experiments

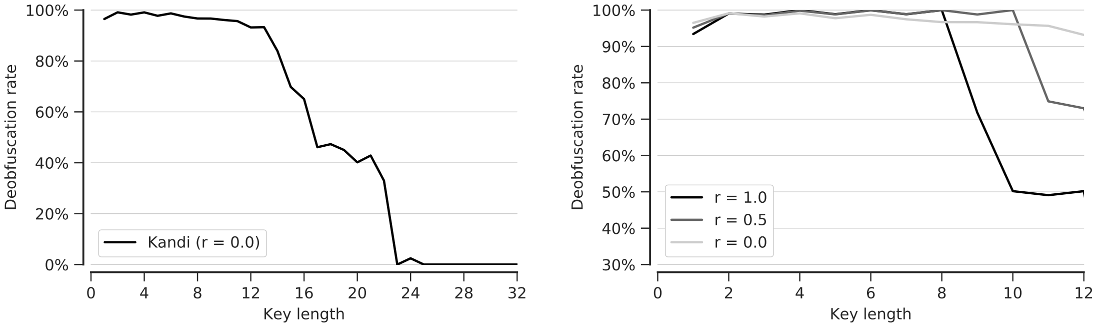
To begin with, we evaluate KANDI in a controlled setting with known ground truth, where we are able to exactly tell if a deobfuscation attempt was successful or not. In particular, we conduct two experiments: First, we obfuscate plain Windows PE files and apply KANDI to them. In the course of that, we measure the runtime performance and throughput of our approach. Second, the obfuscated PE files are embedded in benign Word documents in order to show that KANDI not only works on completely encrypted data, but is also capable of deobfuscating files embedded inside of documents.

Evaluation Datasets. In order to create a representative set of PE files for the controlled experiments, we simply gather all PE files in the system directories of Windows XP SP3 (`system` and `system32`) and Windows 7 (`System32` and `SysWOW64`). This includes stand-alone executables as well as libraries and drivers and yields a total of 4,780 files. We randomly obfuscate each of the PE files with a Vigenère cipher using either XOR, ADD or SUB. We draw random keys for this obfuscation and vary the key length from 1 to 32 bytes, such that we finally obtain 152,960 (32×4,780) unique obfuscated PE files.

To study the deobfuscation of embedded code, we additionally retrieve one unique and malware-free Word document for each PE file from VirusTotal and use it as host for the embedding. Malware appearing in the wild would be embedded at positions compliant with the host's file format. This theoretically provides valuable information where to look for embedded malware. As KANDI does not rely on parsing the host file, we simply inject the obfuscated PE files at random positions. We end up with a total of 152,960 unique Word documents each containing an obfuscated PE file.

Deobfuscation of Obfuscated PE Files. To demonstrate the capability of our method to break Vigenère-based obfuscations, we first apply KANDI to the 152,960 obfuscated PE files. The probable plaintexts for this experiment are retrieved as described in Section 6.3 without further refinements. Figure 6.4a shows results for this experiment, where the key length is plotted against the rate of deobfuscated PE files. For key lengths up to 13 bytes, the obfuscation can be reliably broken with a success rate of 93 % and more. This nicely illustrates the potential of KANDI to automatically deobfuscate malware. We also observe that the performance for keys longer than 13 bytes drops. While our approach is not capped to a specific key length, the limiting factor at this point is the collection of plaintexts and in particular the length of those.

To study the impact of the plaintext length, we additionally apply KANDI with different values for the overlap ratio r as introduced in Section 6.5. The corresponding deobfuscation



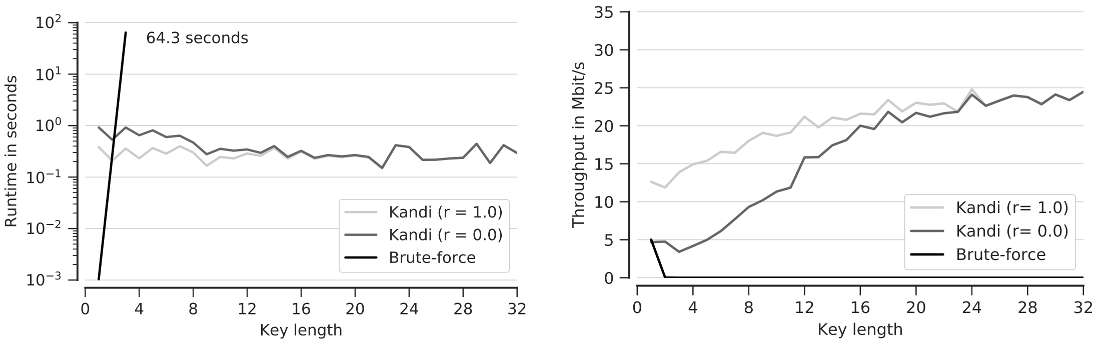
(a) Obfuscated PE files

(b) Influence of the overlap ratio

Figure 6.4: Deobfuscation performance of Kandi on obfuscated PE files. Figure (b) additionally shows the performance for different overlap ratios.

rates are visualized in Figure 6.4b. Although a high value of r potentially increases the performance, it also reduces the number of plaintexts that can be used. If there are too few usable plaintexts, it gets difficult to estimate the correct key. As a result, Kandi attains a deobfuscation performance of almost 100 % for $r = 1.0$ if the keys are short, but is not able to reliably break obfuscations with longer keys.

Runtime Performance. We additionally examine the runtime performance of Kandi. For this purpose, we randomly draw 1,000 samples from the obfuscated PE files for each key length and repeat the previous experiment single-threaded on an Intel Core i7-2600K CPU at 3.40 GHz. As baseline for this experiment, we implement a generic brute-force attack that is applied to the first 256 bytes of each file. Due to the defined starting point and the typical header structure of PE files 256 bytes are already sufficient to reliably break the obfuscation in this setting. Note that this would not be necessarily the case for embedded malware.



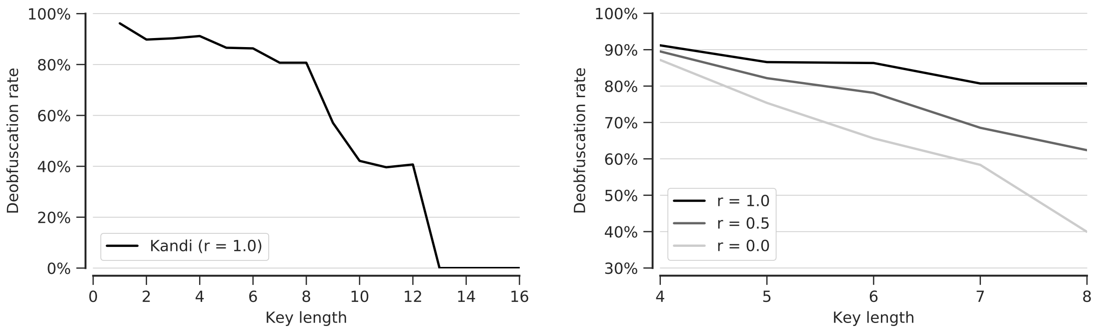
(a) Average runtime per file

(b) Average throughput

Figure 6.5: Runtime performance of Kandi in comparison to a brute-force attack on a batch of 1,000 randomly drawn obfuscated PE files.

The results of this experiment are shown in Figure 6.5 where the runtime and throughput of each approach are shown on the y-axis and the key length on the x-axis. Obviously, the brute-force attack is only tractable for keys of at most 3 bytes. By contrast, the runtime of Kandi does not depend on the key length and the method attains a throughput of 16.46 Mbit/s on average, corresponding to an analysis speed of 5 files of ∼400 kB per second. Consequently, Kandi's runtime is not only superior to brute-force attacks but also significantly below dynamic approaches like OmniUnpack (Martignoni et al., 2007) or PolyUnpack (Royal et al., 2006) and thus beneficial for analyzing embedded malware at large scales.

Deobfuscation of Injected PE Files. As last controlled experiment, we study the deobfuscation performance of Kandi when being operated on obfuscated PE files that have been injected into Word documents. Figure 6.6a shows the results of this experiment. For keys with up to 8 bytes, our method deobfuscates most of the injected PE files—without requiring the document to be parsed. Moreover, we again inspect the influence of the overlap ratio r in this setting. Similar to the previous experiment, a larger value of r proves beneficial for short keys, such that keys up to 8 bytes are broken with a success rate of 81 % and more. This influence of the overlap ratio gets evident for keys between 4 and 8 bytes as illustrated Figure 6.6b. For keys of length $l = 8$, a high value of r even doubles the deobfuscation performance in comparison to the default setting.



(a) Obfuscated PE files

(b) Influence of the overlap ratio

Figure 6.6: Deobfuscation performance of Kandi on Word documents containing obfuscated PE files. Figure (b) additionally shows the performance for different overlap ratios.

Due to this, we use an overlap ratio of $r = 1.0$ for the following experiments on real-world malware. We expect embedded malware found in the wild to mainly use keys of 1 to 8 bytes. The reasons for this assumption is that such keys fit into CPU registers and therefore implementations are more compact. Furthermore, 4-byte keys are already intractable for brute-force attacks.

6.6.2 Real-World Experiments

To top off our evaluation we proceed to demonstrate how Kandi is able to deobfuscate and extract malware from samples seen in the wild. To this end, we have acquired four datasets of real-world malware embedded in documents and images with different characteristics.

Malware Datasets. Embedded malware is typically executed by exploiting vulnerabilities in document viewers. For the first dataset (*Exploits 1*) we thus retrieve all available Word, Powerpoint and RTF documents from VirusTotal that are detected by an anti-virus scanner and whose label indicates the presence of an exploit, such as `exploit.msword` or `exploit.ole2`. Similarly, we construct the second dataset (*Exploits 2*) by downloading all documents that are tagged as one of the following exploits: `CVE-2003-0820`, `CVE-2006-2492`, `CVE-2010-3333`, `CVE-2011-0611`, `CVE-2012-0158` and `CVE-2013-0634`.

As our method specifically targets PE files embedded in documents, we additionally compose two datasets of malware droppers. The first set (*Dropper 1*) contains all available Word, Powerpoint and RTF documents that are detected by an anti-virus scanner and whose label contains the term `dropper`. The second dataset (*Dropper 2*) is constructed similarly by retrieving all malicious images labeled as dropper. An overview of all four datasets is given in Table 6.3. We deliberately exclude malicious PDF files from our analysis, as this file format allows to incorporate JavaScript code. Consequently, the first layer of obfuscation is often realized using JavaScript encoding functions, such as Base64 and URI encoding. Such encodings are not available natively for other formats and hence we do not consider PDF files in this work.

Dataset name	**Type**	**`Formats`**	**Samples**
Exploits 1	Documents	`DOC, PPT, RTF`	992
Exploits 2	Documents	`DOC, PPT, RTF`	237
Dropper 1	Documents	`DOC, PPT, RTF`	336
Dropper 1	Images	`PNG, GIF, JPG, BMP`	52
Total			1,617

Table 6.3: Overview of the four datasets of malicious documents and images.

Deobfuscation of Embedded Malware. We proceed to apply Kandi to the collected embedded malware. Due to minor modifications by the malware author, it is not always possible to extract a valid PE file. To verify if a deobfuscation attempt was successful, we thus make use of a PE checker based on strings such as Windows API function (e.g., `LoadLibrary`, `GetProcAddress`, `GetModuleHandle`) and library names as found in the import table (e.g., `kernel32.dll`, `user32.dll`). Additionally, we look for the MZ and PE header signatures and the DOS stub. We consider a deobfuscation successful if either a valid PE file is extracted or at least five function or library names are revealed in the document.

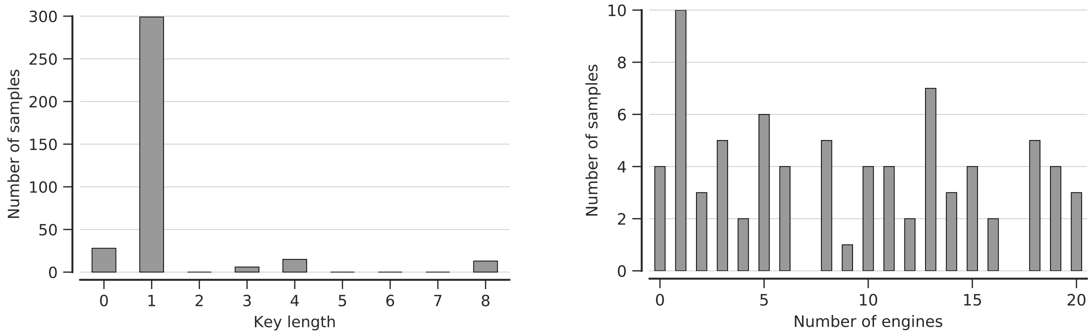
We observe that for 359 of the samples no deobfuscation is necessary, as the embedded malware is present in clear. Kandi identifies such malware by simply returning an obfuscation key of `0x00`. We support this finding by applying the PE checker described earlier. The remaining 1,258 samples are assumed to be obfuscated. Every fourth of those samples contains malware obfuscated with the Vigenère cipher and is deobfuscated by Kandi. That is, our method automatically cracks the obfuscation of 334 samples and extracts the embedded malware—possibly multiple files per sample. Table 6.4 details the results for the individual datasets. A manual analysis of the remaining files on a sample basis does not reveal obvious indicators for the Vigenère cipher and we conclude that Kandi deobfuscates most variants used in real-world embedded malware.

Dataset	Not Obfuscated	Obfuscated	Deobfuscated by Kandi
Exploits 1	211	781	180 (23.1 %)
Exploits 2	35	203	64 (31.7 %)
Dropper 1	86	250	81 (32.4 %)
Dropper 2	27	25	9 (36.0 %)
Total	359	1,258	334 (26.6 %)

Table 6.4: Deobfuscation performance of Kandi on real-world malware. The last column details the number of samples that were successfully deobfuscated.

Figure 6.7a shows the distribution of the key lengths discovered by Kandi. The majority of samples is obfuscated with a single-byte key and seems to be in reach for brute-forcing. However, to do so one would need to precisely locate the encrypted file, which is not trivial. Moreover, our method also identifies samples with longer keys ranging from 3 to 8 bytes that would have been missed without the help of Kandi. Rather surprising are those samples that use 3 bytes as a key. One would suspect these to be false positives, but we have manually verified that these are correctly deobfuscated by our method.

As the final step of this experiment, we analyze the extracted malware binaries with 46 different anti-virus scanners provided by VirusTotal. Since some of these scanners are prone to errors when it comes to manipulated PE headers, we consider only those 242 deobfuscated malware binaries that are valid PE files (conform to the format specification). The number of detections for each of these files is shown in Figure 6.7b. Several binaries are poorly detected by the anti-virus scanners at VirusTotal. For instance, 19 % (46) of the binaries are identified by less than 10 of the available scanners. This result suggests that the extracted binaries are unknown to a large portion of the anti-virus companies—likely due to the lack of tools for automatic deobfuscation.



(a) Detected key lengths

(b) VirusTotal detections

Figure 6.7: (a) Distribution of key lengths detected by Kandi; (b) Number of anti-virus scanners detecting the extracted malware binaries.

Finally, the analyzed binaries also contain several samples of the MiniDuke malware that has been discovered in early February 2013 (CrySyS Malware Intelligence Team, 2013), at about the same time our initial experiments have been conducted. A few months before, this threat has been completely unknown, such that we are hopeful that binaries deobfuscated by Kandi help the discovery of new and previously unknown malware.

6.7 Related Work

The analysis of embedded malware has been a vivid area of research in the last years, in particular due to the increasing usage of malicious documents in targeted attacks (e.g., Bencsáth et al., 2012; Trend Micro Threat Research Team, 2012; CrySyS Malware Intelligence Team, 2013). Several concepts and techniques have been proposed to locate and examine malicious code in documents. Our approach is related to several of these, as we discuss in the following.

6.7.1 Analysis of Embedded Malware

First methods for the identification of malware in documents have been proposed by Stolfo et al. (2007) and Li et al. (2007). Both make use of content-based anomaly detection for learning profiles of regular documents and detecting malicious content as deviation thereof. This work has been further extended by Shafiq et al. (2008), which refine the static analysis of documents to also locate the regions likely containing malware. Although effective in spotting suspicious content, these methods are not designed to deobfuscate code and thus are unsuitable for in-depth analysis of embedded malware. Another branch of research has thus studied methods for analyzing malicious documents at runtime, thereby avoiding the direct deobfuscation of embedded code (e.g., Li et al., 2007; Engelberth et al., 2009; Schreck et al., 2012). For this dynamic analysis, the documents under investigation are opened in a sandbox environment, such that the behavior of the application processing the documents can be monitored and malicious activities detected. These approaches are not obstructed by obfuscation and can reliably detect malicious code in documents. The monitoring at run-time, however, induces a significant overhead which is prohibitive for large-scale analysis or detection of malware at end hosts.

Recently, a large body of work has focused on malicious PDF documents. Due to the flexibility of this format and its support for JavaScript code, these documents are frequently used as vehicles to transport malware (Stevens, 2011). Several contrasting methods have been proposed to spot attacks and malware in JavaScript code (e.g., Cova et al., 2010; Laskov and Šrndić, 2011) and the structure of PDF files (e.g., Smutz and Stavrou, 2012; Šrndić and Laskov, 2013). While some malicious PDF documents make use of Vigenère-based obfuscation, other hiding strategies are more prominent in the wild, most notably the dynamic construction of code. As a consequence, we have not considered PDF documents in this work, yet the proposed deobfuscation techniques also apply to Vigenère ciphers used in this document format.

6.7.2 Deobfuscating and Unpacking Malware

Aside from specific work on embedded malware, the deobfuscation of malicious code has been a long-standing topic of security research. In particular, several methods have been developed to dynamically unpack malware binaries, such as PolyUnpack (Royal et al., 2006), OmniUnpack (Martignoni et al., 2007) and Ether (Dinaburg et al., 2008). These methods proceed by monitoring the usage of memory and identifying unpacked code created at runtime. A similar approach is devised by Sharif et al. (2009), which defeats emulation-based packers using dynamic taint analysis. These unpackers enable a generic deobfuscation of malicious code, yet they operate at runtime and, similar to the analysis of documents in a sandbox, suffer from a runtime overhead.

Due to the inherent limitations of static analysis, only few approaches have been proposed that are able to statically inspect obfuscated malware. An example is the method by Jacob et al. (2012) that, similar to Kandi, exploits statistical artifacts preserved through packing in order to analyze malware. The method does not focus on deobfuscation but rather efficiently comparing malware binaries and determining variants of the same family without dynamic analysis.

6.7.3 Probable-Plaintext Attacks

Attacks using probable and known plaintexts are among the oldest methods of cryptography. The Kasiski examination used in Kandi dates back to 1863 (Kasiski, 1863) and similarly the key elimination of Vigenère ciphers is an ancient approach of cryptanalysis (see Schneier, 1996). Given this long history of research and the presence of several strong cryptographic methods, it would seem that attacks against weak ciphers are largely irrelevant today. Unfortunately, these weak ciphers regularly slip into implementations of software and thus probable-plaintext attacks based on classic techniques are still successful, as for instance in the cases of WordPerfect (Bergen and Caelli, 1991) and PKZIP (Stay, 2002).

To the best of our knowledge, Kandi is the first method that applies these classic attacks against obfuscation used in embedded malware. While some high-profile attack campaigns have already moved to stronger ciphers, such as RC4 or TEA, the convenience of simple cryptography and the risk of introducing detectable patterns with involved approaches continues to motivate attackers to use weak ciphers for obfuscation.

Chapter 7

Conclusion and Outlook

Manually crafting detection rules is by no means sufficient anymore to keep pace with the constant growth of new threats on the Internet. Even automatically generated signatures used by traditional methods often do not scale well, as they lack context and thus generalize poorly to new attack variants. We hence are in urgent need for alternative means of attack detection that are able to cope with the increasing automation of attacks.

In this thesis, we have developed methods to detect attacks using machine learning that push forward reactive computer security. While machine learning is a promising tool to provide generic and automated detection, it is not void of bias. To be effective, learning-based approaches require constant training over time: Both, the classification of malware, as well as the notion of normality established for anomaly detection need to be retrained and kept up-to-date. We have developed detectors that are trained in linear time with up to 580 Mbit/s and are able to effectively provide timely protection. We thus counteract the trend of compensating increasing runtime requirements with resources. To this end, the underlying models are built using algorithms from the field of efficient string processing and analyzed using implementations based on probabilistic data structures, such as Bloom filters and Count-Min sketches, that allow to efficiently store and access features.

In particular, we have used language models and have shown that these are especially well suited for efficiently modeling the content and structure of attacks in a generic manner. In addition, we have developed suitability criteria of data in this representation for the two prevalent learning-based attack detection schemes, classification and anomaly detection, to decide when to favor the one over the other. For both schemes, we have proposed highly performant solutions, first, for detecting Flash-based malware and, second, for detecting network intrusions in industrial control systems. The developed methods do not only meet our expectations with respect to runtime complexity, but also outperform related approaches by up to an order of magnitude in detection performance. We thereby demonstrate that effective and efficient machine learning for attack detection is possible in practice. Moreover, we have shown that the same techniques can further be used for revealing malicious content that lies hidden in passive file formats, such as Microsoft Office documents. We have implemented a method that operates irrespective of the used key length in constant time, and thus succeeds where previous approaches fail.

In the following Section 7.1, we look upon the results of the individual components presented in this thesis in more detail, before we discuss possible extensions and future directions of research in Section 7.2.

7.1 Summary of the Results

The results of this thesis fall into two categories: First, we have provided insight in fundamental properties of the problem of attack detection. We have discussed current factors for the success of malware and targeted attacks, arguing for the necessity of advanced attack detection, and have compiled suitability criteria for data represented by language models for learning-based detection. Second, we have developed three specific approaches for (1) detecting malware using classification, (2) detecting network intrusions using anomaly detection, and (3) detecting embedded malware. All put a special focus on efficiently processing data in linear time. Subsequently, we address the results of each of these parts in detail.

Necessity of Advanced Attack Detection (Chapter 2). We have identified two factors that are crucial for the success of attacks in practice: The availability of an attack vector such as software vulnerabilities and second, existing methods for attack detection that are not able to provide timely protection. In line with this, we have first reviewed 64-bit migration vulnerabilities as one example for the diverse landscape of software flaws and show that even in mature, well-tested software, developers unjustifiably treat the unsigned type `size_t` and (`unsigned`) `int` as equal to a large extend. Second, we have inspected the prevalence of static signature matching as a fall-back mechanism in modern virus scanners by conducting a series of black-box tests that allows to derive such signatures. For 38 % of our malware data set it has been possible to identify specific pattern-based signatures, two third of which can be implanted into arbitrary files at random locations, indicating that these signatures are checked without context. Effective attack detection can only succeed, if detection methods incorporate context and semantics of malicious code. Dynamic approaches as well as language models, as proposed in this thesis, implicitly include these relations and thus are more suitable for reliably detecting attacks in practice.

Language Models and Data Suitability (Chapter 3). We have studied the use of language models, and models based on n-grams in particular, for classification as well as anomaly detection tasks. As result, we have defined prerequisites that allow to decide whether one of the two schemes is applicable. Moreover, we have developed three suitability criteria that can be computed from n-gram data prior to the design of a detection method. These criteria enable a practitioner to assess the complexity of the detection task and help to select an appropriate learning scheme. Our suitability criteria, however, only provide indications for favoring one scheme over the other and should not be considered alone for designing a detection method. Depending on a particular detection task, it may be possible to operate a learning scheme in a slightly imperfect setting for the sake of other constraints, such as run-time performance. Nonetheless, the criteria can guide the development of detection methods and help to avoid tedious experiments with different learning schemes.

Detecting Malware using Classification (Chapter 4). We have presented a learning-based classification approach using language models and have particularly looked at Flash-based malware. At this, we have combined a structural analysis of the Flash container format with guided execution of embedded ActionScript code—a lightweight and pragmatic form of multi-path exploration. The resulting malware profiles have then be used as basis for automatic, learning-based detection. Our evaluation on 26,600 Flash samples shows that our method covers 50 % more code than a naive execution, and exposes indicative patterns that would have been missed otherwise. This increased coverage enables to identify 90–95 % of malware shortly after its appearance in the wild, where training is conducted in linear runtime with up to 580 Mbit/s. Our method can be used to bootstrap the current process of signature generation and point an analyst to novel malware samples and thereby provides a valuable step towards the timely protection of end users. Moreover, we have shown that simpler learning approaches based on substring statistics can, with certain trade-offs, keep pace with more advanced learning schemes.

Detecting Network Intrusions using Anomaly Detection (Chapter 5). In contrast to prior believe, we have shown that content-based anomaly detection is very well applicable to protocols with high-entropy data. The frequency values of features carry valuable information for modeling normality and constructing detection models—especially for proprietary binary protocols. The combination of learning message-type specific prototype models and de-noising these enables attack detection in linear time at 30 Mbit/s with particularly few false positives. In an extensive evaluation using 210 GiB of network traffic from two industrial facilities, we have shown that our method is able to significantly improve detection performance compared to related approaches. For instance, our method detects 97.1 % of the attacks in our dataset with as few as 2 false-alarms out of 100,000 network messages, while related approaches only yield 16.5 % with a 500× higher false-positive rate of 1 %. Moreover, we have studied the influence of polymorphic blending attacks on our detector and have shown that our method's de-noising functionality effectively limits an attacker's reach of play and improves the resistance against this type of mutation attacks.

Detecting Embedded Malware using Substring Statistics (Chapter 6). We have proposed a method based on the analysis of substring statistics that exploits well-known weaknesses of obfuscations based on the Vigenère cipher, as used by embedded malware. We have empirically demonstrated the efficiency of this approach on real malware, where our method is able to uncover the code of every fourth malware in popular document and image formats in constant time. While our approach targets only one of many possible obfuscation strategies, it helps to strengthen current defenses against embedded malware and proofs the feasibility of using substring statistics for malware detection. With up to 25 Mbit/s, our method is fast enough to be applied on end hosts and thereby enables regular virus scanners to directly inspect deobfuscated code and to better identify some types of embedded malware. Moreover, by statically exposing details of the obfuscation, such as the key and the operations used, our method can also be applied for the large-scale analysis of malicious documents and is complementary to time-consuming dynamic approaches.

7.2 Future Work

Based on this work a number of extensions and lines of research are possible. In the following, we discuss some that appear to be particularly promising, such as ways to improve learning efficiency to speed up the overall process, hardware acceleration of our methods, adversarial learning, as well as alternative alphabets for representing data and different platforms.

Learning Efficiency. Neither traditional methods driven by manual or semi-automated analysis, nor approaches using machine learning detect one hundred percent of new threats. Looking at historic data of detections, as provided by VirusTotal for malware, allows to identify samples that, apparently, have been particularly difficult to detect (Wressnegger and Rieck, 2017). Incorporating these samples during training and validation may help to steer the learning process such that less data needs to be processed. In contrast to improving the runtime of the underlying method, as presented in this thesis, this may help to speed up training by improving the quality of the used data.

Hardware Acceleration. The most obvious potential for runtime improvement however is to use dedicated hardware for detection. Such attempts are subject of ongoing research for intrusion detection, for instance, by incorporating the GPU (Vasiliadis et al., 2008), multi-core machines (Nam et al., 2015), or new algorithms that reduce memory access and cache misses (Choi et al., 2016) for multi-pattern matching. The probabilistic data structures we use heavily rely on hash computations and maintain a small memory footprint, and thus are well suited for being implemented in hardware. Recent work has demonstrated implementations of Bloom filters (Nathaniel McVicar, 2017) and Count-Min sketches (Tong and Prasanna, 2015, 2018) that enable a throughput of several Gigabits per second.

Adversarial Learning. Attacks on machine learning methods and learning in adversarial settings have gained a lot of attention recently. This ranges from data poisoning (Barreno et al., 2006), over the evasion of detectors (Fogla and Lee, 2006; Biggio et al., 2013), to inference attacks (Shokri et al., 2017; Nasr et al., 2018). With our experiments, showing that pruning rare features increases the resistance against mimicry attacks, we took a first step towards exploring the use of our methods in adversarial settings. More recent attacks and the sensitivity to membership inference has not been investigated so far. These however pose some interesting questions for future research.

Alternative Alphabets. Choosing appropriate input representations and the right alphabet for language models is crucial for the effectivity of our methods. Compression-based metrics such as the Normalized Compression Distance (Li et al., 2004; Cilibrasi and Vitányi, 2005) and extensions to it (Raff and Nicholas, 2017, 2018) have recently been shown to allow for efficiently measuring the similarity of string-based data. Using compression tables or, more general frequent substrings of varying lengths, as an alphabet may be a worthwhile choice to abstract the input data, reduce and implicitly cleanse its representation, and improve detection performance, while simultaneously speeding up detection.

Different Platforms. Finally, the small memory footprint and the runtime efficiency of the methods presented in this thesis are very well suitable for platforms with hardware restrictions, such as cars, trains or even airplanes. The automotive sector currently strives for ever-increasing interconnection between road users and devices. This opens the gates for remote attackers (Koscher et al., 2010; Checkoway et al., 2011; Miller and Valasek, 2015) and undermines the need for reliable, on-board attack detection. Applying our methods to alternative networks and communication systems, such as the CAN bus, is an interesting field of application. In contrast to this thesis, such work would lay stronger focus on the runtime efficiency during detection.

Chapter A

Supplementary Information

A.1 Data models

A data model defines the width of integer types for a specific platform. Table A.1 provides an overview of common data models used in the present and past, exemplary operating systems using them, as well as the number of bytes assigned to each type. For all models, the width of pointers and the `size_t` type correspond to the architectures' register size, e.g.,IP16 and LLP64 specify the size of pointers as 2 byte and 8 byte, respectively.

The motivation behind the different definitions of basic integer types lies in preserving their relations as good as possible when migrating code between data models. Due to our focus on the transition from 32 bit to 64 bit, ILP32 serves as a reference point in this paper, as it is used on most 32-bit architectures. That is, we assume that a given program works as intended for ILP32 and look upon the differences when compiling the same program using a 64-bit data model.

If we compare ILP32 to LLP64 and LP64, as used by 64-bit Windows and most 64-bit Unix systems, respectively, we see that the type `int` is 32-bit wide for all three data models. While for ILP32 this means that `int` and pointers have the same width, on 64-bit data models `int` is only half as wide as the pointer type. The same holds true for the type `long` on LLP64. As a consequence, on both 64-bit data models an `int` variable can no longer be used to address the full range of memory. While there also exist other 64-bit data models, such as ILP64 and SILP64, these are only used on few platforms only and define the same width for `int`, `long` and pointers, which renders migrating code less problematic.

data model	**IP16**	**IP16L32**	**LP32**	**ILP32**	**LLP64**	**LP64**	**ILP64**	**SILP64**
data type	(PDP-11 Unix)		(Win16)	(Win32, Linux)	(Win64)	(Linux)	(HAL)	(UNICOS)
`pointer/size_t`	2	2	4	4	8	8	8	8
`short`	-	2	2	2	2	2	2	8
`int`	2	2	2	4	4	4	8	8
`long`	-	4	4	4	4	8	8	8
`long long`	-	-	8	8	8	8	8	8

Table A.1: Widths of basic integer types in bytes for different data models and exemplary operating systems using them (Lauer, 1996; Mashey, 1996).

Bibliography

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

Adobe Systems. ActionScript virtual machine 2 (AVM2) overview. Technical report, Adobe System Inc., 2007.

Adobe Systems. SWF file format specification. Technical report, Adobe System Inc., 2013.

Charu Aggarwal. A framework for clustering massive-domain data streams. In *Proc. of the International Conference on Data Engineering (ICDE)*, pages 102–113, 2009.

Charu C. Aggarwal. *Outlier Analysis*. Springer-Verlag New York, 2013.

Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):80–82, 1975.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

Saed Alajlouni and Vittal Rao. Anomaly detection in liquid pipelines using modeling, co-simulation and dynamical estimation. In *Proc. of the International Conference on Critical Infrastructure Protection (ICCIP)*, pages 111–124, 2013.

Alexa Internet Inc. Alexa Top Global Sites. `https://www.alexa.com/topsites`, 1996–2018.

Frances Elizabeth Allen. Control flow analysis. In *Symposium on Compiler Optimization*, pages 1–19, 1970.

Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, February 2014.

Daniel Arp, Erwin Quiring, Christian Wressnegger, and Konrad Rieck. Privacy threats through ultrasonic side channels on mobile devices. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 35–47, April 2017.

Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

John Aycock. *Computer Viruses and Malware.* Springer, 2006.

Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. Doug Tygar. Can machine learning be secure? In *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 16–25, 2006.

Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology (JICV)*, 2(1):67–77, 2006.

Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2009.

Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Felegyházi. Duqu: Analysis, detection, and lessons learned. In *Proc. of the European Workshop on System Security (EUROSEC)*, 2012.

Helen A. Bergen and William J. Caelli. File security in WordPerfect 5.0. *Cryptologia*, 15(1): 57–66, 1991.

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 *Proc. of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 387–402, 2013.

Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. Poisoning behavioral malware clustering. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISEC)*, pages 1–10, November 2014.

Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.

Frank Boldewin. OfficeMalScanner. `http://www.reconstructer.org/code.html`, 2009.

Damiano Bolzoni, Sandro Etalle, and Pieter Hartel. POSEIDON: A 2-tier anomaly-based network intrusion detection system. In *Proc. of the IEEE International Workshop on Information Assurance (IWIA)*, pages 144–156, 2006.

Andrew P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

Andrei Broder and Michael Mitzenmacher. Network applications of Bloom Filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Xiaodong Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2007.

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.

Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proc. of the USENIX Security Symposium*, 2011.

Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 169–182, 2012.

Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, pages 197–206, April 2011.

Dan Caselden, Corbin Souffrant, and Genwei Jiang. Flash in 2015. `https://www.fireeye.com/blog/threat-research/2015/03/flash_in_2015.html`, 2015.

Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 143–163, 2008.

William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *Proc. of the Symposium on Document Analysis and Information Retrieval*, pages 161–175, April 1994.

Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. of the USENIX Security Symposium*, 2011.

Anton Cherepanov. Win32/industroyer – a new threat for industrial control systems. Technical report, ESET, June 2017.

Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. DFC: Accelerating string pattern matching for network applications. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 551–565, 2016.

Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *Proc. of the USENIX Security Symposium*, pages 99–116, 2017.

Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, (4):1523–1545, 2005.

Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 110–125, 2009.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.

Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

Graham Cormode and S. Muthukrishnan. Approximating data with the count-min sketch. *Journal of IEEE Software*, 29(1):64–69, 2012.

Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 269–278, 2006.

Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. of the International World Wide Web Conference (WWW)*, pages 281–290, 2010.

Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2006.

Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 81–95, 2008.

CrySyS Malware Intelligence Team. Miniduke: Indicators. Budapest University of Technology and Economics, February 2013.

Weidong Cui, Vern Paxson, Nicholas C. Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2006.

Mark Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with JavaScript. In *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

Andreas Dewald, Thorsten Holz, and Felix Freiling. ADSandbox: Sandboxing JavaScript to fight malicious websites. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 1859–1864, 2010.

Sarang Dharmapurikar and Vern Paxson. Robus TCP reassembly in the presence of adversaries. In *Proc. of the USENIX Security Symposium*, pages 65–80, 2005.

Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *Proc. of the International Conference on Software Engineering(ICSE)*, pages 760–770, 2012.

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, 2008.

Van Long Do, Lionel Fillatre, and Igor V. Nikiforov. A statistical method for detecting cyber/physical attacks on SCADA systems. In *Proc. of the IEEE Conference on Control Applications (CCA)*, pages 364–369, 2014.

Holger Dreger, Michael Mai, Anja Feldmann, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Proc. of the USENIX Security Symposium*, pages 257–272, 2006.

Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 1285–1298, 2017.

Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. John Wiley & Sons, second edition, 2000.

Patrick Düssel, Christian Gehl, Pavel Laskov, Jens-Uwe Bußer, Christof Störmann, and Jan Kästner. Cyber-critical infrastructure protection using real-time payload-based anomaly detection. In *Proc. of the Critical Information Infrastructures Security CRITIS*, pages 85–97, 2009.

Markus Engelberth, Carsten Willems, and Thorsten Holz. MalOffice: Detecting malicious documents with combined static and dynamic analysis. In *Proc. of the Virus Bulletin Conference*, 2009.

Nicolas Falliere, Liam O. Murcho, and Eric Chien. W32.stuxnet dossier. Technical report, Symantec Corp., 2011.

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, 2008.

Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

Cheng Feng, Tingting Li, and Deeph Chana. Multi-level anomaly detection in industrial control systems via package signatures and LSTM networks. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*, pages 261–272, 2017.

Peter Ferrie. Anti-unpacker tricks 1. *Virus Bulletin*, December 2008.

Prahlad Fogla and Wenke Lee. Evading network anomaly detection systems: Formal reasoning and practical techniques. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 59–68, 2006.

Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proc. of the USENIX Security Symposium*, pages 241–256, 2006.

Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious flash advertisements. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 363–372, 2009.

Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

Igor Nai Fovino, Andrea Carcano, Thibault De Lacheze Murel, Alberto Trombetta, and Marcelo Masera. Modbus/dnp3 state-based intrusion detection system. In *Proc. of the IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 729–736, 2010.

Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An Inquiry Into the Nature and Causes of the Wealth of Internet Miscreants. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 375–388, 2007.

William Friedman. The index of coincidence and its applications in cryptology. Technical report, Riverbank Laboratories, Department of Ciphers, 1922.

William Friedman and Lambros Callimahos. *Military Cryptanalytics*. Aegean Park Press, 1985.

Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Proc. of the International Conference on Security and Privacy in Communication Networks (SECURECOMM)*, pages 330–347, October 2015.

Carrie Gates and Carol Taylor. Challenging the anomaly detection paradigm: A provocative discussion. In *Proc. of the New Security Paradigms Workshop (NSPW)*, pages 21–29, 2006.

Google Inc. VirusTotal. `https://www.virustotal.com/`, 2004–2018.

Google Inc. Google Safe Browsing. `https://safebrowsing.google.com/`, 2008–2018.

Dina Hadžiosmanović, Lorenzo Simionato, Damiano Bolzoni, Emmanuele Zambon, and Sandro Etalle. N-gram against the machine: On the feasibility of the n-gram network analysis for binary protocols. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 354–373, 2012.

Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H. Hartel. Through the eye of the PLC: Semantic security monitoring for industrial processes. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 126–135, 2014.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction.* Springer Series in Statistics. Springer-Verlag New York, New York, N.Y., second edition, 2009. ISBN 978-0-387-84858-7.

Timo Hirvonen. Dynamic Flash instrumentation for fun and profit. In *Proc. of Black Hat USA*, 2014.

Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security (JCS)*, 6(3):151–180, 1998.

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

httparchive. `http://www.httparchive.org`, 2016.

Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C. Snoeren, and Kirill Levchenko. Botcoin: Monetizing stolen cycles. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

Wenyi Huang and Jack W. Stokes. MtNet: A multi-task neural network for dynamic malware classification. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 399–418, 2016.

Kenneth L. Ingham and Hajime Inoue. Comparing anomaly detection techniques for HTTP. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 42–62, 2007.

Luca Invernizzi, Stefano Benvenuti, Paolo Milani Comparetti, Marco Cova, Christopher Kruegel, and Giovanni Vigna. EvilSeed: A guided approach to finding malicious web pages. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 428–442, 2012.

ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 102–122, 2012.

Martin Johns and Sebastian Lekies. Biting the hand that serves you: A closer look at client-side flash proxies for cross-domain requests. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 85–103, 2011.

Ian T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proc. of the USENIX Security Symposium*, pages 625–642, 2017.

Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of the USENIX Security Symposium*, pages 637–651, August 2013.

Stamatis Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *Proc. of Annual Conference on IEEE Industrial Electronics Society (IECON)*, pages 4490–4494, 2011.

Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

Friedrich Wilhelm Kasiski. *Die Geheimschriften und die Dechiffrir-Kunst*. E. S. Mittler und Sohn, 1863.

Kaspersky Lab. The DUQU 2.0 – technical details. Technical report, Kaspersky Lab, June 2015.

Omer Katz, Noam Rinetzky, and Eran Yahav. Statistical reconstruction of class hierarchies in binaries. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 363–376, 2018.

Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-Force: Forced execution on JavaScript. In *Proc. of the International World Wide Web Conference (WWW)*, pages 897–906, 2017.

Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Proc. of the European Symposium on Algorithms (ESA)*, pages 456–467, 2006.

István Kiss, Béla Genge, and Piroska Haller. A clustering-based approach to detect cyber attacks in process control systems. In *Proc. of the IEEE International Conference on Industrial Informatics (INDIN)*, pages 142–148, 2015.

Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 443–457, 2012.

Jeremy Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research (JMLR)*, 7:2721–2744, 2006.

Radhesh K. Konoth, Emanuele Vineti, Veelasha Moonsamy, Martin Lindorfer, Christopher Kruegel, Herbet Bos, and Giovanni Vigna. An in-depth look into drive-by mining and its defense. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 447–462, 2010.

Georgia Koutsandria, Vishak Muthukumar, Masood Parvania, Sean Peisert, Chuck McParland, and Anna Scaglione. A hybrid network IDS for protective digital relays in the power transmission grid. In *Proc. of the IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 908–913, 2014.

Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* John Wiley & Sons, 2004.

Katharina Krombholz, Heidelinde Hobel, Markus Huber, and Edgar Weippl. Advanced social engineering attacks. *Journal of Information Security and Applications (JISA)*, 22(C), 2015.

Marina Krotofil, Jason Larsen, and Dieter Gollmann. The process matters: Ensuring data veracity in cyber-physical systems. In *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 133–144, 2015.

Christopher Kruegel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 201–208, 2002.

Tammo Krueger, Nicole Kraemer, and Konrad Rieck. ASAP: Automatic semantics-aware analysis of network payloads. In *Proc. of the ECML Workshop on Privacy and Security Issues in Machine Learning*, pages 50–63, September 2010.

Tammo Krueger, Hugo Gascon, Nicole Kraemer, and Konrad Rieck. Learning stateful models for network honeypots. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISEC)*, pages 37–48, October 2012.

Anthony LaForge. Flash and chrome. `https://blog.google/products/chrome/flash-and-chrome`, 2016.

Terran Lane and Carla E. Brodley. Temporal sequence learning and data reduction for anomaly detection. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 150–158, 1998.

Terran Lane and Carla E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information Systems Security*, 2(3):295–331, 1999.

Pavel Laskov and Nedim Šrndić. Static detection of malicious JavaScript-bearing PDF documents. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 373–382, 2011.

Thomas Lauer. *Porting to Win32^{TM}: A Guide to Making Your Applications Ready for the 32-Bit Future of Windows^{TM}*. Springer, 1996.

Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *Proc. of the USENIX Security Symposium*, 1998.

Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. A data mining framework for building intrusion detection models. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 120–132, 1999.

Corrado Leita, Ken Mermoud, and Marc Dacier. ScriptGen: An automated script generation tool for honeyd. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 203–214, 2005.

Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, second edition, 2014.

Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234:34–35, 1971.

Vladimir Iosifovich Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1(1), 1965.

Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1966.

Robert Lewand. *Cryptological mathematics*. The Mathematical Association of America, 2000.

Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M. B. Vitányi. The similarity metric. *IEEE Transactions on Information Theory*, (12):3250–3264, 2004.

Wei-Jen Li, Salvatore J. Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis. A study of malcode-bearing documents. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 231–250, 2007.

Zhou Li, Kehuan Zhang, Yinglian Xie, Famg You, and XiaoFeng Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 674–686, 2012.

Zhiqiang Lin, Xuxian Jiang, and Dongyan Xu. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2008.

Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, 2003.

Richard P. Lippmann, Robert K. Cunningham, David J. Fried, Isaac Graf, Kris R. Kendall, Seth E. Webster, and Marc A. Zissman. Results of the DARPA 1998 offline intrusion detection evaluation. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 1999.

Manuel López-Ibáñez and Ian Lance Taylor. The new Wconversion option. `https://gcc.gnu.org/wiki/NewWconversion`, 2006.

Mike Ter Louw, Karthik Thotta, and V. N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisments. In *Proc. of the USENIX Security Symposium*, pages 371–388, 2010.

Mark Luchs and Christian Doerr. Last line of defense: A novel ids approach against advanced threats in industrial control systems. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 141–160, 2017.

Matthew V. Mahoney and Philip K. Chan. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 220–237, 2004.

Malware Tracker Ltd. Cryptam. `http://www.cryptam.com`, 2012-2018.

Udi Manber and Gene Myers. Suffix Arrays: A new method for on-line string searches. In *Proc. of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 319–327, 1990.

Lorenzo Martignoni, Mihai Christodeorescu, and Somesh Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 431–441, 2007.

Conrado Martínez and Salvador Roura. Randomized binary search tree. *Journal of the ACM*, 45(2):288–323, 1998.

John R. Mashey. The long road to 64 bits. *ACM Queue Magazine*, 4(8):24–35, 1996.

Roy A. Maxion and Kymie M. C. Tan. Benchmarking anomaly-based detection systems. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*, pages 623–620, 2000.

John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Transactions on Information Systems Security*, 3(4):262–294, 2000.

Fei Miao, Quanyan Zhu, Miroslav Pajic, and George J. Pappas. Coding sensor outputs for injection attacks detection. In *Proc. of the IEEE Conference on Decision and Control (CDC)*, pages 5776–5781, 2014.

Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. In *Proc. of Black Hat USA*, 2015.

Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

Yilin Mo, Rohan Chabukswar, and Bruno Sinopoli. Detecting integrity attacks on SCADA systems. *IEEE Transactions on Control Systems Technology (TCST)*, 22(4):1396–1407, 2014.

Modbus.org. Modbus application protocol specification v1.1b3. Technical report, Modbus.org, April 2012.

Iulian Moraru and David G. Andersen. Exact pattern matching with feed-forward bloom filters. *Journal of Experimental Algorithmics (JEA)*, 17, 2012.

Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 421–430, 2007a.

Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 231–245, 2007b.

Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 197(1):3–16, 2008.

Jaehyun Nam, Muhammad Jamshed, Byungkwon Choi, Dongsu Han, and KyoungSoo Park. Haetae: Scaling the performance of network intrusion detection with many-core processors. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 89–110, 2015.

Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

Scott Hauck Nathaniel McVicar, Chih-Ching Lin. K-mer counting using Bloom Filters with an FPGA-attached HMC. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 203–210, 2017.

Gonzahlo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88, 2001.

Gonzalo Navarro and Mathieu Raffinot. Fast regular expression search. In *Proc. of the International Workshop on Algorithm Engineering*, pages 198–212, 1999.

Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

Terry Nelms, Roberto Perdisci, Manos Antonakakis, and Mustaque Ahamad. Towards measuring and mitigating social engineering software download attacks. In *Proc. of the USENIX Security Symposium*, pages 773–789, 2016.

Katarzyna Olejnik, Italo Dacosta, Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. SmarPer: Context-aware and automatic runtime-permissions for mobile devices. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 1058–1076, 2017.

Serkan Özkan. CVE Details. `http://www.cvedetails.com`, 2010–2018.

Ruoming Pang and Vern Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proc. of the Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, pages 339–351, 2003.

Ruoming Pang, Vern Paxson, Robin Sommer, and Larry L. Peterson. binpac: a yacc for writing application protocol parsers. In *Proc. of the Internet Measurement Conference (IMC)*, pages 289–300, 2006.

Masood Parvania, Georgia Koutsandria, Vishak Muthukumar, Sean Peisert, Chuck McParland, and Anna Scaglione. Hybrid control network intrusion detection systems for automated power distribution systems. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*, pages 774–779, 2014.

Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. Attack detection and identification in cyber-physical systems. *IEEE Transactions on Automatic Control*, 58(11):2715–2729, 2013.

Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2466, December 1999.

Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 1757–1771, 2017.

Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-executing binary programs for security applications. In *Proc. of the USENIX Security Symposium*, pages 829–844, 2014.

Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul I. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 17–31, 2006.

Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 5(6):864–881, 2009.

Alessandro Pignotti. Lightspark. `https://github.com/lightspark`, 2011–2018.

Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 54–73, 2006.

Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 287–296, December 2010.

Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection.* Addison-Wesley Longman, 2007.

William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

Edward Raff and Charles K. Nicholas. An alternative to NCD for large sequences, Lempel-Ziv Jaccard distance. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1007–1015, 2017.

Edward Raff and Charles K. Nicholas. Lempel-Ziv Jaccard distance, an effective alternative to ssdeep and sdhash. *Digital Investigation*, 24:34–49, 2018.

Rapid7 LLC. The metasploit project. `https://www.metasploit.com/`, 2003–2018.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proc. of the USENIX Security Symposium*, pages 169–186, 2009.

Konrad Rieck and Pavel Laskov. Detecting unknown network attacks using language models. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 74–90, July 2006.

Konrad Rieck and Christian Wressnegger. Harry: A tool for measuring string similarity. *Journal of Machine Learning Research (JMLR)*, 17(9):1–5, March 2016.

Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 31–39, December 2010.

Konrad Rieck, Christian Wressnegger, and Alexander Bikadorov. Sally: A tool for embedding strings in vector spaces. *Journal of Machine Learning Research (JMLR)*, 13(Nov): 3247–3251, November 2012.

Marco Rocchetto and Nils Ole Tippenhauer. Towards formal security analysis of industrial control systems. In *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, pages 114–126, 2017.

Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 289–300, 2006.

Shai Rubin, Somesh Jha, and Barton P. Miller. Automatic generation and analysis of NIDS attacks. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 28–38, 2004.

Gerard Salton, A. Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

Rob Savoye, Sandro Santilli, Bastiaan Jacques, Benjamin Wolsey, Zou Lunkai, Tomas Groth, Udo Giacomozzi, Hannes Mayr, John Gilmore, and Markus Gothe. GNU Gnash. `https://www.gnu.org/software/gnash`, 2005–2018.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

Bruce Schneier. *Applied Cryptography.* John Wiley and Sons, 1996.

B. Schölkopf, J. Platt, J. Shawe-Taylor, A.J. Smola, and R.C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels.* MIT Press, Cambridge, MA, 2002.

Thomas Schreck, Stefan Berger, and Jan Göbel. BISSAM: Automatic vulnerability identification of office documents. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 204–213, 2012.

Thomas Schreiber. Session riding – a widerspread vulnerability in today's web applications. Technical report, SecureNet GmbH, 2004.

Franka Schuster, Andreas Paul, René Rietz, and Hartmut König. Potentials of using one-class SVM for detecting protocol-specific anomalies in industrial networks. In *Proc. of the IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 83–90, 2015.

Robert Sedgewick and Kevin Wayne. *Algorithms.* Addison-Wesley, fourth edition, 2011.

Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

M. Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 88–107, 2008.

Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 94–109, 2009.

John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

Yun Shen, Enrico Mariconti, Pierre-Antoine Vervier, and Gianluca Stringhini. TIRESIAS: Predicting security events through deep learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 3–18, 2017.

Chris Sinclair, Lyn Pierce, and Sara Matzner. An application of machine learning to network intrusion detection. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 371–377, 1999.

Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 239–248, 2012.

Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Nils Provos. ShellOS: Enabling fast detection and forensic analysis of code injection attacks. In *Proc. of the USENIX Security Symposium*, August 2011.

Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 305–316, 2010.

Yingbo Song, Michael E. Locasto, Angelos Stavrou, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 541–551, 2007.

Yingbo Song, Angelos D. Keromytis, and Salvatore J. Stolfo. Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2009.

Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode: Re-thinking the role of learning in intrusion detection systems. *Machine Learning*, 81(2):179–205, November 2010.

Michael Stay. ZIP attacks with reduced known plaintext. In *Proc. of the International Workshop on Fast Software Encryption (FSE)*, pages 125–134, 2002.

Didier Stevens. Malicious PDF documents explained. *IEEE Security & Privacy*, pages 80–82, 2011.

Didier Stevens. XORSearch. `http://blog.didierstevens.com/programs/xorsearch/`, 2014.

Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 1419–1430, 2015.

Salvatore J. Stolfo, Ke Wang, and Wei-Jen Li. Towards stealthy malware detection. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 231–249. Springer US, 2007.

Ching Y. Suen. N-gram statistics for natural language understanding and text processing. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1(2):164–172, April 1979.

Symantec Security Response. W32.duqu – the precursor to the next stuxnet. Technical report, Symantec Corp., June 2011.

Peter Ször. *The art of computer virus research and defense.* Symantec Press, 2005.

Kymie M. C. Tan and Roy A. Maxion. "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 188–201, 2002.

André Teixeira, Saurabh Amin, Henrik Sandberg, Karl Henrik Johansson, and Shankar S. Sastry. Cyber security analysis of state estimators in electric power systems. In *Proc. of the IEEE Conference on Decision and Control (CDC)*, pages 5991–5998, 2010.

The Bro Project. The Bro Network Security Monitor. https://www.bro.org, 1994–2018.

Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6): 419–422, 1968.

Da Tong and Viktor K. Prasanna. High throughput sketch based online heavy hitter detection on FPGA. *ACM SIGARCH Computer Architecture News*, (43):70–75, 2015.

Da Tong and Viktor K. Prasanna. Sketch acceleration on FPGA and its applications in network anomaly detection. *IEEE Transactions on Parallel and Distributed Systems*, (29): 929–942, 2018.

Trend Micro Threat Research Team. The Taidoor campaign: An in-depth analysis. Technical report, 2012.

Trustwave Holdings. Trustwave global security report. Technical report, Trustwave Holdings Inc., 2016.

Robert Udd, Mikael Asplund, Simin Nadjm-Tehrani, Mehrdad Kazemtabrizi, and Mathias Ekstedt. Exploiting bro for intrusion detection in a SCADA system. In *Proc. of the ACM International Workshop on Cyber-Physical System Security*, pages 44–51, 2016.

David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 1092–1105, 2016.

Steven van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

Timon van Overveldt, Christopher Kruegel, and Giovanni Vigna. FlashDetect: ActionScript 3 malware detection. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 274–293, June 2012.

Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 116–134, 2008.

Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 21–30, 2004.

Nedim Šrndić and Pavel Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2013.

Ognjen Vukovic and György Dán. On the security of distributed power system state estimation under targeted attacks. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 666–672, 2013.

Ke Wang and Salvatore J. Stolfo. One-class training for masquerade detection. In *Proc. of the ICDM Workshop on Data Mining for Computer Security*, pages 10–19, 2003.

Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, 2004.

Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 227–246, 2005.

Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 226–248, 2006a.

Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2006b.

Yong Wang, Zhaoyan Xu, Jialong Zhang, Lei Xu, Haopei Wang, and Guofei Gu. SRID: State relation based intrusion detection for false data injection attacks in SCADA. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 401–418, 2014.

Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

Peter Weiner. Linear pattern matching algorithms. In *Proc. of Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Jeffrey Wilhelm and Tzi-cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 219–235, 2007.

Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2008.

Jeong Wook Oh. AVM inception - how we can use AVM instrumentation in a beneficial way. In *Shmoocon*, 2012.

Christian Wressnegger and Konrad Rieck. Looking back on three years of flash-based malware. In *Proc. of the ACM European Workshop on Systems Security (EuroSec)*, April 2017.

Christian Wressnegger, Frank Boldewin, and Konrad Rieck. Deobfuscating embedded malware using probable-plaintext attacks. In *Proc. of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 164–183, October 2013a.

Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A close look on n-grams in intrusion detection: Anomaly detection vs. classification. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISEC)*, pages 67–76, November 2013b.

Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Comprehensive analysis and detection of flash-based malware. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 101–121, July 2016a.

Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 541–552, October 2016b.

Christian Wressnegger, Kevin Freeman, Fabian Yamaguchi, and Konrad Rieck. Automatically inferring malware signatures for anti-virus assisted attacks. In *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 587–598, April 2017a.

Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. 64-bit migration vulnerabilities. *Information Technology (IT)*, 59(2):73–82, April 2017b.

Christian Wressnegger, Ansgar Kellner, and Konrad Rieck. ZOE: Content-based anomaly detection for industrial control systems. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*, pages 127–138, June 2018.

Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, Charles Lee Ray, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 499–510, November 2013.

Mikio Yamamoto and Kenneth W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1): 1–30, 2001.

Dayu Yang, Alexander Usynin, and J Wesley Hines. Anomaly-based intrusion detection for scada systems. In *Proc. of the intl. topical meeting on nuclear plant instrumentation, control and human machine interface technologies*, pages 12–16, 2006.

Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 129–140, 1996.

Adam Young and Moti Yung. Cryptovirology: The birth, neglect, and explosion of ransomware. *Communications of the ACM*, 60(7):24–26, 2017.

Vyacheslav Zakorzhevsky. New Flash Player 0-day (CVE-2014-0515) Used in Watering-hole Attacks. `https://securelist.com/blog/incidents/59399/new-flash-player-0-day-cve-2014-0515-used-in-watering-hole-attacks/`, 2014.

Index

64-bit migration vulnerability, 11, **13**

ActionScript, **42**
Acyclic control-flow graph (ACFG), **47**, 48
Adobe Flash, 41
Anagram, 4, 26, **76**, 77
Anomaly detection, **4**, 68
Anti-virus software, 8, 19
Area under the curve (AUC), **5**, 71, 74

Bloom filter, 7, **32**, 41, 50, 52
Bounded AUC, **5**, 71, 74
Byte *n*-gram, **26**

Classification, **3**, 50
Clustering, 2, 63, **66**, 67
Count-Min sketch, 7, **33**, 52, 68
Content-based anomaly detection, **65**, 68
Control-flow graph (CFG), 46, **47**

Data model, 12, 13, **109**
De-noising, 65, **68**, 74
Density, **27**, 37, 68, 78
Difference stream, 86, **92**
Document frequency, 7, 68, **87**

Embedded malware, **83**, 84
Embedding, *see* Vector space embedding
Evasion, **78**, 106

False alarm, *see* False positive
False positive, **5**, 32, 74
False-positive rate (FPR), **5**, 54, 75
Feature space, 4, **25**, 25, 66
Flash, *see* Adobe Flash
Flash-based malware, **41**, 43
FLASHDETECT, **54**, 56

GORDON, 43
GORDON-SVM, **51**, 55
GORDON-TCA, **52**, 56
Guided code-execution, 41, **46**

Hash table, 31

Indicative code, **46**, 48
Industrial control system (ICS), **63**, 65
Integer overflow, 14
Integer signedness, 13
Integer truncation, 13

KANDI, **86**, 94, 97
Kasiski examination, 86, **90**

Language model, **25**, 50, 63, 66, 68, 103
Longest common prefix (LCP), 30, **87**

Machine Learning, 2
Migration vulnerability, 13
Mimicry attack, 106
Multi-path exploration, 41, **48**, 60

n-gram, 25, **26**, 51, 52, 66
n-gram model, 25
n-gram statistics, 50, **52**, 56

Overflow, *see* Integer overflow

Perturbation, **27**, 36, 51
Polymorphic blending, 71, **78**, 79
Prefix tree, 7, **29**

Probabilistic data structure, 31, **38**, 41, 68, 83
Probable-plaintext attack, 86, 91, **92**
Prototype model, 66, **67**, 68, 75

Radix tree, 29
Receiver operator characteristics (ROC), 5
Recursion, 132
ROC curve, **5**, 75, 77

Saturation, 39
SCADA, 2, **63**, 79
Semantic gap, 3, **5**
Signature-based detection, 19
Software vulnerability, 11
Substring statistics, **6**, 30, 83, 87, 90
Suffix array, 7, **30**, 87
Suffix tree, 7, **30**
Suitability, 27
Support Vector Machine (SVM), 41, 44, 51

Term frequency, 7, **87**
Token n-gram, **26**, 50
Trie, 7, **29**
True positives, **5**, 74
True-positive rate (TPR), **5**, 54, 74
Truncation, *see* Integer truncation

Variability, **28**, 37, 78
Vector space, 7, 8, **26**, 51
Vector space embedding, 8, **25**, 50
Vigenère cipher, 83, 84, **85**
Virus scanner, 8, 19
Vulnerability, *see* Software vulnerability

Zoe, **65**, 74, 77