Machine Learning and Security of Non-Executable Files

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät

der Eberhard Karls Universität Tübingen

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

Nedim Šrndić

aus Maglaj/Bosnien-Herzegowina

Tübingen

2017

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	4.10.2017
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Andreas Zell
2. Berichterstatter:	Prof. Dr. Michael Menth

To my mother and my father and all people of good will coming after me.

Mojoj majci i mom tati i svim ljudima dobre volje koji dolaze poslije mene.

Abstract

Computer malware is a well-known threat in security which, despite the enormous time and effort invested in fighting it, is today more prevalent than ever. Recent years have brought a surge in one particular type: malware embedded in non-executable file formats, e.g., PDF, SWF and various office file formats. The result has been a massive number of infections, owed primarily to the trust that ordinary computer users have in these file formats. In addition, their feature-richness and implementation complexity have created enormous attack surfaces in widely deployed client software, resulting in regular discoveries of new vulnerabilities.

The traditional approach to malware detection – signature matching, heuristics and behavioral profiling – has from its inception been a labor-intensive manual task, always lagging one step behind the attacker. With the exponential growth of computers and networks, malware has become more diverse, wide-spread and adaptive than ever, scaling much faster than the available talent pool of human malware analysts. An automated and scalable approach is needed to fill the gap between automated malware adaptation and manual malware detection, and machine learning is emerging as a viable solution. Its branch called adversarial machine learning studies the security of machine learning algorithms and the special conditions that arise when machine learning is applied for security.

This thesis is a study of adversarial machine learning in the context of static detection of malware in non-executable file formats. It evaluates the *effectiveness*, *efficiency* and *security* of machine learning applications in this context. To this end, it introduces 3 data-driven detection methods developed using very large, high quality datasets. PJScan detects malicious PDF files based on lexical properties of embedded JavaScript code and is the fastest method published to date. SL2013 extends its coverage to all PDF files, regardless of JavaScript presence, by analyzing the hierarchical structure of PDF logical building blocks and demonstrates excellent performance in a novel long-term realistic experiment. Finally, Hidost generalizes the hierarchical-structure-based feature set to become the first machine-learning-based malware detector operating on multiple file formats. In a comprehensive experimental evaluation on PDF and SWF, it outperforms other academic methods and commercial antivirus systems in detection effectiveness.

Furthermore, the thesis presents a framework for security evaluation of machine learning classifiers in a case study performed on an independent PDF malware detector. The results show that the ability to manipulate a part of the classifier's feature set allows a malicious adversary to disguise malware so that it appears benign to the classifier with a high success rate. The presented methods are released as open-source software.

Kurzfassung

Schadsoftware ist eine gut bekannte Sicherheitsbedrohung. Trotz der enormen Zeit und des Aufwands die investiert werden, um sie zu beseitigen, ist sie heute weiter verbreitet als je zuvor. In den letzten Jahren kam es zu einem starken Anstieg von Schadsoftware, welche in nicht-ausführbaren Dateiformaten, wie PDF, SWF und diversen Office-Formaten, eingebettet ist. Die Folge war eine massive Anzahl von Infektionen, ermöglicht durch das Vertrauen, das normale Rechnerbenutzer in diese Dateiformate haben. Außerdem hat die Komplexität und Vielseitigkeit dieser Dateiformate große Angriffsflächen in weitverbreiteter Klient-Software verursacht, und neue Sicherheitslücken werden regelmäßig entdeckt.

Der traditionelle Ansatz zur Erkennung von Schadsoftware – Mustererkennung, Heuristiken und Verhaltensanalyse – war vom Anfang an eine äußerst mühevolle Handarbeit, immer einen Schritt hinter den Angreifern zurück. Mit dem exponentiellen Wachstum von Rechenleistung und Netzwerkgeschwindigkeit ist Schadsoftware diverser, zahlreicher und schneller-anpassend geworden als je zuvor, doch die Verfügbarkeit von menschlichen Schadsoftware-Analysten kann nicht so schnell skalieren. Ein automatischer und skalierbarer Ansatz ist gefragt, und maschinelles Lernen tritt als eine brauchbare Lösung hervor. Ein Bereich davon, Adversarial Machine Learning, untersucht die Sicherheit von maschinellen Lernverfahren und die besonderen Verhältnisse, die bei der Anwendung von machinellem Lernen für Sicherheit entstehen.

Diese Arbeit ist eine Studie von Adversarial Machine Learning im Kontext statischer Schadsoftware-Erkennung in nicht-ausführbaren Dateiformaten. Sie evaluiert die *Wirksamkeit*, *Leistungsfähigkeit* und *Sicherheit* von maschinellem Lernen in diesem Kontext. Zu diesem Zweck stellt sie 3 datengesteuerte Erkennungsmethoden vor, die alle auf sehr großen und diversen Datensätzen entwickelt wurden. PJScan erkennt bösartige PDF-Dateien anhand lexikalischer Eigenschaften von eingebettetem JavaScript-Code und ist die schnellste bisher veröffentliche Methode. SL2013 erweitert die Erkennung auf alle PDF-Dateien, unabhängig davon, ob sie JavaScript enthalten, indem es die hierarchische Struktur von logischen PDF-Bausteinen analysiert. Es zeigt hervorragende Leistung in einem neuen, langfristigen und realistischen Experiment. Schließlich generalisiert Hidost den auf hierarchischen Strukturen basierten Merkmalsraum und wurde zum ersten auf maschinellem Lernen basierten Schadsoftware-Erkennungssystem, das auf mehreren Dateiformaten anwendbar ist. In einer umfassenden experimentellen Evaulierung auf PDF- und SWF-Formaten schlägt es andere akademische Methoden und kommerzielle Antiviren-Lösungen bezüglich Erkennungswirksamkeit.

Überdies stellt diese Doktorarbeit ein Framework für Sicherheits-Evaluierung von auf

machinellem Lernen basierten Klassifikatoren vor und wendet es in einer Fallstudie auf eine unabhängige akademische Schadsoftware-Erkennungsmethode an. Die Ergebnisse zeigen, dass die Fähigkeit, nur einen Teil von Features, die ein Klasifikator verwendet, zu manipulieren, einem Angreifer ermöglicht, Schadsoftware in Dateien so einzubetten, dass sie von der Erkennungsmethode mit hoher Erfolgsrate als gutartig fehlklassifiziert wird. Die vorgestellten Methoden wurden als Open-Source-Software veröffentlicht.

Acknowledgements

This thesis would not have been possible without the support of Prof. Andreas Zell and Dr. Pavel Laskov. I kindly thank Prof. Zell for his backing during both good and hard times and for providing valuable advice and guidance. I owe a particular debt to Pavel for his continuous mentorship and friendship during this entire journey and for teaching me the value of perseverance in scientific investigations. Furthermore, I kindly thank my colleagues and friends Dr. Blaine Nelson, Dr. Florian Mittag, Dr. Battista Biggio, Goran Huskić and numerous members of the Cognitive Systems Department of the University of Tübingen for their scientific feedback and collaboration. I would also like to thank Prof. Saša Mrdović for helping me start my doctoral studies.

I acknowledge the support of the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) and the German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik, BSI), especially Dr. Robert Krawczyk, in funding a substantial part of this work. I kindly thank VirusTotal for providing access to file data. The opinions in this thesis are those of its author and do not necessarily reflect the opinions of funding sponsors.

Finally, this thesis would not have been completed without the love and support of my family, including my wife Lola, daughter Luna, parents Mirsada and Idriz, and brother Mahir.

Contents

Chapter 1

Introduction

The advent of the computer era has had a transformative effect on our world. The ubiquitous availability of computers has changed the way we communicate, learn, work, socialize, trade, etc. There is an unsurpassed amount of information available around the hour at our fingertips, enabling an unprecedented level of prosperity for our society globally.

The benefits of the computer era, however, do not come without drawbacks. As in the physical world, where every economic niche attracts its share of criminals, there is a thriving ecosystem of unlawful activity in the virtual realm as well. Whether interception of personal communications, use of networks of slave computers for denial-of-service attacks against businesses or ransom demands for the release of private data held hostage – the consequences of such activity reach beyond the virtual realm and affect our daily lives.

An especially wide-spread and effective method for computer-related criminal activities is *malware*, i.e., computer software with a malicious function. Historically, the majority of malware was distributed as executable files, e.g., MS-DOS programs or Portable Executable (PE) files. However, in recent years there has been an increasing focus on malware disguised inside non-executable files of different file formats, e.g., Portable Document Format (PDF), SWF file format or office document formats such as OOXML and ODF. Particularly for targeted attacks, non-executable file formats provide attackers with multiple benefits: *a*) ordinary computer users have learned to distrust executable files but not office documents, *b*) the complexity of popular non-executable file formats whose specifications have thousands of pages has fueled a surge in documented vulnerabilities in recent years, *c*) designed for flexibility, these formats facilitate the hiding of malware using obfuscation, dynamic loading or encryption.

As the complexity and frequency of malware attacks grew, it warranted the development of novel defensive technologies that can automate and scale the work of human professionals in the field. A promising avenue of research was found in the area of machine learning, with many successful applications. Still, as typical in science, many more research questions were opened than answered. Specifically, applications of machine learning for security impose a set of constraints that make them more challenging than in other areas. The cost of false predictions in, e.g., misuse detection, is a lot higher

(financial fraud) than in, e.g., product recommendation (loss of a sales opportunity). Furthermore, while in many other areas, e.g., in speech recognition, the data has a relatively stationary distribution or is evolving slowly, in security the data is often changing rapidly, e.g., in intrusion detection, where there are new threats every day. Finally, a unique characteristic of machine learning in the context of security is that the evolution of the data is a direct response to the machine learning systems themselves. The data is continually being adapted to cheat the evolving systems, leading to an arms race. *Adversarial machine learning* is the scientific discipline studying the interplay between machine learning and security.

This thesis is a study of adversarial machine learning in the context of static detection of malware in non-executable file formats. It evaluates the *effectiveness*, *efficiency* and *security* of machine learning applications in this context. It introduces novel methods for malware detection in non-executable file formats and presents a comprehensive experimental analysis of their effectiveness and efficiency and, where available, an independent evaluation of their security. Furthermore, it presents a framework for security evaluation of machine learning classifiers with a case study of an established independently published system for PDF malware detection.

Static detection is performed by examining the file itself, as opposed to the *dynamic* approach that includes the execution of the file in a monitored environment. Static methods operate orders of magnitude faster and can be efficiently applied to massive datasets. Their inherent drawback is the inability to access malicious content in cases where it is dynamically loaded from otherwise seemingly benign files. Well-implemented dynamic methods solve this problem but are too slow for real-time detection and relatively resource-intensive. They have to cope with stealthy malware and support a combinatorial explosion of different versions of targeted software and libraries. A third type of methods, *hybrid*, combines both, in an attempt to balance their advantages and drawbacks. The focus of this thesis is on high efficiency, i.e., methods that can be applied at network speeds, which limits its scope to static approaches.

In 2011, when work on this thesis started, the state of the art in machine-learning-based detection of malware in non-executable file formats was heavily focused on detecting malware targeting web browsers using JavaScript [14, 22, 23, 29, 59, 72, 76, 80, 107]. In 2008 attackers started increasingly using PDF as their attack vector of choice and it remained the dominant threat for several following years. Work on browser malware detection, especially [80], has motivated the question of viability of detecting malware in PDF files which make use of JavaScript, which forms the basis of the first method published as part of this thesis. Most early PDF malware detectors were at least partly based on dynamic execution [1, 22, 28, 71, 103, 104], but their long runtime is ill-suited for network-level detection. The only 2 static methods [48, 84] are based on analysis of n-grams extracted from binary PDF files. These methods are, however, highly prone to evasion due to the wide-spread obfuscation techniques and file-level encryption built into the PDF file format. Along with PDF, late 2000s saw a rapid rise in SWF malware. The only SWF malware detector proposed before 2011, OdoSwiff, is a hybrid method

also using relatively vulnerable features.

The existing solutions at the time were limited to a single file format and could not scale to data quantities traveling through modern networks. Furthermore, the established practice in experimental evaluation was the use of relatively small datasets (some studies used less than 100 samples) and cross-validation instead of a clear temporal separation of training (old) and test datasets (unseen malware). Finally, published methods were compared to previous academic work but rarely to commercial products. The contributions of this thesis address these shortcomings and can be summarized as follows:

- Implementation and evaluation of a static machine-learning-based detector of malicious PDF files, PJScan, based on lexical properties of JavaScript code used by malware embedded inside PDF files. The experimental evaluation was performed on an unprecedentedly large real-world PDF dataset. It demonstrated solid effectiveness and, with 31 ms/file/CPU, unsurpassed efficiency of the system. Furthermore, it was able to detect 71 % zero-day malicious PDF files missed by all surveyed antivirus scanners.

- Definition of a novel set of features for characterization of file formats internally logically organized as hierarchies, e.g., PDF, SWF, HTML, office file formats, etc.

- Implementation and evaluation of a static machine-learning-based detector of malicious PDF files SL2013 based on the defined structural features. A 10-week simulated real-world deployment of the proposed detector on 440,000 malicious and benign PDF files with weekly retraining achieved detection performance close to that of the best antivirus tools.

- Implementation and evaluation of a static machine-learning-based detector of malicious PDF and SWF files, Hidost, based on the defined structural features. This is the first such system applicable to multiple file formats and extensible to other ones. An experimental evaluation performed on both formats under the same simulated real-world deployment experiment as SL2013 demonstrated that Hidost outperforms all surveyed antivirus engines on PDF and ranks among the best on SWF files.

- The first automated practical attack against a machine-learning-based detector of malicious PDF files (PDFrate) deployed "in the wild", performed without knowledge of the learned model and entirely in problem space. Systematic evaluation of different evasion scenarios with varying degrees of knowledge available to the attacker, implemented in a software framework Mimicus, reveals that simple attacks can lower the average classification score from almost 100 % down to 28 % to 33 %.

- All developed methods and frameworks, i.e., PJScan, SL2013, Hidost and Mimicus, together with code for experiments evaluating Hidost are released as free

and open-source software to guarantee the reproducibility of the published results. Furthermore, all datasets used for the evaluation of Hidost are released in form of extracted feature vectors.

1.1 Structure of the Dissertation

The remainder of this thesis is structured as follows. Chapter 2 describes the technical details of two file formats necessary for the understanding of subsequent chapters, i.e., the Portable Document Format (PDF) and the SWF file format.

Chapter 3 presents a case study of machine learning for malware detection on the example of PDF, introducing PJScan. The subsequent Chapter 4 provides an elaborate presentation of two static machine-learning-based malware detectors for non-executable file formats based on hierarchical logical structure. The first, SL2013, is limited to PDF, while its successor Hidost introduces a set of innovations that extend the basic idea and the same feature set to a second format, SWF. Furthermore, a basic outline is provided how to further extend it to generalize to other non-executable file formats, e.g., HTML, OOXML and ODF.

While the last two chapters concern themselves with the question of applicability of machine learning to detection of non-executable malware, Chapter 5 explores the question of security of machine learning itself in this domain in a case study of evasion of a deployed classifier.

Chapters 3, 4 and 5 present methods published in our earlier scientific publications, cited at the beginning of each chapter. They provide a separate account of respective relevant prior work. Furthermore, each has a section titled "Discussion" concerned with the most important scientific findings, limitations of the proposed methods and relevant later work citing the publications underlying the given chapter.

Finally, Chapter 6 concludes the manuscript. Technical details relevant to Mimicus implementation are described in Appendix A.

1.2 Data Sources

Novel methods developed in this thesis were compared to previous work, both academic and commercial. Dataset collection and comparison to commercial antivirus tools proceeded through VirusTotal[1], a website enabling users to upload files to have them scanned by virtually all commercial antivirus tools. Using data from VirusTotal ensured that all evaluations were performed on diverse real-world malicious and benign files and that the data quantity was plentiful. Scan reports obtained from VirusTotal reflect static detection capabilities of deployed antivirus tools; their dynamic components are not utilized. As such, they can be compared with the fully static methods presented in

[1] VirusTotal – `https://virustotal.com/`.

this thesis. Google Search was used as an independent source of benign PDF data. Detailed descriptions of concrete datasets can be found in sections concerning experimental evaluation in every chapter.

Chapter 2

File Formats

Knowledge of these file formats is indispensable for understanding methods elaborated in the remainder of this manuscript, hence we present in the following sections their minimal self-contained description to serve as a common base for the coming chapters.

2.1 Portable Document Format

The *Portable Document Format (PDF)* is a file format that enables creation of documents that render and print consistently, independent of the underlying environment. It was created by Adobe Systems Inc. in 1993 and published in 2008 as open standard ISO 32000-1:2008 [67], a document referred to as the *PDF Reference* hereinafter. This and later sections concerning the PDF format contain direct citations from the PDF Reference.

The syntax of PDF comprises four main elements and is illustrated in a simplified example PDF file in Fig. 2.1:

- **Objects.** These are the basic building blocks in PDF described in more detail in the remainder of this section.
- **File structure.** It specifies how objects are laid out, accessed and modified in a PDF file. At this level, a PDF file consists of:
  - a *header* with the PDF magic number `%PDF-` followed by the version of the PDF standard utilized in the file, e.g., `1.7`,
  - a *body* with PDF objects that make up the actual content of the document,
  - a *cross-reference table* introduced with the keyword `xref`, identifying indirectly referenced objects in the body and their file offsets,
  - a *trailer*, denoted by the keyword `trailer`, containing the location of the cross-reference table and special objects in the file body and followed by the keyword `%%EOF`.

  Although linear PDF files exist, objects are layed out non-linearly in most PDF files, including our example. In order to parse such a file, a PDF parser must

therefore begin by checking the version number and looking at the file trailer for information about the location of the cross-reference table and some special objects in the file body. Starting from there, it can locate the information necessary for rendering.

The file structure lies at the heart of methods elaborated in Chapter 5.

- **Document structure.** It determines how objects are logically organized to represent the components of a PDF file, e.g., fonts, pages, etc. A detailed description of the elaborate document structure is deferred to Chapter 4.
- **Content streams.** They describe the appearance of the content in PDFs and are not further considered in this manuscript.

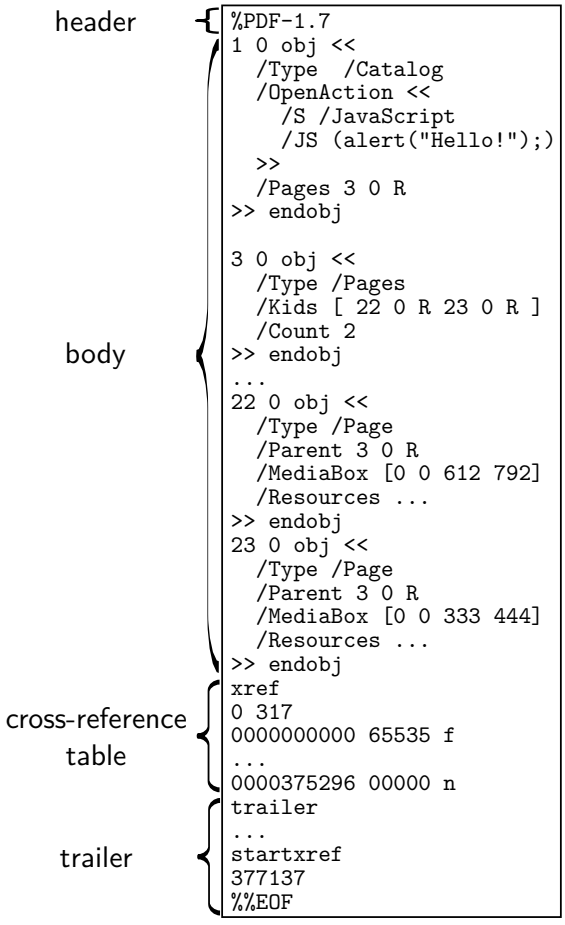


Figure 2.1: An example PDF file, simplified and formatted for brevity.

PDF objects reside in the body of a PDF file. There are 9 basic object types specified in the PDF Reference:

1. *Boolean* objects can take on values `true` and `false`.

2. A *numeric* object may be an integer or a real number.

3. *Strings* may be stored in two ways:
   - as a sequence of literal characters between parentheses '`(`' and '`)`', e.g., the string `alert("Hello!");` in Fig. 2.1,
   - as a sequence of hexadecimal numbers between angle brackets '<' and '>'.

   Strings may be encrypted.

4. A *name* is a sequence of 8-bit characters used as identifier. Names are introduced using the forward slash character ('/') and can contain arbitrary characters except *null* (`0x00`). Examples are `Type`, `Catalog` or `MediaBox` in Fig. 2.1.

5. The *null* object is denoted by the keyword `null`.

   The object types introduced so far, i.e., boolean, numeric, string, name and null, will be referred to as *primitive* types hereinafter.

6. An *array* is a one-dimensional sequence of PDF objects enclosed between square brackets, '`[`' and '`]`', e.g., `[0 0 333 444]`. It may contain heterogeneous PDF objects and be nested.

7. *Dictionaries* are unordered sets of key-value pairs enclosed between symbols '<<' and '>>'. The keys must be *name objects* unique within a dictionary. The values may be of any PDF object type, including nested dictionaries. There are 5 dictionaries in Fig. 2.1, one of them being nested inside the first dictionary.

8. A *stream* is a dictionary followed by a sequence of bytes enclosed between keywords `stream` and `endstream`. Streams are usually used to represent large objects, such as images, in a compact way. The content of the byte sequence may be encoded or compressed and the associated dictionary contains information on whether and how to decode or decompress it. Stream content may also be encrypted. A special type of streams are *object streams* containing arbitrary PDF objects.

9. An *indirect object* is any of the previously defined objects supplied with a unique object identifier and enclosed within keywords `obj` and `endobj`. Fig. 2.1 contains 4 indirect dictionaries. The unique identifier comprises an object number and a generation number, a sort of a version number for tracking changes to objects. Indirect objects can be referenced from other objects via *indirect references* written as a sequence of the object number, the generation number and the capital letter 'R'. For example, `23 0 R` in Fig. 2.1 refers to the dictionary with the object number 23 and generation number 0.

The PDF body is structured as a hierarchy of objects interconnected in a semantically meaningful way to describe pages, outlines, annotations, etc. A central role in the hierarchy belongs to the *Catalog dictionary* pointed to by the `/Root` entry in the cross-reference table.

Having described the PDF file format in sufficient detail, we can now interpret the body of the PDF file illustrated in Fig. 2.1 in its entirety[1]. It contains four indirect objects indicated by two-part object identifiers, e.g., `1 0` for the first object, and the keywords `obj` and `endobj`. They are all dictionaries, as they are surrounded by the symbols '<<' and '>>'. The first indirect object is the *Catalog* dictionary, distinguishable by having `Catalog` for its `Type`. The *Catalog* has 2 further dictionary entries:

- `OpenAction` is an example of a nested dictionary. It has two entries: `S`, a name indicating that this is a JavaScript action dictionary, and `JS`, a string with the actual JavaScript code, `alert('Hello!');`.
- `Pages` is an indirect reference to the object with the ID `3 0`: the *Pages* dictionary that immediately follows *Catalog*. It has an integer, `Count`, announcing that the document has 2 pages, and an array `Kids` identifiable by the square brackets, with two references to 2 *Page* objects describing the appearance of the pages. The remaining objects are interpreted analogously.

Many parsers fail to strictly follow the PDF Reference. Even Adobe's own Adobe Reader is notorious for such lack of compliance. For example, it ignores arbitrary symbols before the header [37] and can dispense with the trailer and cross-references [110].

2.2 SWF File Format

The *SWF File Format* (*SWF*, pronounced *swiff*) is a proprietary binary file format widely used for interactive content on the World Wide Web. Adobe Systems Incorporated has published the partial *SWF Specification* [95]. This and later sections concerning the SWF format contain direct citations from the SWF Specification.

SWF files consist of a header and a sequence of *tags* – well-defined data structures with a set of predefined *fields* and corresponding *values*. There are 65 different types of tags specified, each defining its own set of fields with different names and data types. Some basic SWF data types are:

- 8-, 16-, 32- and 64-bit *integers*, both signed and unsigned, arrays of these types and integers with a variable number of bytes
- *fixed-* and *floating-point numbers* with different widths and precisions
- *integer* and fixed-point numbers with widths that are not exponents of 2

[1] Excepting the omitted parts denoted by ellipses (`...`).

- *strings*
- various *data structures* such as 24- and 32-bit color records, rectangle records, 2D transformation matrices, etc.

```
000  46 57 53 06 24 00 00 00  70 00 0b 9a 00 00 3e 80
010  00 01 02 00 43 02 aa bb  cc 40 00 43 02 11 22 33
020  40 00 00 00
```

Figure 2.2: Hexadecimal view of an example SWF file. Left column shows hexadecimal addresses of rows.

Fig. 2.2 shows a toy SWF file used for illustrative purposes. The physical layout of SWF is too obscure for direct interpretation. Instead, we base our description of the SWF file format on the decoded, human-readable depiction of the same file, illustrated in Fig. 2.3[2].

```
[14:0]: SetBackgroundColor
  [14:0]: Header (Code: 9 Length: 3)
    [14:0]: TagAndLength : 579
  [B0:0]: BackgroundColor
    [16:0]: Red : 170
    [17:0]: Green : 187
    [18:0]: Blue : 204
[19:0]: ShowFrame
  [19:0]: Header (Code: 1 Length: 0)
    [19:0]: TagAndLength : 64
[1B:0]: SetBackgroundColor
  [1B:0]: Header (Code: 9 Length: 3)
    [1B:0]: TagAndLength : 579
  [E8:0]: BackgroundColor
    [1D:0]: Red : 17
    [1E:0]: Green : 34
    [1F:0]: Blue : 51
[20:0]: ShowFrame
  [20:0]: Header (Code: 1 Length: 0)
    [20:0]: TagAndLength : 64
[22:0]: End
  [22:0]: Header (Code: 0 Length: 0)
    [22:0]: TagAndLength : 0
```

Figure 2.3: SWF file depicted in Fig. 2.2, decoded, header omitted for brevity. Every line starts with a hexadecimal number within square brackets denoting the offset, in bytes, of the corresponding tag field from the beginning of the file.

[2]This textual description of the original SWF file was produced using the `ConsoleDumper` class of SWFRETools, an open-source Java toolkit for reverse-engineering SWF files available at `https://github.com/sporst/SWFREtools`.

The illustration shows 5 SWF tags separated by dotted lines: two tags with type `SetBackgroundColor` at bytes `0x14` and `0x1B`, two `ShowFrame` tags at bytes `0x19` and `0x20` and an `End` tag at byte `0x22`. Tags of a SWF file are laid out sequentially. Every tag has a header with an unsigned 16-bit little-endian `TagCodeAndLength` field that comprises a 10-byte tag type identifier and a 6-bit tag length field, with an optional wider length field for tags longer than 62 bytes.

The first tag in this file is used to set the background color of the display. It is a simple tag, defining 3 unsigned one-byte values of the red (`0xAA` = 170), green (`0xBB` = 187) and blue (`0xCC` = 204) color components. The second tag makes the content of the canvas render on screen for the duration of one frame. Following this, the background color is set to #112233 and the screen is refreshed one more time. Finally, the `End` tag signals the end of the file.

Chapter 3

A Case Study of Malicious PDF File Detection

In this chapter we present a method for malicious PDF file detection that specializes in PDF files that contain JavaScript code. At the heart of the detection approach lies the hypothesis that lexical properties of embedded JavaScript code differ between benign and malicious files. Due to its specialization, the proposed method has a restricted applicability and is thus explored in form of a case study in this chapter. A more general detection method for both PDF and Adobe Flash files will be presented in Chapter 4. The remainder of this chapter contains material from our Annual Computer Security Applications Conference (ACSAC) 2011 publication [45].

After the introduction in Section 3.1, we present a survey of prior work (Section 3.2) followed by a brief summary of mechanisms for embedding JavaScript into PDF (Section 3.3). The system design and methodology is presented in Section 3.4. In Section 3.5 we explore the data corpus and analyze its statistical features. Our experimental evaluation is presented in Section 3.6. Limitations of our method and related later work are discussed in Section 3.7, and Section 3.8 concludes this chapter.

3.1 Introduction

Since the discovery of the first critical vulnerability in Adobe Reader in 2008[1], the Portable Document Format has become one of the main attack vectors used by miscreants. PDF-based attacks were the most frequently used remote exploitation technique in 2009 with a proud share of 49 %. Two specific PDF-based vulnerabilities were ranked second and fifth among all discovered in 2009 [96]. Overall, more than 50 vulnerabilities were discovered in Adobe Reader in 2008–2010, which has led to numerous security-related updates.

The vulnerabilities of Adobe Reader can be classified into three categories:

1. The earliest—and the largest—class of vulnerabilities arises from bugs in the implementation of the Adobe JavaScript API. This API significantly extends the Ja-

[1] `collab.CollectEmailInfo` (CVE-2007-5659).

vaScript functionality in the specific context of PDF files.

2. The second class of vulnerabilities is rooted in non-JavaScript features of Adobe Reader but typically requires JavaScript for exploitation, e.g., using heap spraying. Examples of such vulnerabilities are the JBIG2 filter (e.g., CVE-2009-0658) and heap overflow (e.g., CVE-2009-1862) exploits.

3. Finally, the smallest class of vulnerabilities, e.g., the flawed embedded TrueType font handling (CVE-2010-0195), does not involve JavaScript functionality.

Unlike other exploitation techniques such as drive-by-downloads, SQL injection or cross-site scripting, PDF-based attacks have not received significant attention in the research community before 2011. Previous work in this field has mostly focused on dynamic analysis techniques. For example, well-known sandboxes JSand [22] and CWSandbox [109] have been adapted to the analysis of malicious PDF files. Due to heavy instrumentation and security risks associated with dynamic analysis, the practical applicability of such approaches is limited to malware research systems. For end-user systems, some early work on the detection of potential exploits in PDF files [48, 84] has gone largely unnoticed, and in practice the detection of malicious PDF files hinges upon signatures provided by antivirus vendors.

In this chapter we explore *static* techniques for detection of JavaScript-based PDF malware. Our aim is to develop efficient detection methods suitable for deployment on end-user systems as well as in the networking infrastructure, e.g., email gateways and HTTP proxies. We present the tool PJScan[2], capable of reliably detecting PDF attacks with operational false positive rates in the promille range. Its low computational overhead makes it very attractive for large-scale analysis of PDF files.

Conceptually, PJScan is closely related to static analysis techniques for detection of browser-based JavaScript attacks. Similarly to previous work by Rieck et al. [80], our methodology is based on lexical analysis of JavaScript code and uses machine learning to automatically construct models from available data for subsequent classification of new data. The crucial difference from browser-based JavaScript attacks is that reliable ground truth information is hardly available for PDF files. It is especially difficult to identify benign JavaScript-bearing PDF files. Firstly, as our study will show, such examples are indeed much more rare than malicious ones. Secondly, while it is relatively easy to verify that web content at a certain URL is benign, e.g., by using Google Safe Browsing[3], it is much more difficult to extract and analyze JavaScript code in PDF files. These implications necessitate a conceptual re-design of detection methods. We therefore resort to *anomaly detection* to learn only from malicious examples.

Furthermore, reliable extraction of JavaScript code from PDF files poses a challenge in itself. Not only is PDF very complex, it is also rich with features that can be used for

[2]The source code of PJScan can be found at `http://sf.net/p/pjscan`.

[3]Google Safe Browsing – `https://developers.google.com/safe-browsing/`.

hiding the presence of JavaScript code. As elaborated in Chapter 2, PDF supports compression of arbitrary objects as well as various encodings for JavaScript content. Such features are routinely used by attackers to avoid detection by signature-based methods. In our experience, none of the previous tools for static analysis of PDF files, e.g., PDFID[4], JSUNPACK[5], or PDF Dissector (discontinued), were able to provide reliable extraction of JavaScript code from PDF files. In the preprocessing component of PJScan, we have developed an interface to the PDF rendering library Poppler[6]. Using this interface, our system is able to handle nearly all potential locations of JavaScript known to us from the PDF Reference[7].

We have evaluated the effectiveness of PJScan on a large real-world dataset comprising 3 months of data uploaded by users to the malware analysis portal VirusTotal. This is the first study of malicious PDF files carried out at such scale. Our analysis has found zero-day PDF malware samples, i.e., malicious PDF files undetected by all antiviruses (we have found 52 such files among more than 40,000 classified by VirusTotal antiviruses as benign). In our experiments, PJScan has attained average detection rates of 85 % for known and 71 % for previously unseen PDF malware with the average operational false positive rate of about 0.37 %. Due to the difference in the nature of benign data, a direct comparison of PJScan with methods for detection of browser-based JavaScript attacks is impossible. Wepawet, unfortunately since discontinued, was the only previous detection method suitable for PDF-based JavaScript attacks. Much to our surprise, while attaining a perfect false positive score and being very good in detection of novel PDF attacks (90 %), Wepawet has shown poor performance on *known* PDF attacks, for which it only reached the detection accuracy of 63.6 %. As a dynamic analysis tool, Wepawet has been conceived for offline processing and therefore incurs a significant runtime overhead.

In summary, this chapter provides the following contributions:

- **Robust extraction of JavaScript from PDF files.** We provide a detailed account of mechanisms for embedding JavaScript into PDF files and present a methodology for its reliable extraction using the open-source PDF parser Poppler.

- **Fully static detection of malicious JavaScript.** We describe a method for discrimination between malicious and benign JavaScript instances based on lexical analysis and anomaly detection. Unlike previous work, the proposed method does not require manual ground truth labeling. This is especially important for PDF files for which it is difficult to establish ground truth.

- **High performance.** The key advantage of static analysis is that it allows several orders of magnitude higher processing speed. Our system PJScan attains an

[4]PDFID – `http://blog.didierstevens.com/programs/pdf-tools/`.
[5]JSUNPACK – `https://github.com/urule99/jsunpack-n`.
[6]Poppler – `https://poppler.freedesktop.org/`. Version 0.14.3 was used in our implementation.
[7]The sole exception, JavaScript code in XFA forms, will be discussed in Section 3.7.

average runtime of 31 ms/file/CPU, the fastest of all published methods.

- **Comprehensive evaluation.** We present the results of the first large-scale evaluation of malicious PDF detection on a real-world dataset comprising over 65,000 PDF files. PJSCAN has detected 85 % PDF files labeled by at least 5 antivirus tools deployed at VIRUSTOTAL as malicious. Furthermore, it detects 71 % of zero-day malicious PDF files missed by all VIRUSTOTAL scanners. The promille-range false positive rate of PJSCAN makes it suitable for practical deployment.

- **Source code release.** Source code of PJSCAN is released as open-source software.

3.2 Prior Work

The earliest approaches to identification of malware in PDF files [48, 84] were based on *n*-gram analysis of *raw file content*. The scope of experimental evaluation in this work was rather limited. It included self-generated malicious PDF files as well as a relatively small number of examples (less than 300) from the outdated VXHEAVENS malware repository. Due to the wide-spread use of evasion techniques in modern PDF malware, especially object compression and code-level obfuscation, we find the analysis of raw content of PDF files inadequate. Consequently, the approach taken in PJSCAN is fundamentally different from the above-mentioned work in that our methods spend a lot of effort in discovering and utilizing the appropriate lexical features of PDF.

Other prior work on PDF file classification has emerged from existing tools for static and dynamic analysis. Early dynamic approaches were based on software emulation [1, 71] and abstract payload execution [103]. However, they were shown to be susceptible to evasion and computationally intensive. Besides the portal WEPAWET considered in our experiments and based on the sandbox JSAND [22], some other tools use a combination of static and dynamic analysis. One example is MALOFFICE [28], which also employs heuristics. Its static analysis is based on the utility PDFTK[8], and its dynamic analysis builds on CWSANDBOX [109]. Detections are made by combining scores from various heuristics and policies attached to the analysis tools. Another example is MDSCAN, proposed in [104]. From the architectural point of view, MDSCAN is similar to our approach. It also uses static analysis to extract JavaScript content (although using a self-made parser) and a heuristic approach for the extraction of JavaScript code. The extracted code is interpreted using SPIDERMONKEY and the detection is carried out dynamically by utilizing the shellcode detection tool NEMU [71]. The method was evaluated on a set of 197 malicious PDF files artificially generated using the Metasploit framework and 2000 benign files. Compared to MDSCAN, we use SPIDERMONKEY only for token extraction and perform detection statically, which brings a performance improvement of two orders of magnitude.

[8] PDFTK – `https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/`.

A significant body of prior work has addressed the detection of malicious JavaScript in web content, especially in the context of drive-by-downloads. One cannot directly compare the accuracy of such methods with PJSCAN due to the fact that the data corpora used for the experimental evaluation of respective methods are very different. We will hence focus on methodical comparison of our approach with such methods.

Similar to PDF malware, the methods for detection of malicious JavaScript in web content can be classified into static, dynamic and hybrid. Purely dynamic methods deploy various techniques for monitoring the run-time execution of processes accessing web content, e.g., full-fledged host virtualization [107], client virtualization [59], instrumentation of a JavaScript engine [29] or heap monitoring [76]. Dynamic methods have high detection accuracy and are hardly prone to false positives, however, due to their performance overhead they are usually limited to "post-mortem" analysis.

Hybrid methods aim to minimize run-time overhead while retaining high detection accuracy. Several such methods have methodical affinity with PJSCAN. JSAND [22] uses instrumented versions of HTMLUNIT[9], a Java-based browser simulator, and the Mozilla RHINO[10] JavaScript interpreter to extract heuristic features while monitoring code execution. These features are used to train an anomaly detection system by running JSAND on benign web pages. CUJO [80] is another interesting combination of static and dynamic methods. Its static part is similar to PJSCAN, with the exception that PJSCAN employs anomaly detection instead of two-class classification in its learning component. CUJO's dynamic component extracts symbolic features from the light-weight sandbox ADSANDBOX [25] and deploys similar *n*-gram analysis and learning techniques as the static part. As a "mostly static" detection system, ZOZZLE [23] avoids dynamic analysis except for unraveling source-code obfuscation before using statistical feature extraction and supervised learning for classification. Compared to these hybrid methods, PJSCAN uses "reverse" anomaly detection—since only malicious data is widely available for PDF files—and completely dispenses with run-time analysis. Another hybrid method has been proposed by Provos et al. [72]; however, the lack of a technical presentation in this reference prevents us from a detailed comparison.

The only fully static method in the web domain is PROPHILER [14]. It deploys techniques similar to JSAND except that its features are extracted from a JavaScript engine at the parsing stage *without running the code* (a similar idea is used in our method but one step earlier, by stopping SPIDERMONKEY after the lexical analysis). However, PROPHILER has a high false positive rate and is intended to be used as a filter for subsequent dynamic analysis.

[9] HTMLUNIT – `http://htmlunit.sourceforge.net/`.
[10] RHINO – `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino`.

3.3 JavaScript in PDF

PDF provides several mechanisms for inclusion of JavaScript code. They are important for the realization of interactive features, such as forms, dynamic content or 3D rendering. Some PDF usage scenarios relying on these features cannot be realized without JavaScript.

The main indicator of JavaScript code is the presence of the keyword `/JS` in a PDF dictionary. The JavaScript source code can be supplied directly as a string (literal or hexadecimal) or stored in another object accessed *via* an indirect reference. In the latter case, it is usually stored as a compressed or encrypted stream. Examples of typical syntax for embedding of JavaScript code are shown in Fig. 3.1.

```
1 0 obj <<
  /Type /Catalog
  /Pages 2 0 R
  /OpenAction <<
    /S /Rendition
    /JS 23 0 R
  >>
>>
endobj
```

```
1 0 obj <<
  /Type /Catalog
  /Pages 2 0 R
  /OpenAction <<
    /S /JavaScript
    /JS (alert('Hello World!');)
  >>
>>
endobj
```

Figure 3.1: Examples of syntactic constructs for embedding JavaScript code in PDF files. Left: code is placed in another object (omitted) pointed to by an indirect reference. Right: code is supplied as a literal string.

A simple search for `/JS` patterns in PDF files – as it was realized in some tools for PDF file analysis, e.g., PDFID – does not suffice for identification of JavaScript locations. It can be easily evaded by placing dictionaries containing JavaScript into object streams. PDF stream compression makes the keyword `/JS` in that case appear obfuscated when viewed in plain text. The simple search may also yield multiple references to identical code if different revisions of the same content are present.

In order to reliably extract JavaScript code, files must be processed at the semantic level, i.e., considering potential uses of JavaScript in the file. In general, the use of JavaScript code in PDF files is bound to the so-called *action dictionaries*. Such dictionaries may be tagged by a keyword/value pair `/Type/Action`, but such explicit qualification is optional and cannot be relied upon. A mandatory feature of all action dictionaries is the keyword `/S` which may take on 18 different name values. Two of those, `/JavaScript` and `/Rendition`, are important for the search for JavaScript code. The former must, and the latter may have a keyword `/JS` [67], as shown in Fig. 3.1. Content associated with the keyword `/JS`, i.e., the code itself, must be encoded using one of two encodings: the encoding defined in the PDF Reference as PDFDocEncoding or the UTF-16BE (big-

endian) Unicode encoding. In the rest of this chapter we denote JavaScript source code located in or referred to by one *JavaScript* or *Rendition* action dictionary as a *JavaScript snippet*.

JavaScript or *Rendition* action dictionaries can be found at the following locations of the PDF object hierarchy:

- the *Catalog* dictionary's `/AA` entry may define an additional action specified by a JavaScript action dictionary,
- the *Catalog* dictionary's `/OpenAction` entry may define an action to be run when the file is opened,
- the file's name tree may contain an entry `JavaScript` that maps name strings to file-level JavaScript action dictionaries executed when the file is opened,
- the file's *Outline* hierarchy, referenced by the `Outlines` entry of the *Catalog* dictionary, may contain references to JavaScript action dictionaries,
- pages, file attachments and forms may also contain references to JavaScript action dictionaries[11].

Besides being directly embedded in a PDF file, JavaScript code may also reside in a different file on the local host computer or even be retrieved from a remote location using the directives `/URI` or `/GoTo`. However, newer versions of PDF viewers restrict this behavior to increase security. JavaScript also supports dynamic code execution using the `eval()` function or its equivalent, `setTimeOut()`. Such code is generally inaccessible to static analysis; however, it has to be fetched from existing snippets inside the file, i.e., *entry point code*, which we have access to.

3.4 System Design

Conceptually, our system comprises two main components, *feature extraction* and *learning*, as shown in Fig. 3.2. The first component extracts JavaScript code embedded in PDF files and performs lexical analysis on it, outputting as result one sequence of lexical tokens for every file. The second component operates in two steps. In the first step, i.e., at *training* time, it takes the extracted token sequences and trains an anomaly detection algorithm on examples of *malicious* files, thus learning a *model* of lexical properties of JavaScript code in malicious PDF files. In the second step, i.e., during *deployment*, this model is used for *classification* of new files for which the ground truth labels are unknown. Classification is performed by measuring the deviation of a new file from a learned model and comparing it against a predefined threshold usually determined at

[11] PJScan's handling of file attachments and forms is described in Section 3.7.

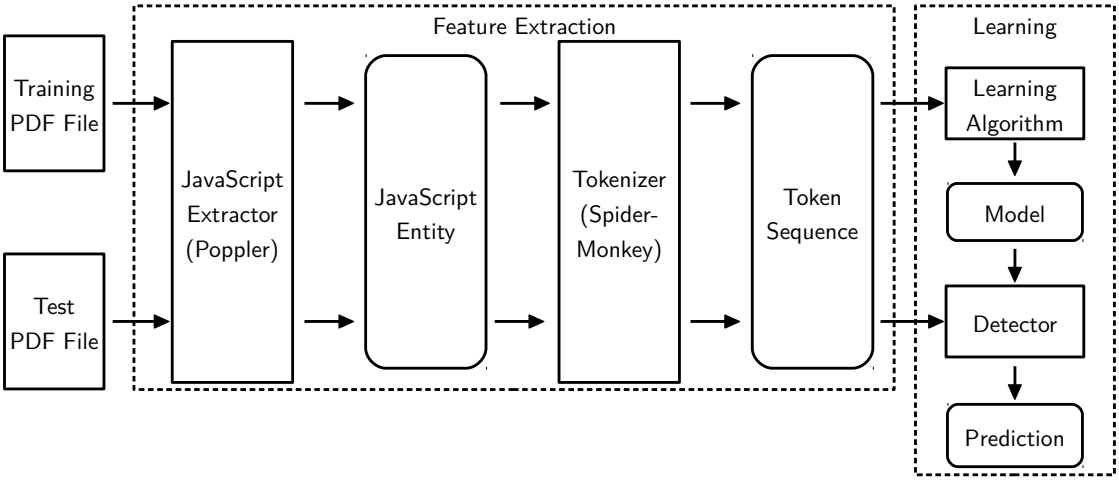


Figure 3.2: Architecture of PJScan.

training time. Files falling close to the learned model are classified as malicious and otherwise as benign. Details of individual components depicted in Fig. 3.2 are reported in the following sections.

3.4.1 Extraction of JavaScript Content

The main technical challenge in extracting JavaScript code from PDF files lies in the decoding of object streams and handling of string encodings. Furthermore, a parser must be robust against potential incompatibilities with the PDF Reference. For this reason, contrary to the approach taken in [104], we have decided against parsing PDF files "by hand" and instead utilize the popular open-source PDF parser Poppler.

Our JavaScript extractor begins by opening the PDF file and initializing Poppler and its internal data structures. Next, the *Catalog* dictionary is retrieved which serves as the starting point in the search for action dictionaries. All candidate locations listed in Section 3.3 are checked, and the found action dictionaries are queried for their type. If the type is *Rendition* or *JavaScript* and a dictionary contains the `/JS` key, the value of this key (or the referenced object in case of an indirect reference) is retrieved. The string containing JavaScript is then decompressed and decoded where necessary.

The peculiarity of our approach is that we fully process only those objects in which *JavaScript* and *Rendition* action dictionaries may occur. This strongly reduces the computational effort for JavaScript extraction and is crucial for efficient batch processing of large datasets. Files without JavaScript are not processed beyond this stage.

3.4.2 Lexical Analysis

At the lexical level, source code is represented as a series of tokens denoting individual lexical elements, e.g., keywords, literals, variable and function names, punctuation, etc. Let us illustrate the tokenization process using an example. The malicious JavaScript snippet

```
bvb('var lBvXSUfYYL7RK = ev' + 'al;'); // a real example
lBvXSUfYYL7RK('var uzWPsX8 = this.info' +
   z("%2e%46%61%6b") + 'erss;');
```

is translated into the lexical token sequence shown in Table 3.1, in order from top to bottom.

Value	Symbolic name	Description
29	TOK_NAME	identifier
27	TOK_LP	left parenthesis
31	TOK_STRING	string constant
15	TOK_PLUS	plus
31	TOK_STRING	string constant
28	TOK_RP	right parenthesis
2	TOK_SEMI	semicolon
29	TOK_NAME	identifier
27	TOK_LP	left parenthesis
31	TOK_STRING	string constant
15	TOK_PLUS	plus
29	TOK_NAME	identifier
27	TOK_LP	left parenthesis
31	TOK_STRING	string constant
28	TOK_RP	right parenthesis
15	TOK_PLUS	plus
31	TOK_STRING	string constant
28	TOK_RP	right parenthesis
2	TOK_SEMI	semicolon
0	TOK_EOF	end of file

Table 3.1: Example token sequence.

We were motivated to use token sequences instead of other properties of JavaScript code because their extraction and processing can be implemented very efficiently and they capture the essential program flow.

Lexical analysis is efficiently performed by the open-source JavaScript interpreter SpiderMonkey[12] developed by the Mozilla Foundation. We have patched SpiderMonkey to stop immediately after lexical analysis. This ensures that code does not get executed and thus PJScan is guaranteed to be a static method. Our extractor queries SpiderMonkey for lexical tokens until an end-of-file or an error is encountered. Tokens in SpiderMonkey are defined as named integer constants ranging from -1 (error) to 85.

Some semantics of the code are lost during lexical analysis. For example, all identifiers are translated to the same token `TOK_NAME`, regardless of their names. We refer to this as *identifier erasure*. Similarly, calls to functions with different names but otherwise identical signatures are translated into equal token sequences. As a result, two JavaScript snippets with different source code may translate to identical token sequences. We say for such snippets that they are distinct at the source code level but equivalent at the lexical level.

Besides the tokens recognized by SpiderMonkey, we have defined extra tokens that are indicative of malicious JavaScript snippets. The newly-introduced tokens are listed in Table 3.2. Their impact on classification performance of is evaluated in Section 3.6.4.

Value	Symbolic name	Description
101	`TOK_STR_10`	string literal of length < 10
102	`TOK_STR_100`	string literal of length < 100
103	`TOK_STR_1000`	string literal of length < 1,000
104	`TOK_STR_10000`	string literal of length < 10,000
105	`TOK_STR_UNBOUND`	string literal of length > 10,000
120	`TOK_UNESCAPE`	call to `unescape()`
121	`TOK_SETTIMEOUT`	call to `setTimeOut()`[13]
122	`TOK_FROMCHARCODE`	call to `fromCharCode()`[14]
123	`TOK_EVAL`	call to `eval()`

Table 3.2: Custom tokens.

3.4.3 Learning and Classification

In the last step of our processing chain, the learning component of PJScan determines whether a PDF file is benign or malicious. After the feature extraction steps described in

[12] SpiderMonkey – `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

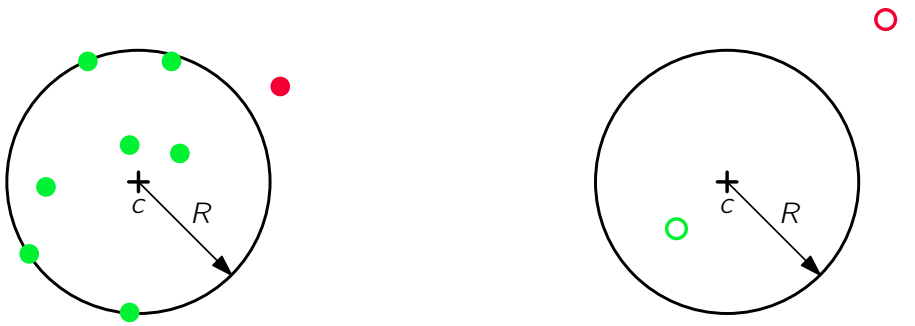
[13] In the PDF JavaScript API, the function `setTimeOut()` of the `app` object can be used as a replacement for `eval()` to execute arbitrary JavaScript code after the specified timeout.

[14] `fromCharCode()` is a static method of the *String* object that converts Unicode values to characters. In malicious files, it is used to decode encoded strings for execution using `eval()`.

Sections 3.4.1 and 3.4.2 are completed, the distance of the token sequence from the new file to the model is computed. Our system employs the One-Class Support Vector Machine (OCSVM) [101] as the learning method[15], a specialization of the Support Vector Machine classifier described in more detail in Section 4.4.3. Its main advantage is that it only requires examples of one class, in our case malicious, to build a model. This is necessary for our problem since examples of benign PDF files with JavaScript content are quite rare, and it takes a lot of manual effort to verify that they are indeed benign. On the other hand, examples of malicious PDF files abound on malware collection systems, and their maliciousness can be ascertained with high confidence if they are detected by antivirus products.

The learning stage of OCSVM, illustrated in Fig. 3.3a, amounts to finding the center c and radius R of a high-dimensional hypersphere such that the total percentage of all data points lying outside of the hypersphere is at most ν. A hypersphere may be extended to arbitrary surfaces by a non-linear transformation to a special feature space equipped with the so-called "kernel function". The type of the employed kernel function and the training rejection rate ν are the only parameters to be specified for training of OCSVM. The learned model comprises the center of the sphere c and radius R.

In the classification stage of OCSVM, we compute the distance between the data point of interest and the center of the hypersphere. If the distance is greater than R , i.e., the data point lies outside of the hypersphere, then it is considered an anomaly and labeled benign. The radius thus serves as a threshold that is automatically determined at the training stage. The classification stage of OCSVM is illustrated in Fig. 3.3b.



(a) Learning stage: the center c and the radius R of the sphere are determined.

(b) Classification stage: new data are accepted (green) or rejected (red).

Figure 3.3: One-Class Support Vector Machine operation.

OCSVM cannot be directly applied to token sequences emitted by PJScan's feature extraction component. The reason for this is that OCSVM expects the data points to be numeric values lying in a high-dimensional space equipped with typical mathemat-

[15]The popular open-source SVM implementation LibSVM [17] version 2.86 was used in our experiments, extended to support one-class SVM.

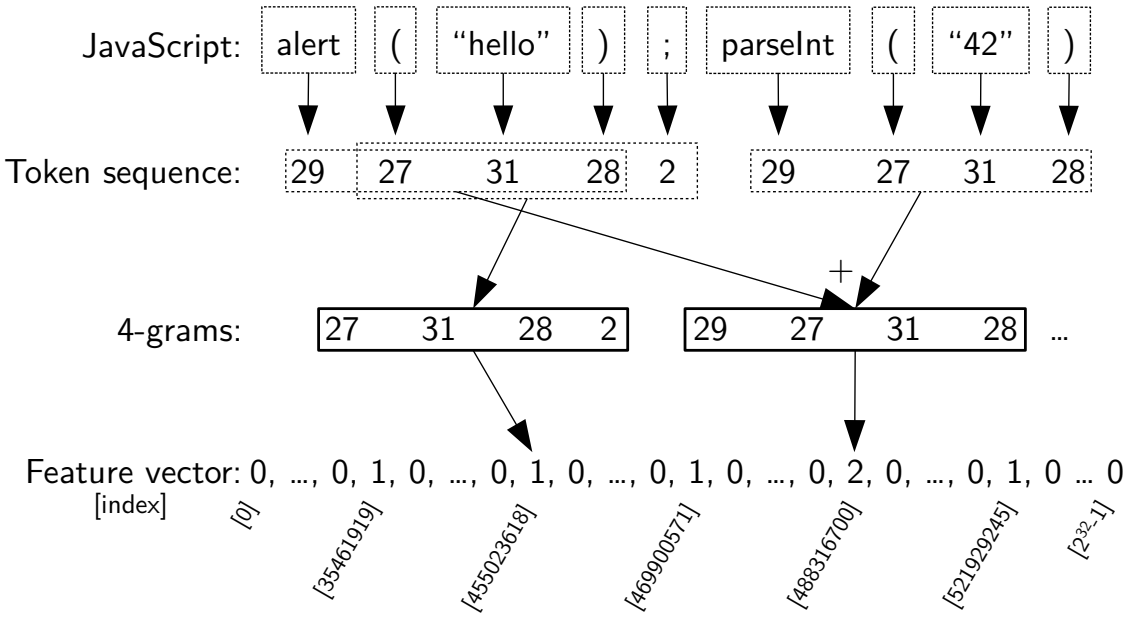


Figure 3.4: PJScan feature extraction. JavaScript is transformed into token sequences, from which 4-grams are extracted and counted (for clarity, only 2 of 5 distinct ones shown). Finally, the resulting counts are assigned to corresponding dimensions of the feature space.

ical operations such as addition, multiplication with a constant and the inner product. Sequential data does not form such a space: it is not immediately clear how to add or multiply two strings. A solution to this problem involves a well-established technique of embedding sequences in metric spaces [79]. By counting the occurrences of substrings in data points and assigning the resulting numeric values to coordinate axes, the required mathematical properties can be enforced.

Fig. 3.4 illustrates this in more detail. In the first step, the token sequence is obtained by lexical analysis of the given JavaScript snippet. Next, all 4-grams are extracted from the token sequence. Every n-gram corresponds to one dimension of the feature space, as determined by the function $t : G_n \to \overline{1,|G_n|}$, where G_n is the set of all n-grams. Thus, for G_4 and an alphabet of 87 tokens, the feature space has 87^4 dimensions. However, for performance reasons, PJScan uses 8 bits per token, resulting in a feature space of $2^{8^4} = 2^{32}$ dimensions. Function t operates by concatenating the 8-bit representations of the 4 tokens and interpreting the resulting 32-bit number as an unsigned integer. For example,

$$t(29,27,31,28) = 29 \cdot 2^{24} + 27 \cdot 2^{16} + 31 \cdot 2^8 + 28 = 488316700.$$

Multiple occurrences of the same n-gram are summed, as illustrated on the example of n-gram `29,27,31,28`.

Sequence embedding solves the problem of handling multiple JavaScript snippets in the same file in an elegant way. To obtain an aggregated representation of all JavaScript snippets, it suffices to sum them using the addition operation provided by the embedding. To avoid the dependence on sequence length, the values in individual dimensions are binarized (by setting any non-zero values to 1) and normalized so that the Euclidean norm of the resulting vectors equals 1.

3.5 Data Collection and Analysis

The success of any learning-based approach crucially depends on the quality of data available for training. Likewise, the viability of a learned model can only be demonstrated on up-to-date real data. The evaluation of our method rests on an extensive dataset collected from the research interface to VirusTotal, a web service that enables ordinary users to upload suspicious files to be scanned by most commercially available antivirus tools[16]. Our dataset comprises 65,942 PDF files with the total size of nearly 59 GB. This data has revealed some interesting features, and is worth looking at in some detail.

We downloaded 3 batches of data at 3 separate occasions:

1. November 3, 2010
2. January 19, 2011
3. February 17, 2011.

Every downloaded batch contains all PDF files available at the given date. The data is kept on VirusTotal only for 30 days. In fact, we were originally unaware of the 30-day lifespan and started a periodic collection of snapshots only in January. There is surprisingly little overlap between subsequent months as we have observed at most 200 identical files across different snapshots. Our corpora were split into two parts:

- Detected, comprising files labeled malicious by at least one antivirus product,
- Undetected, comprising supposedly benign data.

It is instructive to look at the statistical properties of our data presented in Table 3.3 where a number of revealing observations can be made. Files in the Detected corpora with an average size of 0.106 MB are an order of magnitude smaller than those in Undetected (1.39 MB), suggesting an utter lack of meaningful content in malicious PDF files, a finding confirmed by a manual investigation of a subset of them. Furthermore, the percentage of files with JavaScript in the Detected corpora (59.5 %) is in stark contrast to the Undetected corpora (2.4 %), providing further indication that JavaScript plays a crucial role in PDF-related exploits.

Further narrowing our focus to files containing JavaScript, we notice that there are on average over 30 times *less* JavaScript snippets per file in Detected (7.2) than in Undetected datasets (241.1). This observation seemed counter-intuitive before we analyzed

[16] During our evaluation there were 42 deployed antivirus tools.

	03. Nov. 2010		19. Jan. 2011		17. Feb. 2011		
	det.	undet.	det.	undet.	det.	undet.	Total
Dataset size	873 MB	13 GB	429 MB	13 GB	1.5 GB	29 GB	59 GB
Files in the dataset	7592	7768	6465	9993	11634	22490	65942
Files with JavaScript	6626	272	1127	196	7526	492	16239
JavaScript snippets	26372	75199	33418	42265	50269	113994	341517
Distinct JS snippets	8597	5178	2376	3774	9238	6827	35990
Distinct token sequences	1108	429	815	356	2947	764	N/A
Distinct feature vectors	538	115	358	95	1900	237	N/A

Table 3.3: Statistics of PDF files collected from VirusTotal.

the snippets themselves, but we discovered that Undetected data usually contains hundreds of very simple ones such as:

```
this.zoom=100;this.pagenum=39
```

while Detected ordinarily contains large snippets, i.e., complete programs. Furthermore, distinctness of JavaScript snippets at the code level is 3.2 times higher in Detected corpora than in Undetected (16.9 % vs. 5.2 %), suggesting that benign JavaScript in PDF files essentially boils down to boring and redundant code.

Similar effects take place at the lexical level (Table 3.3, second-to-last row). One can observe a further decrease in distinctness (6,419 vs. 35,990, or 17.8 %) due to lexical analysis. This effect can be explained by identifier erasure as elaborated in Section 3.4.2. As with source code, Detected remain more distinct than the Undetected sub-corpora on the lexical level as well. Finally, the last row in Table 3.3 reveals that many files contain identical sets of token sequences, i.e., when all their token sequences are embedded and summed they result in the same feature vector. This can be explained by common code reuse in both types of files.

To enable a quantitative evaluation of detection accuracy in the forthcoming experiments, we have manually labeled the Undetected part of our data. Among 960 benign files with JavaScript, we found 52 that we believe to have been falsely classified as benign by all antivirus tools at VirusTotal. However, no cases were found where PDF files belonging to the same group of lexically distinct files were assigned conflicting labels.

3.6 Experimental Evaluation

The real-world nature and the sheer size of the VirusTotal data make the evaluation challenging. Firstly, the distinction between Detected and Undetected corpora is somewhat vague, as classifications by antivirus engines cannot be fully trusted. Secondly, the huge size of the Detected corpus makes its manual analysis infeasible. On the other hand, the

size of the labeled JavaScript-bearing part of the UNDETECTED corpus is too small to be used for training purposes.

As the baseline for comparison we consider WEPAWET, a web-based service using JSAND [22]. WEPAWET performs both static and dynamic analysis of PDF files based on their JavaScript content and can detect both malware that it has a signature for (labeled as *malicious*), as well as unknown malware (labeled as *suspicious*) using statistical features. In the evaluation, we treat both categories as detections. Similar to our system, WEPAWET generally does not recognize PDF malware that lacks JavaScript. Table 3.4 shows WEPAWET's classification on the DETECTED and UNDETECTED parts of the three data corpora at our disposal. In some cases file uploads were rejected by WEPAWET, referred to as *fail*, or resulted in internal errors despite multiple submissions, referred to as *error*. We treat such cases (about 1.7 % of the total data) as benign.

	03. Nov. 2010		19. Jan. 2011		17. Feb. 2011	
	detected	undetected	detected	undetected	detected	undetected
Fail	12	38	9	25	19	73
Error	15	1	5	0	83	0
Benign	3860	212	502	167	1050	397
Suspicious	1474	11	149	0	257	0
Malicious	1265	10	462	4	6117	22

Table 3.4: WEPAWET classification results.

3.6.1 Objectives and Evaluation Criteria

Our experiments address the following questions:

1. *How well do* PJSCAN *and* WEPAWET *detect known malicious files?*

   This question may appear meaningless: why bother detecting something that is already detected? In practice, however, it is infeasible to deploy all antivirus tools from VIRUSTOTAL. For a single method, attaining the detection accuracy close to that of the combined accuracy of all established antivirus tools is a very challenging goal[17]. The corresponding metric is the *true positive rate on known attacks* TPR_{known}, defined as the percentage of correctly identified malicious JavaScript-bearing files in the DETECTED corpus.

[17] Unfortunately we were unable to compare against the *best* detector at VIRUSTOTAL. At the time of collection, labels in batch data from VIRUSTOTAL reflected only the number of detections but not the specific detectors that classified a file as malicious.

2. *How well do both methods detect attacks that were missed by all* VirusTotal *detectors?*

   We consider files in the Undetected corpus as novel attacks if they are shown to be malicious during manual analysis. The *true positive rate on unknown attacks* $TPR_{unknown}$ is defined as the percentage of correctly identified malicious JavaScript-bearing files in the Undetected corpus. In Section 3.5 we have identified just 52 such files.

3. *How many normal files are classified as malicious by the methods in question?*

   The *laboratory false positive rate* FPR_{lab} is defined as the percentage of incorrectly classified benign JavaScript-bearing files in the Undetected corpus. There are only 960 such files.

   The *operational false positive rate* $FPR_{operational}$ is the ratio between the number of incorrectly classified benign files and the total number of files in the Undetected corpus, i.e., including benign files without JavaScript.

   The distinction between the laboratory and the operational false positive rates is essential for estimation of the expected impact of false positives in practical deployment.

3.6.2 Experimental Protocol

Our experiments were carried out using the following procedure. We merged all 3 corpora from different dates keeping only the distinction between "detected" and "undetected" parts, thus creating two data sets, Detected and Undetected. Next we extracted token sequences from files – the most computationally intensive task in our experiments. We randomly split the Detected dataset into two non-overlapping partitions in such a way that they contained approximately the same number of distinct token sequences. One of these data sets is used to train PJScan, the other to evaluate TPR_{known}. To decrease the impact of non-determinism via random splitting, we repeat the experiment the second time by swapping the training and the evaluation datasets and averaging the detection accuracy. This process is known as *2-fold cross-validation*.

To determine the detection accuracy on unknown data, we apply the trained model on the full Undetected corpus. We use the ground truth information to compute $TPR_{unknown}$, FPR_{lab} and $FPR_{operational}$. The reported results are also averaged over the two partitions of the training data.

Since the models used by Wepawet do not depend on our training data but rather on the data it was trained on, the results presented here for Wepawet reflect its performance on our complete datasets (Detected and Undetected).

Some experimentation was needed to choose the parameters for OCSVM. We selected the training rejection rate $\nu = 0.15$ and the n-gram length of 4, which seem to provide the best trade-off between true positive and false positive rates, using a grid search.

3.6.3 Experimental Results

The results of a comparative evaluation of PJSCAN and WEPAWET according to the criteria specified in Section 3.6.1 are presented in Table 3.5. Two configurations of PJSCAN were considered:

- using only native JavaScript tokens
- using a set of additional heuristic tokens introduced in Section 3.4.2.

Detection method	TPR_{known}	$TPR_{unknown}$	FPR_{lab}	$FPR_{operational}$
PJSCAN	84.80	71.15	16.35	0.3694
PJSCAN with extra tokens	85.17	71.15	17.35	0.3918
WEPAWET	63.60	90.38	0.0	0.0

Table 3.5: Detection performance comparison.

It can be seen that PJSCAN significantly outperforms WEPAWET on known malicious data but is less accurate on previously unknown attacks. The cause for the low score on novel attacks is a set of 11 very similar files in the already very small set of 52 novel malicious files. The 11 files in question were all mislabeled as benign by our method due to their minimalist JavaScript code. Namely, it comprised the following snippet:

```
app.setTimeOut(this.info.dgu,1)
```

The identifier `dgu` differed from file to file, but the snippet in question nevertheless translates to a unique token sequence in all of them. The peculiarity of this example is that the attack code resides *not in the JavaScript snippet* itself but in other code – a string stored as a member of the PDF *Info* dictionary[18] to which PJSCAN has no access. The PDF JavaScript API allows for direct references to members of the *Info* dictionary like this. The `app.setTimeOut()` function is equivalent to `eval()` but executes its code after some specified time.

To fully cover such cases, a dynamic execution engine for PDF files is required which WEPAWET has but which needs orders of magnitude more time for code extraction. With an exception of this kind of attack, the detection rate of PJSCAN would have also reached the 90 % mark.

It is not clear to us why WEPAWET has performed relatively poorly on known malicious data. In a related comparative evaluation against CUJO [80] in the context of web-based JavaScript attacks (drive-by-downloads), WEPAWET was a clear winner with a detection rate of 99.8 % compared to 94.4 %. Most likely, the reason for worse performance of

[18] An *Info* dictionary is used to store meta-data about the PDF file, e.g., author name, creation date, creator software, etc.

Wepawet in our experiments lies in technical problems with the extraction of JavaScript code from PDF files.

A relative disadvantage of PJScan is the high false-positive rate. Measured against only the JavaScript-bearing benign files it reaches the painful 17 %; however, due to the rare presence of JavaScript in benign files, its operational false-positive rate remains acceptable and corresponds, for our data, to 1.7 false alarms per day.

One can also see that heuristic tokens do not improve the performance of PJScan and even lead to a slight degradation of the false-positive rate. The causes for this effect as well as for the false positives are elucidated in the following section.

3.6.4 Significant Features

As noted by Sommer and Paxson [89], a security practitioner would always be interested to know what a learning method has actually learned. The model created by the OCSVM (the center c of the sphere) produces a numeric ranking of essential features encountered in malicious JavaScript code. Since no benign data is used for training, this ranking does not reflect the differences between two classes but rather describes only one class known to it. Examples of the 5 most important and the 5 least important features in one of the models learned by PJScan (created for one half of the data) are shown in Table 3.6.

Feature		
Rank	Value	Weight $\times 10^5$
1	`NAME . NAME (`	5285
2	`NAME ASSIGN STR ;`	5106
3	`NAME ( NAME )`	5092
4	`( NAME ) ;`	4574
5	`; VAR NAME ASSIGN`	4314
4051	`) NAME ( THIS`	2
4052	`+- NAME !== NAME`	2
4053	`] ) - NAME`	2
4054	`NAME ] ) -`	2
4055	`TRUE } ; IF`	2

Table 3.6: Features of the center point.

Although these features do not look particularly malicious, the top 5 clearly correspond to typical lexical patterns of programming languages: member function dereferencing (1), string variable assignment (2), function calls (3 and 4) and variable declarations (5). On the other end of the spectrum are the features that are obviously very atypical for programming languages.

The scoring of a new data point in the detection stage involves the identification of an overlapping subset of features between this data point and the learned model. The smaller the "weighted overlap" between the new point and the center (i.e., the sum of weights in the model corresponding to their common features), the larger the distance from the center. This property is confirmed by the examples of accepted and rejected points presented below.

For the accepted points, the main contributions are made by the top features of the trained model. Table 3.7 shows a comparison of top features of one true and one false positive observation. Both were ranked the most positive, i.e., closest to the center of the hypersphere, among all observations in their class. In our example, 5 out of 7 top features of malicious files are present in the false positive example. Such points are virtually indistinguishable in our model, and this explains a high "laboratory" false positive rate observed in our experiments.

Feature		Weight $\times 10^5$	
Rank	Value	True Positive	False Positive
1	`NAME . NAME (`	456	554
2	`NAME ASSIGN STR ;`	441	535
3	`NAME ( NAME )`	439	
4	`( NAME ) ;`	395	
5	`; VAR NAME ASSIGN`	372	452
6	`; NAME ( NAME`	340	413
7	`NAME ( STR )`	318	386

Table 3.7: Comparison of top features of a true and a false positive observation.

It turns out, however, that *very few benign examples* share the "normal" programming language features captured by the learned model. For the two examples of rejected points (Table 3.8, one true and one false negative), the top features have much lower ranks in the learned model. The majority of benign examples have a small "weighted overlap" with the model and hence are rejected.

The investigation of significant features in our models suggests that the key property that enables effective discrimination between malicious and benign code in PDF files is the fact that *benign usage of JavaScript is very rudimentary* from the programming point of view. Anecdotally, the benign example with the highest rejection score corresponds to the code `print(true)`.

3.6.5 Throughput

Throughput was measured on a commodity PC with a quad-core Intel Core i7 860 CPU, 8 GB of RAM and a 7,200 rpm SATA hard disk drive. Eight processes were run

Feature		Weight $\times 10^5$	
Rank	Value	True Negative	False Negative
1	`NAME . NAME (`		1593
7	`NAME ( STR )`	390	
8	`VAR NAME ASSIGN NAME`	390	
10	`) ; NAME ASSIGN`	359	
14	`THIS . NAME (`	338	
15	`ASSIGN NAME . NAME`	338	
98	`. NAME . NAME`		490
141	`( THIS . NAME`		394
154	`THIS . NAME .`		372
355	`NAME ( THIS .`		177

Table 3.8: Comparison of top features of a true and a false negative observation.

concurrently for performance measurement.

Each stage of PJScan was run on a respective data partition (training on one half of Detected corpus, evaluation on the other half and on the full Undetected corpus). Unlike the accuracy measurement, we learned and classified using *all* files, ignoring their redundancy. Learning with thousands of files instead of a few hundred distinct token sequences reduces performance, but due to the fast learning and classification algorithms the difference is negligible.

Processing times for all stages are shown in Table 3.9. In total, parsing 65,942 PDF files, tokenizing 341,517 JavaScript snippets, learning on 15,279 "detected" files with JavaScript and classification of 960 "undetected" files with JavaScript took 1547 s. All measurements are expressed in wall clock time[19]. One can observe that JavaScript extraction represents the most time-consuming stage of PJScan.

	Total (s)	Average (s)	Percentage
Extraction	1356	$(205 \pm 15) \times 10^{-4}$	87.65
Tokenization	180	$(32 \pm 392) \times 10^{-4}$	11.63
Learning	10.19	N/A	0.0009
Classification	0.014	$(15 \pm 9) \times 10^{-6}$	0.66

Table 3.9: Processing time for different stages of PJScan in batch mode.

When using a single process, 2041 s are required to handle the same dataset – just 30 %

[19]Wall clock time measures real time that elapses between the beginning and the end of a task. It includes CPU time, I/O time and any overhead such as the time process spends waiting for execution. It is a good indicator of real performance but is affected by system load.

more than with 8 processes – resulting in an average processing time of 31 ms per file. Overall the CPU usage was very low in our experiments (up to 40 %, with I/O waiting of up to 30 %), while at the same time disk utilization remained above 95 % during the extraction phase. Therefore, we conclude that disk throughput represents the main performance bottleneck for this application. Using a faster storage device or streaming files through a fast network would likely further improve the performance of PJScan.

	Throughput		Time	Speed
	(file/s)	(Mbit/s)	(s)	(ms/file)
Detected	75.8	64.3	339	13
Undetected	33.3	370.5	1208	30
Combined	42.6	303.5	1547	23

Table 3.10: PJScan throughput measurements.

Table 3.10 shows that throughput has a strong dependency on the ground truth of observations. "Detected" files are much faster to process than "undetected", which is consistent with much larger file size of benign samples. The average throughput of 303.5 Mbit/s is suitable for batch processing tasks even for organizations that have a very high volume of PDF traffic. The average processing speed is 23 ms per file. To the best of our knowledge, no other software package achieves lower processing times for PDF to date.

3.7 Discussion

The reported experimental results confirm the practical feasibility of our static, machine-learning-based approach for detection of malicious JavaScript-bearing PDF files. PJScan's preprocessing component can aid in the manual extraction and analysis of JavaScript code from PDF files. The main benefit of its learning component is the ability to *extract knowledge from large-scale malware corpora*. PJScan can derive light-weight models from heuristic knowledge of several dozen virus detectors and tens of gigabytes of collected data. Such models can be deployed with no manual interaction and negligible performance overhead (<50 ms per file). The operational false-positive rate of less than 0.4 % is admissible in practice. Even at a highly visible site like VirusTotal with a strong bias for suspicious data, this corresponds to an average rate of 1.7 false alarms per day for our data collection period (148 out of ca. 40,000 benign files over 90 days).

PJScan's main limitation is its incapability of processing PDF files without JavaScript. However, even in the domain of JavaScript-bearing PDF files, PJScan's coverage is incomplete. After the publication of PJScan in [45], the authors have learned of another method of embedding JavaScript code into PDF files that is not part of the PDF Reference

but of the therein referenced Adobe XML Forms Architecture (XFA). That technology enables the inclusion of XML-based forms into PDF documents and the scripting of forms is performed using JavaScript. In its published implementation, PJSCAN does not have the ability to access code in XFA forms and is, therefore, vulnerable to malicious JavaScript snippets in XFA. However, this is a technical limitation and not a design flaw. With access to JavaScript from XFA forms, PJSCAN restores its detection capability. This was demonstrated in a later study that extended PJSCAN [16], as elaborated later in this section.

The high "laboratory" false-positive rate of PJSCAN (i.e. the rate measured only for those benign files that contain JavaScript) indicates that our learning setup may indeed have difficulty with accurate discrimination between malicious and benign JavaScript content. This observation is also indirectly supported by our analysis of the learned features. Learning from two classes, as it has been done in prior work on web-based JavaScript content, e.g. [14, 23, 80], may be the right way to avoid this limitation. However, the insufficient quantity of available benign JavaScript-bearing PDF data has prevented us from evaluating this scenario for PDF files.

Another limitation of PJSCAN is its inability to detect malicious JavaScript located in parts of the PDF file which were not designed to store JavaScript code. In this case, elaborated in Section 3.6.3, only a very short code fragment is available in a standard JavaScript entry point location that loads the "hidden" malicious code. This is an inherent limitation of all static methods. While individual cases can be handled statically, e.g., by implementing extraction procedures for known hiding locations, there is no general way of extracting hidden code without using dynamic execution.

PJSCAN is in active use by the German Federal Office for Information Security (*Bundesamt für Sicherheit in der Informationstechnik*)[20].

3.7.1 Later Work

Following its publication, PJSCAN was evaluated in multiple studies and compared to numerous other detection methods [6, 16, 39, 54, 56, 57, 61, 62, 82, 85, 111, 112]. Most direct comparisons with prior work confirm its superiority, while methods developed afterwards tend to have better detection accuracy but orders of magnitude slower processing time. In some cases a direct comparison was impossible due to PJSCAN's limitation to JavaScript-bearing files.

Maiorca et al. find its performance on par with the worst-performing antivirus engines on one experiment [56], while in a later study the same authors demonstrate that it outperforms other benchmarked methods, even more recent ones, on detecting malicious JavaScript code injected into benign PDF files [57]. They also show that PJSCAN fails to

[20]Disclosed in the keynote talk "Fighting Targeted Attacks on Government Networks" at the SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2013 in Berlin: `http://dimva.sec.t-labs.tu-berlin.de/slides/dimva.2013.sanitised.pdf`.

detect malicious portable executable (PE) and PDF files embedded inside benign PDF files. The PDF format allows for embedding files inside a PDF file which can subsequently be opened from a PDF viewer. We find this to be a technical limitation which can be alleviated with a very simple extension to PJSCAN that would process embedded files recursively.

Corona et al. demonstrate PJSCAN's ability to detect completely novel attacks performed years after its training, albeit with a lower detection rate, caused in part by its inability to process files with certain CVEs [20]. Similar insights were made in a recent study by Jerome et al. [39], where PJSCAN was tested on a dataset around 2 years newer than its training dataset. It was able to process only 3.9 % of the dataset, likely due to JavaScript in unreachable locations, but still attained an accuracy of 82.19 %. A study by Smutz and Stavrou confirms PJSCAN's incapability to handle files with JavaScript in nonstandard locations [85]. Liu et al. demonstrate the higher detection performance of more recent methods [51].

Cao and Yang implement a successful *poisoning attack* against 4 machine learning systems, including PJSCAN [15]. In adversarial machine learning, a poisoning attack is a situation in which an adversary manipulates the output of a classifier by injecting mislabeled, i.e., *polluted*, data points into its training set. Trained on such a dataset, the classifier will produce wrong predictions when given similar data. In their paper, Cao and Yang have managed to reduce our system's accuracy from 81.5 % to 69.3 % when polluting 21.8 % of its training set and to 46.2 % with 28.2 % polluted data. Although the employed experimental dataset is very small, i.e., 65 malicious PDF files, and the polluted training data subset relatively large, such an attack is a high risk in computer security applications because it is inherently successful against all One-Class SVMs. To mitigate this risk, Cao and Yang propose an approach for making learning algorithms forget certain data samples from their training set. They call this process *machine unlearning* and implement it as an extension to PJSCAN in just 30 lines of code, completely restoring PJSCAN's initial detection accuracy.

Very recently, Carmony et al. have proposed a novel attack against PDF malware detectors called the *PDF parser confusion attack* [16]. It works by exploiting the discrepancy between the detectors' PDF parsing implementations and that of Adobe Reader. Subtle deviations from Adobe Reader, itself deviant from the PDF Standard and inconsistent between versions, were recognized as vulnerabilities in third-party parsers that enable malware to hide from them altogether. The authors were able to produce a malicious PDF file that combines a number of these implementation inconsistencies that has successfully evaded all parsers in their evaluation. Apart from the missing implementation of XFA forms, they have identified 2 additional inconsistencies in PJSCAN's PDF parsing library, POPPLER, related to the handling of encryption in PDF[21], that prevent it from opening PDF files with recent implementations of encryption even if the encryption password is empty. Furthermore, the study authors developed a reference JavaScript extractor

[21] No support for revisions 5 and 6 of security handlers.

that closely mimics that of Adobe Reader for 2 versions, 9.5.0 and 11.0.08, and created a modified version of PJSCAN that employs this accurate extractor. With this enhancement only and without altering its remaining system components, i.e., lexical tokenization, learning and classification, PJSCAN's true positive rate jumps from 68.34 % to over 94 %. This result, along with its solid detection performance on timestamped datasets collected years after training [39], indicate that PJSCAN's underlying detection methodology has remained both effective and efficient in the detection of rapidly-evolving malware since its publication in 2011.

Studies that have measured runtime performance in their evaluations verify that PJSCAN remains the fastest PDF malware detector to date [16, 20].

For a very detailed recent survey of PJSCAN and other academic PDF malware detection methods, we refer the reader to [62]. Finally, a comparison of PJSCAN to our own later work is presented in Section 4.5.3.

3.8 Conclusions

We have proposed a new static approach to detection of malicious JavaScript-bearing PDF files based on lexical properties of JavaScript in malicious PDFs. The main advantages of our approach are its high performance and no need for special instrumentation, such as virtual machines or sandboxing. It can attain about 85 % of the combined detection accuracy of *all* antivirus engines at VIRUSTOTAL while being the fastest published method to date with a performance overhead of less than 50 ms per file. It is only marginally affected by text-level obfuscation since the resulting JavaScript code remains very conspicuous at the lexical level. Due to these advantages, our method is suitable for deployment in email gateways and HTTP proxies.

The computational efficiency of our system PJSCAN has enabled us to evaluate it on an unprecedentedly large real-life data corpus (over 65,000 PDF files) collected from VIRUSTOTAL. This evaluation has confirmed a high detection accuracy of our method for both known and unknown malware. PJSCAN is more prone to false positives than dynamic approaches; however, its operational false positive rate still lies in the promille range, which is feasible for practical deployment.

Thanks in part to its open-source availability, PJSCAN was evaluated in multiple studies and compared to numerous other detection methods in the years following its publication. Most direct comparisons to prior work confirm its superiority, while methods developed afterwards tend to have better detection accuracy but orders of magnitude slower processing time. PJSCAN has been favorably evaluated on PDF files collected years after its model was trained, demonstrating its generalization ability. Finally, it was extended in two publications that repaired a flaw in its JavaScript extraction module and raised its robustness against adversarial attacks. PJSCAN's underlying detection methodology remains both effective and efficient in the detection of PDF malware.

Chapter 4

A General Approach for Malware Detection in Non-Executable Files

Recent targeted malware campaigns extensively use non-executable malware, most commonly PDF, Microsoft Office and SWF files, as a stealthy attack vector. There exists a substantial body of previous work on the detection of non-executable malware, including static, dynamic and combined methods. While static methods perform orders of magnitude faster, their applicability has been hitherto limited to specific file formats.

This chapter introduces HIDOST, the first static machine-learning-based system capable of detecting malware in *multiple file formats*. To this end, it combines file content with logical structure defined by various file format specifications. The system has been implemented and evaluated on two formats, PDF and SWF. However, thanks to its modular design and general feature set, it is extensible to other formats whose logical structure is organized as a hierarchy. HIDOST was published in the EURASIP Journal on Information Security [93] as an extension of our previous work, SL2013, described in the conference paper [91] at Network and Distributed System Symposium (NDSS) 2013[1].

Following the introduction in Section 4.1 and an overview of prior work in Section 4.2, we present the main properties of hierarchically structured file formats with a particular focus on PDF and SWF in Section 4.3. Sections, 4.4 and 4.6, provide details about the system design of SL2013 and HIDOST, respectively, while sections 4.5 and 4.7 present their comprehensive experimental evaluation. We discuss later work, our main findings and extension to other file formats in Section 4.8 and conclude in Section 4.9.

4.1 Introduction

One of the most effective tools for breaking into computer systems remains malicious software, i.e., malware. Malware has developed several insidious traits in the recent decade to serve the needs of criminal business. One of them is the infection of files in well-known formats used to exchange documents between businesses and individuals. Such infection offers the following benefits to attackers:

[1]Parts of this chapter previously published in [91], including text, tables and figures, are copyrighted by the Internet Society.

1. It is easier to lure users into opening documents than into launching executable programs.

2. A steady stream of new vulnerabilities has been observed in the recent years in document viewers due to their high complexity, caused, in turn, by the complexity of document formats.

3. Flexibility and versatility of document formats offer ample opportunities for obfuscation of embedded malicious content.

The same features also hinder the identification of malicious documents and increase the computational burden on the detection tools.

The favorite formats used by attackers are PDF (targeting Adobe Reader), SWF (targeting Adobe Flash Player) and various Microsoft Office formats [18, 97]. In 2012, the pioneering exploit kit Blackhole targeted Java, PDF and SWF files, and its successors have continued this practice [90]. In 2013, the non-executable malware delivered through the web was dominated by PDF and SWF files targeting Adobe Reader and Microsoft Office applications [18]. Flash has seen wide deployment recently for malicious advertising, i.e., placement of malware on legitimate web sites by means of advertising networks. Even some of the most prominent web sites have fallen victims to such attacks [90]. Although prevalently used for redirection to sites serving exploit kits, it is not uncommon for SWF files to target Flash Player directly.

Non-executable files are especially popular as a means for *targeted attacks*. Recent years have brought a range of high-profile targeted attacks against governments and industry, and they are getting more common and ever stealthier. The Miniduke targeted attack campaign against European government agencies used sophisticated PDF files exploiting an Adobe Reader zero-day vulnerability. Four different zero-day vulnerabilities in Microsoft Office were used in the Elderwood attack against the defense industry. The group APT1/CommentCrew used zero-day vulnerabilities in Adobe Reader and Microsoft Office against government and industry targets [98]. Among the 24 zero-days discovered in 2014, 16 targeted Adobe Reader and Flash Player (cf. Fig. 4.1), while Microsoft Word files dominated the list of file types used for targeted attacks [97, 99]. In the first 9 months of 2015, 8 out of top 10 vulnerabilities leveraged by exploit kits were reported to be Flash Player vulnerabilities [77].

The main difficulty in detecting malicious non-executable files is the necessity to understand complex formats. While such difficulty is marginalized in the methods based on dynamic analysis, i.e., rendering a file in an instrumented sandbox, such methods are in general rather slow. Static analysis methods, known for their high performance, usually deploy format-specific detection techniques which do not generalize across formats. To alleviate this problem, we propose a *new static analysis method* with the potential of being more portable across formats. Our experiments demonstrate that, with the incorporation of an appropriate format parser, it can be applied to both PDF and SWF files.

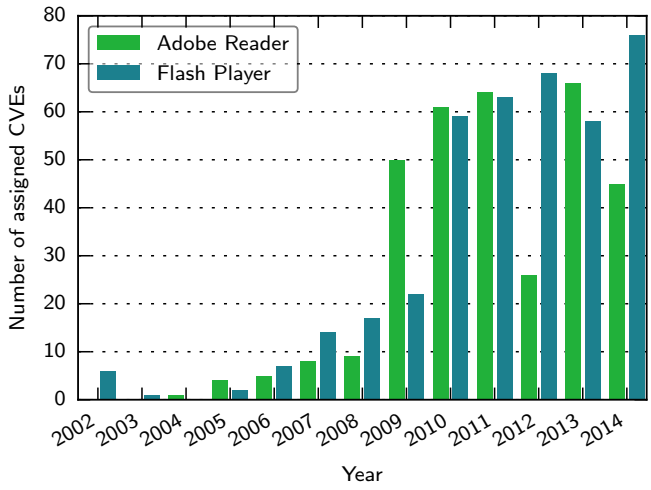


Figure 4.1: Common Vulnerabilities and Exposures (CVEs) with the phrases "Adobe Reader" or "Flash Player" in their description.

The proposed detection method is based on the analysis of hierarchical document structure and is henceforth abbreviated as Hidost. It was originally designed for and evaluated on the PDF file format in our publication [91]; we refer to the first version as SL2013 hereinafter. In our subsequent publication [93] it was generalized to other file formats with a hierarchical logical structure and evaluated on PDF and SWF. This chapter presents both methods chronologically.

The novelty introduced by SL2013 was the use of *logical structure* for characterization of malicious and benign PDF files. PDF logical structure is a high-level construct defined by the PDF Reference that organizes basic PDF building blocks into a functional document. Results published in [91] show that properties of malicious files such as the presence of JavaScript and minimal use of benign content can be accurately determined from their logical structure. Utilizing proper PDF parsing, SL2013 is less affected by PDF obfuscation and physical structure falsification that plague methods based on superficial file examination, e.g., using regular expressions. Evaluated on a real-world dataset comprising 660,000 PDF files, SL2013 has demonstrated a combination of detection performance and throughput that remains unrivaled among antivirus engines and published scientific work. Nevertheless, in a realistic sliding window experiment on timestamped data, the detection performance of SL2013 was shown to be inconsistent. Its feature definition created a blind spot exploitable by evaders and its oversized feature set created difficulties for more memory-intensive machine learning classifiers than the employed Support Vector Machine.

Hidost inherits all the advantages of SL2013. It maintains the nearly perfect detection performance and high throughput on PDF files that tailored SL2013 for centralized deployment on busy networks. As a further advantage of a deep static approach, Hidost is

immune to PDF obfuscation and physical structure falsification.

HIDOST furthermore addresses certain shortcomings of SL2013 discovered after its publication. In particular, it was extended to support a novel technique used to merge similar features called *structural path consolidation* (SPC). Such consolidated features better preserve the semantics of logical structure and reduce the dependency of the feature set on the specific dataset. The benefits of SPC are three-fold: *a*) the attack surface for evasion is reduced; *b*) the feature set is more stable with regard to fluctuations of PDF malware over time, i.e., feature set drift is reduced; and *c*) the total number of features is drastically reduced. Together, these improvements render HIDOST much more secure and practical than SL2013.

Most importantly, however, this chapter introduces a novel system design for HIDOST that enables its generalization to multiple unrelated file formats. To the best of our knowledge, HIDOST is the first static machine-learning-based malware detection system applicable to multiple file formats. Its generality was achieved by extending the feature definition based on the PDF logical structure to a second file format with a hierarchical logical structure, SWF. Finally, going beyond, HIDOST considers not only the logical structure of the file but its content as well, enabling a higher degree of precision on formats with less discriminative structure such as SWF.

We experimentally evaluated HIDOST on two formats: PDF and SWF. Our evaluation protocol is intended to model the practical deployment of a data-driven detection method and to account for a natural evolution of malicious data. In our protocol, a detection model is trained on a fixed-size window of data and is deployed for a limited time period. Once the model is deemed to be too old, it is re-trained on another window of more recent data and again evaluated for a limited time period. Unlike the classical cross-validation methods common in evaluation of machine learning algorithms, our experimental protocol accounts for a temporal nature of data in security applications and never predicts the past data from the future one.

In summary, the main contributions of this chapter are as follows:

- A novel set of features is defined enabling the effective discrimination based on file structure between benign and malicious files of two formats, PDF and SWF, extensible to further ones.

- Using the proposed structural features, the design and evaluation of the PDF malware detector SL2013 is presented.

- A 10-week simulated real-world deployment of the proposed detector on 440,000 malicious and benign PDF files with weekly retraining achieved detection performance close to that of the best deployed antivirus tool on VIRUSTOTAL.

- The throughput of SL2013 is evaluated and shown to be comparable to state-of-the-art static detection techniques.

- Using the proposed structural features, the design and evaluation of a combined PDF and SWF malware detector HIDOST is presented, the first such system applicable to multiple file formats.
- An experimental evaluation of HIDOST is performed on both formats under the same simulated real-world deployment experiment as SL2013, showing that HIDOST outperforms all antivirus engines at VIRUSTOTAL on PDF and ranks among the best on SWF files.
- A prototype implementation of HIDOST for two file formats, PDF and SWF, is released as open-source software.
- Source code required to reproduce the results of the evaluation of HIDOST, including experiments and plots, is released as open-source software.
- Datasets required to reproduce the results of the evaluation of HIDOST are released in form of feature vectors.

Before presenting the main features of the proposed method, we review prior work.

4.2 Prior Work

In Section 3.2 of the previous chapter we described relevant work in the area of PDF malware detection that emerged prior to the publication of our PJSCAN system in 2011. In this section we discuss publications produced afterwards, up to the publication of SL2013, and leave the later work for Section 4.8.3.

Two simple static learning-based methods were proposed subsequent to PJSCAN utilizing heuristic features, MALWARE SLAYER [56] and PDFRATE [85]. Both achieve excellent classification accuracy in their evaluations. However, in contrast to SL2013 and HIDOST which perform PDF parsing in conformance to the PDF Reference, they base their feature extraction on unparsed raw bytes of PDF files, i.e., the PDF physical structure. A common vulnerability of these methods is the relative ease of falsification of the PDF physical structure, as we demonstrate on the example of PDFRATE in Chapter 5.

The family of dynamic detectors was also extended. Snow et al. proposed to employ hardware virtualization and evaluated their system SHELLOS [88] on PDF malware. While the dynamic approaches tend to be more accurate than the static ones, their execution time renders them inadequate for detecting malicious documents on busy networks in real time. Furthermore, building and maintaining a dynamic detector capable of emulating every version of a vulnerable software product in combination with every version of each of its supported operating systems and libraries is a costly and technically challenging task. On the other hand, it suffices to omit one combination of target software from the detector and a threat designed for that specific version will go undetected.

As a combined static and dynamic method, MPScan hooks into Adobe Reader for JavaScript extraction and deobfuscation and performs static exploit detection [53]. Due to its design, it is suitable for malware detection only on a single version of Adobe Reader, and its dynamic component takes seconds to run.

Compared to PDF, research on detecting Flash malware has been scarce with only two methods proposed in the recent years. The OdoSwiff system from 2009 used a heuristics-based approach on features obtained with both static and dynamic analysis [32]. It was succeeded in 2012 by FlashDetect, which upgraded its detection from ActionScript 2 to ActionScript 3 exploits and replaced its threshold-based approach with a Naive Bayes classifier [65]. Both methods are based on an empirical approach, striving to encode the knowledge of domain experts about existing ways of SWF exploitation. These *expert features* perform very well. For example, FlashDetect's machine learning classifier was evaluated using a training dataset comprising only 47 samples of each class, but even this small sample size was enough to achieve a high detection accuracy. However, as the authors point out, some employed heuristics-based features are not robust against steadfast evaders. Furthermore, embedded malware may detect the employed dynamic execution environment based on its difference to Adobe Flash Player, covering its behavior as a reaction. The methods proposed herein use a *data-driven approach* instead of expert features, and their detection is based on structural differences between benign and malicious SWF files. By remaining exploit-agnostic our systems remain open to novel attacks and their static approach enables faster execution.

4.3 Hierarchically Structured File Formats

File formats are developed as a means to store a physical representation of certain information. Some formats, e.g., text files, do not have any logical structure, but others, e.g., HTML, do. HTML files are a physical representation of logical relationships between HTML *elements*. As the example in Fig. 4.2 shows, in an HTML file, a *p* element might be a descendant of the *body* element, which in turn has the *html* element as its parent.

```
<html>
  <body>
    <p>This is a sample HTML file.</p>
  </body>
</html>
```

Figure 4.2: A sample HTML file.

HTML elements have a logical structure in the form of a hierarchy. Work presented in this paper is concerned with the detection of malware in *hierarchically structured file formats*. The physical layout of the file format, which can substantially deviate from

its logical layout, is irrelevant for the operation of the proposed method. Examples of hierarchically structured file formats include:

- Portable Document Format (*PDF*)
- SWF File Format (*SWF*)
- Extensible Markup Language (*XML*)
- Hypertext Markup Language (*HTML*)
- Open Document Format for Office Applications (*ODF*), an XML-based format for office documents
- Office Open XML (*OOXML*), another XML-based format for office documents
- Scalable Vector Graphics (*SVG*), an XML-based format for vector graphics.

In the following we describe the hierarchical logical structure of two file formats implemented in HIDOST, PDF and SWF. We discuss the extension to other file formats in Section 4.8.

4.3.1 Portable Document Format (PDF)

In Chapter 2 we have described the objects and file structure of PDF. Here we introduce the PDF document structure which determines how objects are *logically organized* to represent the contents of a PDF file, e.g., text, graphics, etc.

The body of the example PDF file from Fig. 2.1 is repeated in Fig. 4.3 for convenience, however with primitive data types and references shown in green. Recall that we identified 4 objects in the PDF body, one *Catalog* with ID `1 0`, one *Pages* with ID `3 0` and two *Page* objects with IDs `22 0` and `23 0`. The 4 objects are interconnected via indirect references. *Catalog* points to *Pages*, which in turn points to both *Page* objects. Furthermore, notice that each of the *Page* objects contains a backward reference to the *Pages* object via their `Parent` entry.

The relations between PDF objects, such as in the preceding example, constitute the document structure of PDF files. Notice that the document structure is not a tree but rather a directed rooted cyclic graph, as indirect references may point to other objects anywhere in the document structure. This graph can be reduced to a proper tree, called a *structural tree*, as will be elaborated in Section 4.6.4, and we will henceforth limit our discussion of the PDF document structure to its simplified, tree form. Fig. 4.4 shows the structural tree of our example PDF.

The root node of the document structure is the *Catalog* dictionary. The edges correspond to elements of dictionaries and arrays. Every element of a dictionary, i.e., a key-value pair, corresponds to an edge and a child node; key representing the edge and

```
1 0 obj <<
  /Type  /Catalog
  /OpenAction <<
    /S /JavaScript
    /JS (alert('Hello!');)
  >>
  /Pages 3 0 R
>> endobj

3 0 obj <<
  /Type /Pages
  /Kids [ 22 0 R 23 0 R ]
  /Count 2
>> endobj

22 0 obj <<
  /Type /Page
  /Parent 3 0 R
  /MediaBox [0 0 612 792]
  /Resources ...
>> endobj

23 0 obj <<
  /Type /Page
  /Parent 3 0 R
  /MediaBox [0 0 333 444]
  /Resources ...
>> endobj
```

Figure 4.3: Raw content of an example PDF file. Formatted for easier reading, details omitted for brevity.

value the child node. Edges corresponding to dictionary keys are labeled by the key itself. For example, the key-value pair (`Type`, `Catalog`) corresponds to an edge labeled *Type* and the child node *Catalog*. An array's elements are its children nodes and their edges are not labeled. Every object of a primitive type constitutes a leaf, i.e., terminal node in the document structure. Intermediate nodes may be either arrays or dictionaries, however, empty arrays and empty dictionaries are considered leaf nodes as they have no children by definition.

We define a *path* in the PDF structural tree as a sequence of edges starting at the *Catalog* and ending at a leaf node. For example, in Fig. 4.4 there is a path from the root, i.e., leftmost, node through the edges named `/Pages` and `/Count` to the terminal node with the value 2. This definition of a path in the PDF document structure, which we denote a *PDF structural path*, plays a central role in our approach. We print paths as a sequence of all edge labels encountered during path traversal starting from the root node and ending in the leaf node. The path from our earlier example would be printed as `/Pages/Count`.

The following list shows examples of structural paths from real-world benign PDF files:

```
/Metadata
/Type
```

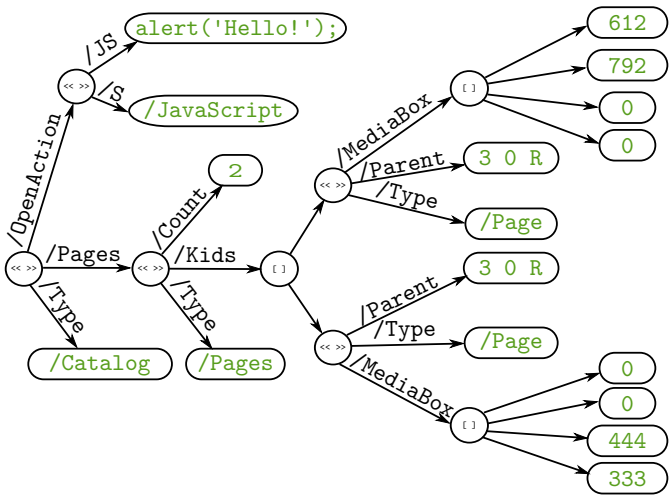


Figure 4.4: Structural tree of the PDF file depicted in Fig. 4.3. Dictionaries are illustrated using the symbol '<< >>', arrays using '[ ]'. Cycles were omitted for simplicity.

```
/Pages/Kids
/OpenAction/Contents
/StructTreeRoot/RoleMap
/Pages/Kids/Contents/Length
/OpenAction/D/Resources/ProcSet
/OpenAction/D
/Pages/Count
/PageLayout
```

Our investigation shows that these are the structural paths whose presence in a file is most indicative that the file is benign, or, alternatively, whose absence indicates that a file is malicious. For example, malicious files are not likely to contain metadata in order to minimize file size, they do not jump to a page in the document when it is opened and are not well-formed so they are missing paths such as `/Type` and `/Pages/Count`.

The following is a list of typical structural paths found in real-world malicious PDF files:

```
/AcroForm/XFA
/Names/JavaScript
/Names/EmbeddedFiles
/Names/JavaScript/Names
/Pages/Kids/Type
/StructTreeRoot
/OpenAction/Type
/OpenAction/S
/OpenAction/JS
```

`/OpenAction`

We see that malicious files tend to execute JavaScript stored within multiple different locations upon opening the document, and make use of Adobe XML Forms Architecture (XFA) forms as malicious code can also be launched from there.

4.3.2 SWF File Format

As a second example of a hierarchically structured file format we present SWF. The essential technical details of the SWF file format were described in Chapter 2. In the following we introduce its logical structure reusing the example file from Chapter 2, this time with values of tag fields colored green, depicted in Fig. 4.5.

```
[14:0]: SetBackgroundColor
  [14:0]: Header (Code: 9 Length: 3)
    [14:0]: TagAndLength : 579
  [B0:0]: BackgroundColor
    [16:0]: Red : 170
    [17:0]: Green : 187
    [18:0]: Blue : 204
[19:0]: ShowFrame
  [19:0]: Header (Code: 1 Length: 0)
    [19:0]: TagAndLength : 64
[1B:0]: SetBackgroundColor
  [1B:0]: Header (Code: 9 Length: 3)
    [1B:0]: TagAndLength : 579
  [E8:0]: BackgroundColor
    [1D:0]: Red : 17
    [1E:0]: Green : 34
    [1F:0]: Blue : 51
[20:0]: ShowFrame
  [20:0]: Header (Code: 1 Length: 0)
    [20:0]: TagAndLength : 64
[22:0]: End
  [22:0]: Header (Code: 0 Length: 0)
    [22:0]: TagAndLength : 0
```

Figure 4.5: Example SWF file. Every line starts with a hexadecimal number within square brackets denoting the offset, in bytes, of the corresponding tag field from the beginning of the file.

Fig. 4.6 illustrates a logical view of our example SWF file in which the file is structured as a tree. Every tag is represented by a tree node and is a direct descendant of the abstract root node. The edge from the root to the tag node is labeled by the tag type name, in our case `SetBackgroundColor`, `ShowFrame` and `End`. Descendants of tag nodes are its header and fields. Headers are connected with an edge simply labeled `Header`. The edges leading to the fields are labeled by their names, e.g., `BackgroundColor`. The

values of tags' fields are considered leaf nodes, e.g., the value of the *Red* field of the first *SetBackgroundColor* tag, 170.

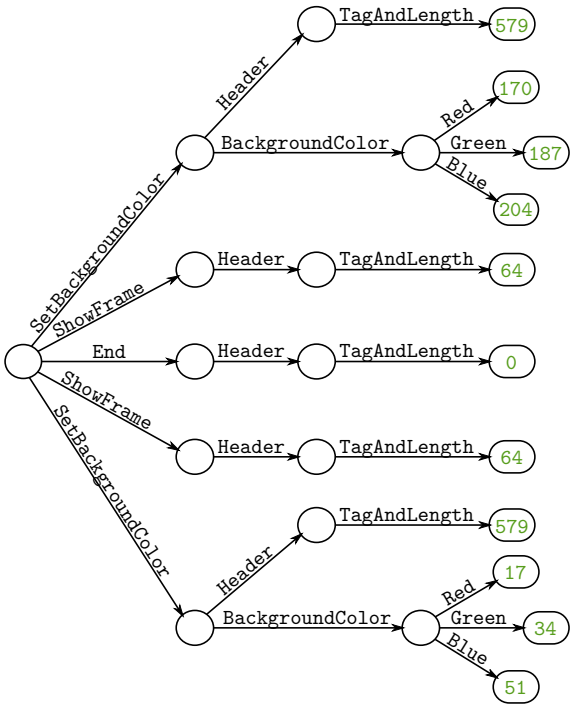


Figure 4.6: Logical structure of the SWF file illustrated in Fig. 4.5.

We define a *path* in the SWF structural tree, analogous to the PDF case, as a series of edges starting in the abstract root node and ending in a leaf node. For example, there is a path from the root node through the edges labeled `End`, `Header` and `TagAndLength` ending in the leaf node with the value 0. For better readability and consistency with PDF, we prepend the forward slash symbol '/' to every edge label when printing a path, hence the path in question prints as `/End/Header/TagAndLength`.

In the following we describe the system design of SL2013 and Hidost, respectively, and show how the two methods utilize logical structure for malware detection.

4.4 SL2013 System Design

The SL2013 system is limited to detecting malware in one file format, PDF. Its processing can roughly be divided into two steps, schematically shown in Fig. 4.7:

1. *Extraction of structural features.* As the basic pre-processing step, the content of a PDF file is parsed and converted into the special form, *bag-of-paths*, which characterizes the document structure in a well-defined way.

2. *Learning and classification.* The detection process is driven by examples of malicious and benign PDF files. In the *learning* step, a model is created from the data with known labels, i.e., training data. The model encodes the differences between malicious and benign data. In the *classification* step the model is applied to new, evaluation data, to classify it as malicious or benign.

The technical realization of these two fundamental tasks is presented below.

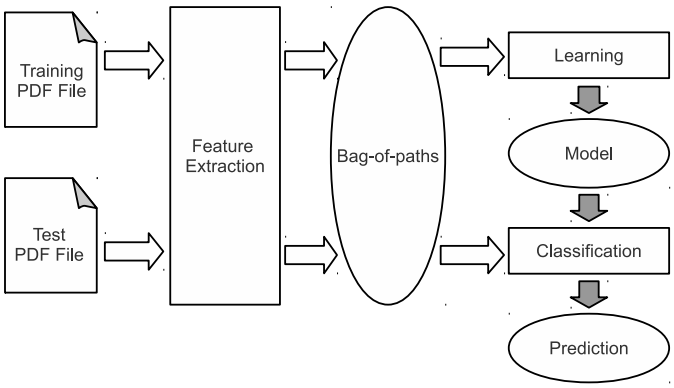


Figure 4.7: SL2013 system design.

Source code for SL2013 is not published separately but as part of the Hidost system in Section 4.6.

4.4.1 Feature Definition

A common approach to the design of data-driven security methods is to manually define a set of "intrinsic features" which are subsequently used for learning and classification. It was successfully applied for network intrusion detection [47, 55], botnet detection [35], detection of drive-by-downloads [14, 22, 23], and other related problems. The challenge in defining features for detection of malicious PDF files lies in the complex structure of the PDF format. We therefore depart from the knowledge-driven strategy mentioned above and consider a richer set of potential features that capture PDF's complexity. These features will be later automatically reduced to a smaller subset based on the available data.

The goal of structural analysis of PDF files is to recover all parent-child relations between their objects. The tree-like structure of PDF documents can be represented by a set of paths from the root to leaves, as shown in Fig. 4.8.

For notational convenience, we will use the forward slash symbol '/' as a delimiter between the names on a structural path[2]. The same path may occur multiple times in

[2]Technically, *null* is the only disallowed character in PDF names and hence, the only suitable delimiter for structural paths.

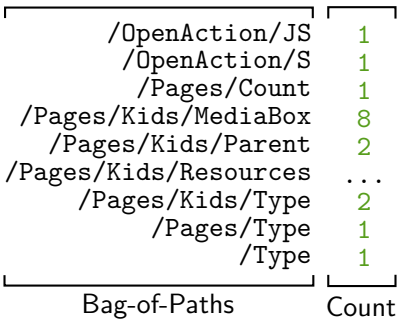


Figure 4.8: Structural paths corresponding to the example PDF file in Fig. 4.3 and their counts.

a document if it crosses an array with multiple elements. Empirical evidence indicates that the counts of specific paths in a document constitute a good measure of structural similarity between different documents. This motivates the choice of the set of structural paths as the intrinsic features of our system.

Due to widespread use of indirect references in PDF files, multiple structural paths may lead to the same object. Indirect references may even form circles in the structural graph, in which case the set of paths becomes infinite. In some semantic constructs of PDF, e.g., page trees, multiple paths to the same object are *required* to facilitate content rendering. Precise treatment of indirect references is only possible with directed graphs. Since the comparison of graphs is computationally difficult, we adhere to the tree-like view of the document structure and introduce additional heuristics in the following section which produce a finite set of structural paths while maintaining a reasonable semantic approximation of the existing relations.

Thus, the main operation to be performed in our feature extraction step is counting the structural paths in a document. Additional transformations, to be referred to as *embeddings*, can be applied to path counts. The binary embedding detects the presence of non-zero counts, the frequency embedding divides the counts over the total number of paths in a document, and the count embedding refers to the path count itself. All three embeddings were experimentally evaluated and the binary one was chosen over the other two for its slightly better detection performance.

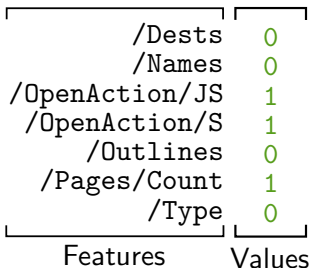


Figure 4.9: Feature vector resulting from the bag-of-paths illustrated in Fig. 4.8.

For our feature set, we selected only structural paths which occur in at least 1,000 files in our corpus (see Section 4.5.1 for a detailed description of the data used in our experimental evaluation). This reduces the number of features, i.e., structural paths, in our laboratory experiments from over 9 million to 6,087. Fig. 4.9 illustrates the binary embedding on a 7-dimensional feature set. We did not use class information for the selection of "discriminative features" as it was done, e.g., in Zozzle [23]. Such manual pre-selection of features introduces an artificial bias to a specific dataset and provides an attacker with an easy opportunity to evade the classifier by adding features from the opposite class to his malicious examples.

4.4.2 Extraction of PDF Document Structure

Extraction of structural features defined in Section 4.4.1 must meet the following requirements:

R1: All paths must be extracted with their exact counts.

R2: The extraction algorithm must be deterministic, i.e., for two PDF files with the same logical structure it must produce the same set of paths.

R3: The choice among multiple paths to a given object should be semantically the most meaningful one with respect to the PDF Reference.

As the first step in the extraction process, the file is parsed using the PDF parser Poppler, version 0.14.3. Its key advantages are the robust treatment of various encodings used in PDF and the reliable extraction of objects from compressed streams. In principle, other robust PDF parsers would be suitable for extraction of structural paths as well. Our choice of Poppler was motivated by its open-source nature. The parser maintains an internal representation of the document and provides access to all PDF objects.

Conceptually, path extraction amounts to a recursive enumeration of leafs in the document structure, starting from the root node, i.e., *Catalog*. The extracted paths are inserted into a suitable data structure, e.g., a hash table or a map, and their counts are accumulated. However, several refinements must be introduced to this general algorithm to ensure that it terminates and that the above requirements are met.

The requirement **R1** is naturally satisfied by the recursive nature of our feature extraction. Since our recursion terminates only if a leaf node is encountered, the algorithm is guaranteed to never underestimate the count of a particular path. However, an overestimation of the path count may occur if there is a cycle in the structural graph leading to infinite recursion. To prevent it, the requirement **R3** must be enforced.

The enforcement of requirements **R2** and **R3** is tightly coupled and ultimately relies on the intelligent treatment of indirect references. Obviously, one cannot always dereference them, as this may result in an infinite recursion. One cannot also avoid their

de-referencing, as the algorithm would hardly ever move beyond the root node. Hence, a consistent strategy for selective de-referencing must be implemented.

In our extraction algorithm, we approach these issues by maintaining a breadth-first search (BFS) order in the enumeration of leaf objects. This strategy assumes that the shortest path to a given leaf is semantically the most meaningful. For example, this observation intuitively holds for various cases when circular relations arise from explicit upward references by means of the *Parent* entry in a dictionary, as demonstrated by our example in Fig. 4.3. We find the path `/Pages` to preserve more semantics than `/Pages/Kids/Parent` which refers to the same object. In Section 4.6.2 we present a refinement of this rough heuristic for selecting semantically more informative paths introduced for HIDOST.

Two further technical details are essential for the implementation of BFS traversal. It is important to keep track of all objects visited during the traversal and backtrack whenever an object was seen before in order to break graph cycles. It is also necessary to sort all entries in a dictionary in some fixed order before descending to the node's children. Since no specific ordering of dictionary fields is required by the PDF Reference, such ordering must be artificially enforced in order to satisfy the requirement **R2**.

4.4.3 Learning and Classification

Once the counts or other embeddings over the set of structural paths are extracted, various learning algorithms can be applied to create a model from the given training data and use this model to classify unknown examples. For an overview of suitable algorithms, the reader may refer to any standard textbook on machine learning, e.g., [9, 36], or use any entry-level machine learning toolbox, such as SHOGUN[3] or WEKA[4]. It is beyond the scope of this manuscript to provide a comprehensive experimental evidence as to which machine learning method is most suitable for detection of malicious PDF files using structural paths. We have chosen two specific algorithms, decision trees and Support Vector Machines, for subjective reasons presented in the following section along with a high-level description of the respective method.

Decision Trees

The decision tree is a popular classification technique in which predictions are made in a sequence of single-attribute tests. Each test either assigns a certain class to an example or invokes further tests. Decision trees have arisen from the field of operational decision making and are especially attractive for security applications, as they provide a clear justification for specific decisions – a feature appreciated by security administrators. An example of a decision tree classifying whether one should take an umbrella when leaving home is shown in Fig. 4.10.

[3] SHOGUN – `http://www.shogun-toolbox.org/`.
[4] WEKA – `http://www.cs.waikato.ac.nz/ml/weka/`.

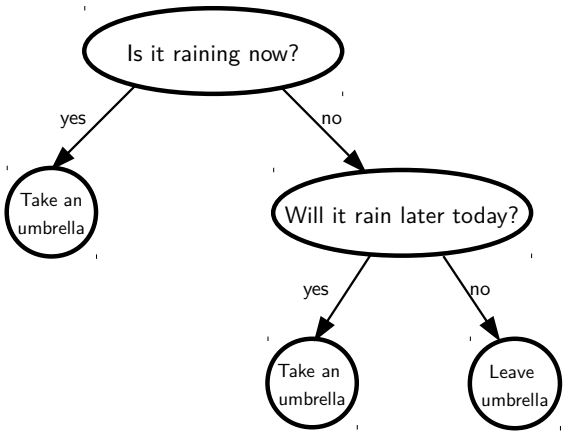


Figure 4.10: An example decision tree.

The goal of *automatic decision tree inference* is to build a decision tree from labeled training data. Several classical algorithms exist for decision tree inference, e.g., CART [11], RIPPER [19], C4.5 [73]. We have chosen a modern decision tree inference implementation C5.0[5] version 2.07 which provides a number of useful features, e.g., automatic cross-validation and class weighting. It can also transform decision trees into rule sets which facilitate the visual inspection of large decision trees.

Support Vector Machines

The Support Vector Machine (SVM) [21] is another popular machine learning algorithm. Its main geometric idea, illustrated in Fig. 4.11, is to fit a hyperplane to data so that the margin M between examples of 2 classes is maximized. In the case of a linear decision function, it is represented by the hyperplane's weight vector w and the threshold ρ which are directly used to assign labels y to unknown examples x:

$$y(x) = w^\top x - \rho$$

Nonlinear decision functions are achieved by applying a nonlinear transformation to input data which maps it into a feature space with special properties, the so-called *Reproducing Kernel Hilbert Space (RKHS)*. The elegance of SVM consists in the fact that such transformations can be done implicitly, by choosing an appropriate nonlinear *kernel function* $k(x_1, x_2)$ which compares two examples x_1 and x_2. The solution α to the *dual* SVM learning problem, equivalent to the primal solution w, can be used for a nonlinear decision function expressed as a comparison of an unknown example x with selected examples x_i in the training data, the so-called "support vectors" (circles with black outlines

[5] C5.0 – `http://www.rulequest.com/see5-info.html`.

in Fig. 4.11):

$$y(x) = \sum_{x_i \in SV} \alpha_i y_i k(x, x_i) - \rho$$

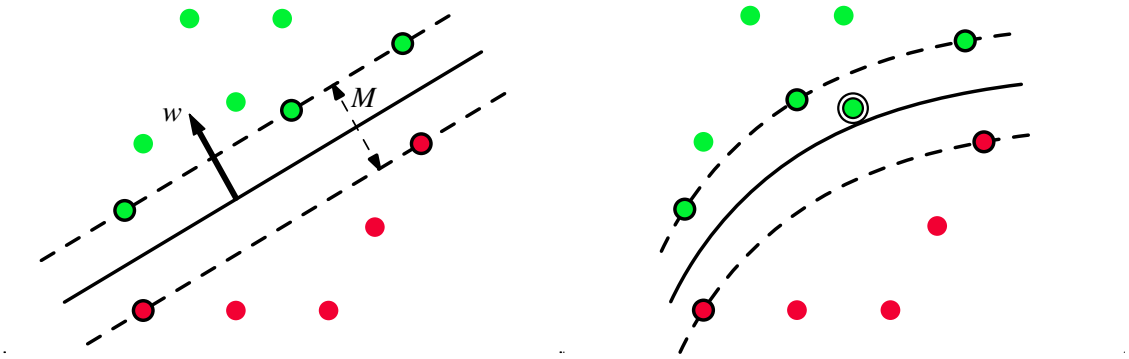


Figure 4.11: Linear and nonlinear SVM. Decision boundary (vector w) between the two classes depicted with two colors is shown with a solid and the margins M with dashed lines; points with a single outline are support vectors; point with a double outline is a test point that falls within the margin.

Efficient implementations of SVM learning are available in various machine learning packages. In our experiments, we used a well-known stand-alone SVM implementation LibSVM[6] version 3.12.

4.5 SL2013 Experimental Evaluation

The goal of experiments presented in this section is to measure the effectiveness and throughput of SL2013 under both laboratory and deployment conditions. In addition, we compare the classification performance of our method to other existing PDF malware detection methods: PJScan and the established antivirus tools. Our evaluation is based on an extensive dataset comprising around 660,000 real-world PDF files.

4.5.1 Experimental Datasets

Dataset quality is essential for the inference of meaningful models as well as for a compelling evaluation of any data-driven approach. For our evaluation, we have obtained a total of 658,763 benign and malicious PDF files (around 595 GB), the largest data corpus hitherto used in evaluation of malware detectors for non-executable files. Our data was collected from Google and VirusTotal and organized into the following 6 datasets:

[6] LibSVM – `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

D1: VirusTotal malicious, containing 38,207 (1.4 GB) files obtained from VirusTotal during 18 days, between the 5th and 22nd of March 2012, labeled by at least 5 antivirus tools as malicious,

D2: VirusTotal benign, comprising 79,200 (75 GB) files obtained from VirusTotal during the same time period, labeled by all antivirus tools as benign,

D3: VirusTotal malicious new, with 11,409 (527 MB) files obtained from VirusTotal 2 months later, during 33 days, between the 23rd of May and 24th of June 2012, labeled by at least 5 antivirus tools as malicious,

D4: Google benign, containing 90,834 (73 GB) files obtained from 1,000 Google searches for PDF files with a date range as the sole query argument; the 1,000 date ranges covering 2,000 days, between the 5th of February 2007 and the 25th of July 2012,

D5: Operational malicious, with 32,526 (2.7 GB) files obtained from VirusTotal during 14 weeks, between the 16th of July and 21st of October 2012, labeled by at least 5 antivirus tools as malicious,

D6: Operational benign, comprising 407,037 (443 GB) PDF files obtained from VirusTotal during 14 weeks, between the 16th of July and 21st of October 2012, which were labeled by all antivirus tools as benign.

The VirusTotal data comprises PDF files used by people from all over the world, which brings us as close to real-world private PDF data as possible. In fact, the benign VirusTotal data is even biased towards being malicious, as users usually upload files they find suspicious. The dataset obtained by Google searches removes the bias towards maliciousness in benign data and attempts to capture the features of regular benign PDF files found on the Internet.

Note that we consider a VirusTotal file to be malicious only if it was labeled as such by at least 5 antivirus tools. Files labeled malicious by 1 to 4 antivirus tools are discarded from the experiments because there is little confidence in their correct labeling, as we verified empirically. Given the lack of reliable ground truth for these files, we assume a zero false positive rate for antivirus tools and cannot directly compare them to SL2013 with respect to the false positive rate.

4.5.2 Experimental Protocol

Two types of experiments were devised to evaluate the detection performance of SL2013: *laboratory* and *operational* experiments. The three laboratory experiments operate on static data, captured in a specific point in time, where training and classification data are intermixed using *5-fold cross-validation*[7]:

[7] 5-fold cross-validation works as follows: we randomly split our data into 5 disjoint subsets, each containing one fifth of malicious and one fifth of benign files. Learning and classification are repeated five

- The CrossVal5 experiment is designed to evaluate the overall effectiveness of our method on known malicious and average benign data. To this end, we use the VirusTotal malicious dataset (**D1**) and the Google benign dataset (**D4**).

- The Suspicious experiment is designed to evaluate the effectiveness of our method on PDF files that ordinary users do not trust. For this experiment, we use VirusTotal malicious data (**D1**) and VirusTotal benign data (**D2**). The classification task in this experiment is harder than in the CrossVal5 experiment since its benign data is biased towards malicious.

- The WithJS experiment is designed to enable the comparison of our method to PJScan. For this experiment, a subset of the datasets used for the CrossVal5 experiment (**D1** and **D4**) was used comprising only those files that contain directly embedded JavaScript which PJScan can extract; i.e., 30,157 malicious and 906 benign files.

In contrast, in the two operational experiments, classification is performed on files which did not exist at all at the time of training, i.e., files obtained at a later time:

- The Novel experiment evaluates our method on novel malicious threats when trained on an *outdated* training set. For this experiment, we apply the models learned in the CrossVal5 experiment to 2 months younger VirusTotal malicious data **D3**. Novel benign data was not evaluated as its observed change in this timespan was not significant.

- The 10Weeks experiment is designed to evaluate the classification performance of our method in a simulated real-world, day-to-day practical operational setup and directly compare it to the results achieved by antivirus tools in the same time period. This experiment is performed on data from the Operational benign (**D6**) and malicious (**D5**) datasets, containing files gathered during 14 weeks. It is run in a sliding window fashion, with one evaluation every week, for 10 weeks starting from week 5. In every iteration, feature selection is performed on files gathered in the past 4 weeks and a new model is learned from scratch on these files using the selected features. This model is then used to classify the files obtained during the current week.

  For example, the data obtained during weeks 1 to 4 is used to learn a model which classifies data gathered in week 5, weeks 2 to 5 are used for week 6, etc. This *periodic retraining* approach is typical for large-scale machine learning deployments in applications where the underlying data distribution changes continually, e.g., in intrusion detection.

times, each time selecting a different combination of four subsets for learning and the remaining one for classification. This experimental protocol enables us to *classify every file exactly once* while ensuring that *no file processed in the classification phase was used in the learning phase for the respective model*.

Source code for the reproduction of this experiment is published as part of HIDOST source code release introduced in Section 4.7.

Note that, in practice, there are no fundamental difficulties for periodic retraining of learning models as new labeled data becomes available. Models deployed at end-user systems can be updated in a similar way to signature updates in conventional antivirus systems. As we show in Section 4.5.4, SVMs are efficient enough to allow periodic retraining of models from scratch. Our decision tree learning algorithm implementation, however, lacked the required computational performance and was not evaluated in this experiment.

4.5.3 Experimental Results

Both the decision tree learning algorithm and the SVM were evaluated in our laboratory experiments. For the SVM, we selected the radial basis function (RBF) kernel with $\gamma = 0.0025$ and a *cost* parameter $C = 12$, based on an empirical pre-evaluation. We define PDF files with malware to be the positive class in our experiments.

The CROSSVAL5 experiment

Table 4.1 shows detection results for both classification algorithms in the CROSSVAL5 experiment. The top part shows the confusion matrices (the number of positive and negative files with true and false classifications) obtained by aggregating the results of all five cross-validation runs. The bottom part shows other performance indicators: the *true* and *false positive rates* and the overall *detection accuracy*.

	Decision tree	SVM
True Positives	38,102	38,163
False Positives	51	10
True Negatives	90,783	90,824
False Negatives	105	44
True Positive Rate	0.9973	0.9989
False Positive Rate	5.6×10^{-4}	1.1×10^{-4}
Detection Accuracy	0.9988	0.9996

Table 4.1: Aggregated results of the CROSSVAL5 experiment.

The CROSSVAL5 experiment evaluates the overall performance of our method under laboratory conditions. As Table 4.1 shows, although the SVM slightly outperforms the decision tree learning algorithm, both algorithms show excellent classification performance. Very high detection accuracy (over 99.8 %) was achieved, while false positives rate remained in the low promille range (less than 0.06 %).

The Suspicious experiment

Results for the Suspicious experiment are shown in Table 4.2. The classification performance of both algorithms indeed decreases when applied to this harder, suspicious data than in the baseline CrossVal5 experiment, but the difference is marginal.

	Decision tree	SVM
True Positives	38,118	38,163
False Positives	68	27
True Negatives	79,132	79,173
False Negatives	89	44
True Positive Rate	0.9977	0.9989
False Positive Rate	8.6×10^{-4}	3.4×10^{-4}
Detection Accuracy	0.9987	0.9994

Table 4.2: Aggregated results of the Suspicious experiment.

The WithJS experiment

Table 4.3 compares the results of both of our algorithms to PJScan. Since PJScan performs anomaly detection; i.e., it learns using only examples of one class (malicious), during its training the benign files are discarded.

	Decision tree	SVM	PJScan
True Positives	30,130	30,149	21,695
False Positives	14	12	1
True Negatives	892	894	905
False Negatives	27	8	8,462
True Positive Rate	0.9991	0.9997	0.7194
False Positive Rate	0.0154	0.0132	0.0011
Detection Accuracy	0.9986	0.9993	0.7275

Table 4.3: Aggregated results of the WithJS experiment.

Overall, our method performs somewhat worse than in the CrossVal5 experiment due to the strong class imbalance in the training dataset. Still, it significantly outperforms PJScan, a method specialized for detecting malicious JavaScript, even on a dataset carefully chosen for it. PJScan's high false negative rate in this experiment can be attributed to its failure to process JavaScript code from XFA forms and code loaded at runtime. The

reported effectiveness of PJScan is consistent with the results presented in Section 3.6.3 and its high false negative rate was elaborated in Section 3.7.

Comparison to Prior Work

Table 4.4 compares SL2013 to prior work in terms of detection performance under laboratory conditions. A direct comparison was not performed due to a lack of available source code. Instead, we show the results reported in the original publications, measured on datasets of various sizes and structure.

	SL2013	MDScan	ShellOS	Malware Slayer
Number of malicious samples	38,207	197	405	11,157
Number of benign samples	90,834	2,000	179	9,989
True positive rate	0.9988	0.8934	0.8024	0.9955
False positive rate	0.0001	0	N/A	0.0251

Table 4.4: Comparison of SL2013 to prior work.

Our laboratory results represent a significant improvement over prior work. The closest reported result, achieved by Malware Slayer, attains a similar true positive rate but at the cost of more than 200-fold increase of false positive rate. Both dynamic methods, MDScan and ShellOS, generate no false positives but detect only 80 % to 90 % of malware; it should be also noted that these results have been measured on an order of magnitude smaller datasets.

The Novel experiment

Table 4.5 shows the results for the Novel experiment[8].

	Decision tree	**SVM**
True Positives	10,681	10,870
False Negatives	728	539
True Positive Rate	0.9361	0.9527

Table 4.5: Aggregated results of the Novel experiment

Even when our learning algorithms are trained on a 2 to 3 month-old dataset, they still achieve respectable 93 % (decision tree) or 95 % (SVM) true positive rates, indicating

[8]Note that some information, such as the true negative count, is missing for this experiment because it was only applied to malicious data, since changes in benign performance were negligible.

a slow pace of malware evolution when viewed through the prism of structural features used in SL2013. The finding that the document structure of malicious PDF files exhibits a small change in the wild even 2 months after the model was trained suggests that there is little evolutionary pressure on malware to adapt in this regard. Our detector is not commercially deployed and therefore malware can safely ignore it. Instead, the pressure is asserted by antivirus tools to produce novel binary code, e.g., using polymorphism, that will evade signature-based detection.

The 10Weeks experiment

Fig. 4.12 shows the comparison of our method to the best VirusTotal antivirus tool[9] in the 10Weeks experiment in terms of true positive rate (TPR). The best antivirus tool achieves an overall TPR of 92.81 %, significantly better than 87.14 % achieved by our method. However, our method consistently outperforms the best antivirus tool in 7 out of 10 weeks, and there is a draw in week 8. The performance degradation of our method in weeks 14 and, most notably, 12 has motivated a further investigation which uncovered a curious trend in VirusTotal submissions, discussed below.

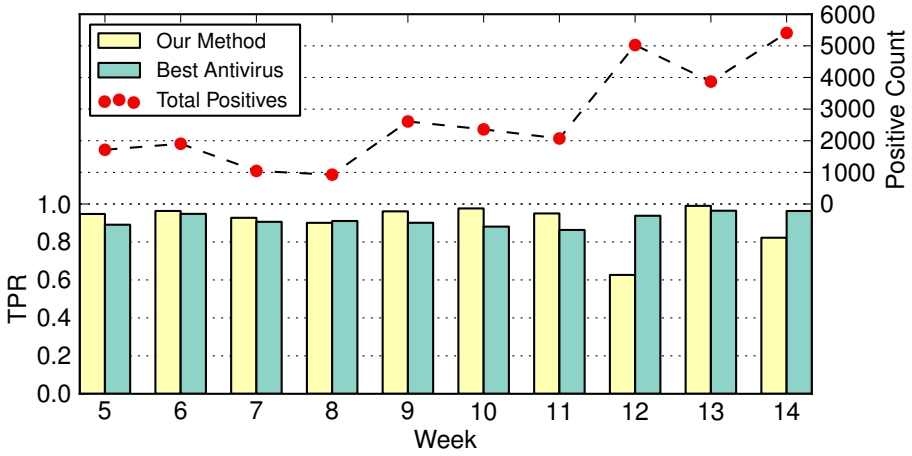


Figure 4.12: Comparison of SL2013 to the best antivirus in terms of the true positive rate achieved in the 10Weeks experiment.

The top half of Fig. 4.12 shows the number of PDF submissions to VirusTotal detected by at least 5 antivirus tools, i.e., the number of positives, per week. In the first week of October 2012, week 12, VirusTotal saw the positive submission count increase by approx. 150 %, from around 2,000 to around 5,000. This elevated level of submissions has persisted to the end of the experiment. A closer inspection of this week's data has revealed that there are two very large groups of submissions, one comprising 1,842

[9]The name of the best antivirus tool is not disclosed as there are several within a 5 % margin of the TPR achieved in this experiment, and their rankings change in time.

and the other 2,595 files. In each group, the files differ byte-wise from each other, but have identical PDF structure, i.e., every file in a group corresponds to the same bag-of-paths. Furthermore, there is a high similarity between the two groups. The bag-of-paths of the smaller group consists of 99 structural paths, all of which are present in the other group as well. The only difference between the groups are additional 11 structural paths in the bigger group. Files with the same bag-of-paths were also submitted later, but not before this week. This finding strongly suggests that the submissions of week 12 and later stem in great part from a single source and were generated using the same tools, e.g., *fuzzing techniques* or a malware toolkit.

The cause for the false negative rate of 37 % in week 12 is that all files in the smaller group (1,842) were mislabeled as benign by our method. The prediction is the same for all files because they all translate into identical bags-of-paths, i.e., the same data point. A wrong classification of one data point in this case led to a false negative rate of more than 1/3 because the test data was heavily skewed by one source producing very similar submissions. In about 20 cases, these files were also missed by *all* antivirus tools.

The data point corresponding to the bag-of-paths of the smaller group of files is located on the wrong side of the SVM decision boundary, although very close to it. The addition of further 11 structural paths positioned the data point corresponding to the bag-of-paths of the larger group significantly over the decision boundary into the positive class. The reason for this lies in the fact that 8 out of 11 added structural paths are strong indicators of maliciousness in the learned SVM model[10]. In the weeks following week 12, these examples showed up in the learning stage and were correctly classified.

The performance drop in week 14 comes from a very high number of submitted files (over 900) which our parser could not open. This anomaly was not further investigated as these are either *a*) malformed PDFs that pose no threat to the PDF renderer application but are, nevertheless, scanned by ignorant antivirus tools, or *b*) parser bugs, in which case it suffices to update or fix the parser or employ a completely different one, as our method is parser-agnostic.

The overall false positive rate of SL2013 in this experiment is 0.0655 %, as in laboratory tests. The antivirus tools do not have false positives by definition of our experiments, as the "undecided" files (the ones between 1 and 4 detections) are filtered.

4.5.4 Throughput

High throughput is an important consideration when dealing with large volumes of PDF data, as is the case with VirusTotal, big companies or governments. Our system was designed to handle such loads and utilize the parallel processing capabilities of modern computers. We have measured the time it takes to perform feature extraction, learning

[10] A linear SVM was trained for the purpose of this feature interpretation which exhibits the same misclassification problem for the smaller group of files. The evaluation was performed by computing and sorting weights of all features.

and classification for datasets **D1**, **D3**, **D2** and **D4** with both decision trees and SVMs. The measurements were made on a quad-core CPU with 8 GB of RAM and a 7,200 RPM SATA hard disk with the memory caches previously cleared.

Feature extraction is the most time-consuming operation, as it requires loading all PDF files from the hard drive. It was performed using 7 parallel processes. In total, 7315 s were spent on feature extraction for the 150 GB of data in the above-mentioned datasets, of which 313 s were spent on malicious and 7002 s on benign files, yielding a throughput of 168 Mbit/s.

Numbers for learning and classification differ for decision trees and SVMs. They are presented in Table 4.6.

	Learning	Classification
Decision tree	391 s	52 s
SVM	83 s	54 s

Table 4.6: Time required for learning and classification in the CrossVal5 experiment.

Since each of the 5 cross-validation runs trains on 80 % of the training data, we divided the total sum of execution times for all runs by four to obtain an estimate of how long training would take for the entire dataset. The classification time is a simple sum of 5 individual classifications, as each deals with 20 % of testing data. Note that executing cross-validation runs in parallel increases performance linearly with the number of processes. Even though decision trees are significantly slower than the SVM, the overall running time is dominated by feature extraction.

The total time required for feature extraction, learning and classification using SVMs in the CrossVal5 experiment with the datasets **D1** and **D4** of 74.4 GB was 3602 s, yielding the total throughput of around 169 Mbit/s and a mean processing time of 28 ms per file. The high performance numbers are achieved by static detection and parallel execution. In contrast, dynamic methods such as MDScan (slightly less than 3000 ms per malicious file, 1500 ms per benign file on average) and ShellOS (on average 5.46 s per file for analysis, plus additional 2 s (benign) to 20 s (malicious, non-ROP) for buffer extraction) require orders of magnitude more time. The only other fully static method with published throughput metrics, PJScan, takes 23 ms per file, because it only extracts a well-defined, limited subset of the entire PDF file.

Having introduced SL2013 and presented a detailed evaluation of its detection performance and throughput, in the following section we describe Hidost, the improvements it brings over SL2013 and its extension to other hierarchically structured file formats.

4.6 HIDOST System Design

HIDOST has been designed as a malware detection system capable of learning to discriminate between malicious and benign files based on their logical structure. Due to the semantic heterogeneity of various file formats it is hard to imagine a single format to act as a "common denominator" for all conceivable hierarchically structured file formats. Yet our design clearly separates format-specific processing steps from the detection methodology. As a result, HIDOST, currently tested on PDF and SWF formats, can be extended to other formats by implementing the format-specific components without rebuilding its general framework. The proposed method was implemented as a research prototype and its feature extraction subsystem was published as open-source software[11]. The published code comprises a toolset for feature extraction from PDF (implemented in C++) and SWF files (implemented in Python and Java) and can extract features for HIDOST as well as SL2013. Experiment reproduction code is published separately, as described in Section 4.7.

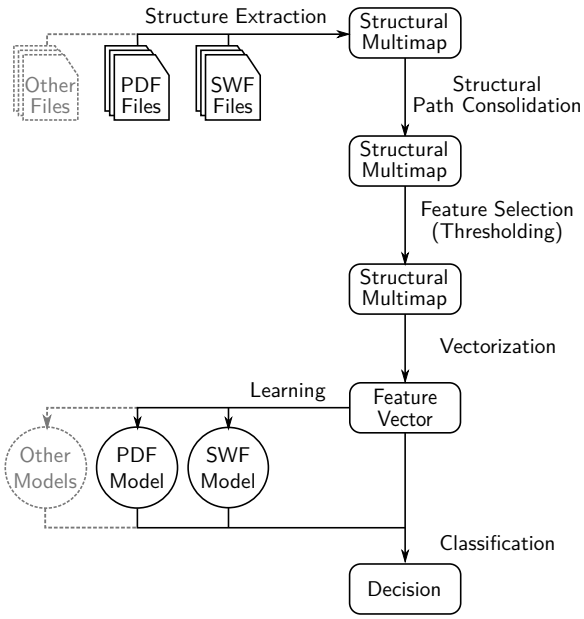


Figure 4.13: Hidost system design.

The system design of HIDOST is illustrated in Fig. 4.13. It comprises 6 main stages:

1. **structure extraction**, transforms structural features of different formats into a common data structure – structural multimap – representing paths in the structural hierarchy,

[11] HIDOST – `https://github.com/srndic/hidost`.

2. **structural path consolidation**, groups similar structural paths into a single feature,

3. **feature selection**, finds the minimum set of features required for a successful machine learning application,

4. **vectorization**, transforms structural multimaps into numeric vectors processed by machine learning methods,

5. **learning**, generates a discriminative model of malicious and benign files based on their properties encoded in feature vectors,

6. **classification**, makes a decision whether a previously unseen sample is malicious or benign based on the learned model.

In the following subsections, the main stages of our approach are presented in detail.

4.6.1 Logical Structure Extraction

The first step of our method transforms files into a more abstract representation, their logical structure. This step is essential to our approach because it achieves two key goals: *a*) use of logical structure for discrimination between malicious and benign files; and *b*) generalization to multiple file formats.

In this manuscript the meaning of the term "logical structure" has a subtle but important difference when used in describing SL2013 versus HIDOST. In SL2013, it was limited to expressing the hierarchical structure of files, enumerating the set of paths from the root node to every leaf, while in HIDOST it encapsulates *both the hierarchy and the content*, i.e., information contained in leaf nodes. This may be strings, numbers or other basic data types that occur in PDF, SWF and other hierarchically structured file formats. As we show experimentally in Section 4.7, it is the incorporation of this additional information that enabled HIDOST to effectively extend to SWF.

One suitable representation for this broader definition of logical structure of hierarchically structured file formats that accomodates both structure and content is a *structural multimap*. A multimap is a generalization of the common map data structure, also known as a dictionary or associative array. While maps provide a mapping between a key and a corresponding value, multimaps map a key to a *set of values*. A structural multimap is a multimap that maps every structural path of a structural tree to the set of all leafs that lie on the given path. In map terminology, the structural paths represent the *keys* and sets of all leafs that a path maps to represent the *values* of the map. It is clear that structural multimaps encode more information about files than the bag-of-paths used in SL2013 because they contain not only the structure, i.e., the paths themselves, but also the content, i.e., values stored in leaf nodes.

Keys	Values
/OpenAction/JS:	alert('Hello!');
/OpenAction/S:	/JavaScript
/Pages/Count:	2
/Pages/Kids/MediaBox:	0|0|612|792|0|0|333|444
/Pages/Kids/Parent:	3 0 R|3 0 R
/Pages/Kids/Resources:	...
/Pages/Kids/Type:	/Page|/Page
/Pages/Type:	/Pages
/Type:	/Catalog

Figure 4.14: A complete structural multimap of the PDF file depicted in Fig. 4.3. This type of structural multimap is not used in Hidost but rather illustrated as an instructive example.

An example of a structural multimap is shown in Fig. 4.14. Multiple values for the same key are delimited using vertical bar symbols '|'. Comparing this structural multimap to SL2013 bag-of-paths from Fig. 4.8, we see the breadth of information that is discarded completely by SL2013. Hidost uses a slightly simplified form of structural multimaps presented later in this section.

The reason why logical file structure requires a multimap instead of a map is that multiple leafs may be reachable by the same structural path. Recall that in PDF files this occurs when a path contains an array with more than one element. For example, the path `/Pages/Kids/MediaBox` contains two arrays and reaches 8 leafs. In case of SWF files, apart from arrays, multiple tags of the same type cause multiple leafs to lie in the same path. Implementation of structure extraction for PDF and SWF is presented in the following two sections.

PDF

The PDF logical structure is organized as a directed rooted cyclic graph. To transform it into a structural multimap, it is first necessary to reduce the graph to a directed rooted tree instead. This step was extensively documented as part of SL2013 in Section 4.4.1 and is mostly the same in Hidost. The only difference lies in the treatment of leaf nodes. While they were previously discarded, they are essential in the new system.

Therefore, when the breadth-first search is performed on the file structure graph, in the new approach we insert all pairs (p, l), where p is a structural path and l is a leaf node located on the path p, into the resulting structural multimap. This ensures that the complete content is retained as part of logical structure.

To be more precise, Hidost does not use full structural multimaps but a simplification thereof. This simplification concerns the treatment of non-numeric data types, i.e., all types except integers, real numbers and booleans. Strings, PDF names and other non-numeric types, all convertible to strings, are replaced with a constant value, 1. The

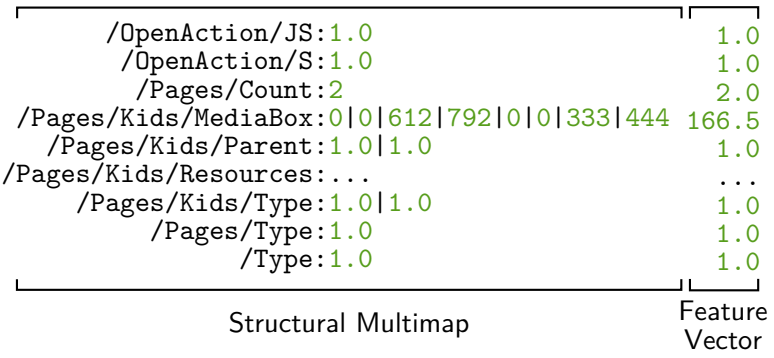


Figure 4.15: Structural multimap and feature vector of the PDF file depicted in Fig. 4.3.

resulting structural multimap is shown in Fig. 4.15. Comparing it to the one shown earlier (Fig. 4.14), we clearly see the binarization of non-numeric values. This choice of treatment is a trade-off between fully discarding non-numeric values and performing their extensive evaluation.

Different approaches to the treatment of string-like data types have been proposed in related work, from static, e.g., embedding strings in metric spaces [45, 80], characterizing them with simple properties such as their length, entropy, or distribution of keywords [56, 85] or testing them for valid CPU instructions [1], to dynamic, e.g., CPU emulation of strings [71, 104] or their execution [53, 88, 100]. However, these approaches either conflict with the desired static system design (dynamic evaluation), lower computational performance (string embedding and testing for CPU instructions), or are easily evadable (simple string properties).

The positive effect of the use of binarized non-numeric values compared to their complete omission was experimentally confirmed. The effects of migrating from purely binary features used by SL2013 to numerical in HIDOST are evaluated in Section 4.7.3.

SWF

The SWF logical structure is more straightforward to extract than PDF as it naturally does not have cycles or ambiguities. The approach implemented in HIDOST begins by employing the `ConsoleDumper` class of the SWFRETools toolkit to parse the SWF file and produce its textual representation, such as the one in Fig. 4.5. Parsing the output generated by this tool suffices to extract the SWF logical structure. Every line of the textual output that starts with zero or more spaces followed by an opening square bracket '[' contains a tag or field name that represents one edge in the structural tree. The distance of this edge from the root node is encoded as the number of spaces before the bracket, divided by 2. Consequently, a line with a bracket preceded by no spaces signals the beginning of a new tag. By keeping track of the most recent edges parsed at each level in the hierarchy it is possible to reconstruct the entire path to the edge in the current line. Finally, if the edge name is succeeded by a colon then this edge represents a tag field and

the remainder of the line encodes that field's value. The pair (p, v), where p is the path at the current line and v the parsed value, is then inserted into the structural multimap. Strings and other non-numeric types are binarized in the same way as with PDF.

```
                /End/Header/TagAndLength:0          0.0
  /SetBackg...Color/Header/TagAndLength:579|579 579.0
 /SetBackg...Color/BackgroundColor/Blue:204|51  127.5
/SetBackg...Color/BackgroundColor/Green:187|34  110.5
  /SetBackg...Color/BackgroundColor/Red:170|17   93.5
        /ShowFrame/Header/TagAndLength:64|64     64.0
```

Structural Multimap | Feature Vector

Figure 4.16: Structural multimap and feature vector of the SWF file depicted in Fig. 2.2.

The structural multimap corresponding to the SWFRETools output of Fig. 4.5 is illustrated on the left-hand side of Fig. 4.16. The following section describes the second processing step of our method, structural path consolidation.

4.6.2 Structural Path Consolidation

The syntactic richness and flexibility of many file formats enables semantically equivalent logical structures to be expressed in syntactically different ways. For example, the path `/Threads/F` represents the first element of a doubly linked list describing a PDF article. The path `/Threads/F/N` points to the next element in the list and is a different path than the previous one, but they both point to an object with the same semantics. Such syntactic polymorphism decreases the detection accuracy because related objects translate to different features. Furthermore, coupled with our thresholding feature selection strategy, it provides an opportunity for attackers to hide their content in spurious locations that are not used in common files. To address this problem, we have developed a heuristic technique for consolidation of structural paths which reduces polymorphic paths to somewhat consistent representation. This technique can be best exemplified for the PDF format.

We have observed that many paths common among PDF files exhibit structural similarities to other paths. In fact, we were able to identify through manual inspection groups of paths similar to each other, yet not completely identical. Paths in these groups exhibited a similarity in one of two ways:

1. all were identical except for exactly one customizable path element,

2. all shared a common repetitive subpath.

Importantly, however, all paths in a group of similar paths refer to objects with the same purpose, i.e., the same *semantics*. For example, `/Pages/Kids/Resources` shares a

Table 4.7: PDF structural path consolidation rules.

	Search regular expression	Substitute regular expression
1.	`/Resources/(ExtGState|ColorSpace|Pattern|Shading|XObject|Font|Properties|Para)/[^/]+`	`/Resources/\1/Name`
2.	`^Pages/(Kids/|Parent/)*(Kids$\|Kids/\|Parent/\|Parent$)`	`Pages/`
3.	`/(Kids/|Parent/)*(Kids$\|Kids/\|Parent/\|Parent$)`	`/`
4.	`(Prev/|Next/|First/|Last/)+`	`<empty string>`
5.	`^Names/(Dests|AP|JavaScript|Pages|Templates|IDS|URLS|EmbeddedFiles|AlternatePresentations|Renditions)/(Kids/|Parent/)*Names`	`Names/\1/Names`
6.	`^StructTreeRoot/IDTree/(Kids/)*Names`	`StructTreeRoot/IDTree/Names`
7.	`^(StructTreeRoot/ParentTree|PageLabels)/(Kids/|Parent/)+(Nums|Limits)`	`\1/\3`
8.	`^StructTreeRoot/ParentTree/Nums/(K/|P/)+`	`StructTreeRoot/ParentTree/Nums/`
9.	`^(StructTreeRoot|Outlines/SE)/(RoleMap|ClassMap)/[^/]+`	`\1/\2/Name`
10.	`^(StructTreeRoot|Outlines/SE)/(K/|P/)*`	`\1/`
11.	`^(Extensions|Dests)/[^/]+`	`\1/Name`
12.	`Font/([^/]+)/CharProcs/[^/]+`	`Font/\1/CharProcs/Name`
13.	`^(AcroForm/(Fields/|C0/)?DR/)(ExtGState|ColorSpace|Pattern|Shading|XObject|Font|Properties)/[^/]+`	`\1\3/Name`
14.	`/AP/(D|N)/[^/]+`	`/AP/\1/Name`
15.	`Threads/F/(V/|N/)*`	`Threads/F`
16.	`^(StructTreeRoot|Outlines/SE)/Info/[^/]+`	`\1/Info/Name`
17.	`ColorSpace/([^/]+)/Colorants/[^/]+`	`ColorSpace/\1/Colorants/Name`
18.	`ColorSpace/Colorants/[^/]+`	`ColorSpace/Colorants/Name`
19.	`Collection/Schema/[^/]+`	`Collection/Schema/Name`

common repetitive subpath, `/Kids`, with `/Pages/Kids/Kids/Resources`, but both refer to PDF dictionaries that have the same purpose – to provide a name for resources required to render a page of a PDF file. Semantically, it is irrelevant which path the structure extraction algorithm took before it visited a page's resource dictionary – all resource dictionaries have the same semantics.

Likewise, path `/Pages/Kids/Resources/Font/F1` normally has a multitude of similar paths, e.g., `/Pages/Kids/Resources/Font/F42`, that only differ in the last path element which the PDF Reference mandates to be user-defined, but both refer to *Font* dictionaries describing fonts for use in the PDF file. Again, regardless of the concrete name a specific PDF writer gives to a font dictionary, all font dictionaries are semantically equivalent.

The existence of such groups of semantically equivalent paths questions the utility of SL2013 feature definition which treats every individual path as a distinct feature. We find it more instructive to preserve the semantics of paths by consolidating equivalent ones to a single feature. This idea, called *structural path consolidation* (SPC), was implemented in HIDOST and experimentally evaluated in Section 4.7.3. It is a preprocessing step for structure extraction that facilitates generalization to other file formats that may have complicated logical structures, e.g., with many redundant elements.

The implementation of SPC is based on the substitution of key path components using regular expressions. Repetitive subpaths are completely removed from the path. For example, both paths indicated above as examples with a common repetitive subpath would be consolidated into the path `/Pages/Resources`, removing the repetitive subpath `/Kids`. On the other hand, user-defined path components are *anonymized*, i.e., replaced with the placeholder path component `/Name`. For instance, both paths from the example above with user-defined font names would be consolidated into the path `/Pages/Kids/Resources/Name` (if the rule concerning repetitive paths was not applied beforehand, of course). Table 4.7 lists SPC rules employed in HIDOST for PDF, implemented using the BOOST.REGEX library. Every rule comprises two regular expressions: one is used to search for a pattern to replace (left) and the other to determine the replacement string (right).

A single consolidation rule can be applied to multiple groups of semantically equivalent paths. For example, `/Pages/Kids/Resources` and `/Pages/Kids/MediaBox` can be consolidated by the same rule, but the resulting paths `/Pages/Resources` and `/Pages/MediaBox` still belong to separate groups and are, therefore, two different features.

SPC rules in Table 4.7 are the result of an empirical investigation of structural paths occurring in our dataset, with the aim of minimizing their total count after transformation. Our analysis was focused on rules that capture generic branches of the PDF document structure instead of dataset-specific artifacts, e.g.:

- anonymized items such as resources (1 and 13),
- entries of various name trees, such as global (5) and structure tree (6),

- dictionaries for mapping custom names into other objects (9),
- color space items (17 and 18),
- or names of embedded files (19).

Other rules are used to flatten hierarchies (2, 3, 7, 8 and 10) and convert linked lists to shallow sets (4) in order to create a generic, unified view of their elements, all on the same level.

Due to the relatively shallow SWF logical file structure and no support for user-defined path elements, only two SPC rules were compiled for this format, listed in Table 4.8, both for handling repetitive subpaths.

Table 4.8: SWF structural path consolidation rules.

Search regex	Substitute regex
`(DefineSprite/ControlTags/){2,}`	`DefineSprite/ControlTags/`
`(Symbol/Name/){2,}`	`Symbol/Name/`

No attempt was made to compile a complete list of SPC rules – there are further ones to discover and extend the list. Especially for PDF, there is ample opportunity for further anonymization and flattening of hierarchies such as name trees and number trees not covered in our rules. However, even this limited set of rules provides the following crucial benefits compared to SL2013:

- **Reduced attack surface.** Without SPC, every distinct path with an occurrence count above a threshold constitutes a feature. An attacker striving to evade detection may in that case perform a *hiding attack* by concealing a malicious payload at a custom path different from any in the feature set. For example, a path to a font with a long, randomly generated name is highly unlikely to have been encountered before. A malicious payload inserted there would be invisible to the detector that does not have this particular path in its feature set. In case of SWF, where user-defined paths are disallowed, payloads may be concealed in very deep hierarchies, not encountered in "normal" files. PDF is affected by this issue as well.

  These *hiding attacks* are cheap to implement and the primary security vulnerability of SL2013. This avenue for evasion is closed in HIDOST with the use of consolidated paths.

- **Limited feature set drift in time.** In real-world machine learning applications, the problem at hand often changes in time. This is especially true in security applications, where defenders are forced to adapt to unpredictable changes in attacks.

This problem is known in machine learning as *concept drift* [108] and has recently started to attract interest in security literature [40].

The continual change in data renders classifiers ever more outdated as time elapses since their training. Therefore, the need arises for regular updates to the learning model in the form of periodic classifier retraining. With data-dependent features such as in this work, it is advisable to perform feature selection anew before every retraining in order to better adapt to concept drift. Periodic feature selection causes the obsolescence of existing features and addition of new ones between two retraining periods. We refer to changes in the feature set caused by periodic feature selection as *feature set drift*. PDF is more susceptible to feature set drift than SWF due to its flexible structure. As Section 4.7.3 shows, SPC is effectively used to reduce feature set drift in HIDOST.

- **Feature space dimensionality reduction.** Finally, SPC has a tremendous impact on the total number of features. Feature space dimensionality directly affects the running time and memory requirements of learning algorithms. In our PDF experiments with periodic retraining, the average feature set size was reduced by an impressive 88 %, from 10,412.5 to 1,237.4 features per training.

  However, there are limits in the effectiveness of SPC against manually crafted paths. Because it has no notion of a semantically valid path, SPC cannot handle unforeseen cases, e.g., arbitrary names in the *Catalog* dictionary. To tackle this final "blind spot" in the coverage of the PDF logical structure, a whitelisting approach would be required with a complete and up-to-date representation – a model of the entire structure – which is out of scope of this work.

The reduced attack surface and limited feature set drift represent an important contribution to the operational security of HIDOST as a machine-learning-based detector. Massively reduced feature count enables its application on even larger datasets. Together, the described improvements bring HIDOST a big step towards applicability in a real-world, operational environment as an accurate, reliable and secure malicious file detector.

4.6.3 Feature Selection

Despite the reduction of syntactic polymorphism via structural path consolidation, there may still exist paths that occur very infrequently in the observed data. Using such paths to build discriminative models increases the dimensionality of the input space without improving classification accuracy. Therefore, feature selection has to be carried out to limit the impact of rare features. Before presenting the specific feature selection techniques, we discuss the reasons why rare features occur in the two formats studied in detail in the paper.

The SWF file format specification [95] strictly defines the names of all tags and all their fields, prohibiting customization. Therefore, HIDOST's feature set for SWF theo-

retically comprises every structural path defined by the SWF specification. However, in practice, no effort has been made to enumerate all paths in the SWF logical structure. Instead, the feature set comprises all paths *observed* in the training dataset, a total of 3,177.

In contrast, the PDF file format specification [67] allows the use of user-defined names in any PDF dictionary, essentially enabling an unlimited number of different paths. Our data indicates that this PDF feature is widely used in practice as we have observed over 9 million distinct PDF structural paths. However, $^2/_3$ of these paths do not occur in more than one file. These and other paths that occur in a small percentage of the dataset are considered anomalous. Therefore, SL2013 solved this problem by selecting paths which occur in more than a fixed number of training files, i.e., 1,000, for its feature set. This threshold controls the trade-off between detection accuracy (more paths) and model simplicity (less paths) and may be freely adjusted.

After SPC and before every training in our periodic retraining experimental protocol, we applied the same occurrence threshold, i.e., 1,000 files, which corresponds to around 1 % of the training set size.

4.6.4 Vectorization

Structural multimaps are a suitable representation of PDF and SWF logical file structure but they cannot be directly used by machine learning algorithms. They first need to be transformed into feature vectors, i.e., points in the *feature space* $\mathbb{R}^N$, in a process called vectorization.

During vectorization, structural multimaps are first replaced by structural maps – ordinary map data structures that map a structural path to a corresponding single numeric value. To this end, every set of values corresponding to one structural path in the multimap is reduced to its median. We selected median as a more robust statistic than mean (here we use the term *robust* in the statistical sense, denoting that median provides a better characterization of the set of values in the presence of outliers, and not that it provides any robustness against adversarial evasion). The only exception to this rule is that sets of values in SWF structural multimaps consisting primarily of booleans are reduced to their means, not medians. Mean preserves more information about booleans than median, which can only be 0, $^1/_2$ or 1, and there is no possibility of outliers. This exception is not implemented for PDF as its logical structure has relatively few boolean values.

Structural maps are transformed into feature vectors $\mathbf{f} \in \mathbb{R}^N$ by reserving a separate dimension for every structural path and using values from structural maps as values of corresponding dimensions. The mapping of individual structural paths to dimensions of feature vectors is defined before feature extraction, during feature selection, and is applied uniformly to all structural maps prior to both training and classification. Consequently, a specific structural path corresponds to the same dimension in every feature vector, enabling the learning algorithms to make sense of feature vectors.

The ordered collection of all features used by a learning algorithm is its *feature set*.

Figs. 4.15 and 4.16 illustrate feature vectors extracted from a PDF and a SWF structural multimap, respectively. They show a simple case when the feature set is identical to the set of keys in the structural multimap and every value of the feature vector is assigned. In practice, however, files usually do not contain all structural paths present in the feature set and the corresponding values in the feature vectors are set to zero. In summary, a feature vector corresponding to a structural multimap m is a point $\mathbf{f} = f_1, f_2, \ldots, f_N$ in feature space $\mathbb{R}^N$ with specific values defined as

$$f_i = \begin{cases} median(m[p_i]), & p_i \in m \\ 0, & \text{otherwise} \end{cases} \quad \forall i \in \overline{1,N} \tag{4.1}$$

Here, p_i denotes the i^{th} path in the feature set and $m[p_i]$ denotes the value in a multimap m associated with that path.

4.6.5 Learning and Classification

The stages presented so far transform samples, i.e., files, into feature vectors suitable as input for machine learning algorithms. The choice of a concrete machine learning classifier depends on a multitude of parameters, e.g., dataset size, feature space dimensionality, available computational resources, robustness against adversarial attacks, etc., and classifiers are tailored for different uses. The published implementation of Hidost therefore does not comprise learning and classification subsystems. Instead, its output can be used with any classifier. For experiments presented in this paper, the Random Forest implementation of the open-source scikit-learn Python machine learning library [68], version 0.15.0b2, was utilized. This part of Hidost was published separately, as part of experimental reproduction code, as detailed in the following section.

Random Forest [10] is an ensemble classifier. It is trained by growing a forest of decision trees using the CART methodology. Each of the t_{RF} trees is grown on its own fixed-size random subset of training data drawn with replacement. At every branching of a tree during training, the feature providing the optimal split is selected from a random subset comprising f_{RF} features not previously used for this tree. During classification, the decision of every tree is counted as one vote and the overall outcome is the class with the majority of votes. Random Forests are known for their excellent generalization ability and robustness against data noise. For the experimental evaluation, forest size was set to 200 trees and all other parameters to their scikit-learn defaults.

4.7 Hidost Experimental Evaluation

An extensive experimental evaluation was performed as part of this work to assess the detection performance of Hidost. The entire source code and datasets needed to reproduce

all experiments and plots as part of the evaluation of HIDOST are available online[12].

4.7.1 Experimental Datasets

Experiments were run on two datasets, one for each file format. Both were collected from VIRUSTOTAL. VIRUSTOTAL enables us to compare HIDOST's detection performance to that of deployed antivirus engines. As in the experimental evaluation of SL2013, we consider files with at least 5 detections as malicious and those with 0 detections as benign. The remaining files, labeled by 1 to 4 antivirus engines as malicious, are discarded from the experiments because of the high uncertainty of their true class label.

For our **PDF dataset** we directly take datasets **D5** and **D6** from the evaluation of SL2013, described in Section 4.5.1. As a reminder, they comprise 439,563 (446 GB) files, 407,037 (443 GB) benign and 32,567 (2.7 GB) malicious, collected during 14 weeks, between July 16 and October 21, 2012. Using the same dataset enables a direct comparison with SL2013.

The **SWF dataset** was collected between August 1, 2013 and March 8, 2014 and comprises 40,816 (14.2 GB) files, 38,326 (14.1 GB) benign and 2,490 (190 MB) malicious. The VIRUSTOTAL SWF data had a benign-to-malicious ratio of around 52:1 during the collection period, therefore a random subsampling of benign data was performed to approximately match the ratio with that of PDF data.

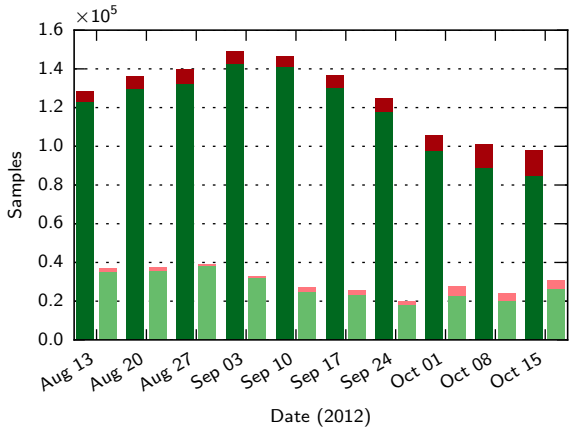
4.7.2 Experimental Protocol

Our experimental protocol has the following two main goals: *a*) to evaluate the performance of HIDOST under realistic conditions; and *b*) to enable the comparison of HIDOST's detection performance on PDF to its predecessor, SL2013. To this end, we adopt the experimental protocol of the 10WEEKS experiment from Section 4.5.2.

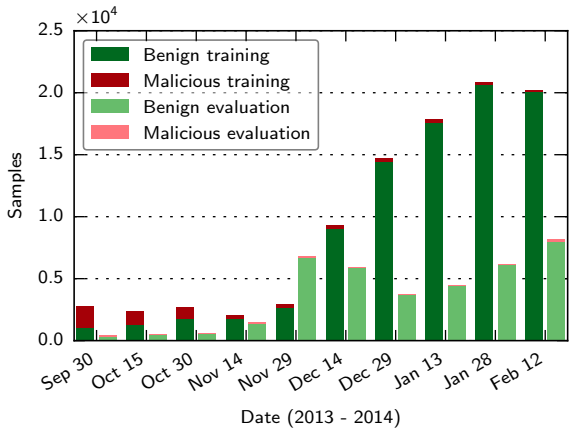
We partitioned our PDF dataset in the same manner as in the original experiment for SL2013. The SWF dataset was partitioned analogously. The time period in which the files were collected was divided into 14 smaller, consecutive time periods. For PDF, every period was exactly one week long, for SWF 15 days, the last one 25 days. Every time period was assigned a bucket, and every file was put into one of the buckets, according to the time period when it was first seen. Then the sliding window approach was applied, joining 4 consecutive buckets into a training dataset and using the following bucket as the corresponding evaluation dataset, resulting in 10 data partitions for periodic retraining. Before every retraining, i.e., every week for PDF, every 15 days for SWF, all 4 steps of feature extraction (i.e., structure extraction, SPC, feature selection and vectorization) were applied to the training dataset.

Our datasets are illustrated in Fig. 4.17. Every retraining event is labeled by the date of training, i.e., the day that marks the beginning of data collection for the evaluation period.

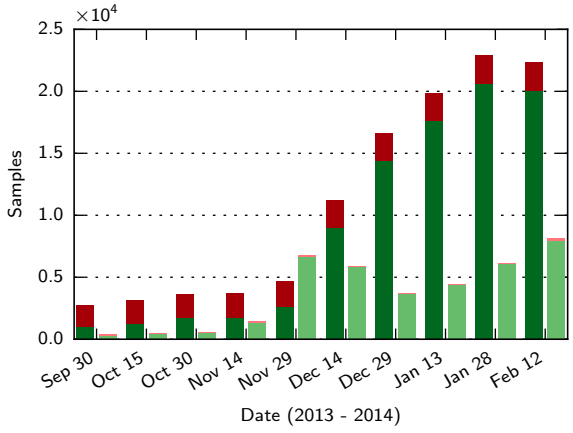
[12] HIDOST-REPRODUCTION – `https://github.com/srndic/hidost-reproduction`.



(a) PDF dataset.



(b) SWF-Normal dataset. Malicious training samples older than 4 time periods are discarded.



(c) SWF-KeepMal dataset. Malicious training samples are kept indefinitely.

Figure 4.17: Experimental datasets.

While the benign-malicious class ratio for PDF is approximately equal throughout all time periods, the distribution of malicious and benign SWF files in time is highly skewed. Around 70 % of malicious SWF files in the SWF-NORMAL dataset were collected before the first evaluation period, while less than 10 % of benign SWF files occur before the fifth evaluation period. The result is a high class imbalance in most training and evaluation datasets.

To quantify the effect of high class imbalance on detection performance, we generated another data partitioning just for SWF data. Labeled SWF-KEEPMAL and illustrated in Fig. 4.17c, it has the property that malicious training samples older than 4 periods are not discarded. Instead, they are used for training in all subsequent periods. By discarding old benign samples and keeping malicious ones throughout the experiment, the class imbalance in training datasets is significantly reduced.

4.7.3 Experimental Results

Experimental results for different methods operating on PDF and SWF data are illustrated in Figs. 4.18 and 4.19, respectively. The methods are compared using 4 performance indicators typical for classification tasks: true positive rate(TPR), false positive rate (FPR), accuracy and area under receiver operating characteristic (AUROC). AUROC, similar to the area under the precision-recall curve, is a good detection performance indicator for both balanced and unbalanced datasets. Due to the stochastic nature of the algorithm, mean values of 10 independent runs are plotted for all Random Forest experiments. The variance of these experiments was omitted from the plots due to its very low value. SL2013 employs a Support Vector Machine (SVM) classifier, a deterministic algorithm, therefore its results are obtained from a single experimental run.

Fig. 4.18 shows results for different variants of HIDOST and SL2013 on PDF data. For better comparison, we have evaluated 2 variants of SL2013: one using its original classifier, SVM, the other with a Random Forest instead. HIDOST is shown with both binary and numerical features. These two variants of HIDOST are also shown in Fig. 4.19 on SWF datasets SWF-NORMAL and SWF-KEEPMAL.

Classification Performance

Fig. 4.18 shows a direct comparison of HIDOST to SL2013 on the PDF dataset. The Random Forest variant of SL2013 was introduced to enable the comparison of the two methods' feature sets and classifiers independently. It can be seen that SL2013 results, especially the true positive rate on October 1, can be promptly improved by using a Random Forest instead of an SVM on the same binary unconsolidated features. On the other hand, the expected classification performance indicated by AUROC is effectively equal for all methods, including the SVM. The maximum difference between any two methods in AUROC in a given time period is a mere 0.006 and all methods have an

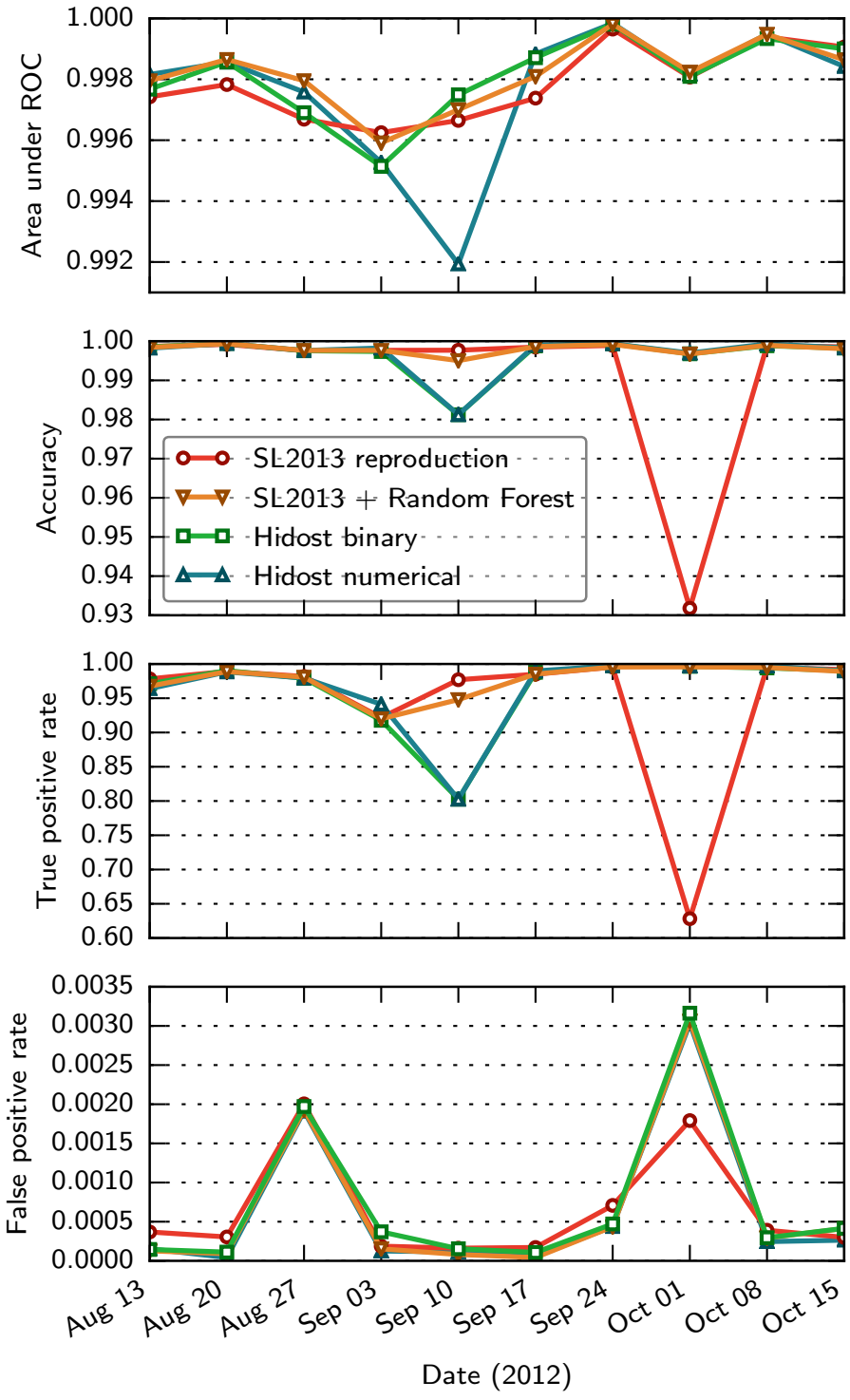


Figure 4.18: Results on PDF data. Performance of SL2013 with an SVM and Random Forest classifier compared to Hidost with binary and numerical features.

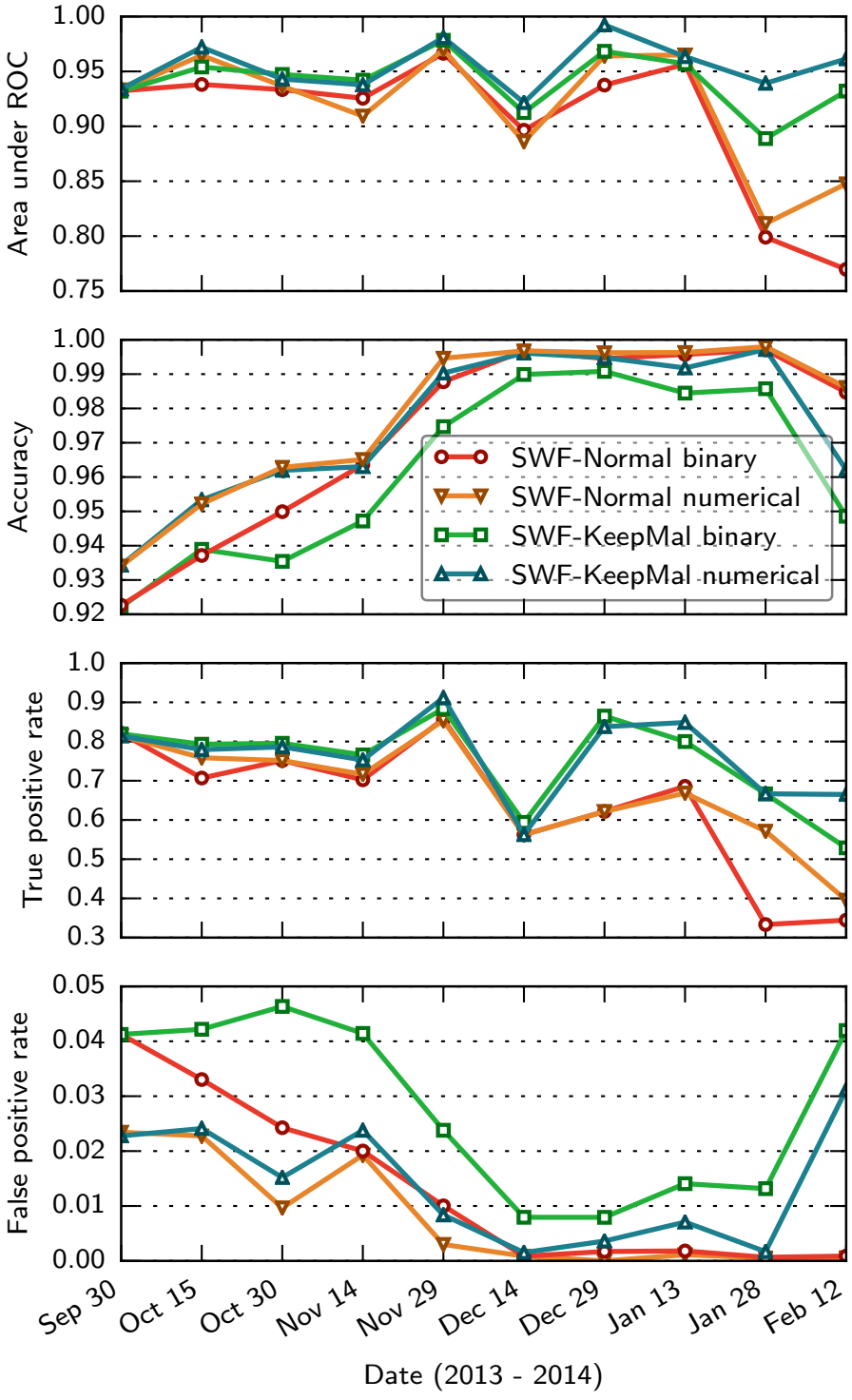


Figure 4.19: Results on SWF data. Performance of HIDOST with both binary and numerical features on two SWF datasets, SWF-NORMAL and SWF-KEEPMAL.

AUROC above 0.99 in every period. It can thus be concluded that HIDOST achieves the excellent classification performance of its predecessor, SL2013, on PDF data.

HIDOST's performance on SWF data is not on par with its success on PDF. Although the mean detection accuracy lies above 95 %, as seen in Fig. 4.19, accuracy is not a meaningful performance indicator due to the large class imbalance of 15:1 in favor of benign samples in the SWF dataset. The class imbalance is even greater in the datasets of individual time periods, shown in Figs. 4.17b and 4.17c, especially in the case of SWF-NORMAL.

The effect of class imbalance is clearly reflected in the results. Applied on SWF-KEEPMAL, where malicious training samples are accumulated over time, HIDOST has an overall much higher AUROC than on SWF-NORMAL, where malware is discarded after 4 periods. The true positive rate on SWF-KEEPMAL in the early stages, when the classes are more balanced, is 5 % to 10 % higher than on SWF-NORMAL. Starting from December 29, after a sharp rise in class imbalance, the advantage jumps to around 20 % – a tremendous improvement. Access to more malicious training data also increased the false positive rate, but the increase for the variant with numerical features remained within bounds, except for the last time period. These findings clearly show HIDOST's potential for further improvement of detection performance, given a greater availability of malicious SWF training data. However, as the SWF dataset only comprises 2,490 malicious samples, it is impossible to accurately quantify the potential for improvement.

Comparison to Antivirus Engines

To get an estimate of HIDOST's detection performance under day-to-day, realistic operational conditions, it is necessary to put it into a wider perspective. A direct comparison with antivirus engines provides such a reality check. We compare the detectors in terms of their true positive count, i.e., the number of malicious samples they have correctly labeled. By definition of our ground truth, samples labeled malicious by at most 4 antivirus engines are filtered out. Therefore, the antivirus engines have no false positives and cannot be compared in that respect.

Figs. 4.20 and 4.21 show the results achieved by HIDOST (average of 10 experimental runs) and antivirus engines deployed by VIRUSTOTAL on both PDF and SWF files. Antivirus detection results were collected *after* the experiments were over, and not immediately after each new file was submitted to VIRUSTOTAL. This provided antivirus engines with the opportunity to update their detection mechanisms in the meantime and correctly detect any file resubmitted between its initial submission and the time when the detection results were collected.

Nevertheless, HIDOST ranks among the best overall. Its PDF detection rate is unsurpassed, and even the SWF true positive count, comparatively much worse than PDF "on paper", ranks among the best when compared to established products under realistic conditions.

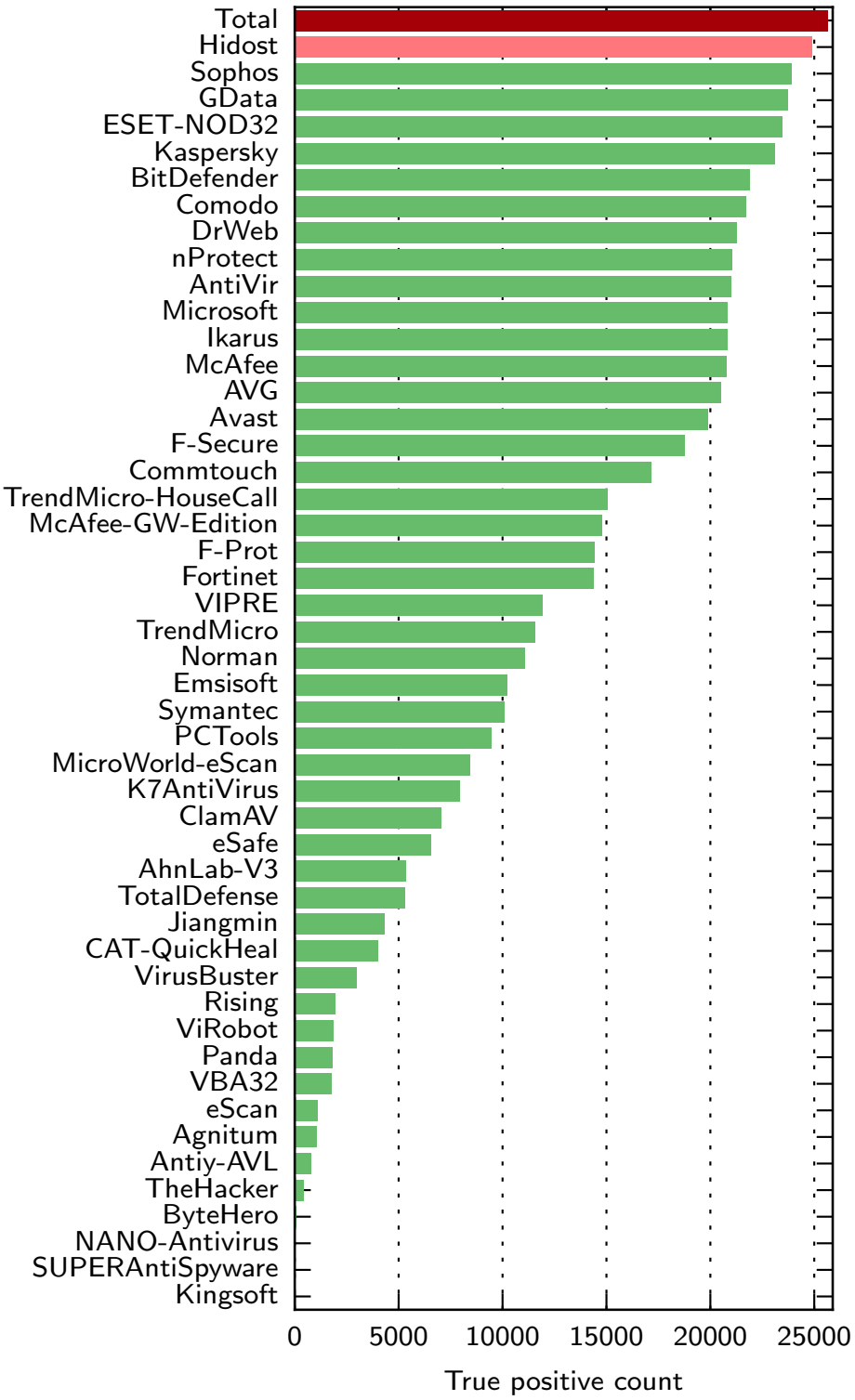


Figure 4.20: Comparison of HIDOST to antivirus engines on PDF data.

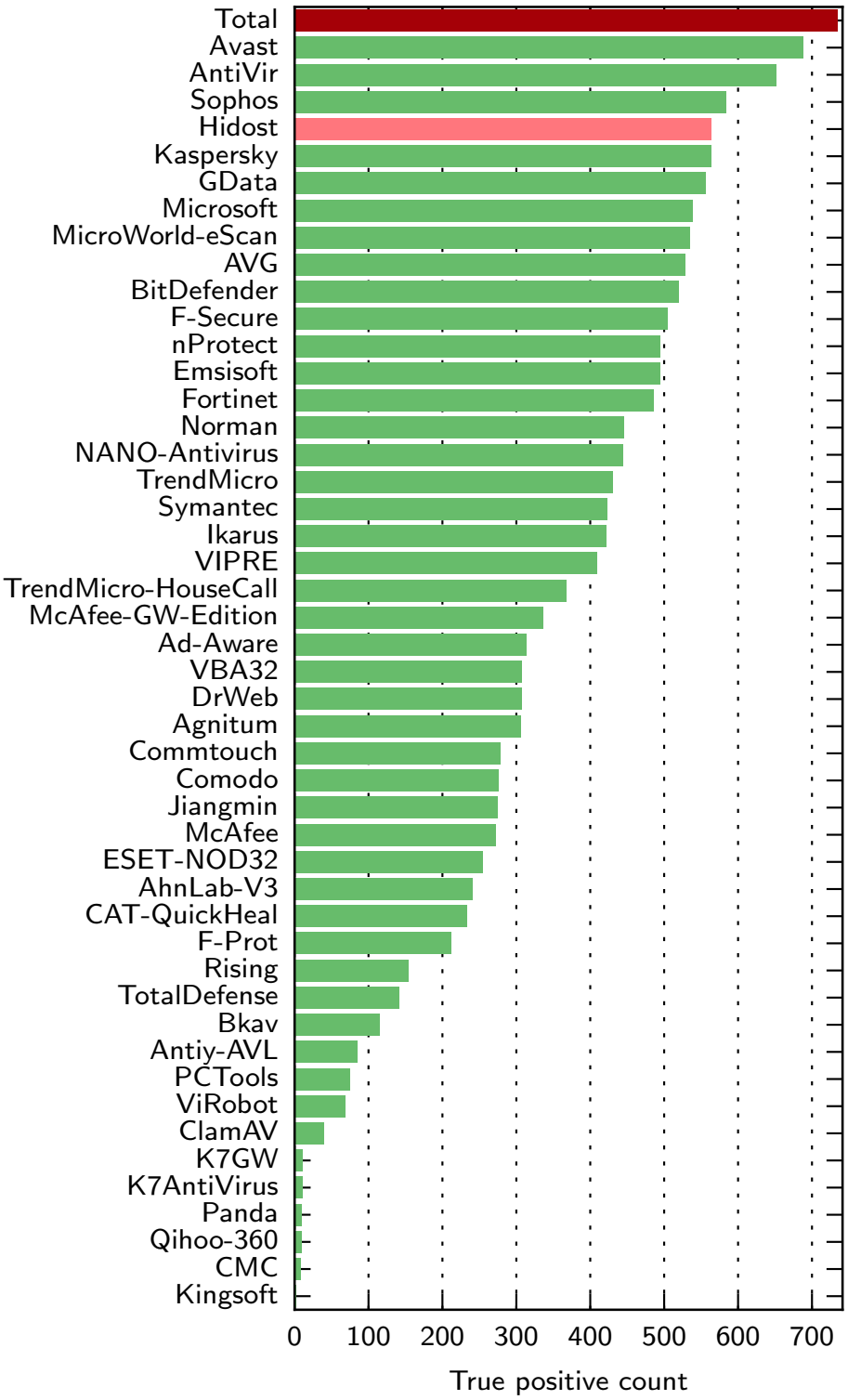


Figure 4.21: Comparison of Hidost to antivirus engines on SWF (right) data.

Effects of Structural Path Consolidation

SPC is one of the main novel features in HIDOST with respect to SL2013, therefore an evaluation of its effects on the performance of the system is only fitting. Fig. 4.18 shows that SPC has no effect on detection performance, neither positive nor negative. Results of SL2013 (no SPC) are virtually identical on PDF to those of HIDOST (with SPC) when the same Random Forest classifier is utilized. Effects on SWF are negligible because its rigid logical structure disallows user-defined paths, resulting in minimal necessity for SPC.

However, SPC has a strong positive effect on feature set drift. Fig. 4.22 illustrates feature set drift in our experiment with periodic retraining and periodic feature selection on PDF data. It can be observed that in the first half of our 10-week experiment the feature set had been expanded with up to 9 % of new features per week, while in the second half many features were found obsolete and were removed from the feature set. Feature removal is especially high in week 8, when almost a fifth of all features from the previous week were deleted when SPC was not used. On the other hand, when utilizing SPC, the overlap between feature sets of consecutive weeks was well above 90 % throughout the entire experiment. Overall, the introduction of SPC in HIDOST reduced feature set drift by around 50 %.

Effects of Merging Content with Structure

Another novelty introduced in HIDOST is the use of numerical instead of binary features, reflecting the transition from learning on pure structure to learning on structure coupled with content. Here we evaluate the impact of content on detection performance.

On PDF, the difference is insignificant, as shown in Fig. 4.18. On the other hand, the effect on SWF is largely positive. As shown in Fig. 4.19, numerical features consistently outperformed binary ones on both SWF datasets. They had the highest impact on false positive rate, reducing it by as much as 50 %. TPR and AUROC also showed a significant overall improvement.

The cause of the discrepancy between results for the two file formats may lie in the nature of attacks against them. Malicious PDF files often use features uncommon in benign files, i.e., their *structure* is different, while malicious SWF files mostly base their attacks on different values, i.e., *content*, at specific paths, although these paths are also common among benign files. While binary features suffice to describe the bare logical structure, the added expressive power of numerical features enables the characterization and, consequently, improved detection of both structure- and content-based attacks.

4.8 Discussion

In our description of SL2013 and HIDOST we strive to depict their *modus operandi* concisely, yet with enough detail to enable a thorough understanding. Having equipped the

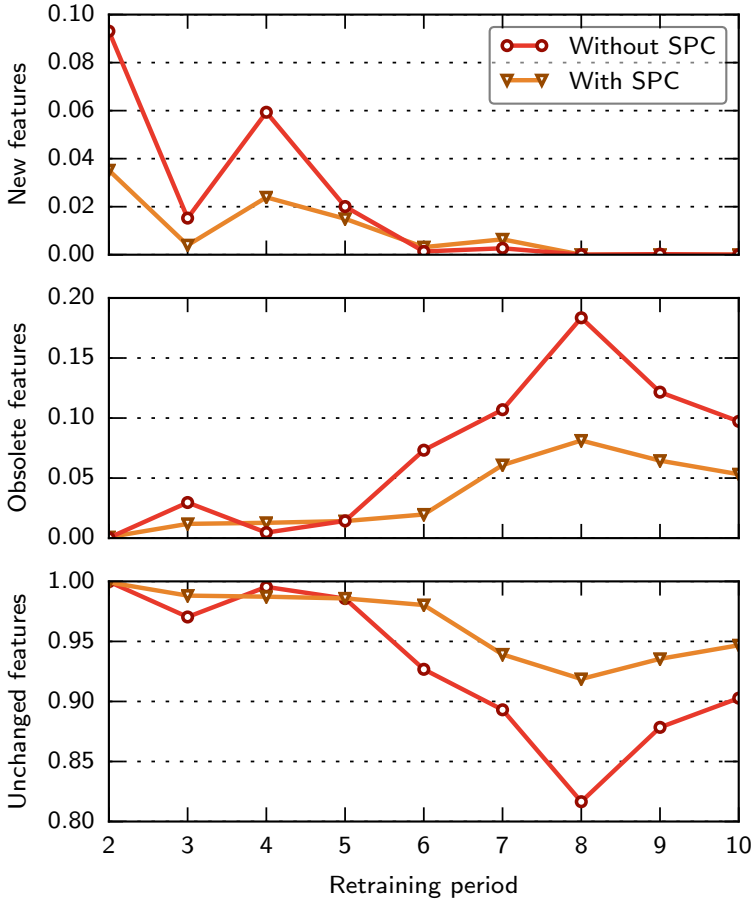


Figure 4.22: Feature set drift in the PDF dataset. Illustrates change in the feature set between consecutive feature selection events in our experiment, performed before every training, i.e., once per week. For weeks 2 to 10, the percentage of features that have been added to (new), removed from (obsolete) or remained (unchanged) in the feature set is plotted, relative to the previous week.

reader with particulars of both systems, in this section we elaborate on their most important characteristics and adversarial considerations. Finally, we present subsequently published related work.

4.8.1 Extensibility to Other File Formats

The main novelty introduced by HIDOST is its applicability to multiple file formats, implemented and experimentally confirmed on PDF and SWF. Its application to other hierarchically structured file formats, e.g., XML, HTML, ODF, OOXML and SVG, requires the instrumentation of an existing parser or the development of a new one for each file format. Given the ability to parse a specific file format, incorporating it into HIDOST amounts to developing a structure extraction module. In the following we discuss file structure and content extraction for various hierarchically structured file formats.

The XML and the related HTML and SVG formats have a very clear and well-defined hierarchical structure that represents one of their cornerstones. For example, Fig. 4.2 depicts an HTML file with the path `/html/body/p`. Furthermore, there exists a number of mature open-source parsers for XML files. We estimate it to be very simple to implement the extraction of both logical document structure and content from XML files.

Although based largely on XML, ODF and OOXML generally combine multiple XML files into a ZIP archive and therefore require some additional processing. Both formats prescribe a set of files and directories in which content, layout and metadata are separately organized. The formats differentiate between textual and graphical content; textual being stored alongside logical structure in XML files and graphical in separate files within the directory hierarchy. We observe that the files and directories are themselves organized hierarchically and that the remaining logical structure is described in XML files. Fig. 4.23 shows a simplified file and directory layout in an ODF file.

```
.
|-- content.xml
|-- manifest.rdf
|-- META-INF
|   \-- manifest.xml
|-- meta.xml
|-- mimetype
|-- settings.xml
|-- styles.xml
\-- Thumbnails
    \-- thumbnail.png
```

Figure 4.23: File and directory layout of an example ODF file.

We consider the directory hierarchy to be the top level of the logical structure. In it, the root directory of the ZIP archive represents the root node of the entire structural

```
<?xml version="1.0" encoding="UTF-8"?>
<office:document-meta
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
  xmlns:ooo="http://openoffice.org/2004/office"
  xmlns:grddl="http://www.w3.org/2003/g/data-view#"
  office:version="1.2">
    <office:meta>
        <meta:initial-creator>John Smith</meta:initial-creator>
        <dc:date>2013-04-11T13:51:33.009039356</dc:date>
        <dc:creator>John Smith</dc:creator>
        <meta:editing-duration>P0D</meta:editing-duration>
        <meta:editing-cycles>1</meta:editing-cycles>
        <meta:document-statistic
          meta:table-count="0"
          meta:image-count="0"
          meta:object-count="0"
          meta:page-count="1"
          meta:paragraph-count="1"
          meta:word-count="2"
          meta:character-count="12"
          meta:non-whitespace-character-count="11" />
    </office:meta>
</office:document-meta>
```

Figure 4.24: Example `meta.xml` file.

hierarchy. XML files can be viewed as sub-trees rooted at the corresponding nodes in the file system hierarchy. For example, ODF prescribes that the file `meta.xml`, depicted in Fig. 4.24, resides within the root directory and has a set of XML tags describing document metadata. Given this structure, the path to the *dc:creator* tag would be:

```
/meta.xml/office:document-meta/office:meta/dc:creator
```

By treating the directory hierarchy as the top level of the logical structure and XML files as sub-trees belonging to it, we ensure the complete and unambiguous extraction of logical structure. Compared to PDF and SWF, we prepend the file system path of a given XML file, relative to the root of the ZIP archive, to structural paths extracted from the file itself.

Multiple parsers for ODF and OOXML exist, of which some are open-source. We believe that it would be possible, with moderate effort, to develop structure extraction modules for both formats. Furthermore, in many cases completely benign OOXML files are used as containers for embedding malicious SWF files and Hidost can already handle them.

Structural path consolidation is the second and final format-specific step in Hidost and

requires some tuning. We expect that different formats have different requirements for SPC and acknowledge the necessity for deeper understanding of file formats for SPC rule development. The variety of SPC rules for PDF versus SWF corroborates this hypothesis.

Finally, HIDOST's applicability to a given file format does not imply its effectiveness on it. For example, despite our firm belief that extending HIDOST to XML is straightforward, its effectiveness, measured in its ability to detect malware disguised in XML files, can only be evaluated experimentally. However, its use of both structure and content for modeling makes it more likely to be successful.

4.8.2 Adversarial Considerations

Given the increasing interest for the application of learning methods to security problems, some previous work has addressed the methodology for security analysis of learning algorithms [2, 3, 44]. In this dissertation, we devote Chapter 5 to the analysis of robustness to adversarial evasion of PDFRATE, another PDF malware detection system. In the publication [91] introducing SL2013, we performed a quantitative analysis of the robustness of our system against deliberate manipulation by malicious third parties. We defined a probable attack scenario against a deployed SL2013 system in which the attacker was motivated to evade detection. In our scenario, we assumed that the attacker had complete knowledge of the system and its classification model. As a second assumption, we postulated that the attacker would start from a malicious PDF that was correctly detected by the system and attempt to add benign content to it. This would affect the file's logical structure and, as a result, influence our classifier's decision, confusing it into labeling the changed file as benign.

Our analysis found decision trees to be trivially evadable. For the linear SVM we were able to implement and evaluate an algorithm that ranked the weights of the features based on their coefficients in the support vectors of the SVM model. Having a list of features sorted by their weights makes it trivial to affect the classifier's decision by modifying those that correspond to the most negative weights (in our implementation the SVM predicted negative values for benign samples). This approach failed for the SVM with the RBF kernel because, due to the utilization of the non-linear radial basis function, the task of modifying a feature vector to yield a negative result becomes a non-convex optimization problem.

As a more potent attack strategy against the RBF SVM, we then implemented a mimicry attack, in which the most benign of a large set of benign files in the attacker's dataset would be selected as the mimicry target. The attack files were generated by introducing *all* features from this highly benign mimicry target into a large set of diverse malicious files. We evaluated this algorithm with 5,000 benign and 5,000 malicious files and our results indicated that at most 30 attack files would have fooled the classifier. As a result, we concluded that the RBF SVM was surprisingly resistant to mimicry attacks using structural features.

However, when attempting to reproduce this mimicry attack at a later point, we discovered a flaw in its source code. Namely, instead of mimicking the most benign sample in the benign set, the flaw caused the most benign sample in the malicious set to be used as the mimicry target. Naturally, the generated attack dataset was successfully detected as such by the RBF SVM. Upon discovering this flaw, we removed it and performed a corrected experiment. This time the attack was highly successful and the accuracy fell to 50 %. We therefore correct the results of the mimicry attack experiment published in [91] and see the success of the attack as evidence against the robustness of SL2013 in an adversarial environment. We present an independent adversarial analysis of our system in Section 4.8.3.

As a final point, we would like to emphasize the importance of releasing source code and datasets that enable the reproducibility of published research, especially in the area of information security where data confidentiality and maliciousness often prevent full disclosure. It is our view that the act of releasing itself motivates the researcher to better scrutinize the released materials. Furthermore, had the flawed source code been released, we strongly believe that the flaw would have been discovered by our peers shortly afterwards. It is perhaps no coincidence that we found no such flaws in other publications that form the basis of this dissertation as their source code and, in some cases, datasets were made available from the start.

4.8.3 Later Work

Of the two publications presented in this chapter, the later one [93] was published very recently and has, therefore, not been cited so far. The initial publication [91] presenting SL2013 has, to the contrary, been referenced in numerous later work.

Kantchelian et al. published a study focused on adversarial drift [40] in which they emphasize the importance of experimental evaluation of machine-learning-based systems that takes into account the dimension of time, i.e., temporal consistency of the data. They specify as example our experiment 10Weeks with a sliding-window design. Their conclusions and recommendations are in accordance with the intuition behind the design of our experiment, subsequently repeated for the evaluation of Hidost. Recently, Miller et al. proposed a novel experiment design that goes one step beyond [58]. It requires temporal consistency not just from the samples used for training and evaluation, but also from the labels. The labels assigned to samples change in time, e.g., when re-evaluating them with updated antivirus labels. They find that detection performance drops significantly if training is performed on temporally consistent labels, collected at the time of training, compared to labels obtained after the entire experiment. The two designs reflect two different assumptions. We assume that relatively reliable ground truth is available, while Miller et al. make the opposite assumption and use the next best source, an ensemble of antivirus engines, as oracle.

In [57], Maiorca et al. demonstrate that a number of static PDF malware detection methods and, by extension, SL2013, are vulnerable to a simple attack where a malicious

PDF or portable executable (PE) file or a malicious JavaScript snippet is injected into a benign PDF file, called a *reverse mimicry attack*. Our method was not evaluated against this attack, nevertheless, we propose potential enhancements to effectively mitigate it: *a*) embedded PE files would have to be extracted and scanned by a specialized PE scanner; *b*) embedded PDF files would have to be recursively processed; and *c*) embedded JavaScript snippets would introduce a small but noticeable change in the PDF structure; given that JavaScript-related structural paths dominate the list of most common paths in malicious files, this might suffice to correctly detect this kind of attack. Without a dedicated evaluation, the effectiveness of the attack remains unknown.

Very recently, Xu et al. presented a much more elaborate evasion attack against PDF malware detectors specialized on structural features. They use Genetic Programming, an evolutionary optimization method, to stitch together evasive samples from a set of 500 correctly detected malicious seeds and a few benign files by inserting, deleting or replacing objects between PDF files. The attack is evaluated against SL2013 and PDFrate, although the authors erroneously refer to SL2013 as Hidost in the manuscript. A successful run produced samples that evade SL2013 for each of the 500 seed PDF files within 2 days on a single computer, producing on average one evasive file in 5 min. We find it a major strength of the attack that it produces actual evasive malware samples instead of operating in the feature space. Furthermore, we find the evasion results plausible, although likely optimistic. The reason for this is the bias of the chosen experimental dataset. The authors take the Contagio PDF dataset with around 10,000 malicious files and filter out files that do not generate network traffic in the Cuckoo sandbox. The remaining 1,384 are scanned by SL2013 and only 502 were correctly detected. The selected 500 files target two vulnerabilities[13] affecting Acrobat Reader 8.1.1. We see two potential causes for overly optimistic results:

- The files in the dataset target only two CVEs and a single version of Acrobat Reader. We find that the dataset has too little diversity to make a confident conclusion about the results.
- Our system generalized very poorly to the utilized malware dataset, detecting only 502 of 1,384 candidate samples. We believe that this is due to an experimental flaw in the analysis, as the authors seem to have utilized features transformed by SPC, while the model was trained on untransformed features. Effectively, this led to a sparsification of the trained model, as only a subset of the original 6,087-dimensional feature set has "survived" SPC unchanged.

It remains an intriguing open question how effective the corrected attack would be against SL2013, but also Hidost with its structural path consolidation and the choice of the Random Forest classifier.

The state of the art in the related category of dynamic detection was also advanced in later work. Tang et al. published a dynamic approach using anomaly detection on

[13] CVE-2007-5659 and CVE-2009-0927.

low-level hardware features [100]. A novel combined static and dynamic method was presented by Liu et al. [51] using document instrumentation and behavior monitoring. Both publications present great detection performance and an evaluation of robustness against adversarial evasion.

In contrast to fully automated methods presented so far, Nissim et. al. propose an *active learning* approach, where a human expert manually labels interesting samples for a machine learning algorithm, with the goal of keeping the detector up-to-date with the newest threats [62]. They outline a design with a combination of signature detection and multiple methods described so far, including the use of structural features inspired by those defined in SL2013, but leave its implementation and evaluation for future work. Recently, Nissim et al. present an implementation of the proposed active learning system with a reimplementation of SL2013 used in one of the steps for pre-labeling unknown PDF files [63].

In the domain of SWF malware, despite its wide spread and growing number of exploited vulnerabilities, there has been only one related later work published very recently by Wressnegger et al. [113], presenting a combined static and dynamic SWF malware detector Gordon. Similar to Hidost, its static component also performs SWF parsing and hierarchical structure extraction with 3 main differences: *a*) Gordon works with SWF tags while Hidost goes one step deeper and considers tag fields as basic structural elements; *b*) in addition to hierarchical structure, Gordon also preserves information about the order of SWF tags by using n-grams; *c*) Hidost preserves values of tag fields. In a sense, Gordon puts more focus on the *order* of SWF tags as opposed to the detailed structural and value perspective taken by Hidost. Although evaluated on a different dataset and therefore not directly comparable, Gordon's performance was measured on a dataset from the same source, i.e., VirusTotal, and in an experiment with a very similar design. With a true positive rate between 80 % and 99 % at a false negative rate of 1 %, its combined static and dynamic approach achieves significantly better results than purely static Hidost.

4.9 Conclusions

This chapter introduces a novel set of features that enable the modeling of file formats which have a hierarchical logical structure. We presented 2 machine-learning-based systems that successfully apply this structural characterization of files in the domain of static detection of malware in non-executable file formats.

The first system, SL2013, utilizes file structure to discriminate between malicious and benign PDF files. A 10-week simulated operational deployment with weekly retraining on an unprecedentedly large real-world dataset has demonstrated its strong performance, while at the same time uncovering a difficulty in handling sudden changes of attack patterns in time. Its computational efficiency is on par with the fastest previously known static detection method PJScan: an order of magnitude higher than hybrid static/dynamic

methods and almost two orders of magnitude higher than dynamic ones.

The second system HIDOST, an extension of SL2013, is the first static machine-learning-based malware detector designed to operate on multiple file types. The generalization is accomplished by extending the purely structural file format model to include file content as well. Evaluated in the same realistic experiment, HIDOST outperformed all antivirus engines deployed by the website VIRUSTOTAL and detected the highest number of malicious PDF files. It also ranked among the best on SWF malware. Compared to its predecessor, it is much less vulnerable to malware hiding in obscured parts of PDF files. HIDOST also became more robust against the continual adaptation of malware to updated defense through periodic retraining. The dramatic reduction of feature set dimensionality achieved using structural path consolidation enables its efficient application on very large datasets. Finally, the open-source availability of the system and the experiment reproduction code, combined with the release of utilized datasets, enable full reproducibility of achieved results.

Chapter 5

A Case Study of Machine Learning Classifier Evasion

Machine learning classifiers are increasingly used for detection of various forms of malicious data including malware, email spam, malicious web pages and advertisements, etc. Although most such systems are confined to academic research environments, some have seen Internet-scale deployment. Criminals have an economic incentive to trick deployed systems, e.g., to evade the spam filter and have their message delivered, boosting their illicit goods sale. They do this by manipulating data, e.g., the content of spam messages. Examples of such attacks have been previously studied under the assumption that an attacker has full knowledge about the deployed classifier. In practice, such assumptions rarely hold, especially for systems deployed online. Nevertheless, a significant amount of information about such systems can be obtained indirectly.

In this chapter, based on our work [92] published at the IEEE Symposium on Security and Privacy 2014, we experimentally investigate the effectiveness of classifier evasion using a deployed system, PDFrate, as a test case. After the introduction in Section 5.1, we develop a taxonomy for practical evasion strategies in Section 5.2. Our taxonomy is based on different levels of knowledge available to attackers about the target system. We describe the details about PDFrate in Section 5.3 and our attack methodology in Section 5.4. Our experimental results, discussed in Section 5.5, reveal a substantial drop in PDFrate's classification scores and detection accuracy after it is exposed even to simple attacks. We further study potential defense mechanisms against classifier evasion. Our evaluation shows that the original defensive technique proposed for PDFrate is only effective if the executed attack exactly matches the anticipated one. Section 5.6 presents a detailed interpretation of results of our attack experiments. In the discussion of the findings of our study in Section 5.7, we analyze some potential techniques for increasing robustness of learning-based systems against adversarial manipulation of data. Finally, we conclude this chapter in Section 5.8.

5.1 Introduction

Data analysis methods such as machine learning are increasingly used in security applications. For tasks like malware analysis, faced with an explosion in data, deployment of automated learning methods has become imperative. Data-driven analysis enables automatic attribution of seemingly heterogeneous malware samples to a modest number of genuine malware families [4, 102]. Recent work has also witnessed several innovative applications of machine learning for detection of various kinds of security violations, e.g., drive-by-downloads [22, 75], malicious web pages [14, 94], compromised accounts and fake identities in social networks [27, 33], unwanted P2P traffic [74] and many others.

Clearly, deployment of learning methods in any security-critical context requires that they can withstand potential attacks. The security of machine learning methods has been previously discussed from conceptual [3], methodical [8, 13, 41, 42] and practical [6, 31, 69] viewpoints. Despite the growing evidence for susceptibility of learning-based approaches to adversarial data manipulation, this seems to be of little hindrance for their acceptance as a versatile tool for data-intensive security tasks. Typically, the security analysis of proposed learning-based techniques is carried out informally and is occasionally supported by experimental evaluation.

Security assessment of learning-based approaches faces several challenges. The main theoretical hurdle is the lack of formal definitions of security in the context of data analysis. In contrast to privacy, for which several formalisms have been proposed, e.g., privacy-preserving data mining [50] or differential privacy [26], no formal connection to established security objectives is known for machine learning. From the practical perspective, the success of attacks against learning algorithms crucially depends on the amount of knowledge available to an attacker. Most successful attacks reported previously assume that the attacker has full knowledge of the learned model [6, 7, 30, 31, 52, 69]. It can, therefore, be argued that reducing the amount of knowledge leaked about the model, as well as a proactive response to potential exploitation of such knowledge should provide adequate protection against adversarial data manipulation.

Still, it remains largely unclear what an attacker may learn about a learning-based method deployed "in the wild" and how this information can be exploited. To investigate this problem, we present the results of a case study we performed on a real learning-based system, PDFrate[1], an online service for detection of PDF malware [85]. For any submitted PDF file, PDFrate provides a probabilistic estimate of its maliciousness. Our study addresses the case when an attacker attempts to evade detection by modifying the submitted PDF file so that its malicious functionality remains intact but the probabilistic score returned by PDFrate is decreased.

We proceed by presenting two classes of evasion strategies suitable for several attack scenarios varying in the amount of knowledge available to the attacker. Since PDFrate

[1] PDFrate – `https://csmutz.com/pdfrate/`.

is a research system, its method and technical details are relatively well documented in the original research paper [85] and the accompanying technical report [86]. Based on this information, it is possible to partially reconstruct the features used for creation and evaluation of models, reproduce the training procedures and even independently obtain some of the training data[2]. To systematically explore the attacker's options, we define an orthogonal set of evasion strategies reflecting various degrees of available knowledge. The general idea of our evasion technique is based on insertion of dummy content into PDF files which is ignored by PDF renderers but affects the computation of features used in PDFRATE. Once we can influence a subset of PDFRATE's features, we develop algorithms for constructing attack instances. In our experiments we evaluate the effectiveness of our strategies on a set of 100 malicious files randomly drawn from a dataset known to PDFRATE.

Our results reveal that even with the smallest amount of available information, i.e., an ability to freely modify 1/6 and increment another 1/6 of the features, our attacks reduce the classification scores of PDFRATE from almost 100% to the median of about 33%. Additional information about the classifier, such as the knowledge of its type (trivial) and possession of the training dataset (somewhat more difficult to obtain), further decreases the median score to about 28%.

We have analyzed the defense strategy suggested and evaluated by the authors of PDFRATE, although we do not know if it is deployed in the online system. The attack scenario of [85] assumes that the attacker instruments a small subset of informative features. It was shown that this attack can be effectively thwarted by including a small portion of the anticipated attack data into the training set. We reconstructed this attack and verified the effectiveness of the original defense strategy. However, such proactive defense turns out to be effective only against the precise "strain" of the attack. Whenever the executed attack does not match the anticipated one, the effect of the proactive defense essentially vanishes, and the detection accuracy falls below 10%.

Our contributions can be summarized as follows:

- We present a general model for practical assessment of security of learning-based detection techniques. This model enables systematic exploration of various kinds of information leaks exploitable by an attacker and is applicable to systems beyond PDFRATE that have a modifiable subset of features.

- We present two evasion attacks that can be staged against a deployed classification model in various scenarios.

- We demonstrate the first automated practical attack against a learning-based classifier deployed "in the wild" performed without knowledge of the learned model and entirely in problem space.

[2]One of the training datasets used by PDFRATE is publicly available for the research community.

- We provide an open-source software framework Mimicus for all experiments carried out in our study for independent verification and extension of our results.

5.2 Evasion Attacks against Learning Systems

Any learning-based system deployed in a real-world environment and for which there exists a critical amount of economic, political or military interest is certain to attract the attention of individuals or groups striving to gain advantage by manipulating the system with the intention to influence its decisions. There are numerous examples of such activity. Besides the computer security applications mentioned in the introduction, potential scenarios include adversarial advertisements [83], spam detection [60, 115], recognition of writing style [12], plagiarism detection [64] and many others.

In this work we focus on *classifiers*, a particular kind of learning systems that classify new data into two or more predefined classes. Classifiers usually make predictions by computing some numeric or probabilistic score and comparing it with a fixed threshold. The goal of an adversary aiming to manipulate a classifier is to *confuse* it into providing a false classification. For binary classification problems, false classifications are called *false positives* and *false negatives*. From an adversarial viewpoint, the more information about a learning-based system is available, the higher the chances become that the system can be successfully gamed.

In the taxonomy of different types of attacks against machine learning systems introduced by Barreno et al. [2], the attacks are divided into 2 groups based on their influence: (*1*) *causative*, influencing the training data to alter the classifier before training (also known as *data poisoning*), and (*2*) *exploratory*, influencing the classifier's decision after it has been trained. This work is concerned with the latter.

The essential components comprising every learning-based classifier system are:

- the **set of features** used by the classifier,
- the **training dataset** used for classifier training,
- the **classification algorithm** with its parameters.

It is, therefore, in the interest of the adversary to maximize their knowledge about the target classifier's components. For example, an adversary A who knows the feature set and training dataset of a certain classifier has a higher chance of evading it than an adversary B who only knows the feature set. In this sense, the two adversaries A and B are operating under different *evasion scenarios*. An evasion scenario is a problem setting for evasion from an adversary's point of view. It describes the classifier system information available to the adversary in a structured way: it outlines whether the adversary has a low or high amount of knowledge about the feature set, training dataset and classification algorithm.

To systematically explore evasion attacks against classifier systems, we propose the taxonomy of evasion scenarios, depicted in Fig. 5.1, based on the amount of knowledge adversaries possess about the three components of a classifier system.

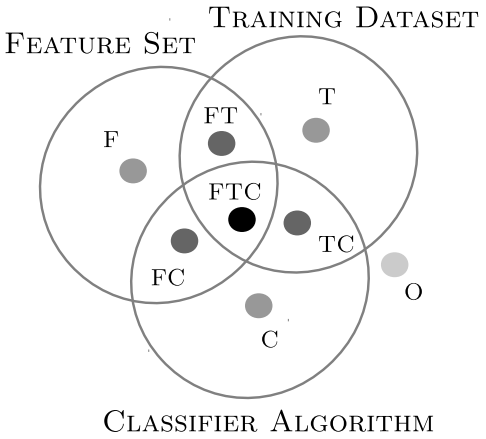


Figure 5.1: Taxonomy of evasion scenarios for classifier systems. In every scenario, represented as a point, the knowledge about a given classifier component is high if the scenario point is within that component's circle, otherwise low.

Our taxonomy comprises 8 evasion scenarios. Their names describe the information available to the adversary. If any of the letters F, T or C, corresponding to the classifier components feature set, training dataset and classifier algorithm, respectively, is present in the name of a scenario, then the level of knowledge about the given classifier component the adversary has in this scenario is high, otherwise low. The scenario named O refers to the case when the adversary has low knowledge about all three classifier components.

In our taxonomy, high knowledge does not necessarily mean complete knowledge, and *vice versa*. There exist no strict criteria for deciding whether the knowledge level about a certain classifier component should be categorized as high or low. We consider the knowledge level high if it can be used to the substantial advantage of the adversary, otherwise low.

Our study is limited to the 4 evasion scenarios in which the level of knowledge about the feature set is high. Without the knowledge of features, the attacker is faced with a major challenge of either deducing them from observation of classification results, or otherwise to attempt to directly measure the sensitivity of a classifier to changes in the original data. A novel technique for inferring the feature set [49] is presented as part of later work in Section 5.7.1.

In the following subsections, we describe high-level algorithms for staging evasion attacks in the 4 scenarios of interest.

5.2.1 Scenario F

In scenario F, only the feature set is available to the adversary, to a varying extent. The adversary might be aware of some or all features, mistakenly consider obsolete features as being used, be capable of reading a subset or all features or be able to modify some

or all features to a varying degree. Manipulation of a sufficient subset of features is, however, required in order to be able to modify samples and proceed with evasion.

An adversary with no knowledge about the classifier and training dataset may still perform evasion. If he has access to data samples certified as benign by the target classifier, he can try to align his malicious examples with known benign examples. This strategy is known as a *mimicry attack*. A particular implementation of this attack for PDFrate is presented in Section 5.4.3. In general, a mimicry attack is most effective if the attacker can submit probes to the target system during the attack, in order to ensure the benign classification of the source examples for the mimicry attack or to choose among multiple benign sources. However, online probing of a target system may be detectable and is therefore less desirable than a fully offline attack, in which only the final result is submitted to the target system.

An adversary that collects a sufficient amount of malicious samples, e.g., those found on the black market, may combine them with a collection of benign samples and thus build a *surrogate dataset*. This dataset can be used to train an off-the-shelf, *surrogate classifier*, which can then be evaded using a special-purpose attack tuned for this particular classifier. The rationale behind the surrogate classifier attack is that the inference of predictive models is based on general statistical properties which are shared among many learning methods. Hence, it is quite likely that one can approximate an unknown classifier with a suitable proxy classifier whose behavior can be controlled by the attacker. The effectiveness of this strategy critically depends on the quality of the data available to the attacker. If surrogate data is a realistic sample of the true distribution of the training data, one can expect the resulting attack to be effective against the original, unknown target classifier. The attack based on the surrogate classifier can be performed offline, with only the final result submitted to the target classifer.

5.2.2 Scenario FT

This scenario enables the adversary to take advantage of the knowledge the target classifier's training dataset, in addition to the known features. The dataset may be fully or partially leaked, enabling more accurate decisions in the process of generating a successful attack sample.

Knowledge of the benign training points enables the adversary to generate evasion samples which closely mimic them, using the mimicry attack, thereby increasing the chances of a successful attack in comparison to scenario F. Training a potent classifier on the *original* dataset creates a surrogate classifier that better approximates the target classifier than the on trained in scenario F, again opening up the way to the use of tailored methods for evasion of the surrogate classifier. Knowledge of training data enables the attacker to perform the entire attack offline before submitting the final result.

5.2.3 Scenario FC

In Scenario FC, the adversary knows the feature set and some details about the classifier, such as its type, parameters or the specific implementation. An adversary with no information about the training dataset at all and without a surrogate dataset has little advantage of knowing the classifier. With a surrogate dataset they can train a surrogate classifier of the right type, yet the accuracy of this approximation depends on the quality of the gathered data. This attack can also be performed offline, similar to other attacks based on surrogate classifiers.

5.2.4 Scenario FTC

The adversary has the best chance of evading the target classifier if he knows the details of all three classifier components. In that case, he can fully reproduce the online classifier in an offline setting, submitting the attack results only when a sufficiently good evading sample has been found. An offline mimicry attack or an offline classifier-specific attack that defeat the offline classifier have a strong probability of defeating the online one as well.

We describe the target system PDFrate and the specific algorithms used to implement the above-mentioned general attack scenarios in the following section.

5.3 PDFrate

PDFrate employs the Random Forest algorithm to classify PDF files into benign or malicious based on their metadata and certain structural features[3]. The following subsections provide an overview of PDFrate's features, classification algorithm, datasets and adversarial considerations; further details can be obtained from the original paper [85].

5.3.1 Features

PDFrate employs a total of 202 integer, floating point and boolean features. 135 of those were described in [86], the rest remain unknown. A subset of features are shown with their values for one specific file in Table 5.1. The features reflect various properties such as size and version of the file, character counts of PDF metadata items such as author name, creation and modification date, structural properties such as the count of Acrobat forms and their relative positions in the file, etc. All features were manually defined by the authors and selected for best classification performance and robustness against adversaries, respectively. The features are extracted by running a set of regular expressions

[3] PDFrate's structural features describe physical rather than logical structure, and are not to be confused with the PDF document structure.

on raw bytes of the PDF file. By not performing proper PDF parsing, authors of PDFRATE have consciously given preference to speed and simplicity rather than completeness and correctness, as some of the features might lay in encoded and/or compressed object streams, beyond the reach of regular expressions.

The features exhibit significant interdependence. When one feature's value is modified, many others may be affected because they directly or indirectly depend on the targeted feature. For example, by modifying the number of lower-case characters of the `Author` metadata field (`author_lc`), the related feature `author_len` will be affected, but so will less directly related ones such as file size (`size`). A change in size triggers further changes of seemingly completely unrelated features `pos_acroform_*`, that denote the relative file offset of one or more keywords `AcroForm`. Feature interdependence makes the adversarial control of feature values difficult.

5.3.2 Datasets

Three datasets were involved in the creation and evaluation of PDFRATE. Three models have been trained on them, which are used separately to assess new data submitted by users.

Two of the three datasets were used in the experimental evaluation of PDFRATE presented in [85]: CONTAGIO and OPERATIONAL. The CONTAGIO dataset is a collection of malicious and benign PDF files contributed by malware researchers, available for download[4]. Training of PDFRATE was carried out on a subsample of the CONTAGIO dataset containing 5,000 benign and 5,000 malicious files[5]. The trained classifier was evaluated on the OPERATIONAL dataset comprising 100,000 PDF files collected "on a large university campus". Presumably, the same dataset was used to train the model currently available as the *George Mason University (GMU)* used by PDFRATE.

The last dataset, *Community*, was created from files submitted and rated by PDFRATE users and was not used in its original evaluation.

5.3.3 Classification Algorithm

PDFRATE employs Random Forests [10], an ensemble learning method comprising a number t_{RF} of independently trained decision trees. In the training step, every tree is learned using the CART methodology, but using only a subset of the available training samples. A different subset is generated for every tree by randomly sampling a fixed number of times from the training data, with replacement – a procedure called bootstrap aggregating or bagging. When a new decision node is added to a tree, only a randomly chosen subset of f_{RF} features is considered, where f_{RF} is less than the total number of

[4]The Contagio archives are available at the following URL: `http://contagiodump.blogspot.de/2010/08/malicious-documents-archive-for.html`.

[5]MD5 sums of those files: `http://pdfrate.com/contagio_md5_class.csv`.

features. A decision is made by majority voting among all decision trees on a given new data point. Random forests are known for their excellent generalization ability and robustness against data noise. PDFrate uses the R port of Leo Breiman's and Adele Cutler's original Random Forest implementation, available as the package randomForest[6]. t_{RF} and f_{RF} are parameters of randomForest called *ntree* and *mtry*, respectively. The values *ntree* = 1000 and *mtry* = 43 are used by PDFrate.

All three classifiers deployed by PDFrate, i.e. the ones trained on the Contagio, *GMU*, and *Community* datasets, produce as their result the output of their decision function, i.e., a real value in the interval $[0, 1]$ denoting the percentage of decisions that have labeled the submitted file as malicious. There is no threshold given in [85] determining at what percentage should a file be considered malicious. Note that by providing this percentage value instead of a binary decision, PDFrate reveals much more information about its classification engines than necessary for decision-making and thus enables the adversaries to make more informed decisions.

5.3.4 Adversarial Considerations

Before describing our attacks, we discuss the properties of PDFrate crucial for the adversarial setting of our study. In our evaluation, we are only concerned with the evasion of the Contagio classifier. We do not consider the *GMU* and *Community* classifiers because their training datasets were unavailable to us and hence we could not evaluate the full spectrum of attack scenarios defined in Section 5.2. Besides being freely available, the Contagio dataset seems to remain static. Periodic retraining, an important security measure, would have complicated the consistent evaluation of effectiveness of our evasion methods, as every classifier update would have rendered previous results outdated. Furthermore, although PDFrate provides a second level of analysis by classifying malicious files into "targeted" and "opportunistic", our study is limited to evading the initial binary classifier.

From an adversarial perspective, the level of knowledge available to attackers about PDFrate is high. The availability of its feature definitions facilitates the creation of manipulated samples. Although robust against data noise, the Random Forest classifier was not designed for resilience against adversarial noise. Periodic retraining is also not carried out in the deployed system. These weaknesses make PDFrate an excellent candidate for our case study. Other, more prominent machine-learning-based malware detectors have features which are either unknown or much more difficult to control.

Despite its weaknesses, adversarial considerations were indeed present in PDFrate's inital design. The attack model considered in [85] assumes that the adversary knows the means and standard deviations of the 6 most important features, i.e., those on top of the list of variable importance measures of the Random Forest model, for the benign training files. The adversarial model assumes that an attacker can create camouflaged

[6] randomForest – `http://cran.r-project.org/web/packages/randomForest/index.html`.

malicious samples in which a subset of top features is set to random values drawn from the normal distributions with the given means and standard deviations characterizing the benign samples. We refer to this attack as "benign random noise" (BRN). It was shown in [85] that the BRN attack can severely degrade the detection accuracy of the classifier. To counter this attack, a proactive defense strategy was proposed: to modify a subset of malicious data points in the training set in exactly the same way as an attacker would proceed. This simple defense strategy proved to be surprisingly effective.

The BRN attack was implemented synthetically, i.e., by modifying the top 6 features *directly in feature space*. Therefore, it does not address the issue of whether real PDF files can be generated with the required feature vectors. Due to strong feature interdependencies, such an assumption is unrealistic in practice. In our evaluation of the defense mechanisms presented in Section 5.5, we depart from the feature space and evaluate this attack using real PDF files. Furthermore, we investigate the robustness of the proposed countermeasure against our own mimicry attack.

5.4 Methodology

Since our study of evasion scenarios assumes a stealthy attacker, the key elements of our methodology involve reimplementation of the methods deployed by PDFrate. We first reconstructed a subset of PDFrate's features using the available public knowledge. The next step was to develop a technique for manipulation of PDF files which affects the selected subset of features. The last step in our methodology was to design attack algorithms for carrying out the generic attack strategies presented in Section 5.2.

The above techniques and methods were implemented in our experimental evasion framework called Mimicus. The framework consists of a Python module which supports:

- feature extraction,
- PDF file modification,
- upload to PDFrate and score retrieval,
- classifier training and
- classifier attack.

Mimicus is free and open-source software, suitable for extension with other attacks and attack targets. It is available for download[7], bundled with all training data (as feature vectors), classifier models and code required to fully reproduce our experimental results. All attack files used in our experiments can be obtained from the Contagio database.

5.4.1 Reimplementation of PDFrate Features

Our four evasion scenarios have one common assumption: the adversary knows the features of the attacked system. The level of knowledge about particular features may, how-

[7] Mimicus – `https://github.com/srndic/mimicus`.

ever, vary widely. The attacker may not be aware of some features' existence at all. Even for features with known description, the attacker may have partial or no control of their values. Finally, interdependence between features prevents the attacker from arbitrary manipulation of their values.

The knowledge about PDFRATE's features comes from three sources: the original research paper [85], the technical report [86], and the behavior of PDFRATE as deployed online. As stated in [85], a total of 202 features are employed by PDFRATE. However, only 135 of them are described in [86], to a varying extent. This limits the set of features potentially under attacker's control to roughly $^2/_3$ of the reported number. Furthermore, it cannot be ruled out that the deployed system uses a different set of features compared to the reported ones due to a natural progress in development.

As a first step, we reimplemented the extraction of 135 known features by following the general guidelines on feature extraction from [86]. Subsequently, regular expressions were developed for each feature except for `size`, which was read directly. During this process we also examined metadata output produced by PDFRATE for a fixed test suite comprising PDF files with a broad range of values for many features. Values of some features, e.g., counts of `Page` or `obj` keywords, can be accurately deduced from the metadata output. The regular expressions were further refined until consistent behavior was achieved across all test files. Although the reimplementation process required time-consuming expert work, it would be a small hurdle for an incentivized adversary.

Thanks to the availability of the CONTAGIO dataset, we were able to verify the correctness of our reimplementation by comparing our classification results on that dataset with those reported in [85]. We, furthermore, verified that despite the discrepancy in the set of implemented features, our local clone of PDFRATE produces similar classification scores as the online system on a benchmark dataset presented in Section 5.5.1.

5.4.2 Modification of PDFRATE Feature Values

The development of the PDF file modification method for our study was guided by the following design goal: once modified, the file in question has to appear indistinguishable from the original to any PDF parser, yet reliably affect PDFRATE's feature extraction. The reason for this is that such a semantics-preserving method can be safely applied to malicious PDF files in our experiments, regardless of the diverse vulnerabilities they may exploit, without the risk of breaking their potentially subtle *modus operandi*.

The feature modification component of MIMICUS can arbitrarily modify values of 35 and increment values of 33 features of PDFRATE, as detailed in Appendix A. Modification of further features would have required delicate changes to the structure of PDF files, increasing both the implementation effort and the risk of breaking the malicious functionality.

Our approach to file modification was motivated by the discrepancy between the operation of PDF readers and PDFRATE. This approach was described in [38] as an example of a *semantic gap* in the interpretation of various file formats. PDFRATE evaluates a set

of regular expressions over the raw bytes in a PDF file, reading *from the beginning to the end* of the file. In contrast, PDF readers parse PDF files in adherence to the PDF syntax prescribed by the PDF Reference. A conformant PDF reader reads a file starting from its end. It checks the trailer to find the location of the cross-reference table (CRT) and then *jumps* directly to it in order to locate the objects in the file body. This difference is illustrated in Fig. 5.2, showing the layout of a PDF file before and after our modifications.

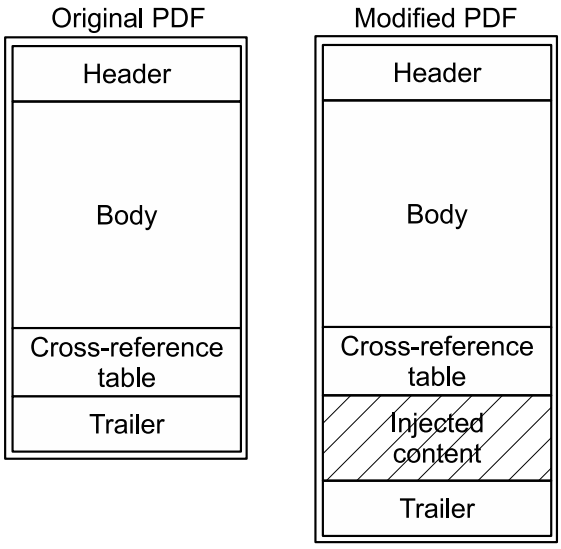


Figure 5.2: The PDF modification method takes the original PDF (left) and injects new content between the cross-reference table (CRT) and the trailer. Such a modified file (right) confuses PDFRATE into accepting the newly-injected content as part of the file, while the PDF readers jump from the trailer directly to the CRT, skipping the injected content completely.

Our solution exploits this semantic gap: as long as the file header, body and CRT are not modified or moved, the trailer can be moved arbitrarily far away from the CRT[8], thereby generating an empty space in the file where arbitrary content can be injected. Such content will be processed by PDFRATE, but PDF readers will *always ignore it*.

The described *content injection approach* leaves behind file modifications which are trivial to detect if one knows what to look for. We believe that it is possible to rewrite the PDF files, modifying the content in-place instead, thereby concealing the modifications altogether. In fact, a method was developed after the publication of our work [92] that achieves this, as presented in Section 5.7.

Our modification method proceeds by injecting a set of whitespace-separated string patterns into the gap between the CRT and the trailer of the target PDF file. The patterns are crafted to make specific PDFRATE regular expressions match them, thereby influencing the extracted feature values. For example, injecting into a file with 5 `obj` keywords

[8]It is only important that the trailer remains at the end of the file.

the string "`obj obj`" will change its `count_obj` feature value from 5 to 7, as PDFRATE's regular expressions will match them all. As another example, the length of the `Author` metadata field can be "reduced" from 10 to 3 by injecting a new `Author` field with 3 characters, "`/Author(abc)`", as PDFRATE tends to only take into account the content of the last metadata field in the file. By injecting our payload just before the trailer we can ensure that this condition is fulfilled.

Using the described modification method it can be safely assumed that the behavior of PDF readers will not be altered[9], but PDFRATE would be tricked into reading the desired feature values from the modified file. Our experiments have confirmed this behavior for two PDF readers, ADOBE READER and EVINCE. In addition, we have submitted all malicious files involved in our evasion experiments to WEPAWET [22] before and after modification and verified that the exploit effectiveness of each sample was unaffected.

As already mentioned, the features of PDFRATE are heavily interdependent, i.e., it is, in general, impossible to perfectly translate data points in feature space into files in problem space. Given a malicious file before the attack F_{before} and a data point P_{attack} generated by the attack algorithm run on F_{before}, the adversary wants to generate the attack file F_{after} that optimally defeats the classifier. This is achieved by modifying the feature values of F_{before} by setting them to P_{attack}. However, due to feature interdependence, the resulting file $F'_{after} \neq F_{after}$ has different features, which may or may not defeat the classifier.

Table 5.1 shows an example using the GD-KDE attack, described in Section 5.4.3. Feature `pos_acroform_min` denotes the relative file offset of the first occurrence of the keyword `AcroForm` and is not modifiable by MIMICUS, however, it was *indirectly* influenced by the increase of the total file size. On the other hand, although feature `author_len`, denoting the length of the `Author` metadata field, is directly modifiable, it got the value 11 instead of the desired 0. This occurred because other modifiable features, i.e., `author_{lc|num|oth|uc}`, denoting different character classes in the `Author` field, drove the total character count to 11.

Another important consideration regarding the translation of data points from feature space to problem space is that algorithms operating in feature space may construct data points which are not feasible in problem space. Examples are the `size` and `version` features to which the attack algorithms attempted to assign negative values. It is neither feasible to enumerate all feature interdependencies and account for their effects *a priori*, nor to identify invalid data points before translation to problem space.

Our approach to dealing with these two limitations is opportunistic: we generate the file from the feature vector by translating features one by one, independently from each other and without taking the limitations into consideration, in the hope that the resulting file's features are not *too far away* from the desired values. Although this approach results in hardly predictable outcome, the resulting files have feature values sufficiently

[9] Provided that they do not parse the injected content, but perform the direct jump prescribed by the PDF Reference instead.

Feature	F_{before}	P_{attack}	F'_{after}
author_lc	0	2	2
author_len	0	0	11
author_num	0	3	3
author_oth	0	5	5
author_uc	0	1	1
count_acroform	1	0	1
count_endobj	11	918	465
count_endstream	1	169	85
count_eof	0	2	2
count_font	0	86	86
count_image_large	0	1	1
count_image_small	0	6	6
count_image_total	0	0	11
count_image_xsmall	0	4	4
count_javascript	3	0	3
count_obj	14	922	922
count_objstm	0	28	28
count_page	0	29	29
count_stream	1	169	85
count_trailer	1	0	1
count_xref	1	0	1
createdate_ts	-1	7.52×10^{8}	7.52×10^{8}
image_totalpx	0	0	813898
moddate_ts	-1	$1.0\ \times10^{9}$	$1.0\ \times10^{9}$
pos_acroform_avg	7.04×10^{-2}	7.04×10^{-2}	7.16×10^{-3}
pos_acroform_min	7.04×10^{-2}	7.04×10^{-2}	7.16×10^{-3}
pos_acroform_min	7.04×10^{-2}	7.04×10^{-2}	7.16×10^{-3}
size	2726	-426760	26782
version	0	-4	0

Table 5.1: Changes of feature values for a subset of features in an example GD-KDE attack in scenario F. The column F_{before} shows the feature values extracted from a malicious candidate file, with the SHA-1 hash *a39cf14b806db14a9e877b665324d203e5a5a666*. GD-KDE transformed these values in feature space into data point P_{attack}. Point P_{attack} was used to modify file F_{before} in problem space and generate the attack file F_{after}. However, feature interdependence caused the file F'_{after} to be generated instead, with slightly different feature values.

close to the desired ones and are suitable for evasion purposes. As a concrete example, compare columns P_{attack} (desired outcome) and F'_{after} (actual outcome) of Table 5.1.

Additional safety mechanisms implemented in Mimicus prevent feature modification if the desired value is outside of valid bounds specific to the feature and the file, e.g., if there was an attempt to modify `size` to a positive value less than the file already had. Reasonable lower and upper bounds enforced by our method were inferred by enumerating the feature values of all files in the dataset, described in Section 5.5.1, and extracting the minimum and maximum values for each feature. Another reason for preventing feature modifications is a feature data type mismatch, e.g., when a data-type-agnostic algorithm wants to set a boolean feature to 7. In the end, the result is a valid PDF file with features close to the desired ones.

5.4.3 Attack Algorithms

The second major component of Mimicus are its attack algorithms. Their main goal is to generate PDF files whose feature vectors are likely to receive low classification scores. To this end, we have adapted two previously known methods to the specific context of PDFrate's features.

Mimicry Attack

The mimicry attack is well-known in the security literature. Its idea is to transform a malicious sample in such a way that it mimics a chosen benign sample as much as possible, making the resulting mimicry sample harder to detect. This attack is simple to implement, can be applied to any classification algorithm, and does not necessarily depend on a particular learned classifier model. Therefore, it is suitable for evaluation in every evasion scenario. Our implementation takes a malicious file and simply attempts to modify all of its modifiable features at once to take on the values of the features of a chosen mimicry target, a benign file. To increase the effectiveness of a mimicry attack, we repeat it 30 times using different benign targets for every attack file. The resulting 30 files are evaluated using a local classifier, and only the sample which best evades the local classifier is submitted to PDFrate.

Due to the existence of undisclosed features and technical limitations discussed in the previous section, it is impossible to generate a file which exactly corresponds to the feature vector resulting from a mimicry attack. It is important that the conversion of a feature vector into a file is performed after the mimicry is complete in the feature space. The latter is technically straightforward: we simply merge a malicious feature vector into a chosen benign one while protecting existing values. Modifying features one at a time while translating them into a file is not a good strategy, as the interdependency between features dominates the transformation and generates a lot of uncontrollable changes. Using a single-step transformation makes such interdependency less prominent.

The generality of the mimicry attack, i.e., its independence of the specific learning algorithm and the underlying dataset, makes it applicable to other learning-based systems which, like PDFRATE, have a known and modifiable subset of features. An inverse attack, performed by injecting malicious content into a benign PDF file, was described in [57] and demonstrated to be effective against PDFRATE in a small-scale experiment.

Gradient Descent and Kernel Density Estimation (GD-KDE) Attack

The second attack evaluated against PDFRATE is based on a method employing gradient descent and kernel density estimation (hence we call it GD-KDE) to defeat a classifier with a known, differentiable decision function [7]. It requires the knowledge of a specific learned model and a set of benign samples. Additionally, because it is based on gradient descent, it is only applicable to differentiable classifiers, e.g., SVM or artificial neural networks, and cannot be applied to the Random Forest. Hence, the GD-KDE attack is applicable only to scenarios with differentiable surrogate classifiers, F and FT.

The GD-KDE algorithm proceeds by following the gradient of the weighted sum of the classifier's decision function and the estimated density function of benign examples. The starting point of the gradient descent is the feature vector of the malicious sample. The starting sample is usually correctly classified as malicious; the goal is to move to the area where the classification algorithm classifies points as benign. In order to avoid moving to infeasible areas of the feature space with negative classifications, the algorithm's objective function has the second term, the density of benign examples. This ensures that the final result lies close to the region populated by real benign examples. The density function must be estimated beforehand, using the standard techniques of kernel density estimation [66]. Similarly to the mimicry attack, we run GD-KDE in feature space to completion before transforming the result into a file.

5.5 Experimental Evaluation

The experiments to be presented in this section assess the effectiveness of evasion techniques presented so far. In our evaluation protocol, we take on the role of an attacker and combine all available means to defeat an up-to-date version of PDFRATE as it is deployed. An attacker has no control over PDFRATE's deployment, hence no guarantees can be provided that the system has not changed between individual experiments. Since our evaluation was carried out against the model trained on a static dataset and took place within one week, it is quite unlikely that any changes in the production system have occurred.

In another set of experiments, we also investigate the impact of our attack on the defensive measures suggested in the original PDFRATE publication [85].

5.5.1 Datasets

Three datasets were used in our experiments: two datasets, CONTAGIO and SURROGATE, were intended for training of local classifiers needed for attack implementation, while the ATTACK dataset consisted of malicious files used as starting points for generating attack samples targeting PDFRATE.

Contagio dataset

This dataset is an exact copy of the original PDFRATE training dataset, described in Section 5.3.2. It contains 5,000 benign and 4,999 malicious PDF files[10]. It is reasonable to assume that an adversary knows that this dataset was used for training and obtains access to it.

Surrogate dataset

This dataset is designed as a dataset that an adversary without access to CONTAGIO data might have collected to approximate it. Malicious files in the dataset are a random subsample of dataset **D1** described in Section 4.5.1, i.e., PDF files uploaded to VIRUSTOTAL between the 5th and 22nd of March, 2012. These files are newer than CONTAGIO data but were known before PDFRATE was published. Four files in the dataset were found to be present in the CONTAGIO dataset and were removed to ensure strict complementarity of data. Benign files in the SURROGATE dataset are randomly subsampled from dataset **D4**, i.e., files obtained using keywordless Google web searches for PDF files published between February 5, 2007 and July 25, 2012. The SURROGATE dataset has the same size and composition as the CONTAGIO dataset.

Attack dataset

This dataset contains 100 malicious files that we used as starting points for all attacks. This dataset was deliberately chosen to be small in order to minimize operational impact on PDFRATE. The adversary has access to these files in all scenarios. The files were randomly drawn from the CONTAGIO dataset. They were already known to the classifier and therefore make evasion even more challenging for the attacker. Both PDFRATE and WEPAWET classify all of them as malicious; PDFRATE with a very high score, as shown in Fig. 5.3, "Baseline". All files are distinct in both the problem and feature space.

5.5.2 Classifiers

Depending on whether the attacker knows the exact classification algorithm employed by PDFRATE or not, he might use either the original or an unrelated, surrogate classifier.

[10]Malicious file with the MD5 hash `35b621f1065b7c6ebebacb9a785b6d69` was missing from the archives.

Our experimental framework Mimicus models these two cases by deploying the Random Forest classifier in scenarios FC and FTC, where the classifier type is known, and a Support Vector Machine (SVM) classifier in scenarios F and FT.

Random Forest

The classifier implementation and parameters are identical to the original classifier of PDFrate described in Section 5.3.3.

Support Vector Machine

We have chosen the SVM [21] as a surrogate classifier because it delivers high classification performance on many problems, including the discrimination between malicious and benign PDF files using PDFrate features, and is unrelated to the Random Forest. The SVC implementation of the scikit-learn [68] machine learning toolkit version 0.13.1 was used.

As elaborated in Section 4.4.3, the SVM learns by mapping labeled training points into a high- or even infinite-dimensional feature space, optionally applying a kernel function to the input feature vectors to make them better separable. It then finds a separating hyperplane in the new space with the largest possible margin, i.e. the distance between the convex hulls of the two classes of points, malicious and benign. The hyperplane vector, represented by support vectors and their weights, constitutes an SVM model. When classifying new data, the distance between the hyperplane and the new data point is calculated. This distance, called the decision function score, is a real value whose meaning is similar to the classification score of PDFrate. A binary decision can be made by taking the sign of the decision function score for the new data point. We assign the positive score to the malicious class by convention. To evade an SVM, the adversary needs to modify a malicious data point so that its decision function score changes sign.

Two kernel functions were evaluated: linear and RBF. The linear kernel

$$k_{linear}(x_1, x_2) = x_1 \cdot x_2 \tag{5.1}$$

provides a linear transformation of two input vectors x_1 and x_2, while the RBF kernel

$$k_{RBF}(x_1, x_2) = exp(-\gamma \|x_1 - x_2\|^2) \tag{5.2}$$

utilizes the Gaussian radial basis function as a nonlinear transformation of its arguments. For optimal evasion results, a grid search was carried out on the two adversary's datasets, optimizing the SVM parameters C for the linear, C and γ for the RBF kernel. The highest achieved accuracy using 10-fold cross-validation and a 60:40 training-test split was 98.7 % on the Contagio and 99.5 % on the Surrogate dataset.

The parameters found in the grid search are the same for both datasets: RBF kernel, $C = 10$, $\gamma = 0.01$. However, the generalization ability of the two learned models for the

two datasets differs greatly. The SVM trained on Contagio data achieves an accuracy of 98.5 % on Surrogate data, but the SVM trained on the Surrogate data performs very poorly on Contagio data, with an accuracy of 61 %. Of course, the adversary in scenarios F and FC, without access to original training data, would be unable to check how well its SVM performs on it. Bad approximation of the training data strongly affects the performance of the Gd-Kde attack.

Due to differences in scale among features and as a general rule when using SVMs, *feature standardization* was performed on extracted data points by subtracting the feature mean from the feature value and dividing the result by the standard deviation of that feature. Means and standard deviations of all features were calculated on the Contagio dataset.

5.5.3 Attack Scenarios

The following subsections elucidate the implementation of the 4 main attack scenarios described in Section 5.2 in our experiments, summarized in Table 5.2.

Table 5.2: Realized Attack Scenarios.

Scenario	Classifier	Dataset	Attacks
F	SVM	Surrogate	Mimicry, Gd-Kde
FC	Random Forest	Surrogate	Mimicry
FT	SVM	Contagio	Mimicry, Gd-Kde
FTC	Random Forest	Contagio	Mimicry

Scenario F

In scenario F, all that the adversary knows about PDFrate is how to read 135 and modify 68 features. Nevertheless, there are two attacks he can perform, depending on the available datasets, as elaborated in Section 5.2.1. The mimicry attack uses randomly sampled benign files from the Surrogate dataset, as the Contagio dataset is unknown. Similarly, the surrogate classifier is trained on the Surrogate dataset for evasion using Gd-Kde. The classifier parameters are optimized using grid search on the Surrogate dataset. Both attacks were performed offline, without classifier feedback, and their results were uploaded to PDFrate for evaluation.

Scenario FT

In scenario FT, besides the limited knowledge of features, the adversary has a complete knowledge of the training dataset. Therefore, the Contagio dataset is used to train a surrogate classifier for the Gd-Kde attack in this scenario, and randomly sampled benign

files from CONTAGIO are used as mimicry targets for the mimicry attack. This time, the surrogate classifier is optimized using grid search on the CONTAGIO dataset. Only the final attack results are submitted to PDFRATE.

Scenario FC

Knowledge about the classifier is added to the limited knowledge of features in this scenario. The adversary knows the original classifier, its implementation and parameters. They use the SURROGATE dataset with the original classifier to produce a surrogate classifier, which they evade offline using the mimicry attack, with mimicry targets randomly selected from the SURROGATE dataset. Results are submitted to PDFRATE for evaluation.

Scenario FTC

Given the limited knowledge of features and complete knowledge of the training dataset and classifier, the attacker creates a local clone of PDFRATE and evades it offline. Only the final attack results are submitted online.

5.5.4 Results

Before the attack experiments were run, all 100 files in the ATTACK dataset were evaluated by PDFRATE. The results of this evaluation, shown in Fig. 5.3, "baseline", provide a baseline with which we compare the attack results. All but 3 files received a 100% malicious classification score.

Our evaluation followed a simple protocol. For every attack, 100 files from the ATTACK dataset were used to generate attack samples. The effectiveness of generated attack files was evaluated by submitting them to PDFRATE and comparing the received classification scores with the baseline. All attack samples were submitted to WEPAWET to verify that they remained malicious after modification.

The summary of PDFRATE's scores for attack files is presented in Fig. 5.3. For each attack, the population of 100 classification scores is represented as a box plot, with the median shown as a thick line, the 25th and 75th percentiles ("interquartile range" or IQR) as a box, scores within 1.5 IQR from the median as "whiskers", and the remaining outliers as single points. Plots are grouped by attack scenario.

The results show that PDFRATE was evaded *in all 4 attack scenarios*. The median score dropped down to 28 % to 42 % for mimicry and 29 % to 34 % for GD-KDE attacks, depending on the scenario. For all attacks except mimicry in scenario F, the 75th percentile of the box plot lies below the 50 % mark, implying that 75 % of attacks would be classified as benign if a 50 % threshold over classification scores were used for decision making. The significance of these results is further emphasized by the fact that only a third of features were modifiable, and the files used for evaluation were already known to PDFRATE at training and hence more difficult to evade.

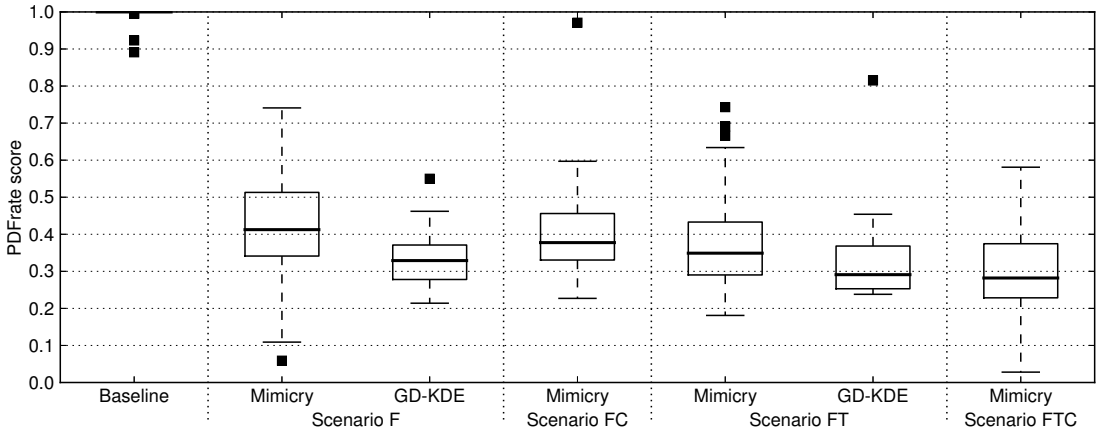


Figure 5.3: Populations of PDFRATE scores of all 100 attack samples from the ATTACK dataset before (baseline) and after each attack (remaining boxes). Attacks are grouped by scenario. The boxes extend from the first to the third quartile, with the median value between them (thick line). The whiskers extend to the farthest datum within 1.5 times the interquartile range from the box, while the squares represent the outliers.

Results in scenarios with a surrogate classifier, F and FT, demonstrate the superiority of GD-KDE over mimicry. Furthermore, the mimicry attack in the scenario with the highest amount of knowledge, FTC, only marginally outperforms GD-KDE in scenario FT. Further insights into the behavior of the GD-KDE attack are given in Fig. 5.4. It shows the decision function values of two SVMs, one trained on the SURROGATE dataset in scenario F, the other on the CONTAGIO dataset in scenario FT, before and after attack. The post-attack SVM scores demonstrate that the GD-KDE attack reliably steers *all* samples far across the decision boundary into the benign region. If the SVM classifier were deployed by PDFRATE, it is very likely that the GD-KDE attack would have attained perfect evasion, driving all scores below zero. Since the SVM only approximately matches the decision function of a Random Forest, attacks against PDFRATE fall far from being perfect, but still significantly decrease the scores.

By careful observation of Fig. 5.4 it is evident that over 25 % of pre-attack samples in scenario F have a negative decision function value, i.e., are classified as benign by the SVM (but not PDFRATE) *before* attack. This is a consequence of operating under scenario F, where the adversary trains using the SURROGATE dataset but attacks using samples from the CONTAGIO dataset. Because of the poor generalization of SVM models in this case, as elaborated in Section 5.5.2, the samples are often misclassified. In scenario FT, where the attacker also trains on the CONTAGIO dataset, the baseline SVM scores are strictly positive.

Another important observation based on Fig. 5.3 is the improvement of attack effectiveness with the increase of adversary's knowledge about the target system. This finding is in agreement with our initial conjecture about the importance of adversary's knowledge

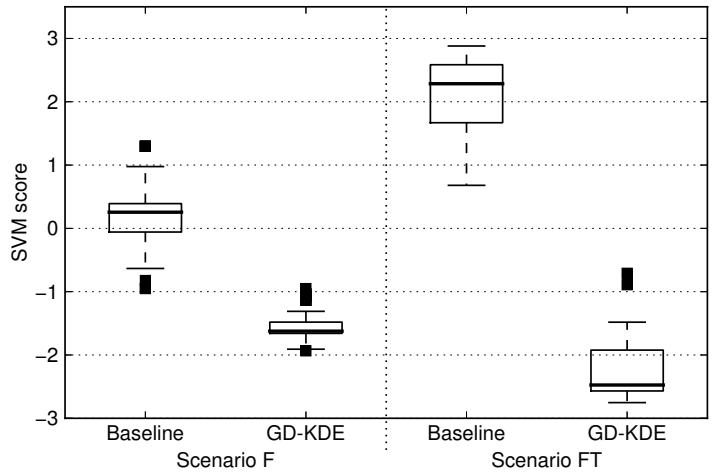


Figure 5.4: Populations of SVM decision function values for all 100 attack samples from the ATTACK dataset before and after GD-KDE attacks in scenarios F and FT. Parameters of the box plot are described in Fig. 5.3.

for classifier robustness. However, it is curious that the improvement from scenario F to scenario FTC is not as dramatic as one might expect: mimicry improves by around 14 % and GD-KDE in scenario F is outperformed by the best overall attack, mimicry in scenario FTC, by a mere 6%. This is an important finding indicating that merely knowing a subset of features might provide the adversary more advantage than previously considered.

The possession of training data is the second most important contribution to the attackers' success, after the knowledge of features. Fig. 5.5 compares the scores of two local Random Forests with that of PDFRATE on mimicry attacks in scenarios FC, using SURROGATE, and FTC, using CONTAGIO data. It can be seen that on SURROGATE data, the locally implemented classifier with the exact parameters used by the target makes an overly optimistic assessment of the attack effectiveness, achieving a median score of about 18 %, while the same files get a median score of 37 % when submitted to PDFRATE. However, when staged with the CONTAGIO dataset, the local estimate of the attack score is almost identical to PDFRATE's (29 % and 28 %, respectively). This similarity is surprising, taking into account that the local classifier was trained using only a subset of PDFRATE's features and that the training process of Random Forests is heavily randomized.

As a final step in our evaluation, we investigate the impact of attacks on the detection performance of PDFRATE. Recall that the classification score still needs to be compared with some threshold for a binary decision to be made. Earlier, we reported that 75 % of attack points would have fallen under the radar if the threshold were set at the specific value of 0.5. To analyze the detection performance *for all possible thresholds*, the Receiver Operating Characteristic (ROC) curves are presented in Fig. 5.6 for the baseline and all attacks.

The ROC curves were obtained on a mixed dataset containing the same 100 attack

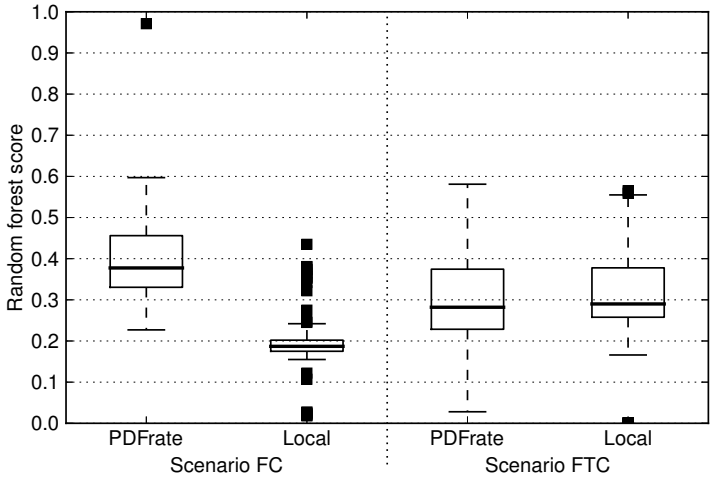


Figure 5.5: Populations of Random Forest scores for all 100 attack samples from the ATTACK dataset by PDFRATE and two local Random Forests on two mimicry attacks. Parameters of the box plot are described in Fig. 5.3.

samples and all 1051 benign samples from the original Contagio database not in the CONTAGIO dataset. It is clear that, especially in the lower range of false alarm rates (less than 0.5 %), the detection performance of PDFRATE is dramatically decreased by the attacks, and the mimicry attack of the FTC scenario has caused a 7 % false positive rate. The relative effectiveness of attacks with respect to detection performance is similar to their relative effectiveness with respect to classification scores (cf. Fig. 5.3).

5.5.5 Defensive Measures

In our last experiment, we have investigated the robustness of defenses proposed in [85] to our evasion technique. To set the baseline, we have reproduced the mimicry attack and the defense technique in exactly the same way as it was originally proposed, using the Random Forest classifier trained on the CONTAGIO dataset. Our classifier ranked the following 10 features significantly above others as most important, in descending order:

`count_font`	`pos_box_max`	`len_stream_min`	`producer_len`
`count_js`	`pos_eof_avg`	`count_endobj`	
`count_javascript`	`pos_eof_max`	`count_obj`	

We have then reimplemented the original "benign random noise" (BRN) attack in feature space and extended it to problem space by materializing the attack's final feature vector as a file. Fig. 5.7 compares these two attack variants as a function of number of modified features.

We observe that the behavior of the synthetic variant of the BRN attack (solid line) closely resembles the results reported in [85], with a slightly higher impact on accu-

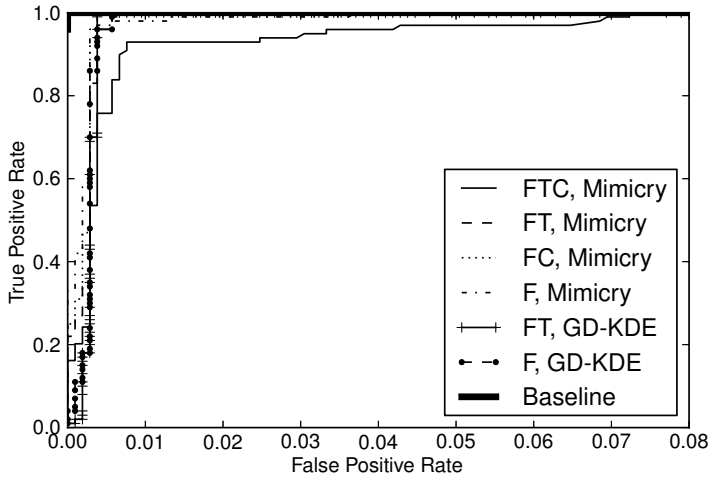


Figure 5.6: ROC curves of the baseline and all attacks.

racy[11]. However, when applied to files, the BRN attack is ineffective (dashed line). Only the modification of one or two features exhibits some impact on detection accuracy. Attempting to modify further features leads to increasing inadvertent modifications which end up steering the mimicry samples towards the benign class. Furthermore, only 5 of the top 10 features are modifiable. Therefore, the BRN attack in problem space is impractical and, compared to other attacks presented in Section 5.5.4, suboptimal.

Finally, we have evaluated the effectiveness of the "vaccination" mechanism proposed by PJScan authors which modifies a fraction of malicious samples in the training dataset in such a way that they are more similar to *expected* attack samples. Two scenarios are considered: when the defender anticipates the (*1*) *right* and the (*2*) *wrong* kind of attack. Fig. 5.8 compares the effectiveness of the vaccination defense against the BRN attack under both scenarios. Our experiment confirms the effectiveness of the vaccination defense when the right kind of attack, i.e., BRN, is anticipated (dashed curve). However, the classifier vaccinated with the BRN attack showed no resistance to our mimicry attack from the FTC scenario (dotted curve).

Repeating the experiment with the vaccination defense, this time seeded using our own mimicry attack, revealed that the resistance to the attack was restored (Fig. 5.9). Hence it can be concluded that the vaccination mechanism is effective against any *correctly anticipated* attack. The latter assumption, however, is rather unrealistic in practice.

5.6 Interpretation of Attacks

From the operational perspective, it is crucial to understand which features contribute most to the success of reported attacks. In general, interpretation of models created by

[11] Accuracy scores vary due to experiment randomization, different models and cutoff values.

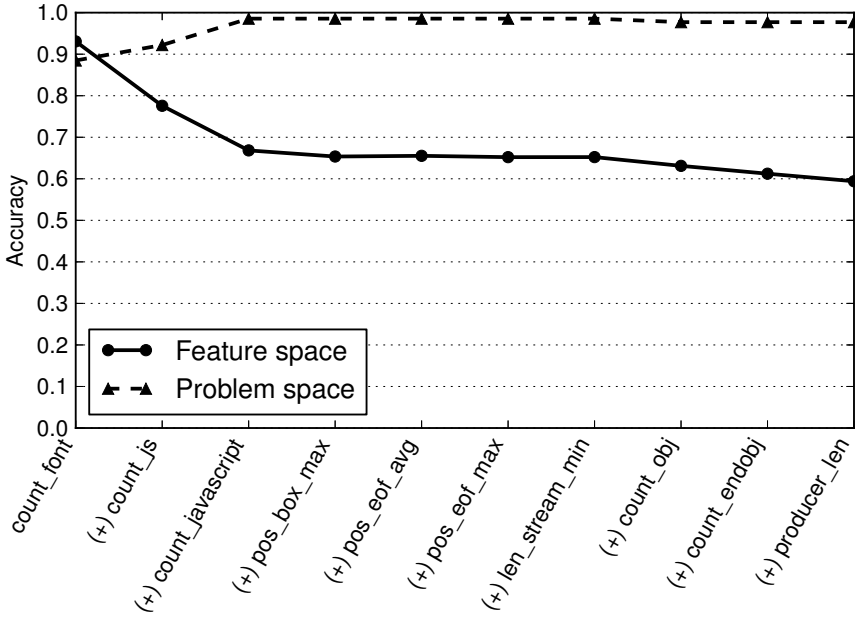


Figure 5.7: Results of the BRN attack applied on data points in feature space versus files in problem space. The attack was applied 10 times, starting with only `count_font` and progressively modifying ever more features.

learning techniques is difficult. Although Random Forest classifiers provide a ranking of features according to their informativeness, which has been crucial for the design of the BRN attack, this information is only indirectly related to the two types of attacks presented in this chapter. Hence, a different analysis technique had to be developed to interpret our attacks.

Our interpretation is based on the binary difference between feature vectors before and after an attack. While it may be tempting to claim that the features with largest change are the most informative, this measure is strongly misleading in our case since the ranges of feature values are vastly different. Even re-scaling the changes to valid value ranges is not suitable since the bounds for specific features can only be determined on the basis of an empirical sample of PDF files and are prone to outliers. Furthermore, only one third of the features is directly modifiable in our approach, yet all of them may be indirectly modified as a result of some other changes.

The only conceivable characterization of the mimicry attack is the empirical support of specific features, i.e., the percentage of files for which a given feature was changed by the attack. The histograms of feature support are shown in Figures 5.10a and 5.10b for the Gd-Kde and mimicry attack, respectively. It can be seen that both attacks perform a significant amount of feature modifications, hence one cannot explain the attacks by a small number of essential features. Between the two attacks, the modifications produced

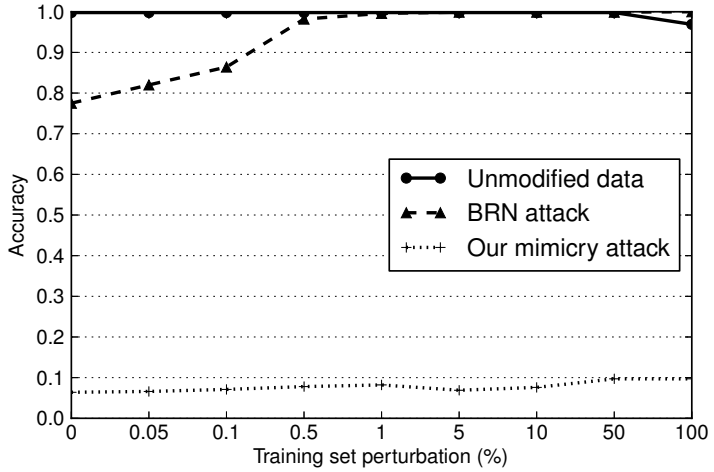


Figure 5.8: Performance of the defensive measure proposed in [85] seeded with BRN attack samples. Averaged results of 5 independent trials, using 10-fold cross-validation.

by GD-KDE are more uniform, having a set of 45 features that are changed in almost every attack. The remaining 23 features are rarely modified, most likely due to the opposite direction of change (recall that 35 features are only incrementable in our setup) or due to the infeasibility of the requested change. The changes effected by the mimicry attack exhibit higher variability of support. It is also interesting to observe that direct modifications are accompanied by an almost balanced amount of indirect modifications. This serves as another example for the high interdependency of PDFRATE's features.

A practical way to interpret attacks is to observe concrete changes in feature values produced by the attacks. Although it does not scale to cases with many features and files, this kind of investigation provides deep insight into the *modus operandi* of the attack at hand. Table 5.1 shows how the features of one specific file changed in the GD-KDE attack. Recall that GD-KDE operates by steering malicious data points across the decision boundary into the benign area using gradient descent, and at the same time utilizes kernel density estimation to push them towards seen benign samples. This "benignization" is evident in the provided example. By comparing the F_{before} and P_{attack} columns, we see that the attack has added an author (`author*` features), set the creation (`createdate_ts`) and modification (`moddate_ts`) date into recent past, reduced JavaScript occurrences (`count_javascript`), added some pages (`count_page`), fonts (`count_font`), images (`count_image*`), etc. – all changes towards the benign class. Some features, e.g., `size` and `version`, were changed to invalid values, possibly due to the influence of the gradient descent component.

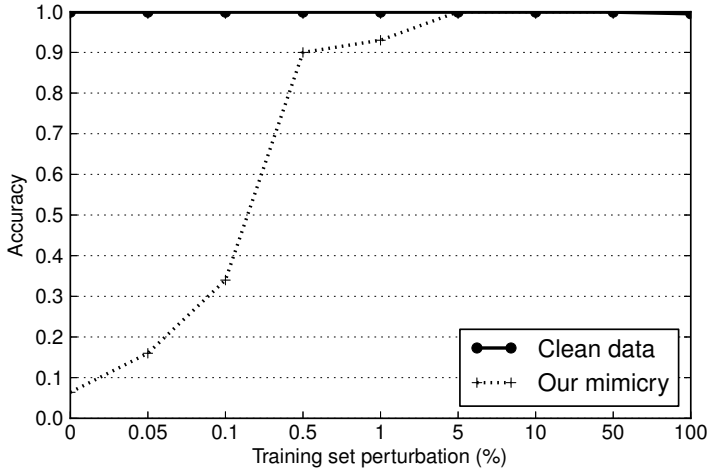


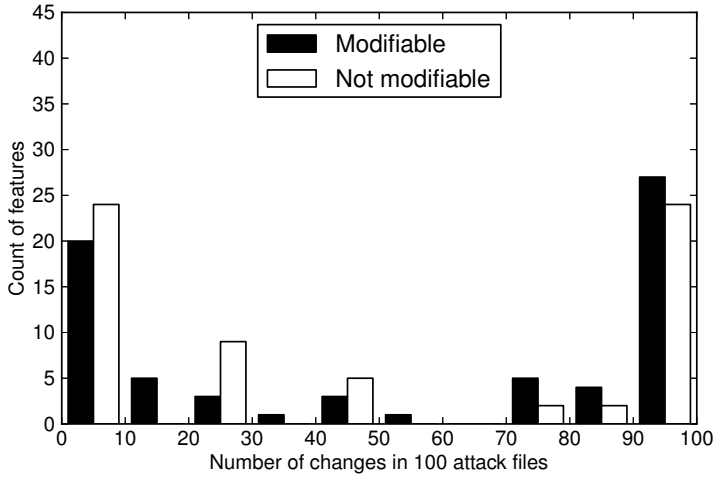
Figure 5.9: Effect of the defensive measure proposed in [85] on our mimicry attack seeded with mimicry attack samples. Averaged results of 5 independent trials.

5.7 Discussion

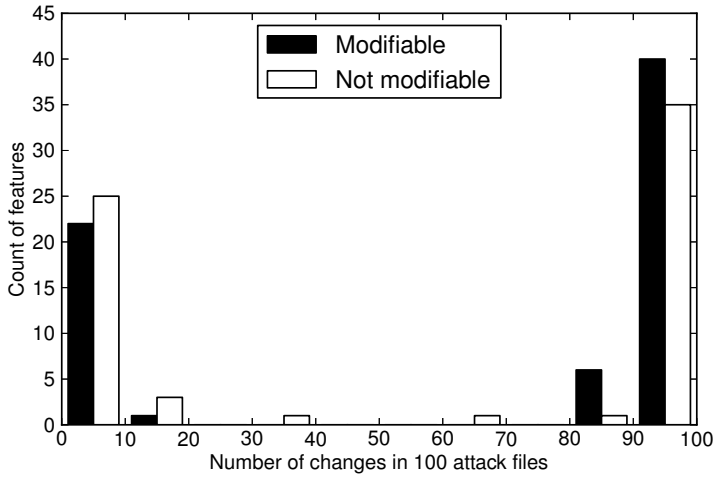
While the results of our study are only applicable to a single system, our findings suggest several important implications. It was the first attempt to perform a comprehensive practical evaluation of a deployed learning-based system, hence we cannot expect that our results can be exactly reproduced for a large number of similar systems. Still, some key issues revealed by our experiments deserve careful consideration, as they pinpoint some general problems that need to be addressed in the design of future data-driven systems.

The main message of our experiments is that an attacker can significantly decrease the accuracy of a learning-based system if he has sufficient knowledge of its features and methods. The main factor that contributes to this insecurity is the knowledge of features. For PDFrate we have observed that even the simplest attack from our arsenal, with no further knowledge of the system except for its features, can reduce maliciousness scores for a chosen representative set of malicious samples from 100 % to the median of 33 %. This was possible despite the fact that roughly $1/3$ of the classifier's features was completely unknown to us and another $1/3$ not modifiable by our tools. Such an impact suggests that even a small amount of knowledge about the features can be exploited to stage evasion attacks. Additional factors such as knowledge of training data and the precise type of the algorithm are helpful but not crucial for the attack, as this information can be approximated well by surrogate sources.

The fundamental problem underlying the insecurity of learning-based approaches lies in the design of features. The growing popularity of machine learning in various kinds of information systems – far beyond security – is largely due to its ability to predict, with greater or lesser success, *causes* from *side effects*. It is this generalization ability that makes machine learning algorithms the means of choice for finding solutions to



(a) GD-KDE attack in scenario FT.



(b) Mimicry attack in scenario FTC.

Figure 5.10: Histograms of the number of features whose values were modified grouped by the number of attack samples (from a total of 100 attacks) in which the modifications occurred.

problems shrouded by uncertainty, when one has neither enough understanding of the problem to *design* a solution, nor can figure it out from looking at the raw data. The prevailing approach for designing features for learning algorithms by hand-picking a set of easily computable side effects, or "expert features", obviously has the peril that the attacker may do exactly the same.

Are there alternative solutions that can make learning methods more robust to evasion? One potential solution is to use features that inherently represent, at least to a reasonable degree, the *causes* to be detected. One example of such features can be found in previous work on shellcode detection and classification, e.g., [43, 78, 84, 106], which uses n-grams, or short byte sequences, as basic features. Similar approach has been explored for detection of JavaScript malware, with the same techniques applied to sequences of syntactic tokens [45, 80]. The discriminative power of these methods lies in the inherent statistical difference between shellcode and usual packet content, as well as between malicious JavaScript code and benign programs. Hence one can expect such features to be less prone to malicious manipulation than "expert features". If fact, it has been shown that exact evasion of n-gram-based features is NP-complete [30], and approximate solutions are widely believed to be difficult in practice.

Another potential solution can be offered by methods attempting to uniformly spread the "discriminative power" across as many features as possible. Some methods of this kind have been proposed for learning on problems with potential feature deletion or corruption [24, 34]. At the cost of a significant increase in the complexity of training problems, such methods offer a reasonable protection for limited amount of feature noise, regardless of the type of features. Assuming that the attacker has modification access to a limited number of features, as it was the case in our study, one can expect such methods to deliver a good trade-off for the cases when no "intrinsic features" can be devised.

Finally, methods based on multiple classifier systems [81] should be mentioned as a potential solution. Evading a number of complementary classifiers can be significantly harder than a single classifier. Some applications of multiple classifier systems in security and other adversarial scenarios have been considered [5, 70].

5.7.1 Later Work

The publication of Mimicus [92] was followed by a number of interesting related works. The authors of PDFrate have recently reproduced our results and published a method for detecting evasion attempts against ensemble classifiers [87]. It follows the intuition that an ensemble decision with a high disagreement among its members is indicative of an evasion attempt. This is implemented by means of an adjustable threshold on the ensemble decision, e.g., labeling samples with less than 25 % positive votes as negative, those above 75 % as positive and the rest as uncertain. An evaluation on a large-scale benign PDF dataset produced around 1.6 % uncertain labels. The authors also attacked their method using Mimicus in all our scenarios with a success rate of 7 % in scenarios F and FC, 10 % in FT and 33 % in FTC, the majority of remaining attack samples labeled

uncertain. We find the method relatively effective in "trapping" attack samples (except in the difficult case of scenario FTC). However, we see the effective redefinition of class labels as potentially problematic when it comes to labeling benign samples. The results show an almost perfect ensemble consensus on this data. However, it is reasonable to expect that it would drop in a more realistic experiment than the employed randomized training-evaluation dataset split[12]. The reason for this lies in the natural emergence of concept drift when the dimension of time is introduced into such a problem. Then, the PDF dataset becomes non-stationary due to normal changes in time, e.g., with new versions of PDF writers. In that case the number of false uncertain labels for benign data would rise, in turn raising the number of costly detailed examinations required for uncertain files. In fact, ensemble disagreement might be useful for concept drift detection and quantification.

Thanks to its open-source availability, Xu et al. were able to utilize our system Mimicus to develop a new attack against PDFrate in their evasion study [114], as we describe in Section 4.8.

Wang et al. evaluated the resilience of multiple algorithms, e.g., Random Forest, SVM, J48 decision tree and Bayesian probabilistic models, against both causative (data poisoning) and exploratory (evasion) attacks in the context of detecting malicious crowdsourcing workers [105]. All algorithms were found to be vulnerable in their experiments. Furthermore, they show that more accurate models consistently show higher vulnerability to attacks. Their finding that attacks are much more effective given complete knowledge about the target classifier corroborates our experimental results, although in our study the difference is less pronounced. Finally, in their experiments Random Forest was the only classifier that in some cases achieved both high accuracy and relative robustness against attacks. As this finding contradicts the results of our case study we see the need to further investigate the limits of Random Forest robustness in future work.

A recent study by Liang et al. [49] demonstrates a highly successful, practical evasion attack against Google Chrome's phishing pages filter (GPPF). Starting from what in our taxonomy would be scenario O, i.e., without any information about the classifier, they perform reverse engineering on the client-side component of GPPF to recover information about classifier type (logistic regression) and manage to recover 85 % of the 2,130 scoring rules based on 1,009 features. Moving from black box to scenario FC, they were able to defeat the classifier on a dataset of 100 recent phishing pages by modifying on average less than 4 features. This study is an example how evasion scenarios outside the 4 evaluated in this work can be reduced to one of them by first inferring the feature set.

[12] For example, with periodic retraining using temporally consistent samples as in our evaluation of Hidost in the previous chapter or using temporally consistent labels as suggested in [58].

5.8 Conclusions

In this chapter, we have presented the first empirical security evaluation of a deployed learning-based system. Our study assumed that an attacker has no specific insider information about the system. It demonstrated, however, that enough information can be gathered from various sources and extended with approximations and automatic inference algorithms in order to stage a successful evasion attack.

We presented a taxonomy of evasion attacks against machine learning classifiers based on the knowledge about 3 classifier components – training dataset, classifier type and feature set – focusing on scenarios with known feature sets. In our experiments carried out on an established system for detection of PDF malware, PDFrate, the significant drop in classification scores (from almost 100 % down to 28 % to 33 %) as well as deterioration of detection rates has been observed.

Decisions made when building and deploying classifiers strongly affect their robustness. Considerations such as the potential for adversarial influence on training or evaluation data, choice of features, level of knowledge about the system available to the adversary and many other ones need to be seriously evaluated. We have observed that simple countermeasures against evasion attacks, such as including a small fraction of attacks in the training data, are only effective if the anticipated attack exactly matches the performed one. Recent work has shown even established black box classifiers to be vulnerable to reverse engineering and subsequent evasion.

Our evaluation methods are applicable to other learning-based systems with modifiable features. The open availability of Mimicus source code has already facilitated other studies and continues to guarantee reproducibility of our results.

Chapter 6

Summary and Conclusions

This thesis presented a study of adversarial machine learning in the *context of static malware detection in non-executable file formats*. It identified 3 main criteria for a successful deployment of machine learning applications in this context: *effectiveness*, *efficiency* and *security*, and evaluated them experimentally. To this end, the thesis introduced 3 novel machine-learning-based methods for detection of malware in different file formats and measured their effectiveness and efficiency. Furthermore, it proposed a framework for the evaluation of security of machine learning applications.

The first two methods, PJScan and Hidost, detect malicious PDF files. PJScan performs anomaly detection based on lexical properties of embedded JavaScript code and is, therefore, limited to PDF files containing JavaScript. SL2013 introduced a novel set of features based on the hierarchical logical internal structure of PDF files and can be applied to any PDF file. The third method, Hidost, generalizes the feature definition of SL2013 to formats beyond PDF. It was successfully implemented and evaluated on malicious PDF and SWF files and is extensible to other hierarchically structured formats, e.g., XML, HTML, SVG, OOXML and ODF. Mimicus, a novel framework for the security evaluation of machine learning applications was presented and used in an experimental evaluation of an independently published, actively deployed PDF malware detector, PDFrate.

Of the 4 published methods, PJScan, Hidost and Mimicus were released as open-source software. Furthermore, source code for experiment reproduction for Hidost and Mimicus was also released as open-source software, and datasets used for evaluation of Hidost were made available for download in an effort to ensure reproducibility of results published in this thesis.

The 3 introduced detection methods were developed in a strictly data-driven fashion. All employed features are well-defined properties of files, extracted directly or with minor transformations. This is in contrast with features based on expert heuristics, i.e., manually defined properties that indicate benign or malicious intent, used by some related work [32, 85]. Intuitively, data-driven features provide a more objective and general characterization of data and remain relatively stable in time, while expert heuristics capture concrete anomalies better but have to be updated for novel attacks. The recent advances in deep neural networks that go one step further and automatically learn sig-

nificant features out of raw data have demonstrated the power of strictly data-driven methods [46]. A crucial requirement for data-driven methods is the availability of a sufficient quantity of high-quality data for learning. The experimental evaluations of the presented detection methods were performed on datasets whose size shadows all previously published work. Data quality is validated through the use of VirusTotal, the data source with the highest quality available to researchers, and long-term data acquisition, i.e., weeks to years per dataset.

In the following we present the main results of this thesis with respect to the 3 main criteria for successful deployment of machine learning methods in our context.

Effectiveness

Effectiveness expresses the detection performance, i.e., the ability to correctly classify previously unseen samples. All presented detection methods have achieved breakthroughs in effectiveness compared to prior work.

Subsequently to its publication as the first fully static PDF malware detector evaluated on a large data corpus, PJScan was compared in numerous studies and found to outperform prior work in terms of detection performance. Its model was able to generalize and retain its accuracy even when evaluated on malware collected years after its training. In the following period, however, a large number of detectors were published of which some were able to surpass PJScan by a large margin. Nevertheless, SL2013 was shown to be even better. It was the first method to be directly compared to antivirus engines, where it ranked among the best. Its excellent detection was only recently matched by Hidost, ranking better than all antivirus engines on PDF data and among the top on SWF data. It is the first static machine-learning-based malware detector applicable to multiple non-executable file formats.

Our experimental evaluation pioneered a novel approach in measuring effectiveness, combining comparison to antivirus engines and a simulated long-term operational deployment. The *status quo ante* in security experiment design involving machine learning was a randomized training-test split or cross-validation, with the comparisons limited to other academic methods. By comparing a method to state-of-the-art practical commercial solutions, the new approach puts it into a wider perspective. Furthermore, a long-term simulated operational deployment with temporally consistent samples ensures the method's evaluation is performed exclusively on novel data. Together, these elements combine to make experimental conditions almost perfectly realistic (the only missing condition is temporal consistency of labels [58]).

Given the demonstrated detection performance of our methods that come close to or even outperform all antivirus engines under realistic experimental conditions, we conclude that machine learning methods are highly effective in the context of static malware detection in non-executable file formats.

Efficiency

We measure efficiency as the amount of computing resources required to process data for detection. In our experiments we measured throughput (processed bits per second) and file processing time.

In machine learning one can distinguish between training and prediction tasks and measure their efficiency separately. However, in the common case of batch learning, e.g., in our 3 methods, only the prediction task is perceived by users as causing latency – training is performed independently ahead of time and causes no delay. In online learning the 2 tasks cannot be decoupled.

All detection methods presented in this thesis are static. Compared to dynamic prior work, they usually run orders of magnitude faster. As reported in our own and independent publications, PJScan is the fastest among all academic methods published to date. Measured on a commodity PC from 2010 in a 2-fold cross-validation experiment using approximately 65,000 (60 GB) PDF files, it achieves a combined training and prediction time of 1547 s, resulting in a throughput of approximately 300 Mbit/s or 23 ms per file on average. Our analysis shows that it was severely limited by hard drive throughput. Feature extraction amounted to almost 90 % of its time, while prediction only took around 15 µs. SL2013 performed around 20 % slower, nevertheless much faster than other work, while the efficiency of Hidost was not measured.

Unfortunately, we have no data on the efficiency of static components of antivirus engines and therefore cannot compare directly. Nevertheless, PJScan and SL2013 demonstrate excellent efficiency of machine learning methods in this context.

Security

In adversarial machine learning, security denotes the resistance of a machine learning algorithm to adversarial manipulation. Attacks are categorized into *causative*, where the adversary manipulates training data, and *exploratory*, where only test samples are manipulated [2]. This thesis described independent third-party security evaluations of the presented methods, where available, and our own security evaluation of an independently published machine learning method, PDFrate.

PJScan was evaluated in multiple studies. In a causative attack, Cao and Yang implement successful *poisoning* of training data [15], drastically reducing accuracy. Subsequently they design a defensive measure called *machine unlearning* and implement it as an extension to PJScan in just 30 lines of code, completely restoring its initial detection accuracy. Others have performed exploratory attacks, exploiting PJScan's inability to extract JavaScript from XFA forms and arbitrary PDF strings. However, Carmony et al. show that implementing part of the missing functionality boosts PJScan's true positive rate in their experiment from 68.34 % to over 94 % [16].

SL2013 was targeted in 2 exploratory attacks. In the reverse mimicry attack [57], malware in form of PE files, JavaScript or PDF files is injected into an otherwise benign

PDF file. Our method was not evaluated against this attack but we proposed a set of enhancements to mitigate it. Xu et al. used Genetic Programming to evolve evading samples from an initial set of malicious PDF files [114]. However, their evaluation is limited to malware exploiting one of only 2 CVEs targeting a single version of Acrobat Reader. Furthermore, the trained model of SL2013 used in the evaluation had poor detection performance on the data even before the attack. The strongest attack against SL2013 was our own mimicry attack that completely defeated the method. However, that attack represents the worst case and was performed only in feature space. HIDOST's security was not evaluated to date.

Our own evaluation of the third-party deployed PDF malware detector PDFRATE using our framework MIMICUS attempted to estimate its robustness from a broader perspective. To this end, we developed a taxonomy of exploratory attacks based on the level of knowledge available to adversaries about the target system. We identified the feature set, training dataset and classifier details as the main components of machine learning based malware detectors. Each of the 8 combinations of high or low knowledge about these 3 components is called an evasion scenario. We limited our analysis to the 4 scenarios in which the feature set is known, guided by the idea that the remaining scenarios can be reduced to one of these 4 if the feature set can be deduced in some way. Our results show that an attacker can significantly degrade the accuracy of a learning-based system if he has sufficient knowledge about the system. The main factor that contributes to this insecurity is the knowledge of features, while the remaining 2 components can be approximated well from surrogate sources. However, even without access to any of the components, it was practically demonstrated in related work that security through obscurity does little more than raise the cost of exploitation [49].

The main findings presented above indicate the need for substantial future research in the security of machine learning applications for malware detection. Adversarial machine learning is a young discipline. The state of the art lacks a precise and comprehensive method for security evaluation and the theoretical and practical limits remain largely unknown. Consequently, a method is demonstrated to be insecure when a vulnerability is discovered, but there exists no mechanism to prove its security in the absence of vulnerabilities. However negative, this result does not present a fundamental limitation to the applicability of machine learning methods in this context. In fact, it is true for most other areas of security as well. Until a provably secure method can be developed, the common approach is to evolve the methods in reaction to novel attacks. A good example is PJSCAN which was independently successfully attacked and improved twice after its publication.

In the long term, more understanding is required about the relationships between effectiveness, efficiency and security. Does there fundamentally exist a trade-off between security and effectiveness as implied by a lot of existing work? Can machine learning be made provably secure, similar to cryptography? Would it still remain efficient? Finding an answer to these and other questions exploring the theoretical and practical limits of adversarial machine learning is an indispensable step towards its final goal – the devel-

opment of effective, efficient and secure methods.

Appendix A

PDFrate Feature Reimplementation

The Mimicus experimental framework supports reading of 135 PDFrate features (66%) described in [86]. The remaining 67 of 202 features were not disclosed. Modification of values of the following 68 features (33%) is supported:

- Features whose value can only be incremented (33):

  count_acroform
  count_acroform_obs
  count_action
  count_action_obs
  count_box_a4
  count_box_legal
  count_box_letter
  count_box_other
  count_box_overlap
  count_endobj
  count_endstream
  count_eof
  count_font
  count_font_obs
  count_image_large
  count_image_med
  count_image_small
  count_image_xlarge
  count_image_xsmall
  count_javascript
  count_javascript_obs
  count_js
  count_js_obs
  count_obj
  count_objstm
  count_objstm_obs
  count_page
  count_page_obs
  count_startxref
  count_stream
  count_trailer
  count_xref
  size

- Features whose value can be both incremented and decremented (35):

  author_dot
  author_lc
  author_num
  author_oth
  author_uc
  createdate_ts
  createdate_tz
  creator_dot
  creator_lc
  creator_num
  creator_oth
  creator_uc
  keywords_dot
  keywords_lc
  keywords_num
  keywords_oth
  keywords_uc
  moddate_ts
  moddate_tz
  producer_dot
  producer_lc
  producer_num
  producer_oth
  producer_uc
  subject_dot
  subject_lc
  subject_num
  subject_oth
  subject_uc
  title_dot
  title_lc
  title_num
  title_oth
  title_uc
  version

Abbreviations

API	Application Programming Interface
CVE	Common Vulnerabilities and Exposures
HTML	HyperText Markup Language
JS	JavaScript
ODF	Open Document Format
OOXML	Office Open XML
PDF	Portable Document Format
PE	Portable Executable
SVG	Scalable Vector Graphics
SVM	Support Vector Machine
XFA	Adobe XML Forms Architecture
XML	Extensible Markup Language

Bibliography

[1] Periklis Akritidis, Evangelos P. Markatos, Michalis Polychronakis, and Kostas G. Anagnostakis. STRIDE: polymorphic sled detection through instruction sequence analysis. In Ryôichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan*, volume 181 of *IFIP*, pages 375–392. Springer, 2005.

[2] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In Ferng-Ching Lin, Der-Tsai Lee, Bao-Shuh Paul Lin, Shiuhpyng Shieh, and Sushil Jajodia, editors, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, March 21-24, 2006*, pages 16–25. ACM, 2006. doi: 10.1145/1128817.1128824. URL http://doi.acm.org/10.1145/1128817.1128824.

[3] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010. doi: 10.1007/s10994-010-5188-5. URL http://dx.doi.org/10.1007/s10994-010-5188-5.

[4] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, 2009. URL http://www.isoc.org/isoc/conferences/ndss/09/pdf/11.pdf.

[5] Battista Biggio, Giorgio Fumera, and Fabio Roli. Multiple classifier systems for adversarial classification tasks. In Jon Atli Benediktsson, Josef Kittler, and Fabio Roli, editors, *Multiple Classifier Systems, 8th International Workshop, MCS 2009, Reykjavik, Iceland, June 10-12, 2009. Proceedings*, volume 5519 of *Lecture Notes in Computer Science*, pages 132–141. Springer, 2009. doi: 10.1007/978-3-642-02326-2_14. URL http://dx.doi.org/10.1007/978-3-642-02326-2_14.

[6] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012. URL http://icml.cc/2012/papers/880.pdf.

[7] 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 Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Zelezný, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III*, volume 8190 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2013. doi: 10.1007/978-3-642-40994-3_25. URL http://dx.doi.org/10.1007/978-3-642-40994-3_25.

[8] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE Trans. Knowl. Data Eng.*, 26(4):984–996, 2014. doi: 10.1109/TKDE.2013.57. URL http://dx.doi.org/10.1109/TKDE.2013.57.

[9] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.

[10] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi: 10.1023/A:1010933404324. URL http://dx.doi.org/10.1023/A:1010933404324.

[11] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN 0-534-98053-8.

[12] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Trans. Inf. Syst. Secur.*, 15(3):12:1–12:22, 2012. doi: 10.1145/2382448.2382450. URL http://doi.acm.org/10.1145/2382448.2382450.

[13] Michael Brückner, Christian Kanzow, and Tobias Scheffer. Static prediction games for adversarial learning problems. *Journal of Machine Learning Research*, 13:2617–2654, 2012. URL http://dl.acm.org/citation.cfm?id=2503326.

[14] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 197–206. ACM, 2011. doi: 10.1145/1963405.1963436. URL http://doi.acm.org/10.1145/1963405.1963436.

[15] Yinzhi Cao and Junfeng Yang. Towards making systems forget with machine unlearning. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 463–480. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.35. URL http://dx.doi.org/10.1109/SP.2015.35.

[16] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing PDF parsers in malware detectors. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL http://www.internetsociety.org/sites/default/files/blogs-media/extract-me-if-you-can-abusing-pdf-parsers-malware-detectors.pdf.

[17] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM TIST*, 2(3):27:1–27:27, 2011. doi: 10.1145/1961189.1961199. URL http://doi.acm.org/10.1145/1961189.1961199.

[18] Cisco. 2014 annual security report, 2014. URL http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf. Accessed 13 Apr 2015.

[19] William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart J. Russell, editors, *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning, Tahoe City, California, USA, July 9-12, 1995*, pages 115–123. Morgan Kaufmann, 1995.

[20] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of API references. In Christos Dimitrakakis, Aikaterini Mitrokotsa, Benjamin I. P. Rubinstein, and Gail-Joon Ahn, editors, *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop, AISec 2014, Scottsdale, AZ, USA, November 7, 2014*, pages 47–57. ACM, 2014. doi: 10.1145/2666652.2666657. URL http://doi.acm.org/10.1145/2666652.2666657.

[21] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. doi: 10.1007/BF00994018. URL http://dx.doi.org/10.1007/BF00994018.

[22] Marco Cova, Christopher Krügel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 281–290. ACM, 2010. doi: 10.1145/1772690.1772720. URL http://doi.acm.org/10.1145/1772690.1772720.

[23] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. ZOZZLE: fast and precise in-browser javascript malware detection. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL http://static.usenix.org/events/sec11/tech/full_papers/Curtsinger.pdf.

[24] Ofer Dekel, Ohad Shamir, and Lin Xiao. Learning to classify with missing and corrupted features. *Machine Learning*, 81(2):149–178, 2010. doi: 10.1007/s10994-009-5124-8. URL http://dx.doi.org/10.1007/s10994-009-5124-8.

[25] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. Adsandbox: sandboxing javascript to fight malicious websites. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 1859–1864. ACM, 2010. doi: 10.1145/1774088.1774482. URL http://doi.acm.org/10.1145/1774088.1774482.

[26] Cynthia Dwork. Differential privacy: A survey of results. In Manindra Agrawal, Ding-Zhu Du, Zhenhua Duan, and Angsheng Li, editors, *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, volume 4978 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2008. doi: 10.1007/978-3-540-79228-4_1. URL http://dx.doi.org/10.1007/978-3-540-79228-4_1.

[27] Manuel Egele, Gianluca Stringhini, Christopher Krügel, and Giovanni Vigna. COMPA: detecting compromised accounts on social networks. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. URL http://internetsociety.org/doc/compa-detecting-compromised-accounts-social-networks.

[28] M. Engelberth, C. Willems, and Holz. T. MalOffice – analysis of various application data files. In *Virus Bulletin International Conference*, 2009.

[29] B. Feinstein and D. Peck. Caffeine Monkey: Automated collection, detection and analysis of malicious JavaScript. In *Black Hat USA*, 2007.

[30] Prahlad Fogla and Wenke Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 59–68. ACM, 2006. doi: 10.1145/1180405.1180414. URL http://doi.acm.org/10.1145/1180405.1180414.

[31] Prahlad Fogla, Monirul I. Sharif, Roberto Perdisci, Oleg M. Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006. URL https://www.usenix.org/conference/15th-usenix-security-symposium/polymorphic-blending-attacks.

[32] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious flash advertisements. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 363–372. IEEE Computer Society, 2009. doi: 10.1109/ACSAC.2009.41. URL http://dx.doi.org/10.1109/ACSAC.2009.41.

[33] David Mandell Freeman. Using naive bayes to detect spammy names in social networks. In *AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, pages 3–12, 2013.

[34] Amir Globerson and Sam T. Roweis. Nightmare at test time: robust learning by feature deletion. In William W. Cohen and Andrew Moore, editors, *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148 of *ACM International Conference Proceeding Series*, pages 353–360. ACM, 2006. doi: 10.1145/1143844.1143889. URL http://doi.acm.org/10.1145/1143844.1143889.

[35] Guofei Gu, Phillip A. Porras, Vinod Yegneswaran, and Martin W. Fong. Bothunter: Detecting malware infection through ids-driven dialog correlation. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007. URL https://www.usenix.org/conference/16th-usenix-security-symposium/bothunter-detecting-malware-infection-through-ids-driven.

[36] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: data mining, inference and prediction*. Springer series in statistics. Springer, New York, N.Y., 2009. 2nd edition.

[37] K. Itabashi. Portable document format malware. Symantec white paper, 2011.

[38] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 80–94. IEEE Computer Society, 2012. doi: 10.1109/SP.2012.15. URL http://dx.doi.org/10.1109/SP.2012.15.

[39] Quentin Jerome, Samuel Marchal, Thomas Engel, et al. Advanced detection tool for pdf threats. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 300–315. Springer, 2014.

[40] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In Ahmad-Reza Sadeghi, Blaine Nelson, Christos Dimitrakakis, and Elaine Shi, editors, *AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, pages 99–110. ACM, 2013. doi: 10.1145/2517312.2517320. URL http://doi.acm.org/10.1145/2517312.2517320.

[41] M. Kearns and M. Li. Learning in the presence of malicious errors. *SIAM Journal on Computing*, 22(4):807–837, 1993.

[42] Marius Kloft and Pavel Laskov. Security analysis of online centroid anomaly detection. *Journal of Machine Learning Research*, 13:3681–3724, 2012. URL http://dl.acm.org/citation.cfm?id=2503359.

[43] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 6:2721–2744, 2006. URL http://www.jmlr.org/papers/v7/kolter06a.html.

[44] Pavel Laskov and Marius Kloft. A framework for quantitative security analysis of machine learning. In Dirk Balfanz and Jessica Staddon, editors, *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence, AISec 2009, Chicago, Illinois, USA, November 9, 2009*, pages 1–4. ACM, 2009. doi: 10.1145/1654988.1654990. URL http://doi.acm.org/10.1145/1654988.1654990.

[45] Pavel Laskov and Nedim Šrndić. Static detection of malicious javascript-bearing PDF documents. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 373–382. ACM, 2011. doi: 10.1145/2076732.2076785. URL http://doi.acm.org/10.1145/2076732.2076785.

[46] Quoc V. Le. Building high-level features using large scale unsupervised learning. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 8595–8598. IEEE, 2013. doi: 10.1109/ICASSP.2013.6639343. URL http://dx.doi.org/10.1109/ICASSP.2013.6639343.

[47] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. A data mining framework for building intrusion detection models. In *1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9-12, 1999*, pages 120–132. IEEE Computer Society, 1999. doi: 10.1109/SECPRI.1999.766909. URL http://dx.doi.org/10.1109/SECPRI.1999.766909.

[48] Wei-Jen Li, Salvatore J. Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis. A study of malcode-bearing documents. In Bernhard M. Hämmerli and Robin Sommer, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 4th International Conference, DIMVA 2007, Lucerne, Switzerland, July 12-13, 2007, Proceedings*, volume 4579 of *Lecture Notes in Computer Science*, pages 231–250. Springer, 2007. doi: 10.1007/978-3-540-73614-1_14. URL http://dx.doi.org/10.1007/978-3-540-73614-1_14.

[49] Bin Liang, Miaoqiang Su, Wei You, Wenchang Shi, and Gang Yang. Cracking classifiers for evasion: A case study on the google's phishing pages filter. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 345–356. ACM, 2016. doi: 10.1145/2872427.2883060. URL http://doi.acm.org/10.1145/2872427.2883060.

[50] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *J. Cryptology*, 15(3):177–206, 2002. doi: 10.1007/s00145-001-0019-2. URL http://dx.doi.org/10.1007/s00145-001-0019-2.

[51] Daiping Liu, Haining Wang, and Angelos Stavrou. Detecting malicious javascript in PDF through document instrumentation. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 100–111. IEEE Computer Society, 2014. doi: 10.1109/DSN.2014.92. URL http://dx.doi.org/10.1109/DSN.2014.92.

[52] Daniel Lowd and Christopher Meek. Good word attacks on statistical spam filters. In *CEAS 2005 - Second Conference on Email and Anti-Spam, July 21-22, 2005, Stanford University, California, USA*, 2005. URL http://www.ceas.cc/papers-2005/125.pdf.

[53] Xun Lu, Jianwei Zhuge, Ruoyu Wang, Yinzhi Cao, and Yan Chen. De-obfuscation and detection of malicious PDF files with high accuracy. In *46th Hawaii International Conference on System Sciences, HICSS 2013, Wailea, HI, USA, January 7-10, 2013*, pages 4890–4899. IEEE Computer Society, 2013. doi: 10.1109/HICSS.2013.166. URL http://dx.doi.org/10.1109/HICSS.2013.166.

[54] Michael Maass, William L. Scherlis, and Jonathan Aldrich. In-nimbo sandboxing. In Laurie A. Williams, David M. Nicol, and Munindar P. Singh, editors, *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security, HotSoS 2014, Raleigh, NC, USA, April 08 - 09, 2014*, page 1. ACM, 2014. doi: 10.1145/2600176.2600177. URL http://doi.acm.org/10.1145/2600176.2600177.

[55] Matthew V. Mahoney and Philip K. Chan. Learning rules for anomaly detection of hostile network traffic. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003), 19-22 December 2003, Melbourne, Florida, USA*, pages 601–604. IEEE Computer Society, 2003. doi: 10.1109/ICDM.2003.1250987. URL http://dx.doi.org/10.1109/ICDM.2003.1250987.

[56] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A pattern recognition system for malicious PDF files detection. In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition - 8th International Conference, MLDM 2012, Berlin, Germany, July 13-20, 2012. Proceedings*, volume 7376 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2012. doi: 10.1007/978-3-642-31537-4_40. URL http://dx.doi.org/10.1007/978-3-642-31537-4_40.

[57] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious PDF files detection. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 119–130. ACM, 2013. doi: 10.1145/2484313.2484327. URL http://doi.acm.org/10.1145/2484313.2484327.

[58] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, Anthony D. Joseph, and J. D. Tygar. Reviewer integration and performance measurement for malware detection. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 2016. doi: 10.1007/978-3-319-40667-1_7. URL http://dx.doi.org/10.1007/978-3-319-40667-1_7.

[59] Jose Nazario. Phoneyc: A virtual client honeypot. In Wenke Lee, editor, *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '09, Boston, MA, USA, April 21, 2009*. USENIX Association, 2009. URL https://www.usenix.org/conference/leet-09/phoneyc-virtual-client-honeypot.

[60] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D. Joseph, Benjamin I. P. Rubinstein, Udam Saini, Charles A. Sutton, J. Doug Tygar, and Kai Xia. Exploiting machine learning to subvert your spam filter. In Fabian Monrose, editor, *First USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '08, San Francisco, CA, USA, April 15, 2008, Proceedings*. USENIX Association,

2008. URL http://www.usenix.org/events/leet08/tech/full_papers/nelson/nelson.pdf.

[61] Nir Nissim, Aviad Cohen, Robert Moskovitch, Asaf Shabtai, Mattan Edry, Oren Bar-Ad, and Yuval Elovici. ALPD: active learning framework for enhancing the detection of malicious PDF files. In *IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014, The Hague, The Netherlands, 24-26 September, 2014*, pages 91–98. IEEE, 2014. doi: 10.1109/JISIC.2014.23. URL http://dx.doi.org/10.1109/JISIC.2014.23.

[62] Nir Nissim, Aviad Cohen, Chanan Glezer, and Yuval Elovici. Detection of malicious PDF files and directions for enhancements: A state-of-the art survey. *Computers & Security*, 48:246–266, 2015. doi: 10.1016/j.cose.2014.10.014. URL http://dx.doi.org/10.1016/j.cose.2014.10.014.

[63] Nir Nissim, Aviad Cohen, Robert Moskovitch, Asaf Shabtai, Matan Edri, Oren Bar-Ad, and Yuval Elovici. Keeping pace with the creation of new malicious PDF files using an active-learning based detection framework. *Security Informatics*, 5 (1):1, 2016. doi: 10.1186/s13388-016-0026-3. URL http://dx.doi.org/10.1186/s13388-016-0026-3.

[64] Gabriel Oberreuter, Gaston L'Huillier, Sebastián A. Ríos, and Juan D. Velásquez. Outlier-based approaches for intrinsic and external plagiarism detection. In Andreas König, Andreas Dengel, Knut Hinkelmann, Koichi Kise, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowlege-Based and Intelligent Information and Engineering Systems - 15th International Conference, KES 2011, Kaiserslautern, Germany, September 12-14, 2011, Proceedings, Part II*, volume 6882 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2011. doi: 10.1007/978-3-642-23863-5_2. URL http://dx.doi.org/10.1007/978-3-642-23863-5_2.

[65] Timon Van Overveldt, Christopher Kruegel, and Giovanni Vigna. Flashdetect: Actionscript 3 malware detection. In Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings*, volume 7462 of *Lecture Notes in Computer Science*, pages 274–293. Springer, 2012. doi: 10.1007/978-3-642-33338-5_14. URL http://dx.doi.org/10.1007/978-3-642-33338-5_14.

[66] Emanuel Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[67] PDFReference. Document management - Portable document format - Part 1: PDF 1.7, 2008. URL https://www.adobe.com/devnet/pdf/pdf_reference.html. Accessed 23 Jan 2015.

[68] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL http://dl.acm.org/citation.cfm?id=2078195.

[69] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M.I. Sharif. Misleading worm signature generators using deliberate noise injection. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 17–31, 2006.

[70] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53(6):864–881, 2009. doi: 10.1016/j.comnet.2008.11.011. URL http://dx.doi.org/10.1016/j.comnet.2008.11.011.

[71] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 287–296. ACM, 2010. doi: 10.1145/1920261.1920305. URL http://doi.acm.org/10.1145/1920261.1920305.

[72] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 1–16. USENIX Association, 2008. URL http://www.usenix.org/events/sec08/tech/full_papers/provos/provos.pdf.

[73] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN 1-55860-238-0.

[74] Babak Rahbarinia, Roberto Perdisci, Andrea Lanzi, and Kang Li. Peerrush: Mining for unwanted P2P traffic. In Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings*, volume 7967 of *Lecture Notes in Computer Science*, pages 62–82. Springer, 2013. doi: 10.1007/978-3-642-39235-1_4. URL http://dx.doi.org/10.1007/978-3-642-39235-1_4.

[75] Moheeb Abu Rajab, Lucas Ballard, Noe Lutz, Panayiotis Mavrommatis, and Niels Provos. CAMP: content-agnostic malware protection. In *20th Annual Network and Distributed System Security Symposium, NDSS*

2013, San Diego, California, USA, February 24-27, 2013. The Internet Society, 2013. URL http://internetsociety.org/doc/camp-content-agnostic-malware-protection.

[76] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 169–186. USENIX Association, 2009. URL http://www.usenix.org/events/sec09/tech/full_papers/ratanaworabhan.pdf.

[77] Recorded Future. Gone in a flash: Top 10 vulnerabilities used by exploit kits, 2015. URL https://www.recordedfuture.com/top-vulnerabilities-2015/. Accessed 15 Nov 2015.

[78] Konrad Rieck and Pavel Laskov. Language models for detection of unknown attacks in network traffic. *Journal in Computer Virology*, 2(4):243–256, 2007. doi: 10.1007/s11416-006-0030-0. URL http://dx.doi.org/10.1007/s11416-006-0030-0.

[79] Konrad Rieck and Pavel Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9:23–48, 2008. doi: 10.1145/1390681.1390683. URL http://doi.acm.org/10.1145/1390681.1390683.

[80] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 31–39. ACM, 2010. doi: 10.1145/1920261.1920267. URL http://doi.acm.org/10.1145/1920261.1920267.

[81] Fabio Roli, Giorgio Giacinto, and Gianni Vernazza. Methods for designing multiple classifier systems. In Josef Kittler and Fabio Roli, editors, *Multiple Classifier Systems, Second International Workshop, MCS 2001 Cambridge, UK, July 2-4, 2001, Proceedings*, volume 2096 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2001. doi: 10.1007/3-540-48219-9_8. URL http://dx.doi.org/10.1007/3-540-48219-9_8.

[82] Florian Schmitt, Jan Gassen, and Elmar Gerhards-Padilla. PDF scrutinizer: Detecting javascript-based attacks in PDF documents. In Nora Cuppens-Boulahia, Philip Fong, Joaquín García-Alfaro, Stephen Marsh, and Jan-Philipp Steghöfer, editors, *Tenth Annual International Conference on Privacy, Security and Trust, PST 2012, Paris, France, July 16-18, 2012*, pages 104–111. IEEE Computer Society, 2012. doi: 10.1109/PST.2012.6297926. URL http://dx.doi.org/10.1109/PST.2012.6297926.

[83] D. Sculley, Matthew Eric Otey, Michael Pohl, Bridget Spitznagel, John Hainsworth, and Yunkai Zhou. Detecting adversarial advertisements in the wild. In Chid Apté, Joydeep Ghosh, and Padhraic Smyth, editors, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 274–282. ACM, 2011. doi: 10.1145/2020408.2020455. URL http://doi.acm.org/10.1145/2020408.2020455.

[84] M. Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In Diego Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings*, volume 5137 of *Lecture Notes in Computer Science*, pages 88–107. Springer, 2008. doi: 10.1007/978-3-540-70542-0_5. URL http://dx.doi.org/10.1007/978-3-540-70542-0_5.

[85] Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In Robert H'obbes' Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 239–248. ACM, 2012. doi: 10.1145/2420950.2420987. URL http://doi.acm.org/10.1145/2420950.2420987.

[86] Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. Technical Report GMU-CS-TR-2012-5, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA, 2012.

[87] Charles Smutz and Angelos Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL http://www.internetsociety.org/sites/default/files/blogs-media/when-tree-falls-using-diversity-ensemble-classifiers-identify-evasion-malware-detectors.pdf.

[88] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. SHELLOS: enabling fast detection and forensic analysis of code injection attacks. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL http://static.usenix.org/events/sec11/tech/full_papers/Snow.pdf.

[89] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*,

pages 305–316. IEEE Computer Society, 2010. doi: 10.1109/SP.2010.25. URL http://dx.doi.org/10.1109/SP.2010.25.

[90] Sophos. Security threat report 2014, 2014. URL http://www.sophos.com/en-us/medialibrary/pdfs/other/sophos-security-threat-report-2014.pdf. Accessed 13 Apr 2015.

[91] Nedim Šrndić and Pavel Laskov. Detection of malicious PDF files based on hierarchical document structure. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. URL http://internetsociety.org/doc/detection-malicious-pdf-files-based-hierarchical-document-structure.

[92] Nedim Šrndić and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 197–211. IEEE Computer Society, 2014. doi: 10.1109/SP.2014.20. URL http://dx.doi.org/10.1109/SP.2014.20.

[93] Nedim Šrndić and Pavel Laskov. Hidost: a static machine-learning-based detector of malicious files. *EURASIP J. Information Security*, 2016:22, 2016. doi: 10.1186/s13635-016-0045-0. URL http://dx.doi.org/10.1186/s13635-016-0045-0.

[94] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady paths: leveraging surfing crowds to detect malicious web pages. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 133–144. ACM, 2013. doi: 10.1145/2508859.2516682. URL http://doi.acm.org/10.1145/2508859.2516682.

[95] SWFSpec. Swf file format specification (version 19), 2012. URL https://www.adobe.com/devnet/swf.html. Accessed 23 Jan 2015.

[96] Symantec. Internet security threat report. Symantec, 2010.

[97] Symantec. 2014 internet security threat report, volume 19, 2014. URL https://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf. Accessed 13 Apr 2015.

[98] Symantec. 2014 internet security threat report, volume 19, appendix, 2014. URL https://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_appendices_v19_221284438.en-us.pdf. Accessed 13 Apr 2015.

[99] Symantec. 2015 internet security threat report, volume 20, 2015. URL http://know.symantec.com/LP=1123. Accessed 15 Apr 2015.

[100] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 109–129. Springer, 2014. doi: 10.1007/978-3-319-11379-1_6. URL http://dx.doi.org/10.1007/978-3-319-11379-1_6.

[101] David M. J. Tax and Robert P. W. Duin. Support vector data description. *Machine Learning*, 54(1):45–66, 2004. doi: 10.1023/B:MACH.0000008084.60811.49. URL http://dx.doi.org/10.1023/B:MACH.0000008084.60811.49.

[102] Olivier Thonnard. *Vers un regroupement multicritères comme outil d'aide à l'attribution d'attaque dans le cyber-espace. (A multi-criteria clustering approach to support attack attribution in cyberspace).* PhD thesis, Télécom ParisTech, France, 2010. URL https://tel.archives-ouvertes.fr/pastel-00006003.

[103] Thomas Toth and Christopher Krügel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002. doi: 10.1007/3-540-36084-0_15. URL http://dx.doi.org/10.1007/3-540-36084-0_15.

[104] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In Engin Kirda and Steven Hand, editors, *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11, April 10, 2011, Salzburg, Austria*, page 4. ACM, 2011. doi: 10.1145/1972551.1972555. URL http://doi.acm.org/10.1145/1972551.1972555.

[105] Gang Wang, Tianyi Wang, Haitao Zheng, and Ben Y. Zhao. Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 239–254. USENIX Association, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/wang.

[106] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In Diego Zamboni and Christopher Krügel, editors, *Recent Advances in Intrusion Detection, 9th International Symposium, RAID 2006, Hamburg, Germany, September 20-22, 2006, Proceedings*, volume

4219 of *Lecture Notes in Computer Science*, pages 226–248. Springer, 2006. doi: 10.1007/11856214_12. URL http://dx.doi.org/10.1007/11856214_12.

[107] 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 *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*. The Internet Society, 2006. URL http://www.isoc.org/isoc/conferences/ndss/06/proceedings/papers/honeymonkeys.pdf.

[108] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996. doi: 10.1007/BF00116900. URL http://dx.doi.org/10.1007/BF00116900.

[109] Carsten Willems, Thorsten Holz, and Felix C. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007. doi: 10.1109/MSP.2007.45. URL http://dx.doi.org/10.1109/MSP.2007.45.

[110] Julia Wolf. OMG WTF PDF. Chaos Communication Congress (CCC), December 2010.

[111] Christian Wressnegger, Frank Boldewin, and Konrad Rieck. Deobfuscating embedded malware using probable-plaintext attacks. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, volume 8145 of *Lecture Notes in Computer Science*, pages 164–183. Springer, 2013. doi: 10.1007/978-3-642-41284-4_9. URL http://dx.doi.org/10.1007/978-3-642-41284-4_9.

[112] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A close look on *n*-grams in intrusion detection: anomaly detection vs. classification. In Ahmad-Reza Sadeghi, Blaine Nelson, Christos Dimitrakakis, and Elaine Shi, editors, *AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, pages 67–76. ACM, 2013. doi: 10.1145/2517312.2517316. URL http://doi.acm.org/10.1145/2517312.2517316.

[113] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Comprehensive analysis and detection of flash-based malware. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes*

in Computer Science, pages 101–121. Springer, 2016. doi: 10.1007/978-3-319-40667-1_6. URL http://dx.doi.org/10.1007/978-3-319-40667-1_6.

[114] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers: A case study on PDF malware classifiers. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL http://www.internetsociety.org/sites/default/files/blogs-media/automatically-evading-classifiers.pdf.

[115] Chao Yang, Robert Chandler Harkreader, and Guofei Gu. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, volume 6961 of *Lecture Notes in Computer Science*, pages 318–337. Springer, 2011. doi: 10.1007/978-3-642-23644-0_17. URL http://dx.doi.org/10.1007/978-3-642-23644-0_17.