In Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), Venice, Italy.

Detection of Malicious Scripting Code through Discriminant and Adversary-Aware API Analysis

Davide Maiorca[1], Paolo Russu[1], Igino Corona[1], Battista Biggio[1], and Giorgio Giacinto[1]

Department of Electrical and Electronic Engineering, University of Cagliari, Italy
{davide.maiorca, paolo.russu, igino.corona, battista.biggio, giacinto}@diee.unica.it

Abstract

JavaScript and ActionScript are powerful scripting languages that do not only allow the delivery of advanced multimedia contents, but that can be also used to exploit critical vulnerabilities of third-party applications. To detect both ActionScript- and JavaScript-based malware, we propose in this paper a machine-learning methodology that is based on extracting discriminant information from system API methods, attributes and classes. Our strategy exploits the similarities between the two scripting languages, and has been devised by also considering the possibility of targeted attacks that aim to deceive the employed classification algorithms. We tested our method on PDF and SWF data, respectively embedding JavaScript and ActionScript codes. Results show that the proposed strategy allows us to detect most of the tested malicious files, with low false positive rates. Finally, we show that the proposed methodology is also reasonably robust against evasive and targeted attacks.

1 Introduction

`JavaScript` and `ActionScript` are two scripting languages that are mostly used in websites and documents to provide the user with additional, multimedia contents. However, their flexibility and interpreted nature allows attackers to use them for exploiting vulnerabilities of applications. This became particularly evident when PDF files containing `JavaScript` attacks started to hamper Adobe Reader security in 2008. More recently, starting from 2015, `ActionScript`-based attacks exploited critical vulnerabilities of the `ActionScript Virtual Machine`, normally used to read Flash files, thus causing severe damage. A very popular case is the attack against the Hacking Team, an Italian security company, which caused a loss of more than 400 GB of classified data [1].

Despite the attempts made by browser developers such as Mozilla and Google to contain the phenomenon and limit the usage of Flash-based web applications, such technology is still widely used to deploy multimedia content. Likewise, `JavaScript` is still considered an essential tool to deploy web content and to improve the readability of documents. For these reasons, security threats concerning these two formats are still actual and need to be addressed [5].

Due to the flexibility that the scripting languages allow when writing the source code (for example, by writing lines of code as strings that can be interpreted at runtime by executing specific functions), common Antivirus solutions suffer from multiple problems at detecting such attacks. The source code can be easily obfuscated by means of automatic tools that make the detection and analysis process considerably harder.

For these reasons, a number of approaches based on machine learning to detect `JavaScript` and `ActionScript` have been introduced [6, 7, 9, 12, 13]. They employ static or dynamic analysis of the scripting code typically carried by PDF and SWF files. Leveraging machine learning guarantees more flexibility in comparison to signature-based approaches, as the former can also be used to detect zero-day and novel attacks. The aforementioned approaches have been specifically tailored either to `JavaScript` or to `ActionScript` files. Moreover, most of these works did not discuss the possibility of *targeted attacks*, i.e., when an attacker attempts to craft malicious samples with the aim of evading the system.

In this paper, we propose a machine-learning approach that can be employed, although with some slight differences, to analyze `JavaScript` or `ActionScript` codes carried by PDF and SWF files. The rationale here is showing that, as the two languages share multiple characteristics, it is possible to extract similar information related to system- or application-based APIs. We show that using such information in combination with proper machine-learning algorithms can not only lead one to attain high detection rates, but also to develop systems that are robust against some targeted attacks.

The main contributions of this work are summarized in the following.

1. We provide a methodology that can be used to detect `JavaScript` and `ActionScript` code embedded within PDF and SWF files.
2. We empirically evaluate it on PDF and SWF data, showing that it can correctly detect a large fraction of malicious files while misclassifying only a small fraction of benign files.
3. We empirically show that our approach also exhibits some degree of robustness against well-crafted, targeted attacks.

Paper Organization. The rest of the paper is organized as follows. Section 2 provides an overview on `JavaScript` and `ActionScript`; Section 3 describes the employed detection methodology; Section 4 provides the experimental evaluation; Section 5 discusses the limitations of our approaches; Section 6 discusses the related work in the field; Section 7 closes the paper with the conclusions.

2 Overview on JavaScript and ActionScript

`JavaScript` and `ActionScript` are both derived from `ECMAScript`, a standardized programming language maintained by Ecma with the ECMA-262 standard [8]. They are object-oriented, interpreted scripting languages. However, while `JavaScript` is mostly used for web applications and to extend functionality of third parties formats such as PDF, `ActionScript` is used as an essential support for delivering Flash-based content. In particular, `ActionScript` is mainly employed in SWF files, although it can also be employed in PDF files to show Flash animations inside a document. Moreover, `ActionScript` code is also compiled into a bytecode (called `ActionScript Bytecode` or `ABC`), as it is executed by a virtual machine that has been specifically designed by `Adobe`. `JavaScript` and `ActionScript` are actively used for exploiting vulnerabilities of the readers of the files that host them. For example, malicious `JavaScript` codes are commonly used to perform attacks against Adobe Reader, by exploiting the fact that such scripts are executed inside a PDF file. Such operation is performed in three steps: *(i)* the reader opens the file and executes the scripting code; *(ii)* the scripting code performs malicious actions by exploiting a vulnerability of the reader; *(iii)* if the vulnerability is correctly triggered, the malicious script may download another executable to infect the victim from a malicious URL, or it may directly execute a binary file *embedded* within the script itself.

Most of these attacks are performed by invoking system-based or application-specific APIs. The underlying reason is that some of the APIs themselves are vulnerable to attacks (e.g., the `collab.getIcon()` method used for PDF files). Likewise, system-based APIs can be used to manipulate memory, and they are often an essential element for performing attacks (e.g., the `flash.utils.ByteArray` class, which allows one to easily manipulate arrays of bytes, and which is often used in buffer overflow or heap-spraying attacks). In the following, we describe possible usages of `JavaScript` and `ActionScript` codes.

Examples of JavaScript code. Example 1 shows three ways of using `Javascript` code inside a PDF file. The functions and attributes belong to the Acrobat `JavaScript` API.

```
// get adobe version number
var version = app.viewerVersion;

// printing date
var d = new Date();
var sDate = util.printd("mm/dd/yyyy", d);

//exploiting a vulnerability (CVE-2009-4324)
try {this.media.newPlayer(null);} catch(e) {}
```

Example 1: Possible usages of the Acrobat `Javascript` API.

In the first case, the `app.viewerVersion` attribute can be used by a malware to infer the version of the PDF reader. The second case uses the `util.printd` function to print the current date. This function can be also used by a malware to fill the system memory. The third case is a popular example of vulnerability exploiting (`CVE-2009-4324`) in which, by passing the null parameter to the `media.newPlayer` function, the attacker may be eventually able to gain full control of the victim machine.

Examples of ActionScript codes. The next Example shows two ways of using the `ActionScript` system classes. These lines are rather frequent in malware as well.

```
import flash.utils.ByteArray
import flash.system.Capabilities

//Write bytes
var mem_block:ByteArray = new flash.utils.ByteArray();
mem_block.writeInt(0x41414141);

//Check if os is windows
var op_sys:String = flash.system.Capabilities.os
```

Example 2: Possible usages of the `ActionScript` system classes.

The first lines use the `writeInt` function belonging to the `flash.utils.ByteArray` system class to fill an array with integers. This is often used by malware for memory manipulation. The last line invokes the `flash.system.Capabilities` system class to infer the operating system executing the script. This is often used by malware, as some exploits are dependent on the operating system. Despite the differences between the two languages, these two examples show the role of system- and application-based API calls in characterizing the behavior of the malicious scripts.

3 API-Based Detection of Scripting Malware

We describe here a general methodology that can be applied to both `JavaScript` and `ActionScript` to detect the corresponding attacks. Our goal is to develop a system that, given an input file containing a `JavaScript` or `ActionScript` code,[1] is able to establish whether that file is malicious or not. This is done in three phases: (*i*) during *pre-processing*, the input file is analyzed (statically or with dynamic instrumentation, depending on the application) to extract the embedded scripting code; (*ii*) during *feature extraction*, each sample is represented

[1]In this paper, we only consider PDF and SWF files.

in terms of a feature vector, whose values correspond to the number of occurrences of each system API found inside the scripting code;[2] and (*iii*) during *classification*, the feature vector of the sample to be classified is provided as input to the machine-learning algorithm, which outputs a decision, i.e., classifies the input file either as benign or malicious. To this end, the machine-learning algorithm has to be previously *trained* on a (labeled) collection of malicious and benign files (called the *training set*) that should be sufficiently representative of the (never-before-seen) samples to be classified during operation. The aforementioned phases are further detailed in the following sections.

3.1 Preprocessing

Preprocessing is the operation with which the `JavaScript` or `ActionScript` files are detected and extracted for further analysis. This operation is performed differently, depending on the analyzed file. For **PDF files** containing `JavaScript`, we locate the scripting code by analyzing the internal structure of a PDF file. Typically, the presence of such code inside the PDF file is highlighted by keywords like `/JavaScript` or `/JS` (for more details, see the PDF specifications[3]). For **SWF files** containing `ActionScript`, we locate the equivalent `ActionScript` bytecode contained in the file by searching for a data structure called `DoABC Tag` (for more details, see the SWF specifications[4]). With respect to the `ActionScript` bytecode, it is worth noting that it contains scripting code that is semantically equivalent to the original source. For the purpose of feature extraction, directly analyzing the bytecode allows us to easily retrieve the API information without further decompilation.

3.2 Feature Extraction

We now describe the feature extraction methodology employed in our approach. The goal of this phase is counting the number of occurrences of each system API contained in the scripting file. The approach is a variant of the one described in [6], with some differences depending on the scripting file that is analyzed.
Javascript. For `JavaScript` codes contained in PDF files, we count the occurrences of the *methods* and *attributes* belonging to the `JavaScript` for Acrobat API list.[5] This is done by *dynamically* instrumenting the execution of the `Javascript` code inside the PDF file, so that all the invoked `JavaScript` APIs could be extracted. Note that we are not fully executing the PDF file, i.e., code extraction is statically performed.
Actionscript. For `ActionScript` scripts contained in SWF files, we count the occurrences of the *classes* belonging to the official `ActionScript 3` API list. This is done by *statically* analyzing the `ABC` bytecode in order to detect all the employed API.[6]

The two strategies have been tailored to the application domain to obtain a *compact* feature set, consisting respectively of 3272 and 2587 features for `JavaScript` and `ActionScript`. We point out that we did not consider the arguments of the system calls (especially with respect to `JavaScript`), as it would have considerably increased the complexity of our analysis. These feature sets are then reduced through feature selection, to obtain a more compact feature set and facilitate the training process of our classifiers, by tackling the so-called curse of dimensionality [4]. In particular, we exploit a feature selection criterion based on *information gain*, and

[2]In particular, for `JavaScript` code in PDF files, we consider the Javascript APIs for Adobe.
[3]`http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf`
[4]`http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf-file-format-spec.pdf`
[5]`http://www.adobe.com/devnet/acrobat/javascript.html`
[6]`http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/`

select the first 100 features with the highest *occurrence score* $S = |p(x_i|M) - p(x_i|B)|$, being x_i the i-th feature value, and M and B the sets of malicious and benign samples [4].

The selected features include functions, attributes and classes that are often used by malware to perform their actions. For example, selected features among `Flash` files are `flash.events.Event`, `flash.utils.ByteArray`, `Math` and other classes that are often used to manipulate memory to perform attacks. With respect to `JavaScript`, selected features include, among others, `app.ViewerVersion`, `app.['eval']`, `app.PlugIns`. Such features are often used to obfuscate code.

3.3 Classification

Different machine-learning algorithms can be exploited for our classification task. Although previous work has shown that non-linear classifiers such as Random Forest or SVMs with the RBF kernel typically perform better at this task than linear classifiers [10], this is not enough for our purposes. The reason is that our application is intrinsically *adversarial*, i.e., input data can be manipulated by a skilled attacker to evade detection during system operation. In particular, it may be possible for an attacker to modify a malicious file by adding features (i.e., API calls) typically used in benign files, with the goal of confusing the classifier by making the feature vector of the resulting malicious file more similar to those exhibited by benign files. Note also that, While it may be easy to add API calls to malicious files, removing them might compromise the intrusive functionality of the malware sample. We thus restrict ourselves to the case of feature increments in this work. This attack strategy is also known as *mimicry*, and it has been shown to be very effective against systems that are not designed to be robust against targeted evasion attempts [3].

To tackle this issue, we exploit an approach similar to that advocated in [2], named one-and-a-half-class (1.5C) classification. The underlying idea is to combine a two-class classifier with a one-class classifier to detect potential, anomalous samples during testing. In fact, most of the evasion samples constructed to evade detection by a two-class classifier can be considered anomalous with respect to the training (benign) data, and can be thus detected using this simple strategy. In particular, we build our 1.5C Multiple Classifier System (1.5C-MCS) using three distinct classifiers: (*i*) a Random Forest classifier trained on both *benign* and *malicious* data; (*ii*) a one-class SVM RBF trained only on *benign* data; and (*iii*) another one-class SVM RBF trained on the outputs of the aforementioned classifiers, using only *benign* data. The latter SVM will basically output an aggregated score to be thresholded to make the final decision.

4 Experimental Results

In this section, we report the experimental results that we attained by applying the methodology described in Sect. 3 on PDF and SWF files. We divided our experimental protocol into two parts: (*i*) a standard evaluation, in which we assessed the performance of our method on two datasets respectively including PDF and SWF files; (*ii*) an adversarial evaluation, in which we assessed the performance of our method against the mimicry attacks described in Section 3.3. We start by first describing how we pre-processed PDF and SWF files, and the datasets employed in our experiments. Then, we describe the results attained for each evaluation protocol.

Data Pre-processing. Data pre-processing was performed with two tools, depending on the file type. We used `PhoneyPDF`[7] to dynamically analyze PDF files and `JPEXS`[8] for SWF files.

[7] https://github.com/kbandla/phoneypdf

[8] https://www.free-decompiler.com/flash/

Both tools are open source and publicly available.

Datasets. We used two datasets in our experiments, respectively containing PDF and SWF files. For the PDF data, we collected 17826 PDF files, 12592 of which are malicious and the remaining 5234 are benign. These samples were collected until 2016 by using the `VirsuTotal`[9], `Malware don't need coffee`[10] and `Contagio`[11] services. It is worth noting that *all samples* in our dataset embed `JavaScript` code. Since benign files do not typically embed `JavaScript` code, it is reasonable to observe that their number is lower in our dataset (which in turn will provide a pessimistic evaluation of our false positive rates).

For the SWF data, we collected 6776 SWF files, 4425 of which are benign and the remaining 2351 are malicious. Differently from the previous case, there are more benign files than malicious ones, as `Flash`-based attacks have only considerably increased since 2015. These files were collected until 2016 by using the `VirusTotal` service. It is worth noting that each of these samples contains `ActionScript 3` code. This avoids that a file is simply recognized as benign because no `ActionScript` code is present.

4.1 Standard Evaluation

In this experiment, we assessed the performance of our approach for the two aforementioned datasets. Performances were evaluated in terms of true and false positive rates. The experiments were performed as follows. For each dataset, we randomly split the data in a training and a test set, respectively consisting of 70% and 30% of the total number of samples. This process was repeated five times, to avoid biases due to the quality of a specific training-test split. We used Random Forest and SVM RBF classifiers, as described in Sect. 3.3. The parameters of each classifiers were optimized through a 5-fold cross validation performed on the training set. For each split, we classified the test set and calculated the average Receiver Operating Characteristic (ROC) curve, which displays the true positive rate (i.e., the fraction of detected malicious files) against the false positive rate (i.e., the fraction of misclassified benign samples).

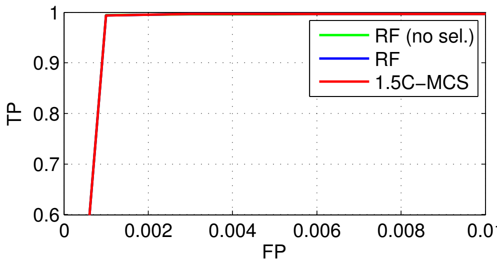
In Fig. 1, we report the results on the `JavaScript` and `ActionScript` data for Random Forests trained either using all features or only using the first 100 features selected with the strategy described in Sect. 3.2), and for the 1.5C-MCS described in Sect. 3.3. Notably, there are clear differences between the results attained on `JavaScript` and `ActionScript`. In particular, although the results are very good for both languages, classifying `ActionScript` files is significantly more difficult than classifying `JavaScript` files. The reason is that benign and malicious `ActionScript` files are more similar, in terms of API calls, than their `JavaScript` counterparts.

Selecting features allows one to attain better performances with Random Forests for `Action-Script` codes. The 1.5C-MCS exhibits essentially the same performance of the best Random Forest classifier. This was somehow expected, as the one-class component has been designed specifically to detect targeted, anomalous attacks that significantly deviate from benign data.

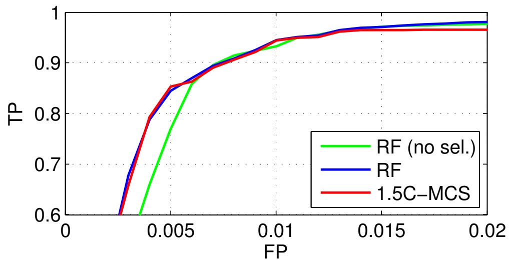
4.2 Adversarial Evaluation

In this experiment, we tested the resilience of the proposed approach against the mimicry attacks described in Section 3.3. To perform our evaluation, we used the same setup of the previous experiment: each dataset was split into training and test set multiple times. The parameters of the classifiers were evaluated by means of a 5-fold cross validation performed

[9] `https://virustotal.com/`
[10] `http://malware.dontneedcoffee.com/`
[11] `http://contagiodump.blogspot.it/`



(a) Performances on `JavaScript`



(b) Performance on `ActionScript`

Figure 1: ROC curves for PDF files embedding `JavaScript` (left) and for SWF files embedding `ActionScript` (right). These curves report the results for Random Forests (either by using all the features or the 100 most discriminant ones, selected accordingly to the the strategy in Section 3.2) and for the 1.5C-MCS described in Section 3.3.

.

on the training set. Finally, the malicious files of the test set were modified according to the *mimicry* strategy.

It is worth noting that we are not building the *real* sample corresponding to the manipulated malicious file, but we are only simulating the effect of the attack at the feature level, i.e., we are simulating changes in the feature values of each malicious sample that can be *practically* implemented also to build a real malware sample. In particular, we only consider *adding* API calls from benign samples. As mentioned in Section 3.3, removing API calls from a malicious sample may compromise the intrusive functionality of the embedded exploitation code.

Fig. 3 shows how the performance of the considered classifiers decreases as the number of benign samples added to a malicious file increases. Classifier performance is measured in terms of detection rate at a given false positive rate for each classifier (for `JavaScript`, we set $FP = 0.1\%$, whilst for `ActionScript` we set $FP = 1\%$). Clearly, the performance of more secure classifiers should decrease more gracefully as the number of added benign files increases.

The first thing to observe is that the attack is tremendously effective against the `ActionScript` dataset. On the `JavaScript` dataset the effect is lower, but it can be increased by raising up the amount of added samples (up to 100, see [6]). Random Forests classifiers are considerably vulnerable to this attack, while the 1.5C-MCS remains significantly secure. The underlying reason is that, in the latter case, the one-class SVM is able to correctly spot the anomalous behavior of the attack samples with respect to the rest of the training data used to learn the classifier. To better explain this phenomenon, in Fig. 2 we report a scatter plot that depicts benign (blue points), malicious (red points) and attack (green points) data in the space characterized by the outputs of the two combined classifiers. In addition, the decision function of the 1.5C-MCS is also shown. Differently to what happens with SVM RBF and stand-alone Random Forest, the circular shape of the 1.5C-MCS encloses all the benign samples, so that malicious and attack samples (which are located in a different position compared to standard malicious samples - see the green points) are considered anomalous. Notably, while the scores assigned to the attack samples by the Random Forest classifier are closer to those assigned by the same classifier to the benign data (i.e., they would evade detection by this classifier), the one-class SVM is able to well-separate them from the rest of the data. This also applies to the 1.5C-MCS, which is able to correctly assign a high score value to the attack samples, and, therefore, to successfully detect them.

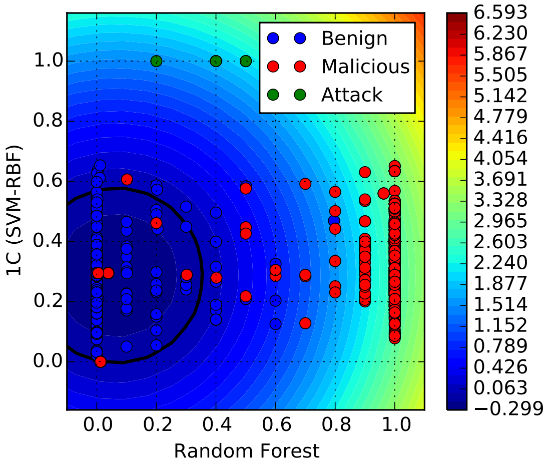
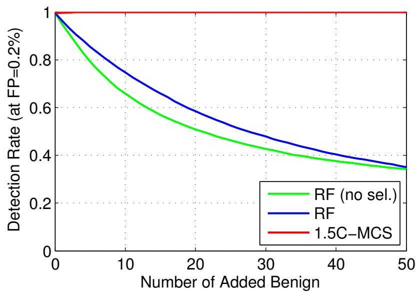
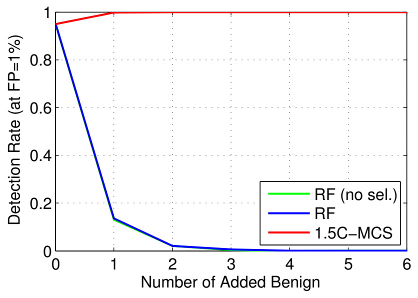


Figure 2: The decision function of our 1.5C-MCS (shown in colors) in the bi-dimensional space spanned by the outputs of the combined classifiers. The decision boundary is highlighted with a solid black line, while blue and red points respectively represent benign and malicious files. Malicious files manipulated with the mimicry attack strategy are reported as green points.



(a) Performances on `JavaScript` (fp=0.1%)



(b) Performances on `ActionScript` (fp=1%)

Figure 3: Detection rate of our classifiers against *mimicry* attacks in which an increasing number of benign samples is added to each malware sample, for `JavaScript` and `ActionScript` files.

5 Discussion and Limitations

The main goal of this paper is showing that information extracted from system API calls, methods and classes can be useful to discriminate between malicious and benign files. However, there are some limitations, especially with respect to the `ActionScript` analysis. As detecting obfuscated files was not our primary goal for this paper, we chose static analysis for Flash files. Using dynamic analysis frameworks such as `Sulo`[12] or `Lightspark`[13] would have been more effective, but overly complex for our purposes (on the contrary, dynamic instrumentation with

[12]https://github.com/F-Secure/Sulo

[13]https://github.com/lightspark/lightspark

PhoneyPDF is rather lightweight). Moreover, JPEXS also employs some static deobfuscation routines that might help detecting obfuscated files. We plan to test our approach against obfuscated Flash files in future work.

The attained results show how the choice of the features and of a proper learning algorithm is crucial to develop systems that are both accurate and robust against evasion attempts. However, in this paper we only evaluate simple mimicry attacks that do not exploit any specific knowledge of the targeted learning algorithm. Conversely, more complex attacks exploit this knowledge in order to increase the probability of evasion while performing a minimal number of modifications to the attack samples [3]. Differently from the attack strategy proposed in this paper, the gradient-descent strategy proposed in [3] only modifies the most discriminant features for the classifier to induce a higher performance decrease with a smaller amount of feature manipulations. Obviously, this requires the attacker to know in detail the features and the type of classifier used. In more practical scenarios, the knowledge of the attacker is more limited, and it may thus be necessary to perform more changes to the attack samples to successfully evade the system.

It is also worth noting that, as the API are publicly available and documented, they might be used in both benign and malicious files. This means that if an attacker was able to craft a malicious sample that looks *exactly* the same to a benign one (in terms of features), the system would be evaded whatsoever. However, performing such operation might be extremely hard. For instance, it may be necessary to replace some APIs with semantically equivalent ones, which may however influence other feature values.

Finally, we plan to extend the output provided by our approach by pointing out which API contributed the most to the classifier decision. At the moment, the user can only see whether or not the file is malicious.

6 Related Work

We discuss here some relevant previous work related to the detection of malicious JavaScript and ActionScript files.

JavaScript Detection. Cova et al. [7] devised a dynamic analysis system that executes JavaScript code from HTML and PDF files, in order to detect malicious activities. To this end, they extracted information related to code obfuscation and analyzed API calls in terms of their sequences and arguments. The emulation of the JavaScript content was performed through JSand,[14] and the extracted information was subsequently used to learn a Bayesian classifier.

The approach proposed in this paper is substantially different. We only observe API calls related to the Adobe JavaScript API (our approach is tailored to PDF detection only). Second, we extract features related to the occurrence of API calls, without looking for other obfuscation-related characteristics or for the arguments of the calls. Another static and dynamic approach to detect JavaScript code inside PDF files was introduced by Tzermias et al. with MDScan [11]. In this case, the scripting code was instrumented with SpiderMonkey[15] to detect shellcodes, which are further analyzed and executed by Libemu[16]. Laskov et al. developed PJScan, a static system to analyze lexical information extracted from JavaScript code to detect malicious PDF files. LuxOR is the system that containts the strategy strategy that has been extended in this work [6]. It focused on JavaScript detection in PDF files by analyzing discriminant APIs.

[14] http://demo-jsand.websand.eu/
[15] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
[16] https://github.com/buffer/libemu

ActionScript Detection. There are two main works for the detection of malicious `ActionScript` inside SWF files. The first one (`FlashDetect`), by Overveldt et al. [12], resorted to dynamic emulation of the SWF files to extract features. This was done in order to extract suspicious function calls that could be useful for classification. Wressnegger et al. developed `Gordon`, a system that statically analyzes the control flow graph of the `ActionScript` code and uses n-grams of instructions and parameters as features for classification. Nevertheless, neither `FlashDetect` nor `Gordon` have been publicly released.

7 Conclusions

In this paper, we have introduced a methodology to detect malicious `JavaScript` and `ActionScript` codes contained in PDF and SWF files. This methodology leverages similarities between the two scripting languages and extracts information from system API methods, attributes and classes. Moreover, the system has been designed from the ground up to be secure, according to the so-called security-by-design principle, by explicitly considering the potential presence of targeted, evasive attacks during system operation. Our empirical results on PDF and SWF files embedding `JavaScript` and `ActionScript` codes have shown that our methodology allows one to detect a very high percentage of malicious files, while only misclassifying a small fraction of benign samples. We have also shown that, by explicitly considering carefully-crafted attacks against our system, it is possible to design a more secure learning-based detector. In future work, we plan to test our approach against obfuscated samples and more sophisticated evasive attacks.

References

[1] ArsTechnica. Hacking teams flash 0-day: Potent enough to infect actual chrome user, 2015.

[2] B. Biggio, I. Corona, Z. He, P. P. K. Chan, G. Giacinto, D. S. Yeung, and F. Roli. One-and-a-half-class multiple classifier systems for secure learning against evasion attacks at test time. In *MCS*, pages 168–180, 2015.

[3] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *ECML PKDD*, pages 387–402, 2013.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1 edition, 2007.

[5] Cisco. Annual security report, 2016.

[6] I. Corona, D. Maiorca, D. Ariu, and G. Giacinto. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *AISec*, pages 47–57, 2014.

[7] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW*, pages 281–290, 2010.

[8] Ecma International. Ecmascript language specification (7th edition), 2016.

[9] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *ACSAC*, pages 373–382, 2011.

[10] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *ACSAC*, pages 239–248, 2012.

[11] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *EUROSEC*, pages 4:1–4:6, 2011.

[12] T. Van Overveldt, C. Kruegel, and G. Vigna. Flashdetect: Actionscript 3 malware detection. In *RAID*, pages 274–293, 2012.

[13] C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. Comprehensive analysis and detection of flash-based malware. In *DIMVA*, pages 101–121, 2016.