Computers & Security 96 (2020) 101901

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Adversarial Detection of Flash Malware: Limitations and Open Issues

Davide Maiorca[a,1,*], Ambra Demontis[a,1], Battista Biggio[a,b], Fabio Roli[a,b], Giorgio Giacinto[a]

[a] *Department of Electrical and Electronic Engineering, University of Cagliari, Piazza dArmi Cagliari, 09123, Italy*
[b] *Pluribus One, Italy*

ARTICLE INFO

Article history:
Received 17 October 2019
Revised 22 May 2020
Accepted 23 May 2020
Available online 26 May 2020

Keywords:
Adobe Flash
Malware Detection
Secure Machine Learning
Adversarial Training
Computer Security

ABSTRACT

During the past four years, Flash malware has become one of the most insidious threats to detect, with almost 600 critical vulnerabilities targeting Adobe Flash Player disclosed in the wild. Research has shown that machine learning can be successfully used to detect Flash malware by leveraging static analysis to extract information from the structure of the file or its bytecode. However, the robustness of Flash malware detectors against well-crafted evasion attempts - also known as adversarial examples - has never been investigated. In this paper, we propose a security evaluation of a novel, representative Flash detector that embeds a combination of the prominent, static features employed by state-of-the-art tools. In particular, we discuss how to craft adversarial Flash malware examples, showing that it suffices to manipulate the corresponding source malware samples slightly to evade detection. We then empirically demonstrate that popular defense techniques proposed to mitigate evasion attempts, including re-training on adversarial examples, may not always be sufficient to ensure robustness. We argue that this occurs when the feature vectors extracted from adversarial examples become indistinguishable from those of benign data, meaning that the given feature representation is intrinsically vulnerable. In this respect, we are the first to formally define and quantitatively characterize this vulnerability, highlighting when an attack can be countered by solely improving the security of the learning algorithm, or when it requires also considering additional features. We conclude the paper by suggesting alternative research directions to improve the security of learning-based Flash malware detectors.

1. Introduction

Malware detection is still a critical priority for researchers and industry, as countless of polymorphic attacks are continually being released (Symantec, 2019). In particular, a very insidious threat is represented by *infection vectors*, i.e., files that exploit vulnerabilities of third-party applications to drop or execute malicious executables. Such vectors are typically documents (e.g., PDF, Office) or multimedia files (e.g. Flash), and attackers leverage their structure to conceal scripting codes that exploit the target vulnerabilities. For example, pages of PDF documents can hide malicious JavaScript codes or even additional executables (Maiorca and Biggio, 2019; Maiorca et al., 2019).

To counteract such attacks, researchers proposed solutions based on machine-learning algorithms applied to information extracted by employing static or dynamic analysis of the embedded code, with excellent results for document-based infection vectors (e.g. Maiorca et al., 2012; Smutz and Stavrou, 2012; Smutz and Stavrou, 2016; Šrndić and Laskov, 2013). However, research also showed that such detection approaches have several *robustness* issues against evasion attacks, namely, well-crafted manipulations of the input samples at test time (Biggio et al., 2013; 2014b; Biggio and Roli, 2018) (also recently referred to as *adversarial examples* in the context of deep learning (Goodfellow et al., 2015; Grosse et al., 2017; Szegedy et al., 2014)). The efficacy of such attacks depends on the information that the attacker possesses about the system, and on her ability to perform enough changes to the extracted features.

The robustness of machine learning approaches against adversarial attacks has been assessed in various case studies. In particular, some works employed code obfuscation through off-the-shelf tools or custom techniques (Kapratwar. et al., 2017; Lin and Stamp, 2011; Maiorca et al., 2015; Scalas et al., 2019; Singh et al., 2016) to perform evasion. Other approaches directly targeted the learning algorithms by employing *black-box* and *white-box* evasion techniques (e.g., by using algorithms such as gradient-descent). Popular case-studies in which these techniques have been systematically studied are the detection of malicious PDF files

[*] Corresponding author.
E-mail addresses: davide.maiorca@unica.it (D. Maiorca), ambra.demontis@unica.it (A. Demontis), battista.biggio@unica.it (B. Biggio), roli@unica.it (F. Roli).
[1] Those authors contributed equally to this manuscript.

https://doi.org/10.1016/j.cose.2020.101901
0167-4048/© 2020 Elsevier Ltd. All rights reserved.

(Biggio et al., 2013; Maiorca and Biggio, 2019; Šrndic and Laskov, 2014a; Xu et al., 2016) and the detection of Android malware Demontis et al., (Calleja et al., 2018; Yang et al., 2017).

Multimedia malware, in particular in the Flash (also known as Adobe Small Web Format - SWF) format, has been studied considerably less from the perspective of adversarial attacks (in particular, against white- and black-box attacks). One of the reasons for such a lack of study is that, albeit Flash malware caused a massive uproar starting from 2015 (due to the significant increment of vulnerabilities for Adobe Flash Player), its technology is going to be discontinued in 2020. Nevertheless, this technology is still active from the perspective of malware-based attacks. Flash-based attacks represent excellent examples of highly obfuscated and evasive infection vectors. Critical vulnerabilities are still released for the Flash platform, and very dangerous attacks are still possible, such as the one that targeted North Korea in 2018 (TrendMicro, 2018).

We believe not only that it will take years before Flash completely stops being used, but also that analyzing Flash attacks constitutes an excellent test-bench for machine learning-based detection. The main reason for this claim is that most infection vectors (including the most used ones at the moment - e.g., PDF, Office documents, etc.) share a similar organization, composed of an external *structure* and a code-based *content*. Research on such infection vectors showed that the analysis of their structure and/or content led to encouraging results in their detection, and the Flash format is no exception in this (Maiorca et al., 2019; Šrndić and Laskov, 2016; Wressnegger et al., 2016). Hence, the lessons learned from the study of adversarial attacks on formats like Flash can provide precious insights into the robustness of systems to detect other file formats.

In this paper, we provide an in-depth analysis of Flash-based learning systems, focusing on their robustness against adversarial attacks. Our analysis aims to give interesting insights into how to design more secure systems for detecting infection vectors, and to point out possible vulnerabilities in the feature representations and the classification algorithms.

To this end, we first propose a representative system for Flash malware detection, named FlashBuster. It is a static machine-learning system that employs information extracted by both the structure and the content of SWF files. Such an approach allows for a more comprehensive assessment of the extracted static information by representing and combining the contents employed by previous state-of-the-art systems (Šrndić and Laskov, 2016; Wressnegger et al., 2016). We show that FlashBuster could detect the majority of malware samples in the wild, by obtaining comparable performances to other systems at state of the art, and demonstrate that it can predict previously unseen attacks. We also tested FlashBuster against popular obfuscation techniques, showing that our approach could also be employed in the presence of obfuscated malware.

We then evaluate FlashBuster robustness by simulating evasion attacks that leverage the knowledge that the attacker may possess about the targeted learning system (Biggio et al., 2013; 2014b; Chen et al., 2017; Dang et al., 2017; Papernot et al., 2017; Tramèr et al., 2016; Xu et al., 2016), against an increasing number of modifications to the input samples. The corresponding *security evaluation curves*, depicting how the detection rate decreases against attack samples that are increasingly manipulated, allow us to understand and assess the vulnerabilities of FlashBuster under attack.

We finally discuss the effectiveness of *adversarial training* against such evasive attacks. To this end, we re-trained FlashBuster on the evasion attack samples used against it, and surprisingly show that this strategy may be ineffective in some cases. We argue that this is due to an *intrinsic vulnerability* of the feature representation, i.e., to the fact that evasion attacks entirely mimic the feature values of benign data, thus becoming *indistinguishable* for the learning algorithm. We define this vulnerability in formal terms, and quantitatively evaluate it by defining a specific metric that measures the extent to which the attack samples converge towards benign data.

Our findings highlight a crucial problem that must be considered when designing secure machine-learning systems, i.e., that of evaluating *in advance* the *vulnerability of the given features*. Indeed, vulnerable information may compromise the whole system even if the employed decision function is robust. In this respect, we sketch possible research directions that may lead one to design more secure machine learning-based malware detectors.

Finally, we note that we publicly released FlashBuster, together with the features employed in the experiments for this paper.[2]

The rest of this paper is structured as follows. Section 2 provides the basics to understand the SWF format and an example of ActionScript code. Section 3 describes the related work in the field. Section 4 describes the architecture of FlashBuster. Section 5 describes the threat model in relation to adversarial environments. Section 6 describes the evasion attacks employed in this paper. Section 7 discusses the vulnerabilities that affect learning-based systems, and introduces a quantitative measure of feature and learning vulnerabilities. Section 8 provides the experimental evaluation. Section 9 provides a discussion on the attained results. Section 10 closes the paper by sketching possible future work.

2. ShockWave Flash File Format

Small Web Format (SWF) is a file type that efficiently delivers multimedia contents, and it is processed by Adobe Software such as Adobe Flash Player.[3]

SWF files are composed of three essential elements: *(i)* a *header* that describes important file properties such as the presence of compression, the version of the SWF format, and the number of video frames; *(ii)* a list of *tags*, i.e., data structures that establish and control the operations performed by the reader on the file data; *(iii)* a unique tag called `End` that terminates the file.

Some tags define actions such as pressing a button, moving the mouse, etc. These actions can be expanded further by employing a scripting language called ActionScript. ActionScript code (the latest release is 3.0) is compiled to a bytecode that is run by the ActionScript Virtual Machine 2 (`ASVM 2`). The computation in the `ASVM 2` is based on the execution of *method bodies* composed of *instructions*. Each method body runs in a specific *context* that defines information such as default parameters. More about SWF and `ASVM 2` can be found on the official SWF and VM references (Adobe, 2007; 2012).

2.1. ActionScript in SWF

In order to better understand the role of ActionScript in SWF files, Listing 1 shows a small snippet of code that is typically found in an ActionScript-based malware, where `ByteArray` structures are accessed. Such structures are employed by malware to store information about encrypted/decrypted URLs and payloads.

The code in the Listing reads an `UnsignedByte` from the object `_loc1_`, which belongs to the class `IG` (which inherits from the `ByteArray` class - see the `coerce` bytecode instruction). The code then performs a subtraction and assigns the output to the variable `isAS3`. This result will then be copied to another array of bytes (we did not report this action for space reasons). Note how the reading is performed by following the *little endian* (using the

[2] https://github.com/davidemaiorca/flashbuster.

[3] https://get.adobe.com/en/flashplayer/.

```
 1 _loc1_ = new IG();
 2 _loc1_.endian = Endian.LITTLE_ENDIAN;
 3 _loc1_.position = 0;
 4 this.isAS3 = _loc1_.readUnsignedByte() - 1;
 5
 6
 7 findpropstrict Qname(PackageNamespace(""),"IG")
 8 constructprop Qname(PackageNamespace(""),"IG") 0
 9 coerce Qname(PackageNamespace("flash.utils"),"ByteArray"
     )
10 setlocal_1
11 getlocal_1
12 getlex Qname(PackageNamespace("flash.utils"),"Endian")
13 getproperty Qname(PackageNamespace(""),"LITTLE_ENDIAN")
14 setproperty Qname(PackageNamespace(""),"endian")
15 getlocal_1
16 pushbyte 0
17 setproperty Qname(PackageNamespace(""),"position")
18 getlocal_0
19 getlocal_1
20 callproperty Qname(PackageNamespace(""),"
     readUnsignedByte") 0
21 decrement
```

Listing 1. An example of ActionScript. This code snippet is represented by its decompiled output (lines 2-5) and by its equivalent bytecode output (lines 11-25).

system-related `flash.utils.Endian` package) byte order. We point out that system API methods and classes are often essential for the attacker to build shellcodes or perform buffer overflows and heap spraying attacks. In fact, official ActionScript APIs allow managing low-level data structures efficiently, so attackers do not need to implement their memory management routines.

From the bytecode perspective, to resolve correctly the package belonging to a specific method or class, the ActionScript Virtual Machine resorts to data structures called `Names`. Such structures are composed of one *unqualified name* (for example, a class name) and one or more *namespaces* that typically represent the *packages* from which classes or methods are resolved. Normally, `Name` resolution occurs at compile time by associating one package to one class or method (`QName`). However, there are cases in which `Names` are resolved at runtime, in particular when there are multiple namespaces (packages) from which the same unqualified name (class) can be obtained.

3. Related Work

In the following, we provide a comprehensive review of the state-of-the-art approaches employed for Flash malware detection. Then, we also describe the prominent works in the field of adversarial machine learning.

3.1. Flash Malware Detection

Even if Flash-based malicious attacks started to grow considerably in 2015, the number of detection approaches is rather limited. `FlashDetect` (Overveldt et al., 2012) is one of the first approaches to the detection of ActionScript 3-based malware. `FlashDetect` uses `Lightspark`, an open-source Flash viewer, to perform dynamic analysis of malicious Flash files. `FlashDetect` was employed inside the `Wepawet` service, which is not available anymore.

`Gordon` (Wressnegger et al., 2016) is an approach that resorts to guided-code execution to detect malicious SWF files, by statically analyzing their ActionScript bytecode (without considering the file structure). In particular, the system selects the most suspicious security paths from the control flow graph of the code. Such paths usually have references to security-critical calls, such as the ones for dynamic code loading. To the best of our knowledge, `Gordon` is not publicly available.

`Hidost` (Šrndic, 2016; Šrndić and Laskov, 2016) is a static system that only focuses on the structure of the SWF file, without analyzing its ActionScript bytecode. More specifically, it considers sequences of objects belonging to the structure of the SWF file as features. The system evaluates the most occurring paths in the training dataset and extracts features based on the training data. Relying on such data might be dangerous from the perspective of targeted attacks, as a malicious test file with entirely different paths might evade detection.

According to the results in Šrndić and Laskov (2016); Wressnegger et al. (2016), both `Gordon` and `Hidost` performed well at detecting Flash malware. For this reason, we designed FlashBuster as a simplified extension of the aforementioned static systems, where information from both the structure and content of the file is extracted, and where the extracted features do not depend on the training distributions.

Besides research-based prototypes, some off-the-shelf tools are often used to analyze SWF files, such as `JPEXS` (Petrik, 2020), `PySWF` (Timknip, 2017), `SWFReTools` (Sporst, 2011) and others. It is also possible to perform obfuscation of SWF files by employing `DoSWF` (DoSWF, 2013). In particular, this tool can either conceal variable names or introducing modifications to the flow graph that do not alter the overall semantics of the code. Additionally, it can perform full encryption of SWF files by dynamically loading them in memory at runtime.

3.2. Adversarial Machine Learning

Several works doubted the security of machine learning since 2004 (Biggio and Roli, 2018). The first two works in the field were proposed by Dalvi et al. (2004) in 2004 and by Lowd and Meek (2005) in 2005. Those works, carried out in the field of spam filtering, demonstrated that it could be easy for the attackers to deceive classifiers at test time (*evasion* attacks) by performing a limited amount of fine-grained changes to emails. Following works (Barreno et al., 2010; 2006; Biggio et al., 2014b) proposed attacker models and frameworks that are currently employed to assess the security of learning-based systems, also against training-time (*poisoning*) attacks. The first gradient-based evasion (Biggio et al., 2013) and poisoning (Biggio et al., 2012) attacks were proposed by Biggio et al. respectively in the 2013 and 2012. In Biggio et al. (2013), the authors introduced two important concepts that are currently adopted in the adversarial field, namely *high-confidence* adversarial examples and the use of a *surrogate* model. The over-mentioned works anticipated the discovery of the so-called *adversarial examples* against deep neural networks (Goodfellow et al., 2015; Szegedy et al., 2014). The vulnerability to evasion attacks has been then especially studied on learning systems designed to detect malware samples (for example, on PDF files (Maiorca et al., 2019; 2013; Šrndic and Laskov, 2014b; Xu and Kim, 2017; Xu et al., 2016)), thus raising serious concerns about their usability under adversarial environments. These assessments have been recently expanded to systems for Android malware detection Demontis et al., (Calleja et al., 2018; Melis et al., 2018; Scalas et al., 2019; Yang et al., 2017) and to binary executables (with the adoption of deep learning) (Kolosnjaji et al., 2018; Raff et al., 2017).

4. FlashBuster Architecture

FlashBuster is a static, machine learning-based system whose aim is to detect malicious SWF files and to distinguish them from benign ones. Its main goal is reproducing a combination of the characteristics of previous state-of-the-art systems (see Section 3) that proved to be effective in detecting SWF malware, in order to assess the efficacy and robustness of the information extracted to recognize these attacks. To this end, FlashBuster leverages information provided by the tag structure and the ActionScript bytecode of

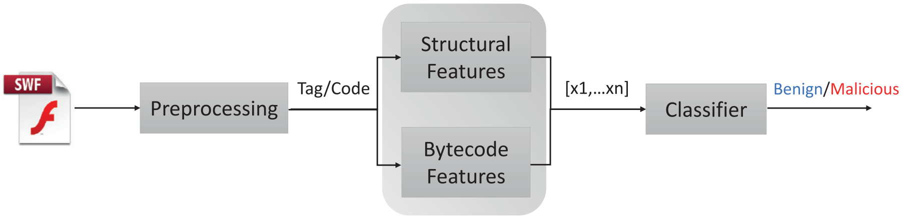


Fig. 1. Graphical architecture of FlashBuster.

the file. Fig. 1 shows the general architecture of the system, which can be divided into three modules:

Parser. This module analyzes the SWF file and extracts information about its structure and its ActionScript bytecode.

Feature Extractor. This module transforms the information obtained from the parser in a vector of numbers, which characterizes the whole SWF file.

Classifier. This module decides on the maliciousness of the SWF file according to the feature vector it receives as input. Such a module is a mathematical function that tunes its parameters by receiving various examples taken from a so-called *training set*. Once its parameters have been set up, the classifier can recognize malicious files that have not been included in the training examples.

In the following, we provide a more detailed description of each component of the system.

4.1. Parser

As previously said, this module performs data pre-processing and selects the information that will be further processed by the other modules. FlashBuster leverages a modified version of JPEXS, a powerful, Java-based Flash disassembler and decompiler (Petrik, 2020). This software is based on `RABCDasm` (CyberShadow, 2019), one of the most popular Flash disassemblers, and it adds new features such as de-obfuscation, file debugging and preview, etc.

More specifically, the parser featured by FlashBuster executes the following operations: *(i)* it performs static de-obfuscation of ActionScript code. This operation is important, as some malicious files might use name obfuscation or other popular techniques to conceal attacks; *(ii)* it extracts the complete SWF structure in terms of *tags; (iii)* it disassembles the ABC bytecode so that it could be read as a plain-text file. This operation includes automatic de-obfuscation of the ActionScript code. Both the tag structure and the ABC bytecode are sent to the feature extractor module for further analysis.

4.2. Feature Extraction

This module represents the core of the whole system, and it converts the information extracted by the parser to a vector of numbers that will be sent to the classifier. Our goal here was not devising a completely novel feature set, but proposing a comprehensive approach that could be comparable, in terms of detection performances, to other state-of-the-art approaches. For this reason, we referred to Wressnegger et al. (2016) to implement a conceptually similar system that would employ information extracted from the structure and the content of the SWF file. However, differently to Wressnegger et al. (2016), we extracted the *number of occurrences* of the extracted information, instead of analyzing their sequences. This intuition derives from the experience in PDF malware detection (Maiorca et al., 2019), where the occurrences of information extracted from the structure and the content of a PDF file proved to be greatly effective in performing detection. We now provide a detailed description of the features extracted in the two cases.

4.2.1. Structural Features (Tags)

These features are related to the information that can be extracted from SWF tags and are crucial to understanding which objects and actions are executed by the SWF file. The main idea here is that malware does not contain particularly complex multimedia contents, such as video with many frames or audio files. Various malware samples simply display images such as rectangles or blank backgrounds. For this reason, we extracted the following 14 features from the file structure, corresponding to the *number of occurrences* of specific SWF tags within the file[4]:

Frames. `ShowFrame` tags that are used to display frames.

Shapes. `DefineShape` (in any of all its four variants) tags, used to define new shapes that will be plotted on the screen.

Sounds. Sound-related events, extracted by examining any of the following tags: `DefineSound`, `SoundStreamHead1`, `SoundStreamHead-2` and `SoundStreamBlock`.

BinaryData. Groups of embedded data, represented by the tag `DefineBinaryData`.

Scripts. ActionScript codes that are contained in the file. Note that a SWF file does not necessarily require scripting code to perform its operations, especially in benign files (ActionScript has been initially devised as an aid to the execution of SWF files). Scripts are discovered by analyzing the following tags: `DoABC`, `DoABCDefine`, `DoInitAction`, `DoAction`.

Fonts. Font-related objects, extracted by detecting any of the following tags: `DefineFont` (in all its variants), `DefineCompactedFont`, `DefineFontInfo` (in all its variants), `DefineFontName`.

Sprites. Sprites extracted by examining the tag `DefineSprite`.

MorphShapes. Morphed shapes (i.e., shapes that might transform into new ones) extracted by examining the tag `DefineMorphShape` (and its variants).

Texts. Text-related objects, extracted by checking any of the following tags: `DefineText` (along with its variants) and `DefineEditText`.

Images. Images contained in the file, extracted by examining any of the following tags (and their variants): `DefineBits`, `DefineBitsJPEG`, `JPEGTables` and `DefineBitsLossless`.

Videos. The number of embedded videos in the file, extracted by examining the tags `DefineVideoStream` and `VideoFrame`.

Buttons. Buttons with which the user can interact with the SWF content. Such counting can be done by examining the tag `DefineButton` (along with its variants).

Errors. Errors made by the parser when analyzing specific tags. Such errors often occur, for example, when the SWF file is malformed due to errors in its compression.

Unknown. Tags that do not belong to the SWF specifications (probably malformed tags).

It is worth noting that we preferred counting the occurrences of each tag (instead of just considering its presence/absence) because

[4] The reader can find more information about these tags on the official SWF specification (Adobe, 2012).

we observed that benign objects contain significantly more tags of the same type in comparison to malicious files. We also observe that structural features must be carefully treated, as benign and malicious files can be similar to each other concerning their tag structure. For this reason, structural features alone are not enough to ensure reliable detection and must be integrated with information from the scripted content of the file.

4.2.2. Actionscript Bytecode Features (API calls)

As structural features (i.e., tags) might suffer from the limitations mentioned in Sect. 4.2.1, we employed an additional set of features that focus on the content of the scripting code that might be included in the files. Although it is not strictly necessary to use ActionScript for benign operations, its role is essential to execute attacks. In particular, as shown in Sect. 2.1, the attacker usually needs to resort to system APIs to perform memory manipulation or to trigger specific events. Moreover, APIs can be used to communicate with external interfaces or to contact an external URL to automatically drop malicious content on the victim's system.

System APIs belong to the official Adobe ActionScript specifications (Adobe, 2015). For this reason, we created an additional feature set that counts the classes and methods belonging to such specifications, leading to 4724 new features. More specifically, this feature set represents the number of specific System methods and classes inside the bytecode. We chose to use only system-based APIs for two reasons: (*i*) the feature vector does not include user-defined APIs, so that the feature list is independent of the training data that is considered for the analysis; (*ii*) system-based calls are more difficult to obfuscate, as the user does not directly implement them.

With respect to the example described in Sect. 2.1, we would therefore consider as valid features the classes `flash.utils.ByteArray` and `flash.utils.Endian`, and the method `readUnsignedByte`. On the contrary, we would not directly consider the class name `IG`, as it was directly implemented by the user. The rationale behind *counting* the occurrences of system-based methods and classes is that an attacker might systematically use functions to manipulate memory or perform suspicious actions. Alternatively, she might attempt to trigger events repeatedly or to access specific interfaces.

Finally, we observe that all features were normalized with the popular tf-idf strategy (Baeza-Yates and Ribeiro-Neto, 1999). This normalization is particularly crucial for SVM classifiers, which typically perform best with normalized features.

4.3. Classification

The features extracted with FlashBuster can be used with different classification algorithms. In the experimental evaluation that we describe in Section 8, we report the results for different classification algorithms. In particular, we focused our attention on SVM (linear and non-linear) and Random Forests, as these were successfully employed in other works on SWF detection (e.g., Wressnegger et al., 2016). Although other classifiers (or even their ensembles) may be employed, we believe that the chosen set is representative of the majority of classifiers in the wild and that similar results would hold for possible alternatives to classification Maiorca and Biggio (2019).

5. Attack Model

To assess the security of FlashBuster against adversarial manipulation of the input data (which can be either performed at training time or at test time), we leverage an attack model originally defined in the area of adversarial machine learning (Biggio et al., 2014a; 2014b; Biggio and Roli, 2018). It builds on the well-known taxonomy of Barreno et al. (Barreno et al., 2010; 2006; Huang et al., 2011) which categorizes potential attacks against machine-learning algorithms along three axes: *security violation, attack specificity* and *attack influence*. By exploiting this taxonomy, the attack model enables defining various potential attack scenarios, in terms of explicit assumptions on the attacker's goal, knowledge of the system, and capability of manipulating the input data.

5.1. Notation

For the sake of clarity, we report here a table of the symbols that will be used during the rest of the paper, along with a brief description. Such a table can be employed as a reference during the reading of the following sections.

5.2. Attacker's Goal

It is defined in terms of two characteristics, i.e., security violation and attack specificity.

Security violation. In security engineering, a system can be violated by compromising its *integrity, availability*, or *confidentiality*. Violating the integrity of FlashBuster amounts to having malware samples undetected; its *availability* is compromised if it misclassifies benign samples as malware, causing a denial of service to legitimate users; and *confidentiality* is violated if it leaks confidential information about its users.

Attack specificity. The specificity of the attack can be *targeted* or *indiscriminate*, based on whether the attacker aims to have only specific samples misclassified (e.g., a specific malware sample to infect a particular device or user), or if any misclassified sample meets her goal (e.g., if the goal is to launch an indiscriminate attack campaign).

We formalize the attacker's goal here in terms of an objective function $\mathcal{W}(\mathcal{A}', \boldsymbol{\theta}) \in \mathbb{R}$ (where $\boldsymbol{\theta}$ is the knowledge possessed by the attacker about the system), which evaluates to what extent the manipulated attack samples $\mathcal{A}'$ meet the attacker's goal.

5.3. Attacker's Knowledge

The attacker may have different levels of knowledge of the targeted system (Barreno et al., 2010; 2006; Biggio et al., 2014a; 2014b; Biggio and Roli, 2018; Huang et al., 2011; Šrndic and Laskov, 2014a). In particular, she may know completely, partially, or do not have any information at all about: (*i*) the training data $\mathcal{D}$; (*ii*) the feature set $\mathcal{X}$, i.e., how input data is mapped onto a vector of feature values; (*iii*) the learning algorithm $\mathcal{L}(\mathcal{D}, f)$, and its decision function $f(\boldsymbol{x})$, including its (trained) parameters (e.g., feature weights and bias in linear classifiers), if any. In some applications, the attacker may also exploit feedback on the classifier's decisions to improve her knowledge of the system, and, more generally, her attack strategy (Barreno et al., 2006; Biggio et al., 2014a; 2014b; Biggio and Roli, 2018; Huang et al., 2011).

The attacker's knowledge can be represented in terms of a space Θ that encodes knowledge of the data $\mathcal{D}$, the feature space $\mathcal{X}$, the learning algorithm $\mathcal{L}(\mathcal{D}, f)$ and its decision function *f*. In particular, we distinguish between limited- and perfect-knowledge attacks.

5.3.1. Limited-Knowledge (LK) Black-Box Attacks

Under this scenario, the attacker is typically only assumed to know the feature representation $\mathcal{X}$ and the learning algorithm $\mathcal{L}$, but not the training data $\mathcal{D}$ and the trained classifier *f*. This assumption is common under the security-by-design paradigm: the goal is to show that the system may be reasonably secure even if the attacker knows how it works but does not know any detail on the specific deployed instance (Barreno et al., 2006; Biggio et al., 2013; 2014a; 2014b; Huang et al., 2011), Demontis et al., (Biggio and Roli, 2018).

In particular, according to the definition proposed by Biggio et al., we distinguish the cases in which either the training data or the trained classifier are unknown (Muñoz González et al., 2017). In the first case, to which we refer as LK attacks with *surrogate data*, it is often assumed that the attacker can collect a surrogate dataset $\hat{\mathcal{D}}$ and that she can learn a surrogate classifier $\hat{f}$ on $\hat{\mathcal{D}}$ to approximate the true f (Biggio et al., 2013; Papernot et al., 2017). Note also that the class labels of $\hat{\mathcal{D}}$ can be modified using the feedback provided from the targeted classifier f, when available (e.g., as an online service providing class labels to the input data). The knowledge-parameter vector can be thus encoded as $\boldsymbol{\theta}_{LK-SD} = (\hat{\mathcal{D}}, \mathcal{X}, \mathcal{L}, \hat{f})$.

In the second case, to which we refer to as LK attacks with *surrogate learners*, we assume that the attacker knows the training distribution $\mathcal{D}$, but not the learning model. Hence, she trains a surrogate function on the same training data. Hence, the knowledge-parameter vector can be encoded as $\boldsymbol{\theta}_{LK-SL} = (\mathcal{D}, \mathcal{X}, \hat{\mathcal{L}}, \hat{f})$.

5.3.2. Perfect-Knowledge (PK) White-Box Attacks

This is the worst-case setting in which also the targeted classifier is fully known to the attacker, i.e., $\boldsymbol{\theta} = (\mathcal{D}, \mathcal{X}, \mathcal{L}, f)$. Although it is not very likely to happen in practice that the attacker gets to know even the trained classifier's parameters, this white-box setting is particularly interesting as it provides an upper bound on the performance degradation incurred by the system under attack, and can be used as a reference to evaluate the effectiveness of the system against the other (less pessimistic) attack scenarios. In the experimental evaluation of this work, we will mostly explore this knowledge scenario, along with limited knowledge with surrogate learners.

5.4. Attacker's Capability

The attacker's capability of manipulating the input data is defined in terms of the so-called *attack influence* and it is based on some application-specific constraints.

Attack Influence. This defines whether the attacker can only manipulate data at test time (*exploratory* influence), or if she can also contaminate the training data (*causative* influence). Such contamination is possible, for instance, if the system is retrained online using data collected during operations that can be manipulated by the attacker (Barreno et al., 2006; Biggio et al., 2014b; Biggio and Roli, 2018; Huang et al., 2011).

Application-specific constraints. According to the given application, these constraints define how and to which extent the input data (and its features) can be modified to reach the attacker's goal. In many cases, these constraints can be directly encoded in terms of distances in the feature space, computed between the source malware data and its manipulated versions (Biggio et al., 2013; Biggio and Roli, 2018; Brückner et al., 2012; Dalvi et al., 2004; Globerson and Roweis, 2006; Lowd and Meek, 2005; Teo et al., 2008). FlashBuster is not an exception to this rule, as we will discuss in the remainder of this section. In general, the attacker's capability can thus be represented in terms of a set of possible modifications $\Omega(\mathcal{A})$ performed on the input samples $\mathcal{A}$.

5.5. Attack Strategy

The attack strategy amounts to formalizing the derivation of the attack in terms of an optimization problem (Biggio et al., 2013; 2014b). Given the attacker's goal $\mathcal{W}(\mathcal{A}', \boldsymbol{\theta})$, along with a knowledge-parameter vector $\boldsymbol{\theta} \in \Theta$ and a set of manipulated attacks $\mathcal{A}' \in \Omega(\mathcal{A})$, the attack strategy is given as:

$$\mathcal{A}^\star = \arg\max_{\mathcal{A}' \in \Omega(\mathcal{A})} \mathcal{W}(\mathcal{A}'; \boldsymbol{\theta}) \, . \tag{1}$$

Under this formulation, one can characterize different attack scenarios. The two main ones often considered in adversarial machine learning are referred to as classifier **evasion** and **poisoning** (Barreno et al., 2010; 2006; Biggio et al., 2013; 2014a; 2014b; 2012; Biggio and Roli, 2018; Muñoz González et al., 2017; Huang et al., 2011; Mei and Zhu, 2015). In the remainder of this work we focus on *classifier evasion*, while we refer the reader to Biggio et al. (2014b); Muñoz González et al. (2017); Mei and Zhu (2015) for further details on *classifier poisoning*.

6. Evasion Attacks and Security Scenarios

Evasion attacks consist of manipulating malicious samples at test time to have them misclassified as benign by a trained classifier. The attacker's goal is thus to violate system *integrity*, either with a *targeted* or with an *indiscriminate* attack, depending on whether the attacker is targeting a specific machine or running an indiscriminate attack campaign. More formally, evasion attacks can be written in terms of the following optimization problem:

$$\boldsymbol{z}^\star = \arg\min_{\boldsymbol{z}' \in \Omega(\boldsymbol{z})} \hat{\mathrm{f}}(\Phi(\boldsymbol{z}')) \, , \tag{2}$$

where $\boldsymbol{x}' = \Phi(\boldsymbol{z}')$ is the feature vector associated to the modified attack sample $\boldsymbol{z}'$, $\boldsymbol{x} = \Phi(\boldsymbol{z})$ is the feature vector associated to the source (unmodified) malware sample $\boldsymbol{z}$, Φ is the feature extraction function, and $\hat{f}$ is the surrogate classifier estimated by the attacker. Concerning Eq. (1), note that here samples can be optimized one at a time, as they can be independently modified.

As in previous work (Biggio et al., 2013; 2014b), Demontis et al., we first simulate the attack at the feature level, i.e., we directly manipulate the feature values of malicious samples without constructing the corresponding real-world samples while running the attack. We discuss in Sect. 6.2 how to create the corresponding real-world evasive malware samples. The above problem can be thus simplified as:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}'} \hat{\mathrm{f}}(\boldsymbol{x}') \tag{3}$$

$$\text{s.t.} \quad \|\boldsymbol{x}' - \boldsymbol{x}\|_1 \leq \varepsilon \, , \tag{4}$$

$$\boldsymbol{x}_{\mathrm{lb}} \preceq \boldsymbol{x}' \preceq \boldsymbol{x}_{\mathrm{ub}} \, , \tag{5}$$

where we have also made the manipulation constraints Ω used to attack FlashBuster explicit. In particular, the box constraint $\boldsymbol{x}_{\mathrm{lb}}{}'\boldsymbol{x}'\boldsymbol{x}_{\mathrm{ub}}$ (in which the inequality holds for each element of the vector) bounds the minimum and maximum feature values for the attack sample $\boldsymbol{x}'$. For FlashBuster, we will only consider *feature injection*, i.e., we will only allow the injection of structural and bytecode features within the SWF file to avoid compromising the intrusive functionality of the malware samples (something that can easily happen by deleting objects or specific calls). This can be simply accounted for by setting $\boldsymbol{x}_{\mathrm{lb}} = \boldsymbol{x}$. The additional ℓ_1 distance constraint $\|\boldsymbol{x}' - \boldsymbol{x}\|_1 \leq \varepsilon$ thus sets the maximum number ε of structural and bytecode features (i.e., tags and API calls) that can be injected into the file. The solution to the above optimization problem amounts to identifying which features should be modified to maximally decrease the value of the classification function, i.e., to maximize the probability of evading detection (Biggio et al., 2013; 2014b). This set of features varies depending on the input sample $\boldsymbol{x}$.

6.1. Evasion Attack Algorithm

If the objective function (i.e., the decision function of the classifier) f is not linear, as for kernelized SVMs and random forests, Problem (3)-(4) corresponds to a non-linear programming problem with linear constraints. The solution is, therefore, typically found at a local minimum of the objective function. Problem (3)-(4) can be solved with standard algorithms, but this is not typically very efficient, as such solvers do not exploit specific knowledge about the evasion problem. We thus devise an ad-hoc solver based on exploring a descent direction aligned with the gradient $\nabla\hat{f}(\boldsymbol{x}')$ using a bisect line search, similar to that used in our previous work (Russu et al., 2016). Its basic structure is given as Algorithm 1. To minimize the number of iterations, we explore one feature at a time (starting from the most promising feature, i.e., the one exhibiting the highest gradient variation in absolute value), leveraging the fact that the solution will be sparse (as the problem is ℓ_1 constrained). We also minimize the number of gradient and function evaluations to further speed up our evasion algorithm; e.g., we only re-compute the gradient of $\hat{f}(\boldsymbol{x})$ when no better point is found on the direction under exploration. Finally, we initialize $\boldsymbol{x}^{(0)}$ twice (first starting from $\boldsymbol{x}$, and then from a benign sample projected onto the feasible domain), to mitigate the problem of ending up in a local minimum that does not evade detection.[5]

Algorithm 1 Evasion Attack

```
Require: $\boldsymbol{x}$, the malicious sample; $\boldsymbol{x}^{(0)}$, the initial location of the
    attack sample;$\hat{f}$, the surrogate classifier (Eq. 3);$\varepsilon$, the maximum number
    of injected structural and bytecode features (Eq. 4);$\boldsymbol{x}_{\mathrm{lb}}$ and
    $\boldsymbol{x}_{\mathrm{ub}}$, the box constraint bounds (Eq. 4);$\epsilon$, a small positive constant.
Ensure: $\boldsymbol{x}'$, the evasion attack sample.
$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    $t' = \arg\min_t \hat{f}(\Pi(\boldsymbol{x}^{(i-1)} - t\nabla\hat{f}(\boldsymbol{x}^{(i-1)})))$
    $\boldsymbol{x}^{(i)} \leftarrow \Pi(\boldsymbol{x}^{(i-1)} - t'\nabla\hat{f}(\boldsymbol{x}^{(i-1)}))$
until $|\hat{f}(\boldsymbol{x}^{(i)}) - \hat{f}(\boldsymbol{x}^{(i-1)})| < \epsilon$
return $\boldsymbol{x}^{(i)}$
```

6.2. Constructing Adversarial Malware Examples

A common problem when performing adversarial attacks against machine learning is evaluating whether they can be truly performed *in practice*. As gradient-descent attacks are performed at the feature level, the attacker is then supposed to solve the so-called *inverse feature-mapping problem*, i.e., to reconstruct from the obtained features the sample that can be concretely used against the target classifier (Biggio et al., 2014b; Huang et al., 2011), Demontis et al..

Such an operation is not smooth to perform in many cases, not only from a more theoretical standpoint (as discussed in Huang et al. (2011)) but also from a practical perspective. In the specific case of Flash malware (as well as malware in general), generating the corresponding real-world adversarial examples may be complicated, as a single wrong operation can compromise the intrusive functionality of the embedded exploitation code Demontis et al.. For example, removing one structural feature such as one frame or script might entirely break the SWF file. For this reason, in this paper, we only considered *injection* of additional content into the SWF file. In particular, we propose a methodology to automatically construct the real evasive samples by automatically injecting features that do not alter the overall functionality of the SWF files. This methodology applies to content-based features, which represent the majority of features employed by FlashBuster, and it works as follows:

1. We disassemble the target SWF file by extracting its ActionScript bytecode (by using, e.g., tools such as `RABCDasm`).
2. We explore the disassembled code until we find `return`-type instructions in functions (e.g., `returnvoid`).
3. We inject a set of instructions that are never parsed by the code (as it comes after the return instruction). In particular, we inject a combination of two instructions: `pushstring`, combined with the name of the API call (e.g., `pushstring ''flash.utils.ByteArray''`); `pop`, which essentially removes the string pushed into the stack. This combination does not alter the memory of the virtual machine.
4. The disassembled code is reassembled back to the original file.

With this technique, it is possible to increase the content-based feature values of FlashBuster without changing the behavior of the file. We tested this methodology on various samples by using the sandbox-based analyzer `Any.run` (Any.Run, 2020)[6]. In particular, we run the samples before and after the reconstruction, by comparing the extracted reports. In the tested cases, we found no differences in behavior between the original and the modified files.

It is worth noting that unreachable code injected after `return` instructions could be discarded by making FlashBuster analyze the applications flow-graph. We plan to implement control flow-based parsing in future work. However, it would still be possible for an attacker to inject unreachable code while defying control flow analysis. For example, attackers may employ *opaque predicates*, which are conditional branches that are typically extremely hard to evaluate statically (Collberg et al., 1998). In this case, unreachable code can be added as an apparently legitimate branch that is never taken, and advanced static (or dynamic) analysis would be required to detect and discard it.

A similar strategy can be employed to modify structural-based features. By using libraries such as `PySWF` (Timknip, 2017), it is possible to extract the structures of tags and add new ones. However, changing the number of tags alters the whole SWF structure, which must be then reconstructed with the correct offsets. This operation is not easy to carry out in an automatic fashion. However, it is possible to use `JPEXS` to edit and reconstruct the SWF structure manually. We plan, in future work, to extend the automatic reconstruction of samples also to structural.

6.3. Summary of the Evasion Methodology

As a further clarification of what we described in Sections 5 and 6, we provide a brief, practical summary of the attack scenario and the evasion methodology that will be employed in the following of this paper.

- The goal of the attacker is to modify Flash applications classified as malicious by the target system to make them misclassified as benign.
- The attacker possesses full knowledge of the target system. She knows which features were used to train the model, the classifier, and the trained model itself (Perfect Knowledge Scenario). The only exception is represented by attacks against non-differentiable Random Forest classifiers, where the attacker

[5] This problem has been first pointed out in Biggio et al. (2013), where the authors have introduced a *mimicry* term to overcome it. Here we consider a different initialization mechanism, which allows us to get rid of the complicated mimicry term in the objective function.

[6] It is not possible to use `Any.run` to perform analysis of groups of samples. Each sample should be analyzed singularly through the graphical interface.

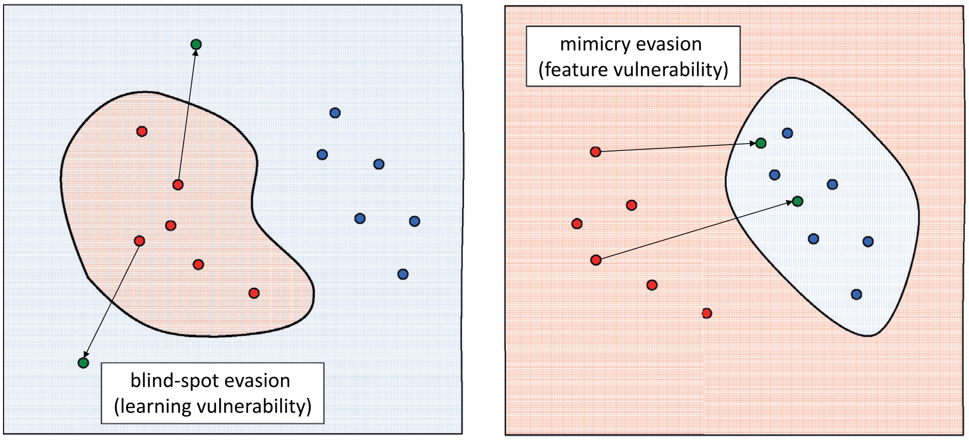


Fig. 2. Conceptual representation of *learning* (left) and *feature* (right) vulnerability. Red, blue and green samples represent, respectively, malicious, benign and attack samples.

employs a differentiable surrogate model to perform the attack (Limited Knowledge with surrogate models).

- The attacker performs evasion against the trained model by employing a gradient descent algorithm. The role of gradient descent is selecting which features should be changed to maximize the probability of evasion with the minimum number of changes to the application. The attack is performed on the feature level and outputs a modified feature vector.
- Once the attacker manages to attain evasion for a specific sample, she can concretely inject the features to the sample in order to obtain the corresponding evasive feature vector.

7. Feature and Learning Vulnerability

We discuss here an interesting aspect related to the vulnerability of learning-based systems, first highlighted in Biggio et al. (2015); Russu et al. (2016), and conceptually represented in Fig. 2, which shows two classifiers on a two-feature space. The classifiers can be defined as surfaces closed around malicious (left) and benign (right) samples. The red, blue, and green samples represent, respectively, malicious, benign, and attack samples. An evasion attack sample is typically misclassified in two scenarios: (*i*) the feature vector related to the sample is *far enough* from those belonging to the rest of known training samples (both malicious and benign), or (*ii*) the feature vector is *indistinguishable* from those exhibited by benign data.

In the first scenario, usually referred to as *blind-spot evasion*, retraining the classifier on the adversarial examples (with *adversarial training*) should successfully enable their detection, improving classifier security. This means that the classification error induced by such attacks could be *reduced* in advance, by designing a learning algorithm capable of anticipating this threat; e.g., building a classifier that better encloses benign data and classifies as malicious the regions of the feature space where training data is absent or scarce (see, e.g., the classifier in the right plot of Fig. 2). We refer to this vulnerability as one induced by the *learning algorithm* (left plot in Fig. 2).

In the second scenario, instead, retraining the classifier would be useless, as the whole distribution of the evasion samples is overlapped with that of benign data in feature space, i.e., the attack increases the Bayesian (non-reducible) error. We thus refer to this attack as *mimicry evasion*, and the corresponding vulnerability as one induced by the *feature representation* (right plot in Fig. 2). In fact, if a malware sample can be modified to exhibit the same feature values of benign data, it means that the given features are intrinsically weak, and no secure learning algorithm can prevent this issue.

This notion can also be motivated in formal terms, similarly to the risk analysis reported in Biggio et al. (2015). From a Bayesian perspective, learning algorithms assume an underlying (though unknown) distribution $p(\boldsymbol{x}, y)$ governing the generation of benign ($y=-1$) and malicious ($y=+1$) data, and aim to minimize the classification error $E(f) = \mathbb{E}_{(\boldsymbol{x},y)\sim p}\ell(y, f(\boldsymbol{x}))$, where $\mathbb{E}$ is the expectation operator, ℓ is the zero-one loss, and f is the classification function returning the predicted class label (i.e., ± 1). Let us denote the optimal classifier achieving the minimum (Bayesian) error on p with $f^\star$. It is clear that, if there is no *evidence* $p(\boldsymbol{x})$ of (training) data in some regions of the feature space (usually referred to as *blind spots*), such regions can be arbitrarily classified by $f^\star$ as either benign or malicious with no impact on the classification error (the expectation on p will be in any case zero in those regions). This is precisely the underlying reason behind the vulnerability of learning algorithms to blind-spot evasion.

Within this setting, evasion attacks can be conceived as a manipulation of the input samples $\boldsymbol{x}$ through a function $a(\boldsymbol{x})$, which essentially introduces a deviation from the source distribution $p(\boldsymbol{x}, y)$. By denoting with $E_a(f) = \mathbb{E}_{(\boldsymbol{x},y)\sim p}\ell(y, f(a(\boldsymbol{x})))$ the error of the classifier f on the manipulated samples, with f' the optimal (Bayesian) classifier on such manipulated data, we can compute the increase in the classification error of $f^\star$ on the manipulated data as the following:

$$E_a(f^\star) - E(f^\star) = \underbrace{E_a(f') - E(f^\star)}_{\text{feature vulnerability}} + \underbrace{E_a(f^\star) - E_a(f')}_{\text{learning vulnerability}}. \tag{6}$$

The first term is the increase in Bayesian error before and after the attack (which characterizes the vulnerability of the feature representation), while the second represents the classification error reducible by retraining on the attack samples (i.e., the vulnerability of the learning algorithm).

Under this interpretation, we can introduce a metric to assess the feature vulnerability quantitatively. To this end, we consider the so-called Bhattacharyya Coefficient (BC):

$$\mathrm{BC} = \int_{\boldsymbol{x}\in\mathcal{X}} \sqrt{p_b(\boldsymbol{x})p_m(\boldsymbol{x})}d\boldsymbol{x} \in \{0, 1\}. \tag{7}$$

This coefficient essentially evaluates the overlapping between the distributions of benign p_b and manipulated attack p_m samples over the whole feature space $\mathcal{X}$. If the two distributions are the same, $\mathrm{BC}=1$, while if they are perfectly separated, $\mathrm{BC}=0$. The convenient aspect of this metric is that it has a closed form for several known distributions; e.g., in the case of multivariate Gaussian distributions, it is given as $\mathrm{BC} = \exp(-D_B)$, where

$$D_B = \frac{1}{8}(\boldsymbol{\mu}_b - \boldsymbol{\mu}_m)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_b - \boldsymbol{\mu}_m) + \frac{1}{2}\log\frac{\det\boldsymbol{\Sigma}}{\sqrt{\det\boldsymbol{\Sigma}_b \det\boldsymbol{\Sigma}_m}}, \qquad (8)$$

and where $\Sigma = 0.5(\Sigma_b + \Sigma_m)$, while μ_b, μ_m, Σ_b and Σ_m are the means and covariance matrices of benign and attack data, respectively. To assess feature vulnerability, we use this expression for BC, and exploit the well-known result that the Bayesian error is upper bounded by $\frac{1}{2}$BC. One may indeed measure the difference between such value computed after and before the attack, which gives us an (approximate) indication of the increase in the Bayesian error induced by the attack, and thus, a quantitative measure of the feature vulnerability (i.e., of the first term in Eq. 6). Therefore, this coefficient can represent a useful measure that can be employed in practice to recognize the presence of feature vulnerabilities. We will use BC to discuss the efficacy of adversarial retraining in Section 8.4.

8. Experimental Evaluation

The experimental evaluation proposed in this paper is divided into two parts, which we describe in the following.

Standard Accuracy Evaluation. In this evaluation, we tested the performances of FlashBuster against benign and malicious files in the wild by comparing them with the results attained by Šrndic (2016); Šrndić and Laskov (2016). FlashBuster and Hidost were trained with a dataset of *randomly chosen* malicious and benign SWF files, and they were tested against many previously unseen malicious and benign files. This experiment provided information on the *general* performances attained by FlashBuster and Hidost regarding true and false positives. The goal of this evaluation was to ensure that the feature set we introduced obtained performances comparable to those attained by other publicly available state-of-the-art tools. Additionally, we tested the capability of FlashBuster to predict previously unseen attacks. To this end, we trained the system with data obtained before 2017 and tested it against data obtained after the same year. Finally, to ensure that FlashBuster could be reliably used also to detect new threats, we also performed an additional evaluation in which our system was tested against obfuscated and encrypted samples.

Adversarial Evaluation. In this experiment (directly linked to the previous one), we evaluated the performances of FlashBuster against adversarial attacks performed according to a gradient descent strategy (see Section 6). It is the first time that such evaluation has been carried out on Flash files and against Flash-based detection systems. Our goal was to understand the robustness of the features extracted by FlashBuster against adversarial modifications by employing possible defenses such as classifier retraining. Moreover, we provide a discussion about the efficacy of adversarial retraining as a possible defense to counteract adversarial attacks.

In the following, we describe the dataset and the basic setup employed for all the experiments.

8.1. Dataset

The dataset used for our experiments is composed of 5828 files, 1593 of which are malicious (an amount comparable to previous works (Overveldt et al., 2012; Šrndić and Laskov, 2016; Wressnegger et al., 2016)) and 4235 are benign (we chose this amount to work on balanced datasets (Rossow et al., 2012)). All malicious files, as well as part the benign ones, were retrieved from the VirusTotal service (Google, 2018). Other benign files were retrieved from the DigitalCorpora repository (Corpora, 2013). Notably, for objective analysis, we only employed samples featuring code belonging to ASVM2 (as the previous version of the Virtual Machine employs different instructions and routines). Table 2 reports the distribution of the malicious samples by the date of the first submission to the service VirusTotal, together with the number of *unique* CVEs that were correctly identified by the service[7].

Table 1
Notation and symbols employed in the paper.

Symbol	Description
$\mathcal{W}$	Objective function
$\mathcal{A}$ and $\mathcal{A}'$	Input samples and modified attack samples
$\boldsymbol{\theta}$	System knowledge
$\mathcal{D}$ and $\hat{\mathcal{D}}$	Training data and surrogate training
$\mathcal{X}$	Feature set
$\mathcal{L}$	Learning algorithm
f, $\hat{f}$	Decision function, surrogate classifier
$f_\star$	Classifier with minimum Bayesian error
Θ	Space that embeds $\mathcal{D}$, $\mathcal{X}$, $\mathcal{L}$, f
Ω	Set of possible modifications
Φ	Feature extraction function
$\boldsymbol{x}$ and $\boldsymbol{x}'$	Original and modified feature vectors
$\boldsymbol{z}$, $\boldsymbol{z}'$ and $\boldsymbol{z}^\star$	Original, modified, and optimal input samples
$\boldsymbol{x}_{\mathbf{lb}}$ **and** $\boldsymbol{x}_{\mathbf{ub}}$	box-constraint bounds
$\boldsymbol{\varepsilon}$	Number of feature modifications
$\boldsymbol{y}$	Labels
$\boldsymbol{E}$	Classification error
ℓ	Zero-one loss
$\boldsymbol{p}$	Data distribution
$\boldsymbol{a}$	Deviation from data distribution
$\boldsymbol{E}$	Classification error
$\boldsymbol{BC}$	Bhattacharyya Coefficient
$\boldsymbol{\mu}$, $\boldsymbol{\Sigma_b}$	Mean and covariance of data

Table 2
Distribution of malicious samples by the date of first submission to the VirusTotal service. The number of unique identified CVEs are reported for each year.

Year	Num.Samples	Num. CVE
2018	233	3
2017	149	2
2016	495	2
2015	453	18
2014	127	9
2013	45	2
2012	37	4
2011	18	6
2010	25	2
2009	9	0
2009	2	0

The number of samples is well-balanced between 2015 and 2018, which were the years when Flash malware was employed the most by attackers. In particular, we report almost 30 unique CVEs in 2015 and 2016. Overall, the employed dataset features a rather large diversity of vulnerabilities. This aspect is further confirmed by Table 3, which shows the top-10 malware families contained in the dataset. We extracted the malware families by employing the popular tool `AvClass` (Sebastián et al., 2016), applied to the VirusTotal data. Note that most malware families are related to *exploit-kits* (e.g., `neutrino`) that contain a specific Flash vulnerability. This aspect shows that the analyzed samples belong to attacks that are consistently employed in the wild. Hence, we claim to have employed a dataset that is representative of the Flash-based malware ecosystem in the wild.

[7] There may be further CVEs in the dataset that were not identified by VirusTotal. One CVE may apply to multiple files.

Table 3
Distribution of the top-10 malware families retrieved from the dataset of Flash-based malware.

Family	Num. Samples
neutrino	114
axpergle	89
exkit	64
angler	45
swfexp	40
swif	38
lodabytor	35
swfdec	21
gwan	21
pubenush	15

8.2. Basic Setup

In this section, we describe the basic setup of the pre-processing, feature extractor, and classification modules for FlashBuster and Hidost. This setup is common to all the evaluations described in this section.

8.2.1. Pre-Processing

Pre-processing was performed by FlashBuster and Hidost as follows:

- **FlashBuster.** As mentioned in Section 4.1, the original JPEXS parser was modified to allow a faster analysis of multiple SWF files, as well as better integration with the other components of FlashBuster. All data related to tags and bytecodes were extracted and dumped into files, to allow for subsequent analyses by the other FlashBuster modules. The extraction time may vary from milliseconds to some minutes for very large files.
- **Hidost.** Hidost employed SWFReTools (Sporst, 2011), a Java-based parser to analyze the structure of SWF files. After pre-processing, the analyzed files were added to a special cache to reduce the extraction times, which may vary from milliseconds to minutes.

8.2.2. Feature Extraction.

We now provide some details about the feature extraction mechanisms employed by FlashBuster. It is critical to point out that our experimental goal was *not* developing the most accurate system possible, but performing an effective security evaluation under reasonable, practical constraints. In particular, there can be high differences in values among the employed features. Thus, attackers can create samples with abnormal values of certain features that would make the adversarial sample utterly different from the typical malware distribution (e.g., using specific functions thousands of times in the same method, while others are only used few times), thus achieving easy evasion. However, this evasion attempt is trivial, as anomalous values would make the sample easy to detect by using anomaly detectors.

To create a realistic situation for the attacker, we established an upper limit on each feature value in our dataset. For our experiments, we chose 10 as a reasonable value that limits how often the same feature can be injected. We established this threshold empirically by performing statistics on how many features of the same type were typically contained in our dataset. Our findings showed that, while some samples may call the same API hundreds of times (especially in benign applications), the majority of the analyzed features were called (on average) less than 10 times. Hence, the choice of this threshold represents the fact that it is unlikely that the majority of the features would appear more than 10 times in a sample. To confirm that limiting the feature values does not influence classification performances, we repeated our experiments with higher upper limits, without noticing significant differences in performances.

8.2.3. Classification and Training Procedures

We used the popular machine-learning suite scikit-learn (Pedregosa et al., 2011), which features the classifiers used in our evaluation,[8] as well as secml to simulate evasion attacks against them (Melis et al., 2019). All the performed evaluations (except the temporal evaluation and the one against obfuscated malware, which were carried out on an entirely different dataset) share the following elements: (a) The dataset was randomly split by considering 70% of it as a training set and the remaining 30% as a test set. The classifier hyperparameters were evaluated with a 5-fold cross-validation performed on the training set to optimize the true positive rate at 1% false-positive rate. We repeated the whole procedure five times, in order to rule out possible biases related to specific train/test divisions. (b) We performed our tests on four classifiers: *(i)* Random Forest (RF); *(ii)* support vector machine with the linear kernel (SVM); *(iii)* support vector machine with the Radial-Basis-Function kernel (SVM-RBF); *(iv)* support vector machine with the Laplacian kernel (SVM-Lap). For support vector machines, we optimized the regularization hyperparameter C and the kernel hyperparameter γ (for kernel-based support vector machines) via cross-validation with C, $\gamma \in \{0.01, 0.5, 0.2, 0.1, 1, 2, 10, 20, 50, 100\}$. For random forests, we optimized the minimum number of samples required to split an internal node in {2, 5}, and either did not set any maximum depth or set it to 30 or 100.

Adversarial Training (AT). Under the same experimental setup, we retrained the same classifiers by adding the attacks generated against them to their training set. This procedure amounts to solving a minimax game known as *adversarial training* (Goodfellow et al., 2015; Szegedy et al., 2014) in which the attacker maximizes the training loss of the classifier by manipulating the training points, while the classifier minimizes the same loss (computed on the manipulated samples) by adjusting its training parameters. We implemented an iterative version of this game in which, at each iteration, we first selected at random 500 malware samples from the training set and optimized them to evade the target classifier. These samples were optimized by randomly-picking $\varepsilon \in \{1, 2, 5, 10, 20, 50, 100\}$. Then, we added such adversarial samples to the training set and retrained the target classifier on the augmented training data. We terminated this procedure after 10 iterations, or earlier if convergence to a stable solution was observed. We denote the corresponding robust classifiers with the suffix AT, i.e., SVM-AT, SVM-RBF-AT, SVM-Lap-AT, and RF-AT.

8.3. Standard Accuracy Evaluation

The standard accuracy evaluation was performed by following the criteria described in Section 8.2.3. For both FlashBuster and Hidost, and for each of the four classifiers tested, we calculated the average (among its splits) Receiving Operating Characteristic (ROC), which reports the detection rate (i.e., the fraction of correctly-detected malware) at different false-positive rates (FPR, i.e., the fraction of misclassified benign samples).

Results are shown in Fig. 3. The leftmost plot shows that FlashBuster detects more than 90% malware samples at 1% FPR, while at 5% the detection rate is higher than 95%. Non-linear models such as Random Forests and SVMs with the Laplacian kernel perform better than their linear counterparts. The middle plot reports the results obtained by retraining the classifiers with adversarial at-

[8] Notably, Hidost was not released with a pre-trained classifier but only with the feature extraction module.

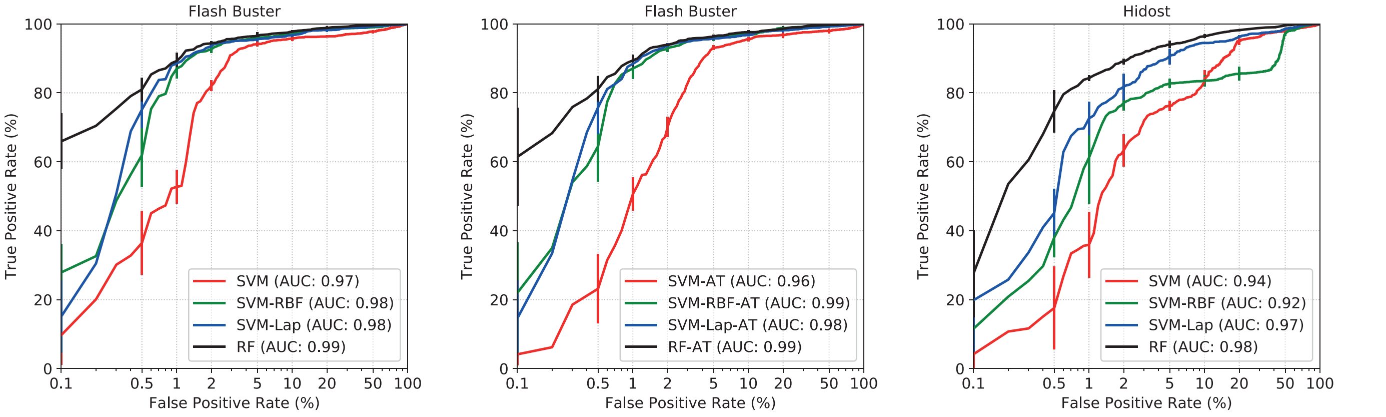


Fig. 3. Average ROC curves and related AUCs obtained on 5 train-test splits on FlashBuster (left - no adversarial retraining; center - with adversarial retraining) and Hidost (right).

Table 4
Accuracy performances on five classifiers on a test set composed of data released from 2017. Each classifier has been trained with data released before 2017. We also report the performances of retrained classifiers.

Classifier	Acc.	Acc. (Retrain)
RF	88%	89%
SVM Laplacian	88%	89%
SVM Linear	85%	89%
SVM RBF	75%	87%

tacks (described more in detail in Section 8.4). The attained results show that, despite adding artificially-generated samples to the training set, the detection rate of the classifiers does not substantially change. We discuss these aspects more in detail in the next section.

The rightmost plot shows the results attained by Hidost. FlashBuster performs better than Hidost with every classifier, with a difference in detection rates between 10% and 20%, depending on the selected classifier. We interpret these results with the fact that Hidost employs only structural features without analyzing the bytecode embedded in Flash applications. Some SWF files may indeed employ very similar structures, but extremely different codes. Hence, such files could be difficult to detect with purely structural approaches. This aspect has also been encountered in PDF files, where purely structural-based methods may fail with malicious files with many employed objects (Maiorca et al., 2019).

Overall, the attained curves show that FlashBuster can be effectively used as a detector of malicious files, outperforming competing state-of-the-art approaches.

8.3.1. Temporal Evaluation

We evaluated the ability of FlashBuster to predict previously-unseen attacks. To this end, we trained the system by only using samples whose first submission date to the VirusTotal service was previous to 2017 (a total of 1211 malicious samples), plus all the benign files in the dataset. The test set was therefore made of those malicious samples released in 2017 and 2018 (383 samples). We used all the classifiers of the previous experiments (along with retrained classifiers), and the parameters of the classifiers were evaluated with a 5-fold cross-validation performed on the training set.

Table 4 shows the results for this evaluation. It is possible to see that RF and SVM-Lap report very high accuracy values (almost 90%) on the test set. This result is also in-line to what showed in Fig. 3. It is interesting to point out that linear classifiers, which are the ones to perform worst against malware in the wild, provide good accuracy (85%) on unseen attacks. SVM-RBF classifiers are the ones to perform worst with a 75% of accuracy. Notably, the linear SVM significantly improves its performances after having been retrained with adversarial samples. We speculate that this behavior is due to similarities between the adversarial generated samples and the novel samples belonging to the test set. Overall, all models prove to be reliable at detecting previously-unseen malware, showing that FlashBuster can be effectively used for this task.

8.3.2. Evaluation against Obfuscated Malware

To test the detection capabilities of FlashBuster against obfuscated samples, we generated two additional datasets of malicious samples by using the popular obfuscation tool DoSWF (DoSWF, 2013), starting from the original malicious dataset. Notably, as DoSWF could not obfuscate all the samples (due to technical limitations of the tool), the two generated datasets featured fewer samples than the starting one. The datasets were organized as follows:

- **Obfuscated.** This dataset was generated by replacing variable names and by performing changes to the control flow graph without modifying the file semantics. After obfuscation, we obtained 1278 obfuscated samples.
- **Encrypted.** This dataset was generated by making each file be dynamically generated at runtime. Typically, encryption is employed to dynamically load files in memory through customized routines introduced by the obfuscator. After encryption, we obtained 1466 samples.

We used a training set composed of all malicious and benign files employed during the experiments reported in Section 8.3. We then tested the various classifiers of FlashBuster (including the ones featuring retrained data) on the two datasets. Results are reported in Tables 5 and 6, and show that FlashBuster is able to detect almost all obfuscated and encrypted samples in the wild.

This result may seem rather surprising because obfuscated malware should be significantly different from the original one. However, there are two explanations to this effect: *(i)* The features employed by FlashBuster are related to system-API calls, and the impact of obfuscation on these features is rather limited. In some cases, encryption may add other system API-based routines, which may increase the maliciousness of the file for the classifier; *(ii)* DoSWF has been often used as a way to conceal malware in the wild. Hence, we speculate that obfuscated and encrypted malware

Table 5
Accuracy performances (in percentage) on five classifiers on a test set composed of obfuscated data. We also report the performances of retrained classifiers.

Classifier	Acc.	Acc. (Retrain)
RF	97%	98%
SVM RBF	95%	96%
SVM Laplacian	94%	94%
SVM Linear	93%	92%

Table 6
Accuracy performances (in percentage) on five classifiers on a test set composed of encrypted data. We also report the performances of retrained classifiers.

Classifier	Acc.	Acc. (Retrain)
RF	100%	100%
SVM RBF	100%	100%
SVM Laplacian	94%	94%
SVM Linear	100%	100%

was already in the employed training set. As a consequence, the classifier learned the characteristics of obfuscated malware and recognized obfuscated attacks due to their characteristics (an aspect also observed in a very recent work (Mantovani et al., 2020)).

8.4. Adversarial Evaluation

The adversarial evaluation aimed to assess the performance of the classifiers employed in the previous experiment after the gradient descent attacks described in Section 6. In this case, we evaluated how the detection rate (at 1% FPR) of the classifiers decreases against an increasing number of injected features ε. According to this security evaluation procedure, a classifier is said to be more robust if its detection rate decreases more gracefully (Biggio and Roli, 2018).

It is important to observe that RF classifiers are not differentiable, so we can not use our gradient-based attack directly to evaluate their security. We thus attacked our RF classifiers by first optimizing the attacks against all the SVM-based classifiers (used as surrogate models), and then evaluating whether such attacks *transfer* correctly to the RF classifiers (Limited Knowledge - surrogate models). In practice, we found that the attacks exhibiting the highest success against the RF classifiers were those optimized against the SVM-Lap models.

The leftmost plot in Fig. 4 provides the results of the evaluation when the classifiers are not retrained with adversarial attacks. While all SVM-based classifiers are completely evaded after $\varepsilon = 100$ changes, Random Forests can still detect 20% of the attacks. This effect may be present because we are using a surrogate model to attack the RF classifiers rather than directly optimizing the attack against them. For this reason, we can not state with certainty that RFs are *generally* more secure; indeed, a more powerful white-box attack as that in Kantchelian et al. (2016) may enable evading them with higher probability. We thus leave a more detailed investigation of this aspect to future work.

The rightmost plot in Fig. 4 shows the security evaluation curves obtained by retraining the classifiers with samples generated through adversarial attacks. While the retraining strategy essentially brings no additional robustness for SVM and SVM-RBF classifiers, we point out a significant increment of robustness in SVM-Lap and RF classifiers. This effect may be due to the decision function learned by these models, which may be better shaped to counter the presence of ℓ_1-norm adversarial attacks (as those simulated in this work, where the attack algorithm manipulates only a few relevant features due to the presence of the sparse constraint $\|\boldsymbol{x}' - \boldsymbol{x}\|_1 \leq \varepsilon$) (Russu et al., 2016).

Discussion on Adversarial Training. We investigate here why adversarial training is only slightly improving robustness to adversarial attacks in this case. To this end, in Fig. 5 we report the average feature values (i.e., the centroid) for benign and malware data (normalized in [0,1] after division by 10), extrapolated from one test set, and the average feature values for adversarial malware samples optimized against SVM-Lap and SVM-Lap-AT. For the same data, we also report in Fig. 6 the variation of the related Bhattacharyya coefficient *BC* (described in Section 7) computed between the malicious and benign distributions after projection on a two-dimensional space via Linear Discriminant Analysis (LDA).

From Fig. 5, it may appear that the centroids of malware and adversarial malware are not that different. This phenomenon is correct since we are only considering the injection of $\varepsilon = 50$ features in each sample. However, the two-dimensional projection elaborated in Fig. 6 unveils an interesting phenomenon. First of all, attacking SVM-Lap and SVM-Lap-AT decreased the initial BC value. This effect is surprising since it means that the attacks are effective even though the Bayesian error (distribution overlap) might decrease. In other words, both attacks evade detection by placing the adversarial malware samples in blind spots, without mimicking the characteristics of the benign distribution. Nevertheless, adversarial malware optimized against the non-robust SVM-Lap classifier is much more separable than the adversarial malware optimized against the robust SVM-Lap-AT classifier from the benign data (cf. the BC values in the middle and rightmost plots). This means that adversarial training, even after several rounds of retraining and addition of up to 5,000 adversarial malware samples in the training set, is capable of enforcing the attack samples to better reproduce the behavior of legitimate samples to be successful. However, even though the classifier has eventually learned a more robust decision surface in our case, this is not sufficient to stop practical attacks in this specific domain. In fact, by injecting more features, the attacker may be able to even better mimic the benign distribution and evade detection with higher probability. In addition, adversarial training is a very costly procedure in terms of computational complexity, both in space and time, especially in high-dimensional spaces like in our case (FlashBuster generates more than 4,000 distinct features), as it requires generating a huge number of samples (potentially exponential in the number of features) before being slightly effective. Accordingly, robust methods based on regularization rather than data augmentation may be better suited to our case Demontis et al., (Lyu et al., 2015; Ross and Doshi-Velez, 2018; Simon-Gabriel et al., 2018).

To summarize, the attained results show that, although the function was retrained against the attacks to reduce the *learning* vulnerability, the *feature* vulnerability related to the employed static features could not be reduced by merely retraining the classifier. In more detail, the problem here is that *none of the employed static features is likely to appear in malware much more frequently than in benign data*, i.e., there is no *invariant* feature that characterizes malware uniquely from benign data (and that can not be removed) (Tong et al., 2017). Such lack of invariance means that, in principle, it is possible to create a malicious sample that is *indistinguishable* from benign ones (even by only injecting content to the input SWF file) and, thus, additional features are required to detect adversarial SWF malware examples correctly.

9. Summary of Results and Discussion

In this Section, we provide a summary of the results attained during the experimental Section, along with a brief discussion on each point.

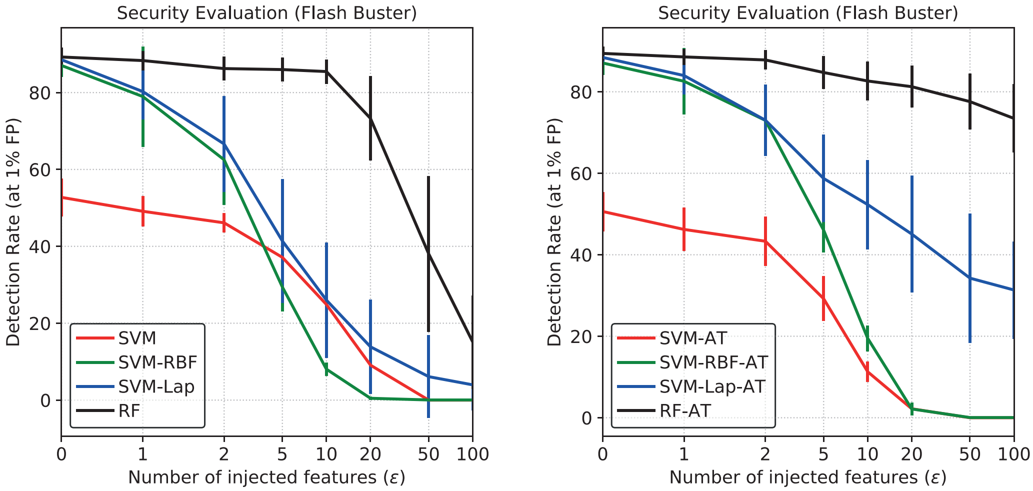


Fig. 4. Security evaluation curves on FlashBuster. On the x-axis, we report the number of changes to the feature values is reported. On the y-axis, we report the decrease of the detection rate as more features are modified. The left-side curves are related to the classifiers that were not retrained with adversarial attacks, while the right-side ones concern the classifiers under adversarial retraining.

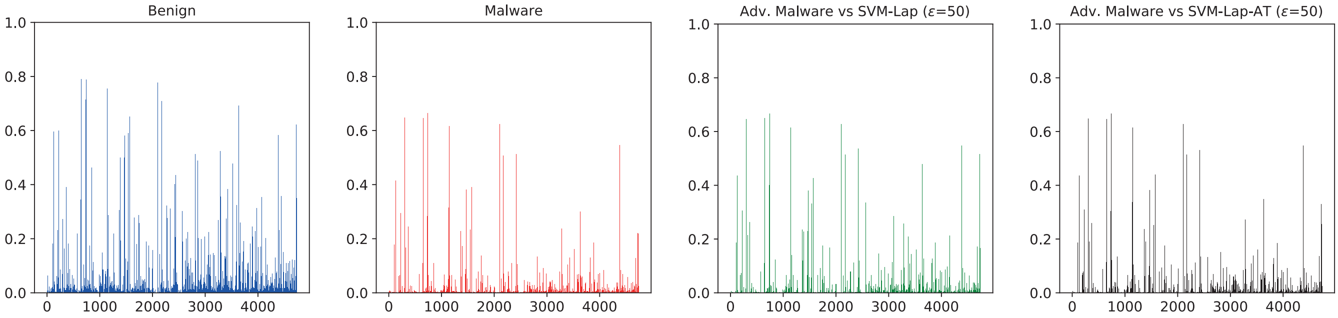


Fig. 5. Average feature values for one test set in which malicious samples (red) were modified with our gradient-based attack (ε is the number of changes) against SVM-Lap (green) and SVM-Lap-AT (black). The Bhattacharyya coefficient *BC* values between these malicious (red, green and black) and benign (blue) samples are reported in Fig. 6.

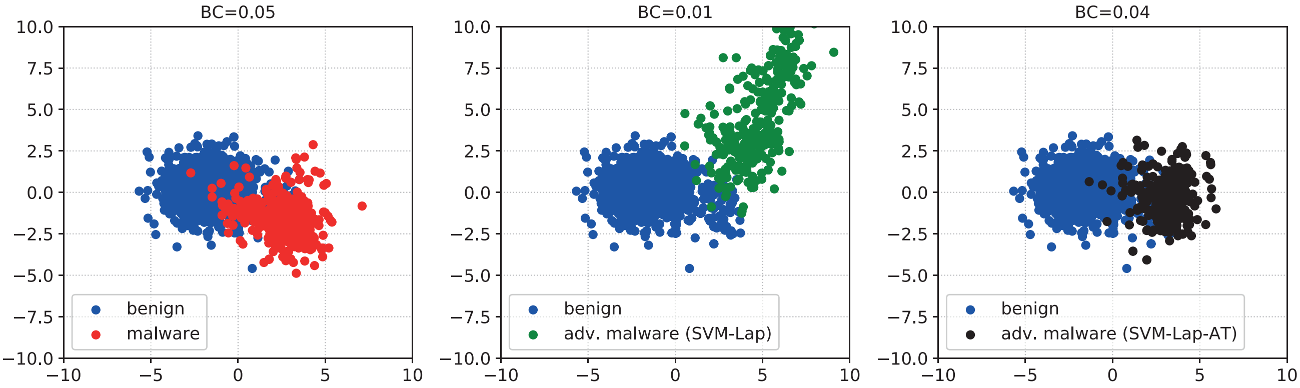


Fig. 6. Two-dimensional LDA projections of benign samples against malware (leftmost plot), adversarial malware targeting SVM-Lap (middle plot) and adversarial malware targeting SVM-Lap-AT (rightmost plot), along with the corresponding BC values.

- **Detection of malware in the wild**. FlashBuster can effectively detect malware in the wild. In particular, it features significantly higher performances than Hidost, a valid state-of-the-art tool for SWF malware detection. Using structural- and content-based features provides a more reliable detection against malware with structures similar to benign files. Overall, we demonstrated that our system could be validly used as a reference for further experiments related to robustness.
- **Detection of previously unseen and encrypted malware**. FlashBuster provides very good accuracy at detecting samples released after training data. Moreover, FlashBuster can detect obfuscated and encrypted (by a popular off-the-shelf tool) malware. In this last case, the distribution of training data is critical to ensure that the malware could be effectively detected. In particular, if the distribution of encrypted malware significantly differs from the one of normal malicious or benign files, it is likely that the system would not be able to detect it.

Conversely, if samples obfuscated with similar techniques are present in the training set, the system's performances can be extremely high.
- **Evasion attacks against classifiers**. When no retraining is employed, gradient-based attacks can completely evade the detection of all classifiers after a reasonable number of changes.
- **The role of adversarial training**. Adversarial training can slightly improve robustness in some cases, but not to a satisfying extent for this application. The main issue here is that defending the classifier remains useless if the adopted feature representation is vulnerable (i.e., the attacks are able to mimic the benign feature values). Moreover, adversarial training remains very computationally demanding, hindering both its effectiveness and scalability in high-dimensional domains such as that considered in this work.

10. Conclusions and Future Work

In this paper, we proposed a security evaluation of static malicious SWF file detectors by introducing FlashBuster, a system that combines structural and content-based information to perform an accurate analysis of such attacks. In particular, we demonstrated that FlashBuster could attain improved performances (in terms of detection rate) compared to other state-of-the-art tools. Moreover, we showed that it could be effectively employed to detect previously unseen, obfuscated, and encrypted attacks. The proposed security evaluation showed an intrinsic vulnerability of the static features used by SWF detectors. In particular, by using gradient descent attacks, we demonstrated how even retraining strategies were not always effective at ensuring robustness. More specifically, we measured and showed how gradient descent attacks made samples more similar to their benign counterparts, thus making adversarial retraining inefficient in some cases. We plan to improve and solve some of the system limitations in future work: for example, reducing its dependence on JPEXS, whose possible failures could compromise the whole file analysis. We also plan to perform more experiments on SWF files that are obfuscated with off-the-shelf tools in order to evaluate the resilience of FlashBuster against them.

In general, our claim for future research is that focusing on improving the classifier decision function can be effective only if the employed features are intrinsically robust, i.e., there should be specific features that are *truly characteristic* of malicious behavior and that cannot be mimicked in benign files (or whose mimicking would require a significant effort from the attacker's side). For example, the maliciousness of a feature could be better pointed out by considering the context in which it was found. There may be API calls that could be solely used for malicious purposes if they operate after (or before) other API calls, with specific parameters, or when the program is in a determined state of execution. We believe that this research direction will be useful not only for Flash malware detection but also for the other malware detection systems.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Davide Maiorca: Conceptualization, Methodology, Investigation, Software, Writing - original draft. **Ambra Demontis:** Software, Methodology, Investigation, Validation, Data curation, Visualization. **Battista Biggio:** Conceptualization, Methodology, Investigation, Supervision, Writing - original draft, Software. **Fabio Roli:** Supervision, Funding acquisition. **Giorgio Giacinto:** Supervision, Funding acquisition.

Acknowledgments

This work was partly supported by the project PON AIM Research and Innovation 2014-2020 - Attraction and International Mobility, funded by the Italian Ministry of Education, University and Research; by the PRIN 2017 project RexLearn, funded by the Italian Ministry of Education, University and Research (grant no. 2017TWNMH2); and by the EU H2020 project ALOHA, under the European Unions Horizon 2020 research and innovation programme (grant no. 780788). The authors would like to thank Maria Elena Chiappe, Denis Ugarte and Michele Scalas for their contributions to FlashBuster.

References

Adobe. Actionscript virtual machine 2 overview. 2007.

Adobe. Swf file format specifications. 2012.

Adobe. Actionscript language specifications. 2015. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html.

Any.Run. Interactive malware hunting service. 2020. https://any.run/.

Baeza-Yates, R.A., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Barreno, M., Nelson, B., Joseph, A., Tygar, J., 2010. The security of machine learning. Machine Learning 81, 121–148.

Barreno, M., Nelson, B., Sears, R., Joseph, A.D., Tygar, J.D., 2006. Can machine learning be secure? In: Proc. ACM Symp. Information, Computer and Comm. Sec., ASIACCS '06. ACM, New York, NY, USA, pp. 16–25.

Biggio, B., Corona, I., He, Z.M., Chan, P.P.K., Giacinto, G., Yeung, D.S., Roli, F., 2015. One-and-a-half-class multiple classifier systems for secure learning against evasion attacks at test time. In: Schwenker, F., Roli, F., Kittler, J. (Eds.), Multiple Classifier Systems. In: Lecture Notes in Computer Science, volume 9132. Springer International Publishing, pp. 168–180.

Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., Roli, F., 2013. Evasion attacks against machine learning at test time. In: Blockeel, H., Kersting, K., Nijssen, S., Železný, F. (Eds.), Machine Learning and Knowledge Discovery in Databases (ECML PKDD), Part III. In: LNCS, volume 8190. Springer Berlin Heidelberg, pp. 387–402.

Biggio, B., Fumera, G., Roli, F., 2014a. Pattern recognition systems under attack: Design issues and research challenges. International Journal of Pattern Recognition and Artificial Intelligence 28, 1460002.

Biggio, B., Fumera, G., Roli, F., 2014b. Security evaluation of pattern classifiers under attack. IEEE Transactions on Knowledge and Data Engineering 26, 984–996.

Biggio, B., Nelson, B., Laskov, P., 2012. Poisoning attacks against support vector machines. In: Langford, J., Pineau, J. (Eds.), 29th Int'l Conf. on Machine Learning. Omnipress, pp. 1807–1814.

Biggio, B., Roli, F., 2018. Wild patterns: Ten years after the rise of adversarial machine learning. Pattern Recognition 84, 317–331.

Brückner, M., Kanzow, C., Scheffer, T., 2012. Static prediction games for adversarial learning problems. J Mach Learn Res 13, 2617–2654.

Calleja, A., Martín, A., Menéndez, H.D., Tapiador, J., Clark, D., 2018. Picking on the family: Disrupting android malware triage by forcing misclassification. Expert Systems with Applications 95, 113–126.

Chen, P.Y., Zhang, H., Sharma, Y., Yi, J., Hsieh, C.J., 2017. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In: 10th ACM Workshop on Artificial Intelligence and Security, AISec '17. ACM, New York, NY, USA, pp. 15–26.

Collberg, C., Thomborson, C., Low, D., 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98. Association for Computing Machinery, New York, NY, USA, pp. 184–196. doi:10.1145/268946.268962.

Corpora D.. Digital corpora - producing the digital body. 2013. https://digitalcorpora.org/.

CyberShadow. RABCDAsm. 2019. https://github.com/CyberShadow/RABCDAsm.

Dalvi, N., Domingos, P., Mausam, Sanghai, S., Verma, D., 2004. Adversarial classification. In: Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). Seattle, pp. 99–108.

Dang, H., Huang, Y., Chang, E., 2017. Evading classifiers by morphing in the dark. In: ACM SIGSAC Conference on Computer and Communications Security, CCS '17. ACM, pp. 119–133.

Demontis A., Melis M., Biggio B., Maiorca D., Arp D., Rieck K., Corona I., Giacinto G., Roli F.. Yes, machine learning can be more secure! a case study on android malware detection. IEEE Transactions on Dependable and Secure Computing In press;.

DoSWF. DoSWF - professional flash SWF encryptor. 2013. http://www.doswf.org/.

Globerson, A., Roweis, S.T., 2006. Nightmare at test time: robust learning by feature deletion. In: Cohen, W.W., Moore, A. (Eds.), Proceedings of the 23rd Int'l Conference on Machine Learning, volume 148. ACM, pp. 353–360.

Muñoz González, L., Biggio, B., Demontis, A., Paudice, A., Wongrassamee, V., Lupu, E.C., Roli, F., 2017. Towards poisoning of deep learning algorithms with back-gradient optimization. In: Thuraisingham, B.M., Biggio, B., Freeman, D.M., Miller, B., Sinha, A. (Eds.), 10th ACM Workshop on Artificial Intelligence and Security AISec '17. ACM, New York, NY, USA, pp. 27–38.

Goodfellow, I.J., Shlens, J., Szegedy, C., 2015. Explaining and harnessing adversarial examples. In: International Conference on Learning Representations.

Google. Virustotal. 2018. http://www.virustotal.com.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P.D., 2017. Adversarial examples for malware detection. In: ESORICS (2). In: LNCS, volume 10493. Springer, pp. 62–79.

Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B., Tygar, J.D., 2011. Adversarial machine learning. In: 4th ACM Workshop on Artificial Intelligence and Security (AISec 2011). Chicago, IL, USA, pp. 43–57.

Kantchelian, A., Tygar, J.D., Joseph, A.D., 2016. Evasion and hardening of tree ensemble classifiers. In: 33rd ICML. In: JMLR Workshop and Conference Proceedings, volume 48. JMLR.org, pp. 2387–2396.

Kapratwar., A., Troia., F.D., Stamp., M., 2017. Static and dynamic analysis of android malware. In: Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1: ForSE, (ICISSP 2017), INSTICC. SciTePress, pp. 653–662. doi:10.5220/0006256706530662.

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In: 26th European Signal Processing Conference, EUSIPCO. IEEE, Rome, pp. 533–537.

Lin, D., Stamp, M., 2011. Hunting for undetectable metamorphic viruses. Journal in Computer Virology 7, 201–214.

Lowd, D., Meek, C., 2005. Adversarial learning. In: Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). ACM Press, Chicago, IL, USA, pp. 641–647.

Lyu, C., Huang, K., Liang, H.N., 2015. A unified gradient regularization family for adversarial examples. In: 2015 IEEE International Conference on Data Mining (ICDM), volume 00. IEEE Computer Society, Los Alamitos, CA, USA, pp. 301–309.

Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G., 2015. Stealth attacks: An extended insight into the obfuscation effects on android malware. Comput Secur 51, 16–31.

Maiorca, D., Biggio, B., 2019. Digital investigation of PDF files: Unveiling traces of embedded malware. IEEE Security & Privacy 17, 63–71.

Maiorca, D., Biggio, B., Giacinto, G., 2019. Towards adversarial malware detection: Lessons learned from pdf-based attacks. ACM Comput Surv 52, 78:1–78:36.

Maiorca, D., Corona, I., Giacinto, G., 2013. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ASIA CCS '13. ACM, New York, NY, USA, pp. 119–130.

Maiorca, D., Giacinto, G., Corona, I., 2012. A pattern recognition system for malicious pdf files detection. In: Perner, P. (Ed.), Machine Learning and Data Mining in Pattern Recognition. In: Lecture Notes in Computer Science, volume 7376. Springer Berlin Heidelberg, pp. 510–524.

Mantovani, A., Aonzo, S., Ugarte-Pedrero, X., Merlo, A., Balzarotti, D., 2020. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In: Network and Distributed System Security (NDSS) Symposium NDSS 20.

Mei, S., Zhu, X., 2015. Using machine teaching to identify optimal training-set attacks on machine learners. In: 29th AAAI Conference on Artificial Intelligence (AAAI '15).

Melis M., Demontis A., Pintor M., Sotgiu A., Biggio B.. secml: A Python Library for Secure and Explainable Machine Learning. 2019.

Melis, M., Maiorca, D., Biggio, B., Giacinto, G., Roli, F., 2018. Explaining black-box android malware detection. In: 26th European Signal Processing Conf., EUSIPCO, IEEE. IEEE, Rome, Italy, pp. 524–528.

Overveldt, T.V., Kruegel, C., Vigna, G., 2012. Flashdetect: Actionscript 3 malware detection. In: Balzarotti, D., Stolfo, S.J., Cova, M. (Eds.), Research in Attacks, Intrusions, and Defenses. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 274–293.

Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z.B., Swami, A., 2017. Practical black-box attacks against machine learning. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '17. ACM, New York, NY, USA, pp. 506–519.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830.

Petrik J.. JPEXS. 2020. https://github.com/jindrapetrik/jpexs-decompiler.

Raff E., Barker J., Sylvester J., Brandon R., Catanzaro B., Nicholas C.. Malware detection by eating a whole exe. 2017. ArXiv preprint arXiv:1710.09435.

Ross, A.S., Doshi-Velez, F., 2018. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In: AAAI. AAAI Press.

Rossow, C., Dietrich, C.J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., Steen, M.v., 2012. Prudent practices for designing malware experiments: Status quo and outlook. In: 2012 IEEE Symposium on Security and Privacy, pp. 65–79. doi:10.1109/SP.2012.14.

Russu, P., Demontis, A., Biggio, B., Fumera, G., Roli, F., 2016. Secure kernel machines against evasion attacks. In: 9th ACM Workshop on Artificial Intelligence and Security, AISec '16. ACM, New York, NY, USA, pp. 59–69.

Scalas, M., Maiorca, D., Mercaldo, F., Visaggio, C.A., Martinelli, F., Giacinto, G., 2019. On the effectiveness of system API-related information for Android ransomware detection. Computers & Security 86, 168–182.

Sebastián, M., Rivera, R., Kotzias, P., Caballero, J., 2016. Avclass: A tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., García-Alfaro, J. (Eds.), Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings. In: Lecture Notes in Computer Science, volume 9854. Springer, pp. 230–253. doi:10.1007/978-3-319-45719-2_11.

Simon-Gabriel C.J., Ollivier Y., Schölkopf B., Bottou L., Lopez-Paz D.. Adversarial vulnerability of neural networks increases with input dimension. 2018. ArXiv e-prints.

Singh, T., Troia, F.D., Corrado, V.A., Austin, T.H., Stamp, M., 2016. Support vector machines and malware detection. Journal of Computer Virology and Hacking Techniques 12, 203–212.

Smutz, C., Stavrou, A., 2012. Malicious pdf detection using metadata and structural features. In: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12. ACM, New York, NY, USA, pp. 239–248. doi:10.1145/2420950.2420987.

Smutz, C., Stavrou, A., 2016. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.

Sporst. SWFREtools. 2011. https://github.com/sporst/SWFREtools.

Symantec. Internet security threat report vol.24. 2019. https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R., 2014. Intriguing properties of neural networks. In: International Conference on Learning Representations. http://arxiv.org/abs/1312.6199

Teo, C.H., Globerson, A., Roweis, S., Smola, A., 2008. Convex learning with invariances. In: Platt, J., Koller, D., Singer, Y., Roweis, S. (Eds.), Advances in Neural Information Processing Systems 20. MIT Press, Cambridge, MA, pp. 1489–1496.

Timknip. PySWF. 2017. https://github.com/timknip/pyswf.

Tong, L., Li, B., Hajaj, C., Xiao, C., Vorobeychik, Y., 2017. Hardening classifiers against evasion: the good, the bad, and the ugly. CoRR abs/1708.08327.

Tramèr, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T., 2016. Stealing machine learning models via prediction apis. In: 25th USENIX Security Symposium (USENIX Security 16). USENIX Association, Austin, TX, pp. 601–618.

TrendMicro. North korean hackers allegedly exploit adobe flash player vulnerability (cve-2018-4878) against south korean targets. 2018.

Šrndic N.. Hidost. 2016. https://github.com/srndic/hidost.

Šrndić, N., Laskov, P., 2013. Detection of malicious pdf files based on hierarchical document structure. In: Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS). The Internet Society.

Šrndic, N., Laskov, P., 2014a. Practical evasion of a learning-based classifier: A case study. In: Proc. 2014 IEEE Symp. Security and Privacy SP '14. IEEE CS, Washington, DC, USA, pp. 197–211.

Šrndic, N., Laskov, P., 2014b. Practical evasion of a learning-based classifier: A case study. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP '14. IEEE Computer Society, Washington, DC, USA, pp. 197–211. doi:10.1109/SP.2014.20.

Šrndić, N., Laskov, P., 2016. Hidost: a static machine-learning-based detector of malicious files. EURASIP Journal on Information Security 2016, 22.

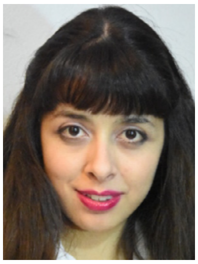
Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K., 2016. Comprehensive analysis and detection of flash-based malware. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, Cham, pp. 101–121.

Xu, M., Kim, T., 2017. Platpal: Detecting malicious documents with platform diversity. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, pp. 271–287. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-meng

Xu, W., Qi, Y., Evans, D., 2016. Automatically evading classifiers. In: Proceedings of the 23rd Annual Network & Distributed System Security Symposium (NDSS). The Internet Society.

Yang, W., Kong, D., Xie, T., Gunter, C.A., 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In: ACSAC. ACM, pp. 288–302.

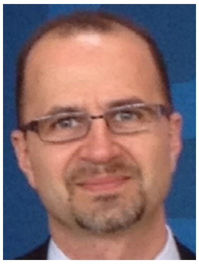
Davide Maiorca received from the University of Cagliari (Italy) the M.Sc. degree (Hons.) in Electronic Engineering in 2012, and the Ph.D. in Computer and Electronic Engineering in 2016. He is currently an Assistant Professor at the Department of Electrical and Electronic Engineering, University of Cagliari. In 2013, he visited the Systems Security group at Ruhr-Universitat Bochum, guided by Prof. Dr. Thorsten Holz, and worked on advanced obfuscation of Android malware. His current research interests include adversarial machine learning, malware in documents and Flash applications, Android malware, and mobile fingerprinting. He has been serving as a reviewer and program committee member for various conferences and journals.



Ambra Demontis is an Assistant Professor at the University of Cagliari, Italy. She received her M.Sc. degree (Hons.) in Computer Science and her Ph.D. degree in Electronic Engineering and Computer Science from the University of Cagliari, Italy, in 2014 and 2018. In 2016, she visited the University of Manchester, UK, where she worked on the security of machine learning to test-time attacks. Her research interests include secure machine learning, kernel methods, biometrics, and computer security. She serves as a reviewer for several international conferences and journals. She is a Member of the IEEE and the IAPR.

Battista Biggio is an Assistant Professor at the University of Cagliari, Italy. In 2015, he co-founded Pluribus One, a research-intensive startup company that develops innovative solutions based on machine learning for cybersecurity. He received the M.Sc. degree (Hons.) in Electronic Engineering and the Ph.D. degree in Electronic Engineering and Computer Science from the University of Cagliari, Italy, in 2006 and 2010. His research interests include adversarial machine learning, kernel methods, biometrics and cybersecurity. In particular, he has provided pioneering contributions in the area of secure machine learning, demonstrating evasion and poisoning attacks, and how to mitigate them, playing a leading role in the establishment and advancement of this research field. He regularly serves as a program committee member for the most prestigious conferences and journals in the area of machine learning and computer security (ICML, NeurIPS, ACM CCS, IEEE SP). He chairs the IAPR TC on Statistical Pattern Recognition Techniques, co-organizes the S+SSPR, DLS and AISec workshops, and serves as Associate Editor for IEEE TNNLS, Pattern Recognition and IEEE CIM. Dr. Biggio is a senior member of the IEEE and member of the IAPR and of the ACM.

Fabio Roli received his Ph.D. in Electronic Engineering from the University of Genoa, Italy. He was a research group member of the University of Genoa ('88-'94). He was adjunct professor at the University of Trento ('93-'94). In 1995, he joined the Department of Electrical and Electronic Engineering of the University of Cagliari, where he is now professor of Computer Engineering and head of the research laboratory on pattern recognition and applications. His research activity is focused on the design of pattern recognition systems and their applications. He was a very active organizer of international conferences and workshops, and established the popular workshop series on multiple classifier systems. Dr. Roli is Fellow of the IEEE and of the IAPR.



Giorgio Giacinto is Professor of Computer Engineering at the University of Cagliari, Italy. Since 1995 he joined the research group on Pattern Recognition and Applications, where he leads the Computer Security unit. His research interests are in the field of pattern recognition and machine learning for malware analysis and detection, web application security, and phishing detection. Prof. Giacinto has been serving either as the coordinator in many R&D projects at the local, national and European level. Prof. Giacinto is an author of around 150 scientific papers in international journals and conferences. He is a senior member of IEEE and ACM.