Šrndić and Laskov *EURASIP Journal on Information Security* (2016) 2016:22
DOI 10.1186/s13635-016-0045-0

EURASIP Journal on Information Security

RESEARCH Open Access

Hidost: a static machine-learning-based detector of malicious files

Nedim Šrndić[1*] and Pavel Laskov[2]

Abstract

Malicious software, i.e., malware, has been a persistent threat in the information security landscape since the early days of personal computing. The recent targeted attacks extensively use non-executable malware 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 paper introduces Hidost, the first static machine-learning-based malware detection system designed to operate on *multiple file formats*. Extending a previously published, highly effective method, it combines the logical structure of files with their content for even better detection accuracy. Our system has been implemented and evaluated on two formats, PDF and SWF (Flash). Thanks to its modular design and general feature set, it is extensible to other formats whose logical structure is organized as a hierarchy. Evaluated in realistic experiments on timestamped datasets comprising 440,000 PDF and 40,000 SWF files collected during several months, Hidost outperformed all antivirus engines deployed by the website VirusTotal to detect the highest number of malicious PDF files and ranked among the best on SWF malware.

Keywords: Machine learning, Security, Malware detection, File formats, PDF, SWF

1 Introduction

One of the most effective tools for breaking into computer systems remains malicious software, i.e., malware. While being a well-known plague since the dawn of personal computing, 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. 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), Flash (targeting Adobe Flash Player), and Microsoft Office files [1, 2]. In 2012, the pioneering exploit kit Blackhole targeted Java, PDF, and Flash files, and its successors have continued this practice [3]. In 2013, the non-executable malware delivered through the web was dominated by PDF and Flash files targeting Adobe Reader and Microsoft Office applications [2]. 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 [3]. Although prevalently used for redirection to sites serving exploit kits, it is not uncommon for Flash files to target Flash Player directly.

*Correspondence: nedim.srndic@uni-tuebingen.de
[1]Cognitive Systems, Department of Computer Science, University of Tübingen, Sand 1, 72076, Tübingen, Germany
Full list of author information is available at the end of the article

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 or CommentCrew used 0-day vulnerabilities in Adobe Reader and Microsoft Office against government and industry targets [4]. Among the recorded 24 0-day discovered in 2014, 16 targeted Adobe Reader and Flash Player (cf. Fig. 1), while Microsoft Word files dominated the list of file types used for targeted attacks [1, 5]. In the first 9 months of 2015, 8 out of top 10 vulnerabilities leveraged by exploit kits were reported to be Flash Player vulnerabilities [6].

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 ad hoc, format-specific detection techniques which do not generalize across the 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 Flash files. Before presenting the main features of the proposed method, we review the related work.

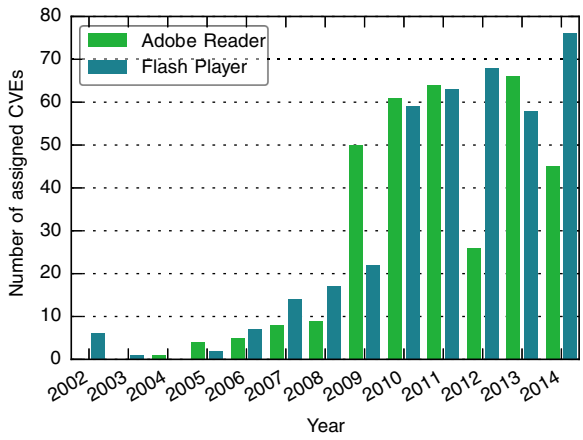

Fig. 1 CVEs with the phrases "Adobe Reader" or "Flash Player" in their description

1.1 Related work

Early work on PDF malware detection focused on n-gram analysis [7, 8] of PDF files on disk. However, PDF is a complex file format [9]. PDF files, especially malicious ones, routinely employ *obfuscation* in the form of compression, the use of different encodings and even encryption. Thus, only full-fledged PDF parsers can properly deobfuscate them. The n-gram approach is in this regard overly simplistic. The first method utilizing a PDF parser was PJScan [10]. It employed anomaly detection based on lexical properties of JavaScript code embedded inside PDF files. However, it could not handle JavaScript code loaded at run time or malware that does not use JavaScript in the first place. Two simple learning-based methods were proposed subsequently, Malware Slayer [11] and PDFrate [12], both utilizing heuristic features based on raw bytes of PDF files. All methods presented so far are commonly referred to as *static* because they do not perform execution or emulation of any part of a PDF file. They can be divided into deep and shallow methods, depending on whether their parsing of PDF files conforms to the PDF Standard [9] (deep) or not (shallow). PJScan is the only deep method presented so far. A common vulnerability of shallow methods is the relative ease of falsification of the PDF physical structure, demonstrated on the example of PDFrate [13]. A common shortcoming of *all* pure static methods is their inability to detect dynamically loaded threats, e.g., when the analyzed file does not contain attack code but instead loads it over the network or from another file.

Along with the described static methods, *dynamic* approaches were developed to leverage the additional information obtained by observing the effects of opening a PDF file at run time. Not reliant on examining the PDF file at all, dynamic methods are immune to PDF obfuscation and physical structure falsification. Early approaches were based on software emulation [14, 15]. However, software emulation was shown to be susceptible to evasion and computationally intensive. Other popular dynamic approaches include Wepawet [16] based on the sandbox JSand [17] and MalOffice based on CWSandbox [18]. Snow et al. proposed to employ hardware virtualization and evaluated their system ShellOS [19] on PDF malware. Tang et al. used anomaly detection on low-level hardware features [20]. 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 an attempt to achieve the speed of static approaches with the accuracy of dynamic ones, *combined static and dynamic* methods were subsequently developed. MDScan performs static JavaScript extraction and dynamic code execution [21], but the complexity of emulation of the PDF JavaScript API with undocumented features prohibits a complete and error-free solution. MPScan, on the other hand, hooks into Adobe Reader for flawless JavaScript extraction and deobfuscation but performs static exploit detection [22]. 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.

In contrast to fully automated methods presented so far, Nissim et al. propose to use 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 [23]. They outline a design with a combination of signature detection and multiple methods described so far but leave its implementation and evaluation for future work. For a more detailed survey of many of the mentioned PDF malware detection methods, we refer the reader to [23].

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 [24]. 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 [25]. Both methods are based on an empirical approach, striving to encode the knowledge of domain experts, i.e., malware analysts, about existing ways of the 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 committed 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. In contrast, the method proposed herein uses a *data-driven approach* instead of expert features, and its detection is based on the structural differences between benign and malicious SWF files. By remaining exploit-agnostic, it remains open to novel attacks, its static approach leads to faster execution and is not vulnerable to run-time evasion.

1.2 Contributions

The proposed detection method is based on the analysis of hierarchical document structure and is henceforth abbreviated as Hidost. It is an extension of previous work published by Šrndić and Laskov in [26], herein referred to as SL2013. The novelty introduced in 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 Standard that organizes basic PDF building blocks into a functional document. Results published in [26] 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. As a deep static method, SL2013 is less affected by PDF obfuscation and physical structure falsification that plague shallow methods. 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 we have discovered later. In particular, we developed *structural path consolidation* (SPC), a technique used to merge similar features. 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 threefold: (a) the attack surface for evasion is reduced; (b) changes in feature set over time are limited; and (c) the number of features is drastically reduced. Together, these improvements render Hidost much more secure and practical than SL2013.

Most importantly, however, this paper 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, Flash's SWF format. Finally, taking a step further, Hidost not only considers 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.

To demonstrate the excellent detection performance of Hidost, we experimentally evaluate it for 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 paper are as follows:

- A static machine-learning-based malware detector, Hidost, the first such system applicable to different file formats based on their logical structure and content.
- An experimental evaluation of Hidost on two formats, PDF and SWF, designed to reflect the operational environment of a malware detector, performed on a dataset of 440,000 PDF files and unprecedented 40,000 SWF files. In our evaluation, Hidost outperformed all antivirus engines at VirusTotal on PDF and ranked among the best on SWF files.
- A prototype implementation of Hidost for two file formats, PDF and SWF, released as Open Source software.
- Source code required to reproduce this work, including experiments and plots, released as Open Source Software.
- Datasets required to reproduce this work.

1.3 Outline

The structure of this paper is as follows. File formats that Hidost is applicable to, i.e., hierarchically structured file formats, are described in Section 2, along with a detailed introduction to logical structures of PDF and SWF. Hidost's system design is described in Section 3 which covers extraction of structural elements from PDF and SWF formats, feature definition, selection, and compaction as well as learning and classification. The experimental evaluation, including the description of datasets and experimental protocols, as well as a discussion of results, is presented in Section 4. We discuss Hidost's extension to other file formats and present a conceptual design for its application to office file formats OOXML and ODF in Section 5. Finally, Section 6 presents conclusions and outlines open questions for future work.

2 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. 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 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 (ODF), an XML-based format for office documents
- Office Open XML (OOXML), a different 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.

2.1 Portable Document Format (PDF)

This section was copied (with adaptation) from [26], copyrighted by the Internet Society.

Portable Document Format (PDF) is an open standard published as ISO 32000-1:2008 [9]. The syntax of PDF comprises these four main elements:

- Objects. These are the basic building blocks in PDF.
- File structure. It specifies how objects are laid out and modified in a PDF file.

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

Fig. 2 A sample HTML file

- Document structure. It determines how objects are logically organized to represent the contents of a PDF file (text, graphics, etc.).
- Content streams. They provide a means for efficient storage of various parts of the file content.

There are nine basic object types in PDF. Simple object types are Boolean, Numeric, String, and Null. PDF strings have bounded length and are enclosed in parentheses "(" and ")". The type Name is used as an identifier in the description of the PDF document structure. Names are introduced using the character "/" and can contain arbitrary characters except *null* (0×00). The aforementioned five object types will be referred to as *primitive* types in this paper. An Array is a one-dimensional ordered collection of PDF objects enclosed in square brackets, "[" and "]". Arrays may contain PDF objects of different type, including nested arrays. A Dictionary is an unordered set of key-value pairs enclosed between the symbols "<<" and ">>". The keys must be *name objects* and must be unique within a dictionary. The values may be of any PDF object type, including nested dictionaries. A Stream object is a PDF dictionary followed by a sequence of bytes. The bytes represent information that may be compressed or encrypted, and the associated dictionary contains information on whether and how to decode the bytes. These bytes usually contain content to be rendered but may also contain a set of other objects. Finally, an Indirect object is any of the previously defined objects supplied with a unique object identifier and enclosed in the keywords `obj` and `endobj`. Due to their unique identifiers, indirect objects can be referenced from other objects via *indirect references*.

The syntax of PDF objects is illustrated in a simplified exemplary PDF file shown in Fig. 3. It contains four indirect objects denoted by their two-part object identifiers, e.g., `1 0` for the first object, and the `obj` and `endobj` keywords. These objects are dictionaries, as they are surrounded with the symbols "<<" and ">>". The first one is the *Catalog* dictionary, denoted by its *Type* entry which contains a PDF name with the value *Catalog*. The Catalog has two additional dictionary entries: *Pages* and *OpenAction*. *OpenAction* is an example of a nested dictionary. It has two entries: *S*, a PDF name indicating that this is a JavaScript action dictionary, and *JS*, a PDF string containing the actual JavaScript script to be executed: `alert('Hello!');`. *Pages* is an indirect reference to the object with the object identifier `3 0`: the Pages dictionary that immediately follows the Catalog. It has an integer, *Count*, indicating that there are two pages in the document, and an array *Kids* identifiable by the square brackets, with two references to Page objects. The same object types are used to build the remaining Page objects. Notice that each of the Page objects contains a backward reference to the Pages object in their *Parent* entry. Altogether, there are three references pointing to the same indirect object, `3 0`, the Pages object.

```
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
```

Fig. 3 Raw content of an example PDF file. Formatted for easier reading. Details omitted for brevity. Primitive data types and references are *colored green*

The relations between various basic objects constitute the logical, tree-like *document structure* of a PDF file. The nodes in the document structure are objects themselves, and the edges correspond to the names under which child objects reside in a parent object. For arrays, the parent-child relationship is nameless and corresponds to an integer index of individual elements. Notice that the document structure is, strictly speaking, 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 3.4, and we will therefore limit ourselves to considering the PDF document structure in its simplified, tree form, as illustrated in Fig. 4.

The root node in the document structure is a special PDF dictionary with the mandatory *Type* entry containing the name *Catalog*. Any object of a primitive type constitutes a leaf, i.e., terminal node, in the document structure.

We define a *path* in the PDF structural tree as a sequence of edges starting in the Catalog dictionary and ending with an object of a primitive type. For example, in Fig. 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 exemplary structural paths from real-world benign PDF files:

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

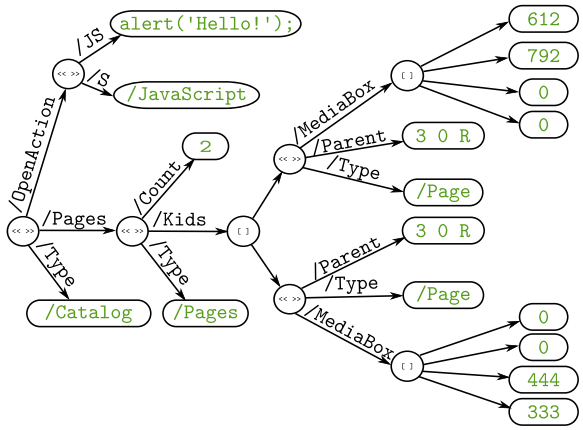

Fig. 4 Structural tree of the PDF file depicted in Fig. 3. Dictionaries are illustrated using the symbol "<< >>", arrays using "*square brackets*". Cycles were omitted for simplicity

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 structural paths from 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.

2.2 SWF file format

SWF File Format (*SWF*, pronounced *swiff*) is a proprietary binary file format, its specification is published in [27]. SWF files consist of a header and a sequence of tags, i.e., data structures with values for predefined fields. There are 65 different types of tags specified, each defining its own set of fields with different names and data types. Some of the basic SWF data types are [27]:

- 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 of different widths and precisions
- Integer and fixed-point numbers with widths that are not exponents of 2
- Strings
- Data structures such as 24- and 32-bit color records, rectangle records, 2D transformation matrices, etc.

Figure 5 shows a very small SWF file used for illustrative purposes. Clearly, the physical layout of SWF is too obscure for direct interpretation. Instead, our description of the SWF logical structure is based on the decoded, human-readable depiction of the same file, illustrated in Fig. 6. This textual description of the original SWF file

```
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
```

Fig. 5 Hexadecimal view of a toy SWF file. *Left column* contains hexadecimal addresses of first bytes of every row

was produced using the `ConsoleDumper` class from the SWFRETools toolkit [28], an open-source Java toolkit for reverse-engineering SWF files.

The illustration skips the file header as it is not used in our method. It shows 5 SWF tags separated by dotted lines: two *SetBackgroundColor* tags at bytes `0×14` and `0×1B`, two *ShowFrame* tags at bytes `0×19` and `0×20` and an *End* tag at byte `0×22`. Tags of a SWF file are laid out sequentially. Every tag has a header with an unsigned 16-b little-endian *TagCodeAndLength* field that comprises a 10-byte tag type identifier and a 6-b tag length field, with an optional wider length field for tags longer than 62 B.

```
[0x14]: SetBackgroundColor
  [0x14]: Header (Code: 9 Length: 3)
    [0x14]: TagAndLength : 579
  [0xB0]: BackgroundColor
    [0x16]: Red : 170
    [0x17]: Green : 187
    [0x18]: Blue : 204
[0x19]: ShowFrame
  [0x19]: Header (Code: 1 Length: 0)
    [0x19]: TagAndLength : 64
[0x1B]: SetBackgroundColor
  [0x1B]: Header (Code: 9 Length: 3)
    [0x1B]: TagAndLength : 579
  [0xE8]: BackgroundColor
    [0x1D]: Red : 17
    [0x1E]: Green : 34
    [0x1F]: Blue : 51
[0x20]: ShowFrame
  [0x20]: Header (Code: 1 Length: 0)
    [0x20]: TagAndLength : 64
[0x22]: End
  [0x22]: Header (Code: 0 Length: 0)
    [0x22]: TagAndLength : 0
```

Fig. 6 SWF file depicted in Fig. 5, decoded. Values are *colored green*. *Every line* starts with a hexadecimal number between *square brackets*, denoting the offset, in bytes, of the tag field specified in the given line from the beginning of the SWF file

The first tag in this file is used to set the background color of the display. It is a simple tag, defining three unsigned 1-B values of the red (`0×AA` = 170), green (`0×BB` = 187), and blue (`0×CC` = 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. The last tag signals the end of the file.

Figure 7 illustrates a logical view of our example SWF file in which the file is structured as a tree. It closely follows the layout presented in the decoded SWF file of Fig. 6. 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 represented as leafs, e.g., the value of the *red* field of the first *SetBackgroundColor* tag, 170.

We define a *path* in the SWF structural tree, analogous to the path in the PDF structural tree, as a series

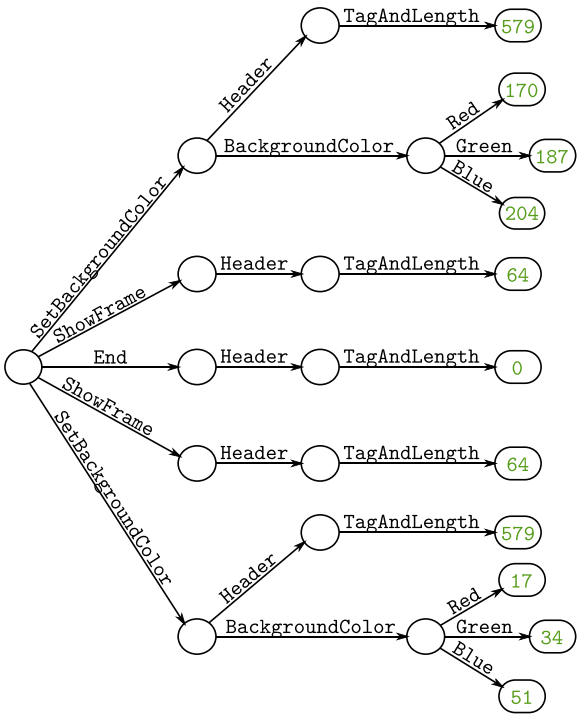

Fig. 7 Logical structure of the SWF file illustrated in Fig. 5

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 section, we describe how the logical structures of PDF and SWF files are processed for use by learning algorithms and describe the system design of Hidost.

3 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, our method, 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 [29]. The published code comprises a toolset for feature extraction from PDF (implemented in C++) and SWF files (implemented in Python and Java). Experiment reproduction code is published separately, as described in Section 4.

The system design of Hidost is illustrated in Fig. 8. There are six main stages in Hidost: structure extraction, structural path consolidation, feature selection, vectorization, learning, and classification. Structure extraction transforms the structural features of specific formats into a common data structure—structural multimap—representing paths in the structural hierarchy. Structural path consolidation is intended to transform structural paths into a more general form, removing artifacts. Feature selection is concerned with finding the minimum set of features required for a successful machine learning application. Vectorization transforms structural multimaps into numeric vectors processed by machine learning methods. Learning generates a discriminative model of malicious and benign files based on their properties encoded in feature vectors. Finally, 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.

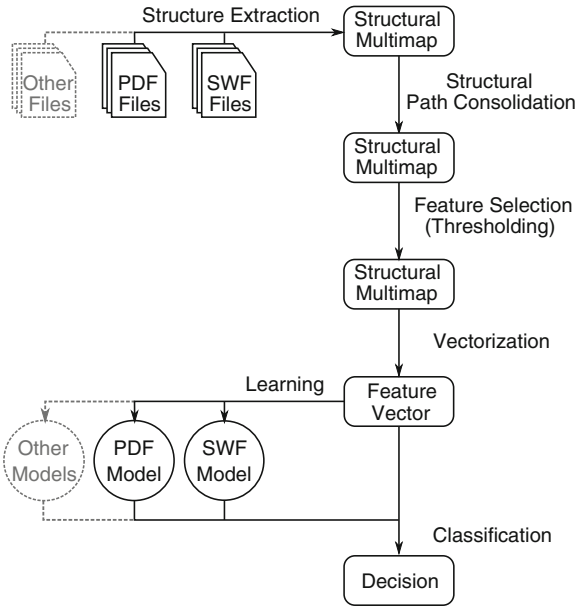

Fig. 8 Hidost system design

3.1 File 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) applicability to multiple file formats.

A suitable representation for the logical structure of hierarchically structured file formats 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. An example structural multimap is illustrated in Fig. 9. Multiple values for the same key are delimited using vertical bar symbols "|". Hidost uses a simplified form of structural multimaps presented later in this section.

The nature of the logical file structure necessitates a multimap instead of a map because multiple leafs may be reachable by a single structural path. With PDFs, this occurs when a path contains an array with more than one element, e.g., the path `/Pages/Kids/MediaBox` contains two arrays and reaches eight leafs. In case of SWF files, apart from arrays, multiple tags of the same type cause multiple leafs to lie in the same path.

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

Fig. 9 A complete structural multimap of the PDF file depicted in Fig. 3. This type of structural multimap is not used in Hidost but rather illustrated as an instructive example

Implementation of structure extraction for PDF and SWF is presented in the following two sections.

3.1.1 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 is achieved by removing all cycles from the graph. There is a cycle in the PDF logical structure when an indirect reference from a tree node at depth d_R points to a tree node lying on the same path at a depth $d_T < d_R$. An example is shown in Fig. 3: the indirect reference at the path `/Pages/Kids/Parent` with depth 3 points to the dictionary located at `/Pages` with depth 1, lying on the same path. Indirect references may also cause inconsistencies in extracted tree structures. When two indirect references located at different paths reference the same object, it is ambiguous which is the "true" one.

Both described problems—cycles in the structural graph and multiple references—are implicitly solved with a simple procedure based on the breadth-first search (BFS) algorithm. A robust PDF parser is required to navigate the PDF structure of obfuscated or malformed PDF files. Hidost utilizes the open-source Poppler PDF rendering library [30], version 0.18.4. The procedure starts by locating the root node of the file structure, i.e., the *Catalog* dictionary. Then, it performs a breadth-first search on the entire file structure graph, inserting 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. Cycles are avoided by skipping indirect references that point to previously visited objects and treating them as leaf nodes instead. Child node enumeration in alphabetical order ensures the consistent resolution of multiple references the same node, so that every traversal of a PDF file's structural graph produces the same structural multimap.

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. Binarization of non-numeric values can be seen by comparing Figs. 9 and 10. 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 [10, 31], characterizing them with simple properties such as their length, entropy, or distribution of keywords [11, 12] or testing them for valid CPU instructions [15], to dynamic, e.g., CPU emulation of strings [14, 21] or their execution [19, 20, 22]. However, all these approaches either conflict with the desired static system design (dynamic evaluation), decrease 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 4.3.4.

3.1.2 SWF

The SWF logical structure is more straightforward to extract than PDF as it contains no 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 depiction, such as the one in Fig. 6. 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

Structural Multimap	Feature Vector
/OpenAction/JS:1.0	1.0
/OpenAction/S:1.0	1.0
/Pages/Count:2	2.0
/Pages/Kids/MediaBox:0|0|612|792|0|0|333|444	166.5
/Pages/Kids/Parent:1.0|1.0	1.0
/Pages/Kids/Resources:...	...
/Pages/Kids/Type:1.0|1.0	1.0
/Pages/Type:1.0	1.0
/Type:1.0	1.0

Fig. 10 Structural multimap and feature vector of the PDF file depicted in Fig. 3

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, i.e., a leaf in the structural tree. 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.

The structural multimap corresponding to the SWFRETools output of Fig. 6 is illustrated on the left-hand side of Fig. 11. The following section describes the second processing step of our method, structural path consolidation.

3.2 Structural path consolidation

The syntactic richness and flexibility of many file formats enable semantically equivalent but syntactically different structures. Such syntactic polymorphism may decrease the detection accuracy and, furthermore, provide a possibility for an attacker to fully avoid detection of specific 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 groups of paths similar to each other but not identical. Paths in these groups exhibited a similarity in one of two ways. Paths in some groups were identical except for exactly one customizable path component, while paths in other groups 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 *semantic meaning*, in PDF. For example, the paths `/Pages/Kids/Resources` and `/Pages/Kids/Kids/Resources` have a common repetitive subpath, `/Kids`, 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 semantic meaning. Likewise, the paths `/Pages/Kids/Resources/Font/F1` and `/Pages/Kids/Resources/Font/F42` only differ in the last path segment, which the PDF Standard 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.

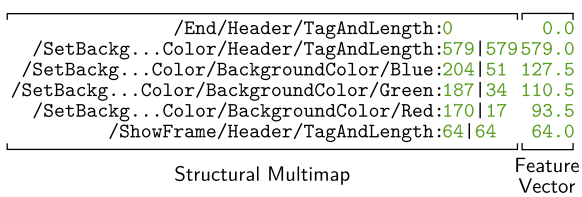

Fig. 11 Structural multimap and feature vector of the SWF file depicted in Fig. 5

The existence of semantically equivalent path groups questions the utility of SL2013 feature definition which treats every path as a distinct feature. It is more meaningful to preserve the semantics of paths by consolidating all semantically equivalent paths to one feature. This idea, called *structural path consolidation* (SPC), was implemented in Hidost and experimentally evaluated in Section 4.3.3.

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 1 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, the paths `/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 1 are a result of an empirical investigation of structural paths occurring in our dataset, with the aim of minimizing their total count after transformation. However, the analysis was limited to rules that capture generic branches of the PDF document structure instead of dataset-specific artifacts. Some of the recognizable branches in Table 1 include anonymized items such as resources (1 and 13), entries of various name trees (global (5) and structure tree (6)), dictionaries for mapping custom names into other objects (9), color space items (17 and 18), or embedded files' names (19). Other rules are used to flatten hierarchies (2, 3, 7, 8, and 10)

Table 1 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`

and convert linked lists to shallow sets (4) in order to create a generic, unified view of their elements, all on the same level. We refer the reader to the PDF Standard [9] for a detailed explanation of these branches of the PDF document structure.

Due to the relatively shallow SWF logical file structure and the barring of user-defined path components, only two SPC rules were compiled for this format, listed in Table 2, both for handling repetitive subpaths.

No attempt was made to compile a complete list of SPC rules. 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. In general, to extend the list, it is advised to read the PDF Standard looking for places in the PDF document structure where user-defined names are allowed or where there is a well-defined list or hierarchy. However, even this limited set of rules provides the following crucial benefits compared to SL2013:

Table 2 SWF structural path consolidation rules

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

- 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 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 also suffers from this vulnerability. Hiding attacks are cheap to implement and the primary weakness 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* [32] and has recently started to attract interest in security literature [33].
The continual change in data renders classifiers ever more outdated as time progresses since their training. Therefore, the need arises for regular updates to the learning model, i.e., 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 the flexibility of its structural path definition. As Section 4.3.3 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 1237.4 features per training. However, there are limits in the effectiveness of SPC against manually crafted paths. Because it knows 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 its 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 bigger 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.

3.3 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 for building discriminative models increases the dimensionality of the input space without improving classification accuracy. Therefore, feature selection—as it is common in other machine learning applications—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 [27] strictly defines the names of all tags and all their fields, prohibiting customization. Therefore, Hidost's feature set for SWF theoretically 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 3177.

In contrast, the PDF file format specification [9] 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, $\frac{2}{3}$ 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, the original SL2013 method selected the PDF paths which occur in more than a fixed number of training files, i.e., 1000, 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., 1000 files, which corresponds to around 1 % of the training set size. This is in contrast with SL2013, where feature selection was performed "in hindsight", once for the entire dataset.

3.4 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, $\frac{1}{2}$, 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 each specific structural path and using values from structural maps as values of specific dimensions. The mapping of individual structural paths to dimensions of feature vectors is defined before feature extraction and 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 the feature vectors.

The ordered collection of all features used by a learning algorithm is its *feature set*. Figures 10 and 11 illustrate feature vectors obtained 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

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

Here, p_i denotes the ith path in the feature set and $m[p_i]$ denotes the value in a multimap m associated with that path.

3.5 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 the reader's classifier of choice. For experiments presented in this paper, the Random Forest implementation of the open-source scikit-learn Python machine learning library [34], 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 [35] is an ensemble classifier. It is trained by growing a forest of decision trees using 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 Experimental evaluation

An extensive experimental evaluation was performed to assess the detection performance of Hidost and compare it to related work. Entire source code and datasets needed to reproduce all experiments and plots have been published as open-source software [36].

4.1 Experimental datasets

Experiments were run on two datasets, one for each file format. Both were collected from VirusTotal [37], a website that performs an analysis of files uploaded by Internet users using many antivirus engines. VirusTotal provides detection results to researchers, enabling us to compare Hidost's detection performance to that of deployed antivirus engines. In our experiments, we consider those files malicious that were labeled as malicious by *at least five* antivirus engines and those files benign that were labeled benign by *all* antivirus engines. The remaining files, labeled by one to four antivirus engines as malicious, are discarded from the experiments because of the high uncertainty of their true class label.

Our PDF dataset comprises 439,563 (446 GiB) files, 407,037 (443 GiB) benign and 32,567 (2.7 GiB) malicious. They were collected during 14 weeks, between July 16 and October 21, 2012. This is the same dataset used for the *10Weeks* experiment in [26], enabling a direct comparison with that work.

The SWF dataset was collected between August 1, 2013 and March 8, 2014 and comprises 40,816 (14.2 GiB) files, 38,326 (14.1 GiB) benign and 2490 (190 MiB) 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.

4.2 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 the same publication.

The *10Weeks* experiment attempts to approximate a real-world, operational environment of a malware detection system exposed to attacks that change and adapt in time. For that purpose, it employs periodic retraining and evaluation. Training and evaluation datasets are assembled in a sliding window fashion, i.e., for every week of evaluation data, the classifier is trained on the previous 4 weeks of training data.

In order to follow the protocol of the *10Weeks* experiment as closely as possible, the two experimental datasets were partitioned as follows. The time period in which the files were collected was divided into 14 smaller, consecutive time periods. For PDF, every period was exactly 1 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 four 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 four steps of feature extraction (i.e., structure extraction, SPC, feature selection, and vectorization) were applied to the training dataset.

Thus, generated PDF file partitioning is identical to the one used in [26], and the SWF dataset follows the same design. The datasets are illustrated in Figs. 12, 13, and 14. 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.

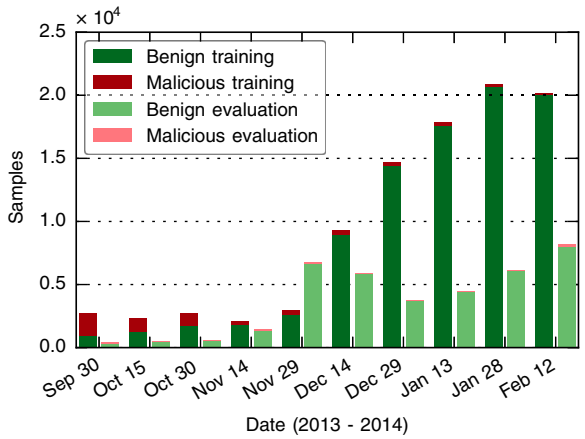

Fig. 13 SWF-Normal dataset. Malicious training samples older than four time periods are discarded

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, as visible in Figs. 13 and 14. As much as 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. In order 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. 14, in which malicious training samples older than four 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 greatly reduced.

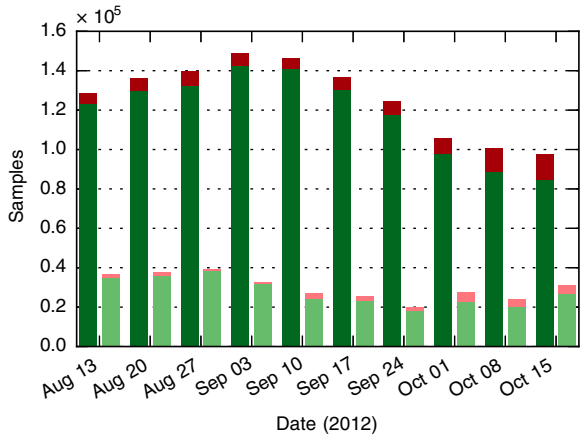

Fig. 12 PDF dataset. Legend in Fig. 13

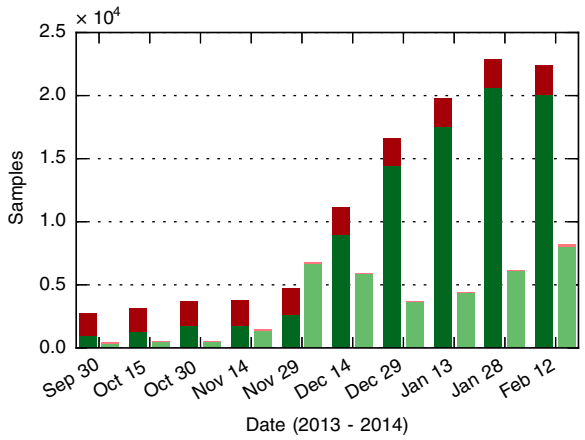

Fig. 14 SWF-KeepMal dataset. Malicious training samples are kept indefinitely. Legend in Fig. 13

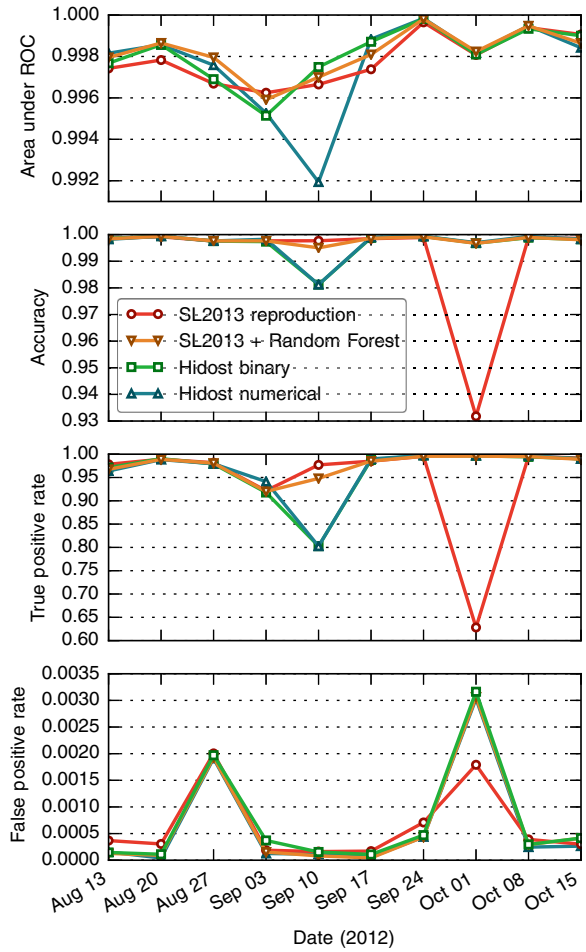

Fig. 15 Results on PDF data. Performance of SL2013 with an SVM and Random Forest classifier compared to Hidost with binary and numerical features

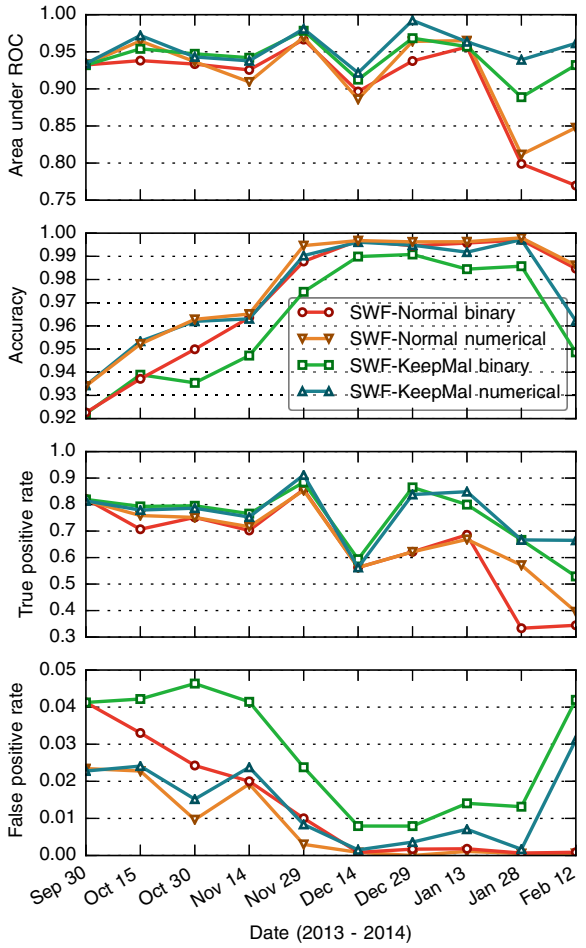

Fig. 16 Results on SWF data. Performance of Hidost with both binary and numerical features on two SWF datasets, SWF-Normal, and SWF-KeepMal

4.3 Experimental results

Experimental results of different methods operating on PDF and SWF data are illustrated in Figs. 15 and 16, respectively. The methods are compared in four performance indicators typical for classification tasks: true (TPR) and 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.

Figure 15 shows results for different variants of Hidost and SL2013 on PDF data. Along with the accurate reproduction of SL2013, the same method is evaluated using a Random Forest classifier instead of the SVM. Hidost is shown with both binary and numerical features. These two variants of Hidost are also shown in Fig. 16 on SWF datasets SWF-Normal and SWF-KeepMal.

4.3.1 Classification performance

Figure 15 shows a direct comparison of Hidost to SL2013 on the PDF dataset. 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 AUROC above 0.99 in every period. It can 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. 16, 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. 13 and 14, 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 four 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 increase of 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.

4.3.2 Comparison to antivirus engines

In order 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, widely used malware detectors most persons rely on for their security, 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, in the course of our experiments. By definition of our ground truth, samples

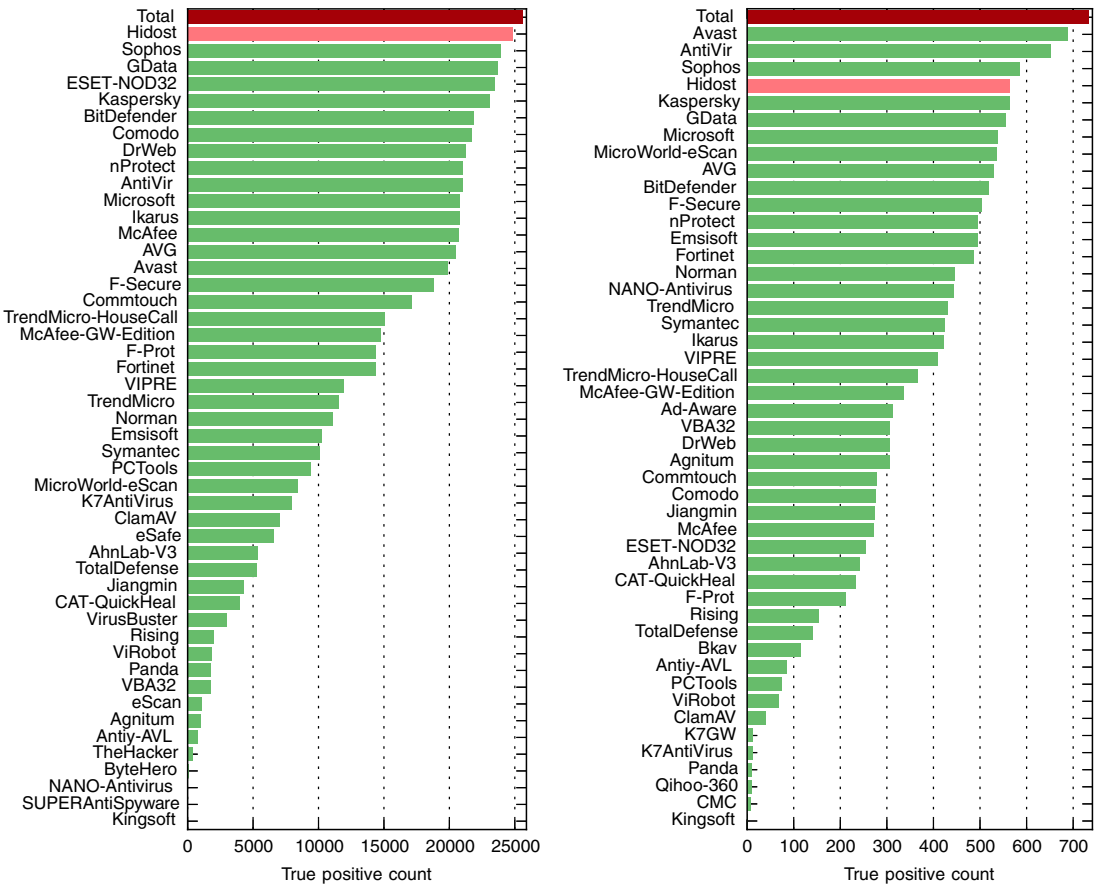

Fig. 17 Comparison of Hidost to antivirus engines on PDF (*left*) and SWF (*right*) data

labeled malicious by at most four antivirus engines are filtered out. Therefore, the antivirus engines do not have false positives and cannot be compared in that sense.

Figure 17 shows 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.

4.3.3 Effects of structural path consolidation

SPC is one of the main novelties in Hidost with respect to SL2013; therefore, an evaluation of its effects on the performance of the system is only fitting. Figure 15 demonstrates that SPC does not affect detection performance, neither positively nor negatively. 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. Figure 18 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 %.

4.3.4 Effects of numerical features

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 numerical features on performance.

On PDF, the difference between binary and numerical features is insignificant, as shown in Fig. 15. On the other hand, the effect on SWF is largely positive. As shown in Fig. 16, numerical consistently outperformed binary features on both SWF datasets. They showed the highest influence 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 might 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 logical structure, the added expressive power of numerical features enables the description, and consequently detection, of both structure- and content-based attacks.

4.4 Reproduction of SL2013 results

Results of all performance evaluation experiments published in SL2013 [26] were accurately reproduced using the original RBF SVM with $C = 12$ and $\gamma = 0.0025$. However, when attempting to reproduce the evasion robustness evaluation experiment, we have discovered a flaw in

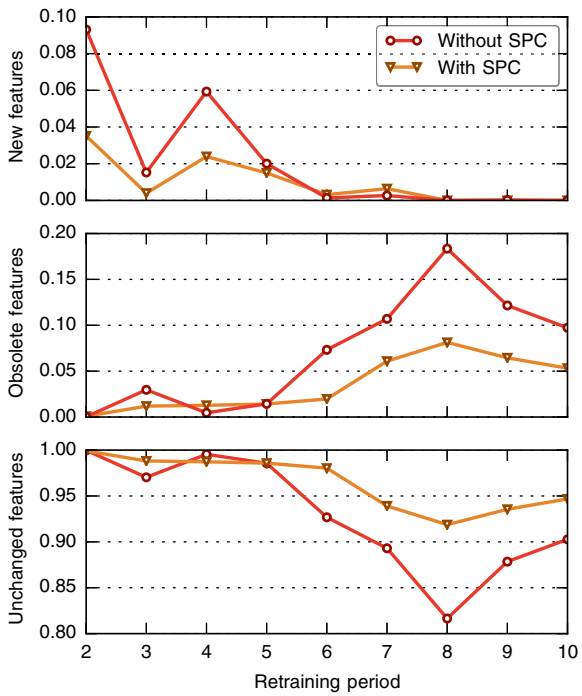

Fig. 18 Feature set drift in the PDF dataset. Illustrates change in the feature set between consecutive feature selection events in our experiment, performed before every training, i.e., once per week. For weeks 2 to 10, the percentage of features that have been added to (new), removed from (obsolete), or remained (unchanged) in the feature set is plotted, relative to the previous week

the source code of the mimicry attack used against the RBF SVM. 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. Thus, 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 retract the results of the mimicry attack experiment published in [26] and see the success of the corrected attack as evidence against the robustness of RBF SVM in an adversarial environment. Robustness guarantees for Random Forests remain a topic for future research.

5 Discussion

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, 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. It is this step that has to be specialized for every file format. In the following, we discuss file structure and content extraction for various hierarchically structured file formats.

XML and the related HTML and SVG have a very clear and well-defined hierarchical structure that represents one of the cornerstones of these formats. For example, Fig. 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. The following shows a simplified file and directory layout in an ODF file:

```
.
|-- content.xml
|-- manifest.rdf
|-- META-INF
|   \-- manifest.xml
|-- meta.xml
|-- mimetype
```

```
<?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>
        <meta:creation-date>2013-04-11T13:34:39.901503738</meta:creation-date>
        <meta:generator>LibreOffice/4.2.8.2$Linux_X86_64</meta:generator>
        <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>
```

Fig. 19 Example `meta.xml` file

```
|-- settings.xml
|-- styles.xml
\-- Thumbnails
    \-- thumbnail.png
```

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 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. 19, 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 beloning 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.

6 Conclusions

In this paper we introduced Hidost, a machine-learning-based malware detection system. It represents an extension of a previously published method, SL2013 [26]. Hidost is the first static machine-learning-based malware detector designed to operate on multiple file types. It accomplishes this by making a model of malicious and benign samples based on their structure and content.

Evaluated on a real-world dataset in a realistic experiment with periodic retraining spanning multiple months, 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, SL2013, 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. Finally, its greatly reduced feature set dimensionality enables its efficient application on very large datasets.

A logical next step for Hidost is its implementation and evaluation on other hierarchically structured file formats. Of particular significance would be an application to formats used by Microsoft Office, as they are widely used for recent targeted attacks. A conceptual design for this application was proposed in this paper. Looking beyond, the development of more advanced string handling methods might prove indispensable to enable detection of malware in formats whose logical structure and numerical content do not provide enough discriminative power.

Acknowledgements
We acknowledge support by Deutsche Forschungsgemeinschaft and Open Access Publishing Fund of University of Tübingen.

Authors' contributions
NS participated in conceiving and designing the study, implemented the method, participated in the data collection, carried out the experiments, and drafted the manuscript. PL participated in conceiving and designing the study, the system design, and the data collection and helped draft the manuscript. Both authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Cognitive Systems, Department of Computer Science, University of Tübingen, Sand 1, 72076, Tübingen, Germany. [2]Munich Office, European Research Center, Huawei Technologies Duesseldorf GmbH, Riessstr. 25 D-3.OG, 80992 München, Germany.

Received: 29 November 2015 Accepted: 7 September 2016
Published online: 26 September 2016

References

1. Symantec, 2014 Internet Security Threat Report, Volume 19 (2014). https://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf. Accessed 13 Apr 2015
2. Cisco, 2014 Annual Security Report (2014). http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf. Accessed 13 Apr 2015
3. Sophos, Security Threat Report 2014 (2014). http://www.sophos.com/en-us/medialibrary/pdfs/other/sophos-security-threat-report-2014.pdf. Accessed 13 Apr 2015
4. Symantec, 2014 Internet Security Threat Report, Volume 19, Appendix (2014). https://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_appendices_v19_221284438.en-us.pdf. Accessed 13 Apr 2015
5. Symantec, 2015 Internet Security Threat Report, Volume 20 (2015). http://know.symantec.com/LP=1123. Accessed 15 Apr 2015

6. Recorded Future, Gone in a Flash: Top 10 Vulnerabilities Used by Exploit Kits (2015). https://www.recordedfuture.com/top-vulnerabilities-2015/. Accessed 15 Nov 2015
7. W-J Li, SJ Stolfo, A Stavrou, E Androulaki, AD Keromytis, in *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. A study of malcode-bearing documents (Springer, 2007), pp. 231–250
8. ZM Shafiq, SA Khayam, M Farooq, in *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Embedded malware detection using markov n-grams (Springer, 2008), pp. 88–107
9. Document management - Portable document format - Part 1: PDF 1.7 (2008). https://www.adobe.com/devnet/pdf/pdf_reference.html. Accessed 23 Jan 2015
10. P Laskov, N Šrndić, in *Annual Computer Security Applications Conference (ACSAC)*. Static detection of malicious JavaScript-bearing PDF documents (ACM, 2011), pp. 373–382
11. D Maiorca, G Giacinto, I Corona, in *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. A pattern recognition system for malicious PDF files detection (Springer, 2012), pp. 510–524
12. C Smutz, A Stavrou, in *Proceedings of the 28th Annual Computer Security Applications Conference*. Malicious PDF detection using metadata and structural features (ACM, 2012), pp. 239–248
13. N Šrndić, P Laskov, in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. Practical evasion of a learning-based classifier: a case study (IEEE Computer Society, 2014), pp. 197–211
14. M Polychronakis, KG Anagnostakis, EP Markatos, in *Annual Computer Security Applications Conference (ACSAC)*. Comprehensive shellcode detection using runtime heuristics (ACM, 2010), pp. 287–296
15. P Akritidis, EP Markatos, M Polychronakis, KG Anagnostakis, in *20th International Conference on Information Security*. STRIDE: Polymorphic sled detection through instruction sequence analysis (Springer, 2005), pp. 375–392
16. Wepawet (2015). http://wepawet.iseclab.org/. Accessed 16 Apr 2015
17. M Cova, C Kruegel, G Vigna, in *International Conference on World Wide Web (WWW)*. Detection and analysis of drive-by-download attacks and malicious JavaScript code (ACM, 2010), pp. 281–290
18. C Willems, T Holz, F Freiling, CWSandbox: towards automated dynamic binary analysis. IEEE Secur. Privacy. **5**(2), 32–39 (2007)
19. KZ Snow, S Krishnan, F Monrose, N Provos, in *USENIX Security Symposium*. ShellOS: enabling fast detection and forensic analysis of code injection attacks (USENIX Association, 2011)
20. A Tang, S Sethumadhavan, SJ Stolfo, in *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17–19, 2014, Proceedings*. Unsupervised anomaly-based malware detection using hardware features, vol. 8688 (Springer, 2014), p. 109
21. Z Tzermias, G Sykiotakis, M Polychronakis, EP Markatos, in *European Workshop on System Security (EuroSec)*. Combining static and dynamic analysis for the detection of malicious documents (ACM, 2011)
22. X Lu, J Zhuge, R Wang, Y Cao, Y Chen, in *System Sciences (HICSS), 2013 46th Hawaii International Conference On*. De-obfuscation and detection of malicious PDF files with high accuracy (IEEE Computer Society, 2013), pp. 4890–4899
23. N Nissim, A Cohen, C Glezer, Y Elovici, Detection of malicious PDF files and directions for enhancements: a state-of-the art survey. Comput. Secur. **48**(0), 246–266 (2015)
24. S Ford, M Cova, C Kruegel, G Vigna, in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. Analyzing and detecting malicious flash advertisements (IEEE Computer Society, 2009), pp. 363–372
25. TV Overveldt, C Kruegel, G Vigna, in *Recent Adances in Intrusion Detection (RAID)*. FlashDetect: ActionScript 3 malware detection (Springer, 2012), pp. 274–293
26. N Šrndić, P Laskov, in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24–27, 2013*. Detection of malicious PDF files based on hierarchical document structure, (2013). http://internetsociety.org/doc/detection-malicious-pdf-files-based-hierarchical-document-structure
27. SWF File Format Specification (version 19) (2012). https://www.adobe.com/devnet/swf.html. Accessed 23 Jan 2015
28. SWFRETools - A collection of tools for reverse engineering Flash files. https://github.com/sporst/SWFREtools. Accessed 4 Feb 2015
29. Hidost - Toolset for extracting document structures from PDF and SWF files. https://github.com/srndic/hidost. Accessed 29 Nov 2015
30. Poppler. http://poppler.freedesktop.org/. Accessed 10 Feb 2015
31. K Rieck, T Krüger, A Dewald, in *Annual Computer Security Applications Conference (ACSAC)*. Cujo: efficient detection and prevention of drive-by-download attacks, (2010), pp. 31–39
32. G Widmer, M Kubat, Learning in the presence of concept drift and hidden contexts. Mach. Learn. **23**(1), 69–101 (1996)
33. A Kantchelian, S Afroz, L Huang, AC Islam, B Miller, MC Tschantz, R Greenstadt, AD Joseph, J Tygar, in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*. Approaches to adversarial drift (ACM, 2013), pp. 99–110
34. F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, E Duchesnay, scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
35. L Breiman, Random forests. Mach. Learn. **45**(1), 5–32 (2001)
36. Hidost Reproduction. https://github.com/srndic/hidost-reproduction. Accessed 29 Nov 2015
37. VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/. Accessed 6 Mar 2015