Towards Building Active Defense for Software Applications

by

Zara Perumal

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2018

Author .....
Department of Electrical Engineering and Computer Science
January 19, 2018

Certified by .....
Kalyan Veeramchaneni
Principle Research Scientist
Thesis Supervisor

Accepted by .....
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Towards Building Active Defense for Software Applications

by

Zara Perumal

Submitted to the Department of Electrical Engineering and Computer Science
on January 19, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

Over the last few years, cyber attacks have become increasingly sophisticated. In an effort to defend themselves, corporations often look to machine learning, aiming to use the large amount of data collected on cyber attacks and software systems to defend systems at scale. Within the field of machine learning in cybersecurity, PDF malware is a popular target of study, as the difficulty of classifying malicious files makes it a continuously effective method of attack. The obstacles are many: Datasets change over time as attackers change their behavior, and the deployment of a malware detection system in a resource-constrained environment has minimum throughput requirements, meaning that an accurate but time-consuming classifier cannot be deployed. Recent work has also shown how automated malicious file creation methods are being used to evade classification.

Motivated by these challenges, we propose an *active defender* system to adapt to evasive PDF malware in a resource-constrained environment. We observe this system to improve the f_1 score from 0.17535 to 0.4562 over five stages of receiving PDF files that the system considers unlabeled. Furthermore, average classification time per file is low across all 5 stages, and is reduced from an average of 1.16908 seconds per file to 1.09649 seconds per file.

Beyond classifying malware, we provide a general *active defender* framework that can be used to deploy decision systems for a variety of resource-constrained adversarial problems.

Thesis Supervisor: Kalyan Veeramchaneni
Title: Principle Research Scientist

Acknowledgments

First, I would like to thank my advisor Kalyan for exploring a new direction in my project. When I started my MENG, I was excited about using machine learning in cybersecurity, but I knew little about the details of what that meant. He took a risk on my project and we dove into a new area of work. Throughout the process of brainstorming, exploring experimental options, identifying interesting problems, and building and deploying a system, Kalyan continuously supported me and provided feedback with new insights on the project. I am incredibly grateful to him for providing me this opportunity to work on such an interesting project.

I would like to thank Professor Rivest for sparking my interest in cybersecurity. Taking 6.857 Senior Spring and having the opportunity to work on election cybersecurity motivated me to pursue cybersecurity.

Thank you to the Data to AI lab for providing feedback, and thank you Brian Jones for helping me set up the Malware sandbox. I would also like to thank Course 6 department and especially Anne Hunter for enabling me to pursue the my Masters of Engineering.

Finally, I am thankful to my family for their love and support. I would also like to thank the friends and mentors who made the last five years not only rewarding, but also fun. I would like to thank Kyra Horne for providing food and support and enabling me to finish my thesis. Finally, I would like to thank Bullet for being the best dog, and a "security expert."

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, cyber attacks have increased dramatically in scale and sophistication. Last spring brought the WannaCry ransomware attack, which crippled the computers of both users and institutions around the world[12]. Soon after came attacks on the Equifax credit reporting agency, which resulted in the release of the personal information of millions of users [24]. In addition, banks and Bitcoin exchanges have been subject to an increasing number of attacks throughout the last few years [2]. A common trend across recent attacks is the increasing sophistication of the attackers. Recent cyber attacks are increasingly attributed to Nation-State actors, or Nation-State sponsored cyber-gangs. These powerful attackers often target individuals or small-scale enterprises. The asymmetry between the resources an adversary can devote to attacking a system and the resources a corporation can devote to preventing such an attack presents a challenging problem.

In addition, the increasing scale of software systems makes it even more difficult to defend against attacks. Even a small social media platform cannot comb through every user's behaviour manually to see if an account has been compromised. As enterprises collect and store more and more data, machine learning is being used to help teams ranging from marketing and sales and human resources to product development and execution. As a result, the interest in developing machine learning-based solutions has increased exponentially[5].

These enterprises also seek a better answer to the question "Can we use machine

learning to detect, predict and defend against cyber attacks?" As researchers study the problem, we have begun to realize that developing solutions for cyber security is one of the most complex machine learning endeavors we can undertake.

Many significant challenges stand in the way of automating security. The first is the evolving nature of cyber attacks. Traditional machine learning operates under the assumption that the data used to model behaviour is similar to what arrives when the system is deployed. This doesn't hold in cybersecurity. As a researcher is designing a defense system by modeling data, attackers are designing automated evasive algorithms to evade these deployed models. This means that machine learning models based on past attacks will quickly become outdated if they cannot automatically adapt to a changing environment.

The second challenge stems from the complex dynamics of the security ecosystem. The actors in a given security problem generally include enterprises who want to defend themselves, sophisticated attackers, overburdened security analysts, and end-users with a limited knowledge of how to protect themselves and subsequently the enterprise. Complications might include detection strategies being public knowledge, or the limited availability of computational resources. Such a dynamic and complex environment means that solutions can fall short in a number of ways. For instance, a highly optimized and accurate attack detection solution could be useless if the enterprise does not have the resources to deploy it.

In the large space of machine learning in cybersecurity, recent research has focused both on the sub-problem of PDF malware detection and, more recently, the automated evasion of detection. 91% of security breaches are caused by low volume phishing attacks[1]. These attacks typically use social engineered messages in combination with malicious URLs or files to gain entry into a system. PDFs are an especially popular method of attack. PDFs appear commonplace, and end users are used to receiving a variety of benign documents as PDFs, including e-books, papers, syllabi, and reports. While users inherently trust PDFs, they are an incredibly flexible document format that make it easy to embed and disguise a variety of malicious code. This makes

[1]http://info.greathorn.com/2017-spear-phishing-report

them a promising method of attack. While PDFs have been used for years as an attack vector, PDF-based malware is a continuous form of attack as vulnerabilities are found in PDF readers such as Google Chrome [2]. Furthermore, document-based malware can be a powerful entry point and has been shown to be able to download other malware, hide malicious scripts within the document, spy on users, or encrypt an end user's computer in a ransomware attack [4].

Because PDF malware is so dangerous, a variety of solutions have been proposed for preventing these attacks, using different methods of classification. Starting at the delivery mechanism, network analysis is used to detect an unusual volume of emails. Next, static classifiers are used to check for known malicious bit-strings or to extract the simple features used in machine learning classifiers. Dynamic classifiers put the file in an isolated environment and look for malicious behaviour. API-based classifiers allow files to be submitted and analyzed across a suite of different and powerful detection methods. Finally, human analysts can manually inspect a file using a variety of software and hardware tools.

These solutions have two problems. First, in a resource-constrained environment, a defense system does not have the time necessary to send every file it receives to an accurate classifier such as a human analyst, and must choose a faster, less accurate option. Second, automated classifiers can be evaded by sophisticated attackers, and such a system can become obsolete if an attacker's behaviour changes after it is deployed [39].

Motivated by these problems, we designed the *SMDA framework*. We aimed to design a framework that could provide accurate detection in a resource-constrained, adversarial environment. A system that can intelligently handle resource limitations and remain robust to a changing environment can be widely deployed. Furthermore, our system uses active learning to provide accurate results in the presence of adversarial evasion algorithms. In this project, we also propose a software system that can generalize to a variety of adversarial problems, and unify researchers and developers in developing active defense solutions.

[2] https://thestack.com/security/2016/06/09/pdf-exploit-found-in-default-google-chrome-reader/

The SMDA framework provides four modules – *Synthesize*, *Model*, *Decide*, *Adapt* – to generate data, develop machine learning models of the data and true labels, classify input data, and adapt to improve the system over time. By developing a general framework, we can use this software system for a variety of use cases and easily benchmark, improve, and deploy classification systems.

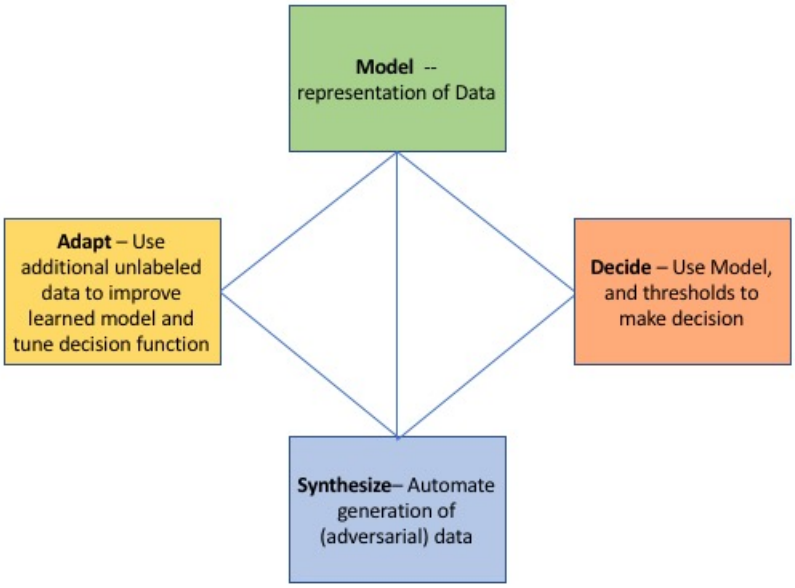

Figure 1-1: Modules of the SMDA framework for building adaptive models in adversarial environments.

For the PDF malware use case, we explored how the SMDA framework could be used to build the *Active Defender* PDF malware classification system. Several recent studies suggest automated methods for PDF malware classification evasion [39, 11, 14, 28]. By deploying the model in the PDF use case and generating evasive data, we are able to easily test how the algorithm performs against evasive adversaries.

1.1 Contributions

In this thesis, we make the following contributions to enable machine learning in cybersecurity.

1. We provide an active learning decision system to maintain high accuracy in the

presence of motivated adversaries using evasive strategies.

2. We propose a resource allocation algorithm to optimize the use of available detection methods in a resource-constrained environment.

3. We provide a general purpose *SMDA* framework to enable the building, evaluating, and deploying of decision systems in an adversarial environment.

4. We present the Max-Diff algorithm, which is shown to confuse even the most sophisticated automated classifier.

1.2 Thesis Organization

The remaining chapters of this thesis are organized as follows:

- Chapter 2 describes PDF malware and the data set used in this project.
- Chapter 3: introduces the SMDA logical framework.
- Chapter 4: describes the *Synthesize* module of *SMDA*.
- Chapter 5: describes the *Learn* module of *SMDA*.
- Chapter 6: describes the *Decision* module of *SMDA*.
- Chapter 7: describes the *Adapt* module of *SMDA*.
- Chapter 8: describes the experimental results and performance of the *active defender* system.
- Chapter 9: describes our key findings and future directions.

Chapter 2

Malware through PDFs

The portable document format (PDF for short) is the most popular format for document sharing. As a result, PDFs are often passed through emails as attachments. Many applications also support PDF uploads, which are used for everything from academic conference paper submissions to government agencies accepting tax forms. Because PDFs are so common, end users often trust them. However, this unassuming document contains a powerful format that enables attackers to embed and hide malicious code. In this thesis, we will focus on PDFs to create an *active defender* using our *smda* framework. This chapter will cover the following core concepts:

- The structure of PDFs
- How malware is embedded in PDFs
- The current detection techniques available

2.1 Structure of a PDF

The PDF file that we see on a daily basis is a rendered version of PDF source code. PDFs use a hierarchical tree structure to store objects. The high-level structure of a PDF contains four main sections.

- Header: The header contains the PDF number and format information.

- Body: The body is the essential element of PDF files. It contains objects of the 8 basic types, namely: Boolean, Numeric, String, Null, Names – representing key labels or name such as "//Page", Arrays, Dictionaries, and Streams, corresponding to an dictionary and sequence of bytes. The objects are organized in a hierarchical tree structure, storing the relations between various objects. This tree structure enables the storage of complex relationships.
- Cross Reference Table (CRT): The CRT indexes the components in the body
- Trailer: The trailer specifies how to find the CRT and other special objects

hello_world.pdf

```
%PDF-1.4

1 0 obj <</Type /Catalog /Pages 2 0 R>>
endobj
2 0 obj <</Type /Pages /Kids [3 0 R] /Count 1>>
endobj
3 0 obj<</Type /Page /Parent 2 0 R /Resources 4 0 R /MediaBox [0 0 500 800] /Contents 6 0 R>>
endobj
4 0 obj<</Font <</F1 5 0 R>>>>
endobj
5 0 obj<</Type /Font /Subtype /Type1 /BaseFont /Helvetica>>
endobj
6 0 obj
<</Length 44>>
stream
BT /F1 24 Tf 175 720 Td (Hello World!)Tj ET
endstream
endobj
xref
0 7
0000000000 65535 f
0000000009 00000 n
0000000056 00000 n
0000000111 00000 n
0000000212 00000 n
0000000250 00000 n
0000000317 00000 n

trailer <</Size 7/Root 1 0 R>>

startxref
406
%%EOF
```

Figure 2-1: Source for a simple hello world PDF

For example, we analyze the simple Hello World PDF described in an IDR solutions tutorial[1]. In this example, we view the source in a text editor in Figure 2-1 and the rendered version in Figure 2-2.

[1]https://blog.idrsolutions.com/2010/10/make-your-own-pdf-file-part-4-hello-world-pdf/

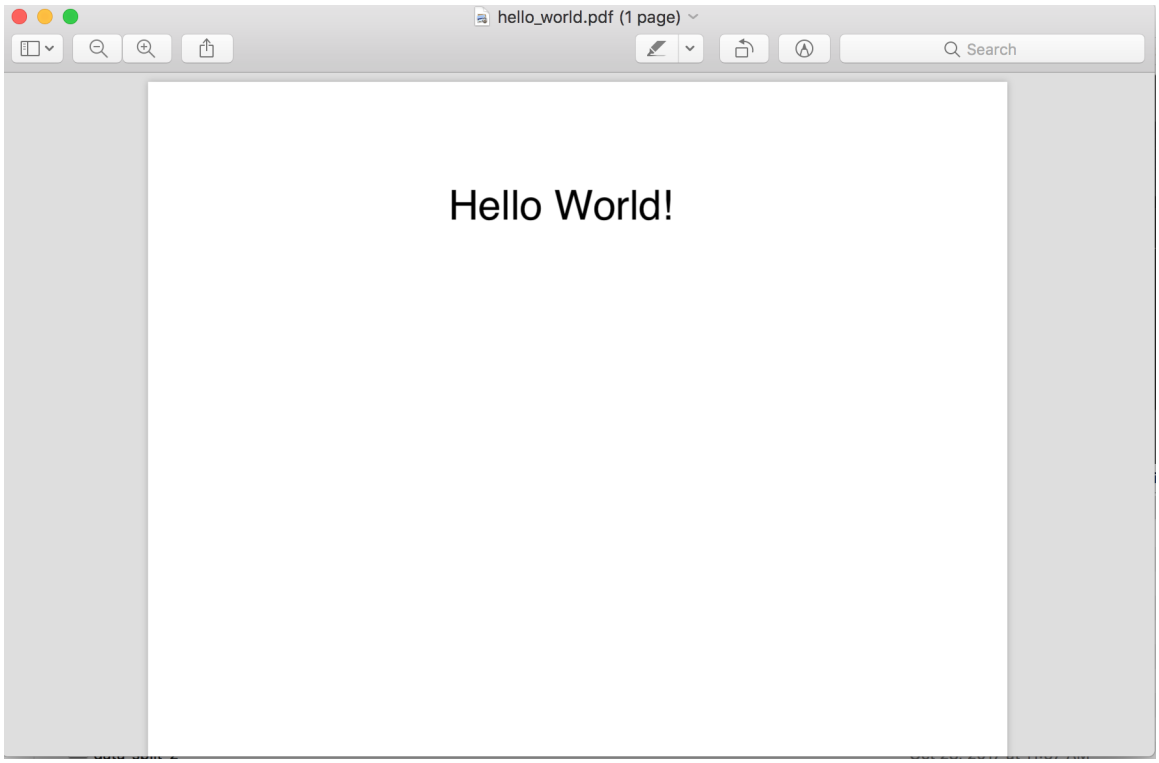

Figure 2-2: Rendered version of simple hello world PDF

In looking through the PDF source in Figure 2-1 we see the header in line 1, the body in lines 3 through 18, the cross reference table (CRT) in lines 19 through 27 and the trailer from lines 31 to 33. The header simply describes the version of PDF used. In the body, the nodes of the tree and the corresponding objects at nodes 1 through 6 are described. In the cross reference table we see "xref" followed by 0, indicating the special node that marks the head of the list and 7, representing the total number of objects in the list including the head of the list. In lines 21 through 27 we see the base list of objects described with the offset, and whether each object is in the body section (lines ending in n) or not. In lines 29 through 33, we see the trailer section which describes the byte offset to the xref key work.

While this is one source code representation that can generate the rendered version shown in Figure 2-2, many other source implementations can result in the same visual document.

2.2 Malware in PDFs

The flexible structure of PDFs means it is easy to manipulate them by changing the file slightly and hiding code deep within branches of the tree structure. Furthermore, the hierarchical structure enables easy mutation by inserting, swapping or deleting elements across multiple files [32].

PDF malware is an effective way for adversaries to gain access to a computer system. Since end users trust PDFs and open them more frequently than other files, they provide a conduit for attackers to gain access to a system using a static document linked to on a website or sent as a shared file. While PDF reader vulnerabilities have been reported for years, new threats are continuously discovered as attackers become more sophisticated, targeting vulnerabilities and hiding their approaches [34]. In the past year alone, over a hundred new vulnerabilities were reported for PDFs on a variety of targets from Acrobat to Chrome [39, 3].

```
19  8 0 obj
20  <<
21  /JS (
22  var pos = [20, 600, 54,642];
23  this.addField("Hello World!","text",0,pos);
24  )
25  /S /JavaScript
26  >>
27  endobj
```

Figure 2-3: Example of un-obfuscated code as a PDF object

Due to the expressive nature of the PDF format, there are many places to hide malicious functionality within a PDF. Most, but not all, PDF exploits are based on Javascript. Javascript is often buried within deep branches of the PDF object structure. These scripts can also be disguised and encoded in streams that are only interpreted as Javascript through the *eval*() function, so that even with manual inspection of the PDF source and Javascript, the purpose may be unclear [31]. Even if

```
19  8 0 obj
20  <<
21  /Producer (evalString.fromCharCode(118,97,114,32,112,111,115,32,61,32,
    91,50,48,44,32,54,48,48,44,32,53,52,44,54,52,50,93,59,10,116,104,105,1
    15,46,97,100,100,70,105,101,108,100,40,34,72,101,108,108,111,32,87,111
    ,114,108,100,33,34,44,34,116,101,120,116,34,44,48,44,112
    ,111,115,41,59,10))
22  /CreationDate (D:20180201112311)
23  >>
24  endobj
25
26  9 0 obj
27  <<
28  Pages 2 0 R
29  /Names <</JavaScript <</Names[() <</S /JavaScript
30  /JS (var a = this.producer;)
31  b=eval\(a.substr\(0,4\)\
32  ......
33  >>
34  endobj
```

Figure 2-4: Example of obfuscated code as a PDF object

the injected code is the same, attackers can mutate other aspects of the file, such as the length and contents, to change the file and confuse classifiers.

Javascript can be embedded as a stream object in a PDF file[2]. In Figure 2-3, we see a representation of a Javascript stream object. In this case, it is a benign alert box. However, using character encoding, adversaries can disguise the purpose of the code used. In Figure 2-4, we see the same code represented as an encoded stream, which is then called in a later object. Here it is unclear what the purpose of the code is, which makes it more difficult to scan a file for known malicious Javascript code. In practice, attackers may use automated tools to inject code to attack known PDF reader vulnerabilities [26].

2.3 Our dataset

To demonstrate the efficacy of different detection techniques in Section 4 we have currently generated a repository of 207,119 total PDFs. From these files, we collected

[2]https://www.cyren.com/blog/articles/how-pdf-files-hide-malware-example-pdf-scan-from-xerox-1247

classifier scores for 79,949 files and created a labeled set. Of these files, 35,269 are malicious files and 44,680 are benign. We created the dataset from a combination of existing, externally provided PDF files, and variations of these PDF files generated via a process we call *mutation*. Here is a list of sources we used to gather these PDFs:

- Contagio: External PDFs were downloaded from the Contagio dataset. The Contagio dataset provided a corpus of 9,000 benign PDFs and 10,597 malicious files[1].

- EvadeML: Evade ML data provided by Weilin Xu contains 16440 malicious PDF files developed using the "EvadeML" algorithm. These files are based off of 500 malicious files in the Contagio data set and are designed to confuse the PDFRate classifier [39].

- Self-generated: PDFs can be generated from existing PDFs according to the "Random-Mutation","Evade-M" [39] or"Max-Dif" algorithms. These are described in more detail in the Chapter 4. Both the "Evade-ML" and "Max-Diff" algorithms are based on genetic programming. These algorithms create pools of samples, score them, and select the best-scoring samples to mutate to create more malicious files. In this data set, we generated 8232 malicious files using the "Max-Diff" algorithm and 35,680 benign files using random mutation.

Base PDF Information We keep the following base information for each PDF in the dataset:

- Filepath: the local location of the file

- Malicious Source: Boolean flag indicating if this is derived from a malicious or benign file

- hash: sha1 hash of the file to uniquely identify the sample

- can_parse: Boolean flag indicating if this is parseable by a PDFRW PDF reader which reads in a PDF file into a JSON tree structure. In practice, most benign

PDFs are parsable, but many malicious PDFs are malformed and while they can be used to exploit a system, cannot be read by the PDFRW parser used to create mutants of files.

2.4 PDF Malware Detection

There have been many approaches to detecting malware within PDFs, operating off of various software security designs. Throughout the many implementations of such detection, there are three main types of PDF malware detection and prevention: network features, static file features, and dynamic behavioural analysis. Other methods use a combination of detection methods to give a final prediction.

2.4.1 Using Network Features

Network detection is the first line of defense, and aims to prevent delivery of malicious content before a user has a chance to download it. Through email analysis such as spam detection or network frequency methods, enterprises can filter out anomalous behaviour and limit the phishing emails and attached malware that makes it to the end-user [15, 21]. This is usually done in combination with static or dynamic analysis.

2.4.2 Using Static Features

Static detection uses static features of the PDF data to process files quickly before passing them on to the users. These methods are preferred for their low latency, but have higher error rates than dynamic methods. Static classification methods include signature based detection methods which could search a received file for unique bit string of known malicious files. Other methods attempt to utilize higher level features of the file. Early solutions were based off of n-gram analysis or javascript pattern recognition [20, 22]. However the most successful static PDF feature based classifiers, have been $PDFRate$ which operates off of PDF meta-data and byte-level filestructure and $Hidost$ operating off of structural paths.

Feature	Name
pdfrate 0	author_dot
pdfrate 1	author_lc
pdfrate 2	author_len
pdfrate 3	author_mismatch
pdfrate 4	author_num
pdfrate 5	author_oth
pdfrate 6	author_uc
pdfrate 7	box_nonother_types
pdfrate 8	box_other_only
pdfrate 9	company_mismatch
pdfrate 10	count_acroform

Table 2.1: Subset of features used by the PDFRate classifier. The full set of PDFRAte features is available in Apprendix A

PDFRate classifier: The PDFRate classifier is a static machine learning classifier which collects features based on the structure and meta data of a PDF document [30]. Once the model is trained, it is fast, taking less than a second to classify a document. However it has been shown to be evaded by adversarial algorithms using genetic programming methods [39]. The classifier works as follows:

- Extract features/attributes: We extract 135 features, described in the PDFRate documentation [6],[29]. The PDFRate features are considered to be set of features for static classification [39]. The features describes statistics and attributes of the of the PDF structure such as sown in Table 2.1 and Appendix A.

- Train a machine learning classifier: The standard PDFRate classifier is a random forest machine learning classifier trained on a labeled features from 5,000 malicious and 5,000 benign files from the Contagio dataset [6, 29]. The classification score is a floating point value where scores close to -1 indicate benign files and scores close to 1 correspond to malicious files. This data is separate from the main dataset we use in for our experiments.

- For a new PDF the detection is made by:

  - Extracting the features

– Generating a score using the trained classifier

– Thresholding a score and making a decision

How good is this classifier? In reviewing the distribution of PDFRate classification scores Figure 2-5,we observe that the distribution does not have clear separation between the benign and malicious scores. The PDFRate classifier is able to separate a portion of the malicious files, however a large portion of the files are misclassified and receive similar scores as the benign files.

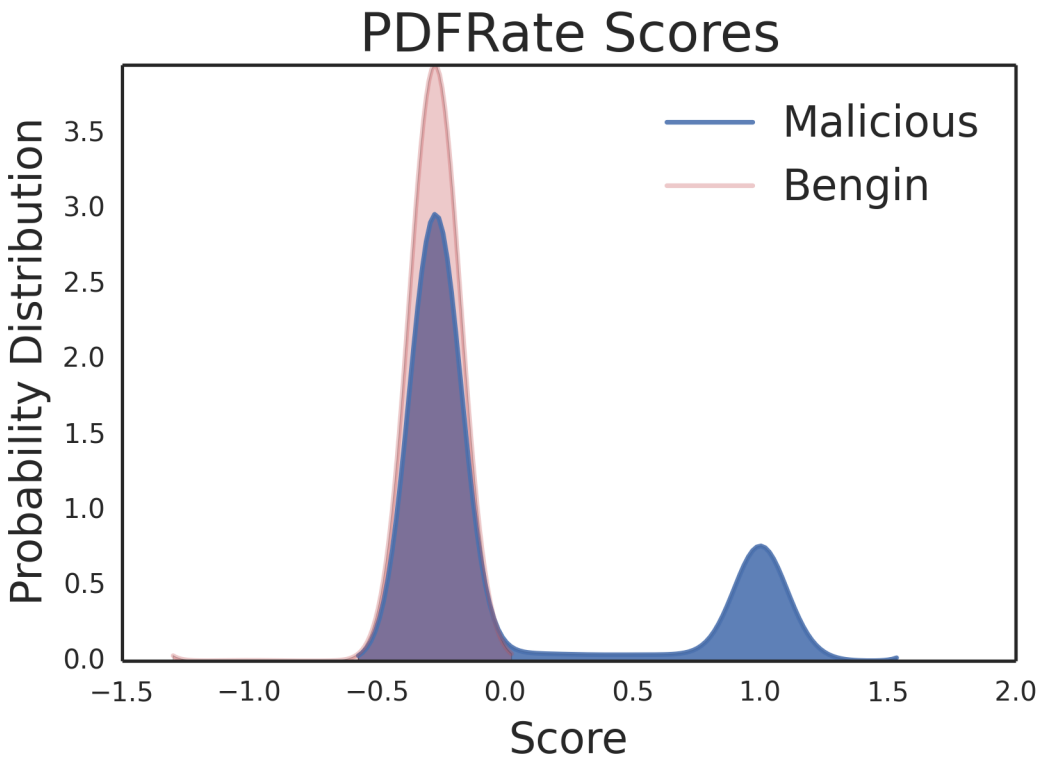

Figure 2-5: KDE approximation of Probability Density for scores generated using the PDFRate classifier. This plot shows the malicious variants in pink and benign variants in blue. The KDE plot was generated with a Gaussian kernel of width 0.1

2.4.3 Dynamic Behavioural Analysis

These methods have been found to have remarkably high accuracy compared to static approaches. Dynamic methods observe the behaviour of a file on an isolated virtual machine or hardware sandbox. Since they monitor the behaviour of a file, rather than

features they can be more difficult to evade. However they can be evaded through environmental detection, delaying malicious behaviour or other methods. Furthermore, these methods have high latency and would significantly reduce performance if they were the only source of detection [31]. Sandboxes such as the *Cuckoo* dynamic sandbox detection framework works well for an Oracle to ensure a mutated file maintains the malicious behaviour of the source file. However, for use in deployed detection systems, this solution has latency that can range from 5 seconds to several minutes to get results. [39].

Cuckoo The Cuckoo Sandbox runs each PDF dynamic analysis sandbox on an isolated "sandboxed" environment. The isolated environment could be virtual machines or an isolated computer. A Cuckoo server runs on the host computer receives files and sends them to a virtual machine for analysis. In the virtual machine, Cuckoo simulates opening PDFs in vulnerable version of Adobe Acrobat, and collects information and compares to a set of known behavioural signatures. Cuckoo is fairly accurate for known malicious signatures, and usually requires 30 seconds of simulation time in a virtual machine. For our experiments we used virtual machines set up in VMWare, running Windows XP and a vulnerable version of Adobe Acrobat Reader 8.1.1. From the results of the Cuckoo classification, we recorded the following outputs for each PDF:

- Signatures observed : This is a list of the known malicious behavioural signatures observed on the virtual machine when the PDF was processed. If the list is empty, no signatures were observed in Cuckoo.

- Cuckoo_Decision: This is a Boolean variable describing if any signatures were observed

How good is this detector? The Cuckoo classifier also is shown to achieve better separation in malicious and benign scores than the PDFRate classifier, as shown in Figure 2-6. However, we still observe a portion of the malicious files achieve the classification scores as benign files.

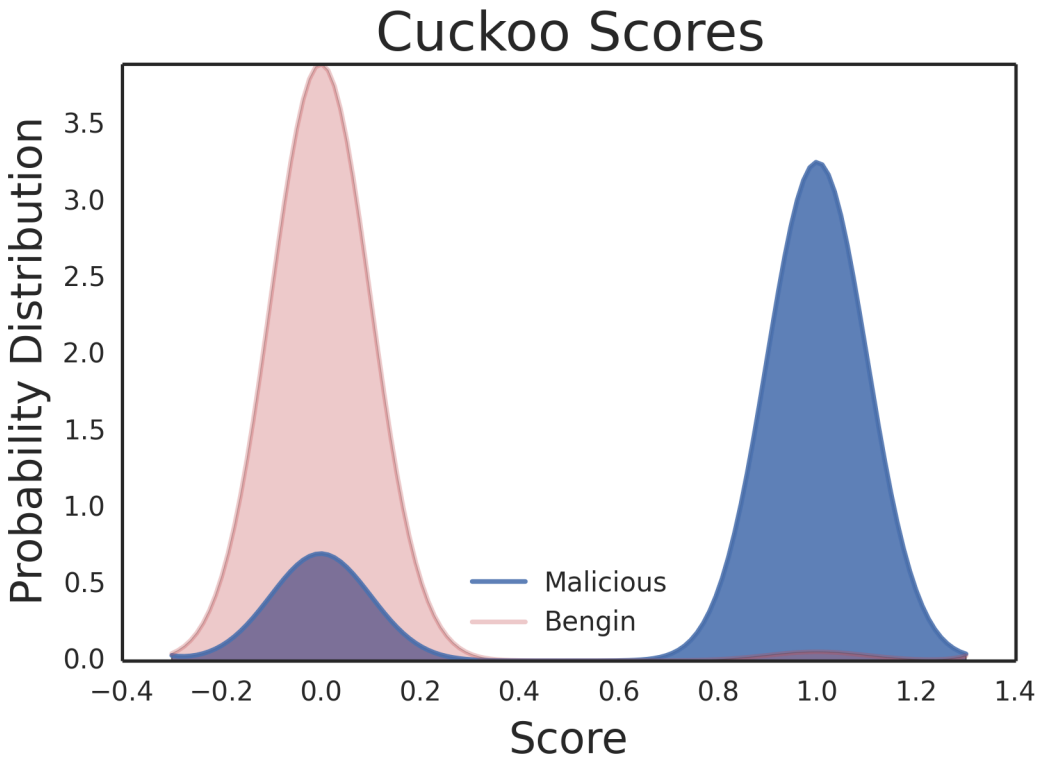

Figure 2-6: KDE approximation of Probability Density for binary Cuckoo classifier scores. This plot shows the malicious variants in pink and benign variants in blue. The KDE plot was generated with a Gaussian kernel of width 0.1

2.4.4 Publicly available APIs

Virus Total is an API based classifier that runs a suite of static classifiers anti-virus engines that run on a submitted PDF. These anti-virus engines often use a combination of classification techniques to return results. Aggregating these results can result in a highly accurate classification, but can be time and resource intensive. We typically observed classification to take about two minutes, but in times of high server load this can be longer. Furthermore corporations may be rate limited by the API and may have to pay for uploads.

We collect the following information for each file from the Virus Total results:

- Percent Malicious : This is the aggregate percent of antivirus engines that classified the uploaded PDF as malicious

- Classifier Results. For each of the scanners used by Virus Total we collect

the following classification attributes: version of scanner used, scan result, and malicious classification. Some of the classifiers used by Virus Total include Endgame, Kaspersky Antivirus, Symantec, and Sophos. See full set of Virus Total Classifiers in appendix [B]

How good is this detector?: Virus Total is considered to be state of art in malicious file scanning and is operated by Google. It runs up to 59 other static dynamic and antivirus classifiers and returns the results of classification. Furthermore it collects a large data set of files from submitters around the world. In 2007 it was listed as one of the best products developed that year [3]. Virus Total is shown to perform the best of all the classifiers and is able to catch many evasive files as shown in Figure 2-7. It performs better than PDFRate and Cuckoo is able to distinguish most malicious files, however there are still some malicious files that receive the same classification scores as benign files.

2.4.5 Human expert analysis

Using human analysts to inspect malware samples is the most accurate form of detection. Analysts can compare samples through a variety of methods comparing networks calls, memory access, or running the sample on a hardware sandbox or comparing activity on a device through a firewall. This is an extremely accurate form of classification; however, the scale of incoming PDFs requires other methods to be used.

How good are humans?: In addition to the automated classifiers we model a human classifier with access to a variety of software an hardware tools. The human is 100% accurate in classification, but is very time intensive and can take several hours to return a result.

[3] https://www.google.com/search?q=best+products+2007+virus+totaloq=best+products+2007+virus+totalaqs=chrome..69i57j69i64.3734j0j4sourceid=chromeie=UTF-8

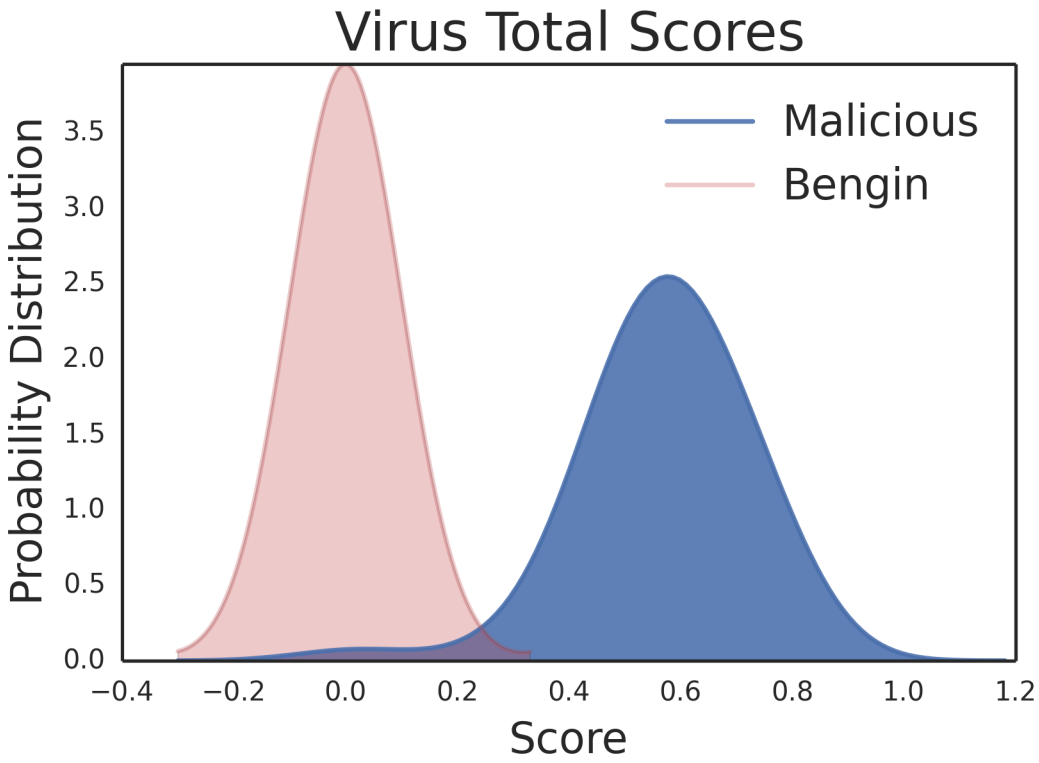

Figure 2-7: KDE approximation of Probability Density for percent of Virus Total classifiers that classify a file as malicious. This plot shows the malicious variants in pink and benign variants in blue. The KDE plot was generated with a Gaussian kernel of width 0.1.

2.4.6 Hybrid systems

Hybrid classification systems have been developed to attempt to achieve increased accuracy without the time constraints of dynamic systems. The ALDOCX system uses active learning combined with human analysts to provide an accurate DOCX classification framework [23]. Other systems such as MDScan combine static document analysis and dynamic code execution on PDFs, however these systems can be expensive or time consuming if the require multiple types if they expensive classifiers every time[36].

2.5 Conclusions

From the data analysis, we observe that neither the initial PDFrate or Cuckoo classifiers perform well on adversarial data. This suggests combining information from classifiers and using the Virus Total classifier at least some point in training is necessary to obtain accurate classification results.

Chapter 3

Active defender system

In this project, we aim to not only provide a method of PDF Malware detection system, but also to provide general purpose framework for building decision systems that perform well in the presence of adversaries and a changing dataset. To this effort, we propose the *SMDA* framework. The goal of this framework is to separate the algorithms used to deploy a decision system into separate logical models.

Separating the process into distinct modules enables researchers and developers to more easily pursue their targeted interests and interface with each other. Researcher scientists can more easily develop and evaluate proposed algorithms for specific components. Developers can more easily combine these modules and choose algorithms that perform best for their use case.

Through studying PDF Malware detection, we found evaluating and deploying classification systems in an adversarial environment can be broken down into the following four logical modules *Synthesize*, *Model*, *Decide*, and *Adapt* (SMDA). Synthesize corresponds to algorithms to create (adversarial) data. Model corresponds to methods for modeling the data. Decide corresponds to the decision function turning an internal representation and features of a sample into a prediction. Adapt corresponds to functions used to update learned models and decision function.

By providing these abstractions and general purpose framework, we hope to facilitate easy integration of state of the art algorithms deployed for a variety of use cases.

Name	Description
n_v	Desired number of files to generate
y	desired label to generate data
S_m	set of malicious samples
S_b	set of benign samples
$o()$	oracle function that indicates if malicious functionality in tact
$S_{mutants}$	set of mutant files
S_m^mutant	set of malicious mutant files
S_{evade}	set of malicious samples
$cutoff$	cutoff specifying the minimum fitness function score for "evasive" files
$f_{mutate}(S_m, S_b)$	function that creates mutations using a set maliciuos and benign files
C	Classifier
p	Individual probabilistic score
C_1	PDFRate (primary) classifier
p_1	Output of C_1
C_2	Cuckoo (primary) classifier
p_2	Output of C_2
C_3	Virus Total (primary) classifier
p_3	Output of C_3
C_4	Secondary classifier using p_1 and p_2 as inputs
p_4	Output of C_4
C_5	Secondary classifier using p_1, p_2, and p_3 as inputs
p_5	Output of C_5
P_1	set of p_1 scores
P_3	set of p_3 scores
S	Samples
$\|S\|$	Number of samples
s	Individual sample
C	Classifier
Y	True Labels
$\hat{Y}$	Predicted Labels
P	Set of probabilities
p	Individual probabilistic score
p_{last}	Output of the last classifier used in making a decisino
D_i	Decision function
t_i^1	Lower threshold for probability score i
t_i^2	Upper threshold for probability score i
t_{last}	Last threshold used in making a decision
N_{pc_used}	Number of primary classifiers used in classifying a file
N_{pc_total}	Number of total classifiers used
γ	value of accuracy score vs. time

Table 3.1: Notation and Definitions used in the SMDA algorithms table (1/2)

Name	Description
$\sum_i r_i$	Total time taken to make decision per file
$\frac{1}{\|S\|}\sum_i r_i$	Average time taken to make decision per file
recall()	Recall
precision()	Precision
f_1	function that computes the f1 score
β	$precision\_recall\_weight$ – weight of precision vs recall
$g(.)$	Function describing accuracy of a system
g_1	Specific evaluation function maximizing f_1 score.
g_2	Specific evaluation function maximizing recall given precision above 0.9
c	Cost function
$e()$	Enumeration function that generates initial threshold sets to evaluate
ϵ	Difference in successive g() scores after which to stop optimizing thresholds
$n_{iterations}$	maximum number of iterations to run in each tuning step
ℓ_{t_1}	list of threshold combinations for (t_1^1,t_1^2)
ℓ_{t_2}	list of threshold combinations for (t_2^1,t_2^2)
ℓ_{t_3}	list of threshold combinations for (t_3^1,t_3^2)
ℓ_{t_4}	list of threshold combinations for (t_4^1,t_4^2)
ℓ_{t_5}	list of threshold combinations for (t_5)
T	set of thresholds $\{t_1^1,t_1^2,t_2^1,t_2^2,t_3^1,t_3^2,t_4^1,t_4^2,t_5\}$
ℓ_T	list of threshold sets
$e_1()$	simple enumeration function
α	Minimum probability used in selecting data
λ	weight between data used for training and tuning
$S_{received}$	Samples received by decision system
$P_{received}$	predicted probability for received samples produced by decision function
$Y_{received}$	predicted labels for received samples produced by decision function
$S_{selected}$	Samples selected for updated the system
S_{Train}	Samples selected for updated the system
$S_{selected}$	Samples selected for updated the system
S_{Train}	Samples used to train classifiers
$S_{TrainPrimary}$	Samples used to train primary classifiers
$S_{TrainSecondary}$	Samples used to train secondary classifiers
S_{Tune}	Samples used to tune the decision function

Table 3.2: Notation and Definitions used in the SMDA algorithms table (2/2)

3.1 Active Defender System

As shown in Figure 3-1, Active Defender utilizes the SMDA framework to maintain high accuracy while reducing classification time and resource usage. In subsequent chapters, we describe the SMDA abstractions in detail and how they are used in active defender.

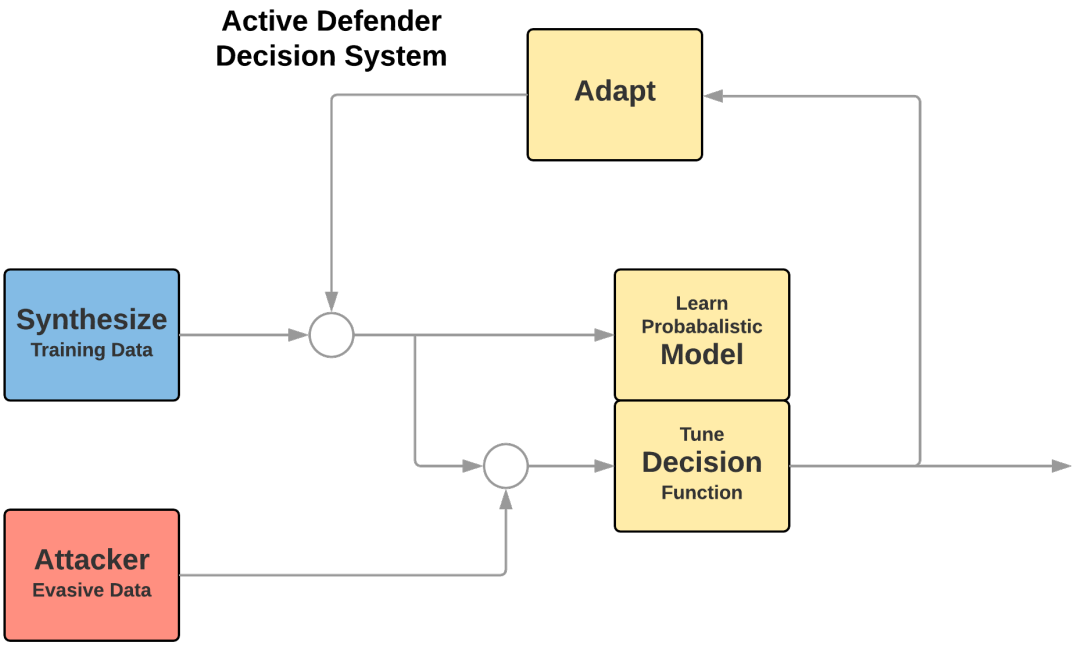

Figure 3-1: Active Defender Sytem Design: The Active Defender system uses the SMDA framework design and deploy a system to build and deploy a decision system. First the system is initialized by *Synthesizing* training data, learning a probabilistic *model*, and tuning the *decision function*. After the system is deployed it is used to *decide* on new data including evasive data generated by the attacker. After a decision is made on newly received data, the system *adapts* to update the *model* and *decision* function

Chapter 4

Synthesizing training data

To develop a adaptive machine learning solution, we need labeled training examples from past. For any given malware type - PDF or others - one may start with some examples made available from the past. Some are either available publicly or provided privately by security teams. Additionally, security teams can create malware directly to test their own defenses. In recent years, machine learning has been used to automatically create malware samples. In this chapter we describe multiple methods used to create malware samples and devise a strategy of our own. Our focus is still on PDFs.

4.1 Machine learning to create malicious samples

Beyond the direct methods used to inject malicious code and create PDF files, attackers can mutate of existing PDF malware in order to avoid detection. Automatic creation of malicious samples can be approached in two ways (i)*supervised* and (ii) *unsupervised*. In the *supervised* method the focus is on creating samples that are able to evade a detector - typically a machine learning classifier itself. A second approach, is *unsupervised* which attempts to create samples that are distant in feature space but are still malicious. In the next subsection, we present the supervised methods.

In the case of PDF Malware, attackers can create malware by either building malicious seeds through injecting javascript into a PDF file, or automated generation

Name	Description
n_v	Desired number of files to generate
y	desired label to generate data
S_m	set of malicious samples
S_b	set of benign samples
$o()$	oracle function that indicates if malicious functionality in tact
$S_{mutants}$	set of mutant files
$S_m^m utant$	set of malicious mutant files
S_{evade}	set of malicious samples
p_1	output of C_1
p_3	output of C_3
P_1	set of p_1 scores
P_3	set of p_3 scores
$cutoff$	cutoff specifying the minimum fitness function score for "evasive" files
$f_{mutate}(S_m, S_b)$	function that creates mutations using a set malicious and benign files

Table 4.1: Notation and Definitions used in the *Synthesize* algorithms.

of mutants from existing malicious seeds. Many features in PDF malware are used used manipulate the presented features of a file without modifying the underlying functionality.

In this case, there are two types of Synthesize functions used. The first type *Initial Sample Synthesis*. This could be done by loading a folder of files and generating labels.

4.1.1 Methods to evade classifiers

Many recent studies have focused on methods for generating adversarial PDF files to evade machine learning classifiers. In almost all cases, these methods rely on feedback from the classifier - that they are trying to evade - to create new variants. Hence we categorize them as *supervised* methods.

The mimicus framework presents a method to manipulate PDF classification using mimicry attack through modifying mutable features and through gradient descent methods using attributes of the model [6, 19]. The EvadeML framework presents a blackbox genetic programming approach to evade a classifier when the classification score is known [39]. The EvadeHC method, evades machine learning classifiers that evades classifiers without knowledge of the model or classification score [14]. The

SeedExploreExploit framework presents another evasion method for deceiving black box classifiers by allowing adversaries to prioritize level diversity and accuracy to generate samples [28].

Other methods operate on the feature space and generate evasive features that could confuse classifiers, however it often unclear how to convert evasive features back into a malicious file [17].

Many other methods have been presented to deceive machine learning classifiers based on the stationarity assumption not holding in an adversarial environment [13, 27, 14, 25, 16, 35, 13, 9, 8, 18, 7]

These attacks often focus on complex classifiers like deep learning systems, where classifiers can be over fit to rely on features that are correlated with malware rather than those necessary for malware. In [38], Wang et al showed that complex classifiers are able to be evaded with the presence of even one unnecessary feature.

EvadeML

Of all the methods under this paradigm of creating evasive variants, we focus on EvadeML. The authors of EvadeML have made their software open source. EvadeML uses a genetic programming method to produce tree-structure variants of malicious seeds - to evade static classifiers such as Hidost and PDFRate. These variants are then tested against the Cuckoo sandbox to ensure maintained malicious activity then scored using static classification scores [39]. With these parameters, EvadeML was able to achieve was able find variants that recieved classification scores of < 0 with PDFRate classification scores in range -1 (benign) to 1 (malicious) for all 500 malicious seeds. This indicates it was successfully able to confuse the PDFRate classifier. The algorithm works as follows:

Step 0 : Start with an empty set $S_{evade} = \{\}$

Step 1 : Create a set of mutant files using f_m using the set of S_m. Call this set $S_{mutants}$.

Step 2 : Check which among the mutants are malicious using the oracle function $o()$. In our case this is the Cuckoo classifier. Call this set S^m_{mutant}

Step 3 : Apply PDFRate classifier to the set S^m_{mutant} and generate classification scores.

Step 4 : Select the mutants that have classification scores greater than the *cutoff*. Add these to the set S_{evade}. These files represent the ones that are able to evade the PDFrate classifier.

Step 5 : Repeat steps 1 -4 until $|S_{evade}| \geq n_v$.

Max-Diff algorithm

We propose the *Max-Diff* algorithm as an alternative way to generate malicious files. The *Max-Diff* algorithm is similar to the EvadeML algorithm in that it uses a malicious and benign pool of variants, scores the malicious variants, mutates mutates the best scoing variants, adds them to the pool of malicious files and continues. However, unlike the Evade-ML algorithm, it does not seek to find files that receives a classification score less than the $cutoff$ for a single classifier. Instead, it selects for files that receive different classification scores with different classifiers in the system. In the case of Active Defender system, Max-Diff targets files that evade PDFRate or Virus Total.

The algorithm works are follows:

Step 0 : Start with an empty set $S_{evade} = \{\}$

Step 1 : Create a set of mutant files using f_m using the set of S_m. Call this set $S_{mutants}$.

Step 2 : Check which among the mutants are malicious using the oracle function $o()$. In our case this is the Cuckoo classifier. Call this set S^m_{mutant}

Step 3 : Apply PDFRate classifier to the set S^m_{mutant} and generate classification scores. Collect these scores in P_1

Step 4: Upload the set to the Virus Total website and generate virus total classifier scores for S^m_{mutant} and generate classification scores. Collect these scores in P_3

Step 5 : Select the mutants that have $(p_1 - p_3)$ greater than a threshold specified by *cutoff*. Add these to the set S_{evade}. These files represent files that different scores between PDFRate and Virus Total and could confuse a classification system.

Step 6 : Repeat steps 1 -4 until $|S_{evade}| \geq n_v$.

4.1.2 $f_{mutate}()$

Evasive algorithms require a mutation function, $f_{mutate}()$, that creates variants of malicious files. The mutation function requires a pool malicious files S_m and a pool of benign files. The malicious files are mutated using components from the pool of benign files S_b.

The mutation function implemented using a modified version of the PDFRW software package[1], works as follows:

Step 1 : Load all PDF files are loaded into a tree structure.

Step 2 : Mutate each malicious PDF by randomly selecting one of the following methods:

- Insert randomly selected sub-tree from a randomly selected sub-tree from the benign file.
- Swap a randomly selected sub element with randomly selected sub-tree from benign file.
- Delete a randomly selected element in the malicious tree representation.

Step 3 : Write tree representation of mutated malicious files to PDF files.

4.1.3 Evasive Performance

In analyzing these algorithms, we characterize their performance on PDFRate – the fastest classifier and Virus Total the most accurate classifier. As we see in Figure 4-1 in this case malicious files generated using the *Evade-ML*, are effective in evading

[1]https://github.com/mzweilin/pdfrwructure

classification. As shown in Figure 4-2, we observe that evasive files generated using the *Max-Diff* algorithm are especially effective at evading the Virus Total classifier and achieve the same scores as benign files. In comparing these results we see that the more time consuming classifier, Virus Total, does achieve higher accuracy against evasive variants than the PDFRate classifier. However, even Virus Total is not foolproof which motivates the need for use of human analysts.

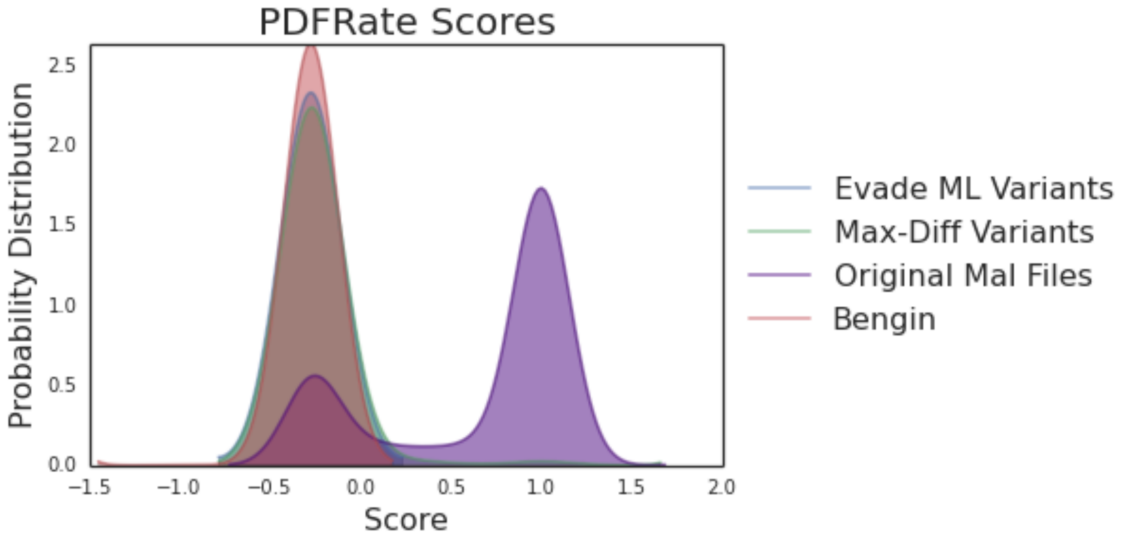

Figure 4-1: KDE approximation of Probability Density for the PDFRate scores. This plot shows the classification scores for different types of files. The Benign files are shown as pink, the Contagio malware samples are shown in purple, the EvadeML variants are shown in blue and the Max-Diff variants are shown in green. The KDE plot was generated with Gaussian kernel of width 0.15, and 0.15 , 0.17,.17, for the Benign, Contagio, EvadeML, and Max-Diff files respectively. In this case the EvadeML, Max-Diff, and Benign files had very similar probability densities. In order to differentiate them, differing width kernels were used.

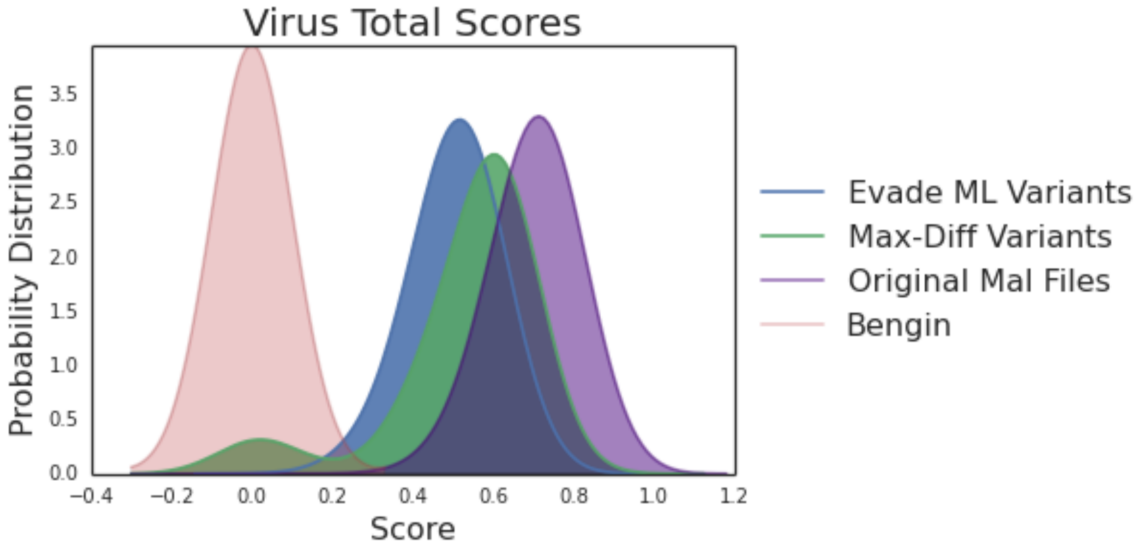

Figure 4-2: KDE approximation of Probability Density for percent of Virus Total-Engines classifiers that classify a file as malicious. This plot shows the classification scores for different types of files. The Benign files are shown as pink, the Contagio malware samples are shown in purple, the EvadeML variants are shown in blue and the Max-Diff variants are shown in green. The KDE plot was generated with Gaussian kernel of width 0.15, and 0.15 , 0.17,.17, for the Benign, Contagio, EvadeML, and Max-Diff files respectively.

Chapter 5

Learning models from training data

In our *active defender* system, we use the training data provided to us in the form of S_m and S_b files to train multiple models. We divide the classifiers into two types *primary* and *secondary*. Primary classifiers/models take the files as input and produce a probabilistic score p. Secondary classifiers/models take output of the primary classifiers and deliver a probabilistic score. All secondary models are machine learning models, whereas not all primary models are. These models enable us to develop an incremental decision system as we will describe in Chapter 6 that in turn allows us to trade off between accuracy and resources used.

5.1 Primary classifiers

The *active defender* system uses the the Model framework to call the PDFRate, Cuckoo, and Virus Total classifiers. For simplicity we describe *samples* as an array of file_paths and labels as array of 1 for malicious and 0 for benign.

5.1.1 PDFRate (C_1)

The PDFRate classifier takes labeled filenames as inputs and extracts static features of the file as discussed in Chapter 2 and Appendix A. A random forest machine learning model is then trained on the set of features and labels.

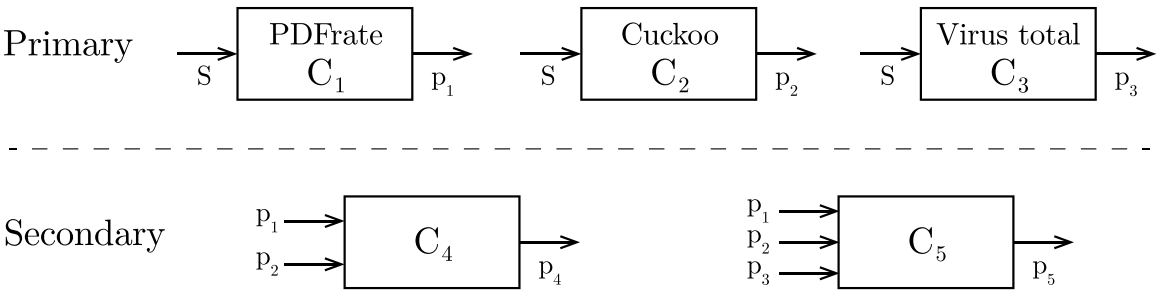

Figure 5-1: Active Defender Classifiers: In active primary classifiers (C_1,C_2,C_3) receive samples and produce probabilistic scores (p_1,p_2,p_3). Secondary classifiers operate off of probabilistic scores as inputs. Secondary classifier C_4 uses inputs (p_1,p_2) to produce the probabilistic score p_4. Secondary classifier C_4 uses inputs (p_1,p_2,p_3) to produce the probabilistic score p_5.

5.1.2 Cuckoo (C_2)

The Cuckoo classifier (C_2) does not require feature extraction or training. When the Cuckoo model is used, it sends files to a running Cuckoo server that accepts files and returns scores p_2 indicating if known behavioural signatures of malicious files were detected .

5.1.3 Virus Total (C_3)

Similarly, Virus Total (C_3) does not require feature extraction or training. When the Virus Total model is used, it uploads files to the Virus Total API and outputs the percent Virus Total classifiers that classify the file as malicious (p_5) as described in Chapter 2 and Appendix B

5.2 Secondary Classifiers

Secondary classifiers are designed taking in output score from the primary classifiers and learning a machine learning model. Two secondary classifiers are developed in our *active defender* system. They are:

– C_4 uses the outputs of PDFRate (C_1) and Cuckoo (C_2) as inputs and produces a probabilistic score (p_4).

Name	Description
S	Samples
S_m	Malicious samples
S_b	Benign samples
C	Classifier
p	Individual probabilistic score
C_1	PDFRate (primary) classifier
p_1	Output of C_1
C_2	Cuckoo (primary) classifier
p_2	Output of C_2
C_3	Virus Total (primary) classifier
p_3	Output of C_3
C_4	Secondary classifier using p_1 and p_2 as inputs
p_4	Output of C_4
C_5	Secondary classifier using p_1, p_2, and p_3 as inputs
p_5	Output of C_5

Table 5.1: Notation and Definitions used in the *Model* algorithms.

– C_5 uses the outputs of PDFRate (C_1), Cuckoo (C_2), and Virus Total (C_3) as inputs and produces a probabilistic score (p_5).

Chapter 6

The decision system

In the previous chapter, we presented multiple classifiers that we can train using the data available to us. In real time, in order to determine whether or not a new input file s is malicious, we apply a hierarchical decision system that makes use of multiple classifiers. In this system, we use three primary classifiers. Classifier C_1, PDFRate, is the cheapest of all in terms of the computational time required to make a decision, but is also the most inaccurate and could be evaded easily. The Cuckoo classifier, C_2, requires the dynamic analysis of the file, and thus needs more time than C_1. VirusTotal, C_3, requires us to use their `API`, and it takes about 2 minutes on an average to receive the scores back. Among the three, Virus Total is the most accurate. For these reasons, in developing a decision system, we considered the following goals:

- Increase throughput: We would like to make decisions for PDFs as fast as possible. Because PDFRate is the fastest and Virus Total is the slowest in giving us the prediction, we would like to use PDFRate to make a decision for as many cases as possible.

- Maintain accuracy: While it is easiest to increase throughput by choosing to use the PDFRate classifier every time, this will lead to a lot of false positives if we have to maintain a high recall for detection of malware (see Chapter 2). To maintain a recall of 90% or higher, we would have to augment and use Cuckoo or VirusTotal.

To achieve the goals above, we propose the following:

- a bi-level decision function for classifiers described in section 6.1,
- a hierarchical, tunable decision system, described in section 6.2,
- A cost function that evaluates the efficacy of a given decision system, described in section 6.3,
- a tuning algorithm that produces a decision system that can be used, described in section 6.4.

6.1 Bi-level decision function

Given a classifier C_i and its output score p_i, a bi-level decision function allows us to make a decision, D_i based on two decision thresholds, t_i^1 and t_i^2,as depicted in Figure 6-1 and more formally given by:

$$D_i \begin{cases} \text{Benign} & \text{if } p_i < t_i^1 \\ \text{Uncertain, output p}_i & \text{if } p_i \geq t_i^1 \text{ and } p_i < t_i^2 \\ \text{Malicious} & \text{if } p_i \geq t_i^2 \end{cases} \tag{6.1}$$

This allows us to make a decision when we are absolutely confident, and enables us to postpone the decision in a region where we are uncertain.

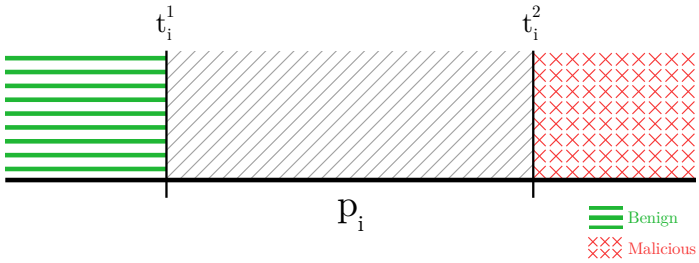

Figure 6-1: Bi-level decision function. Using an input score of p_i, the bi-level decision returns a result if it is certain of the classification. It classifies an input as benign if $p_i < t_i^1$ and malicious if $p_i \geq t_i^2$
. If $t_i^1 < p_i < t_i^2$ the decision function returns p_i as it is uncertain of the result.

6.2 Hierarchical tunable decision system

The hierarchical decision system is shown in Figure 6-2 and formally in Algorithm 1. This system determines a final classification result (y) and a probabilistic score (P_{final}) for each input sample using layers of $bi-level$ classifiers.

The P_{final} score is calculated using the output of the last classifier (p_{last}), the threshold used in the last decision (t_{last}) , the number of primary classifiers used (N_{pc_used}), and the total available primary classifiers (N_{pc_total}) classifiers as shown below:

$$P_{final} = \frac{N_{pc\_used}}{N_{pc\_total}} * (p_{last} - t_{last}) \tag{6.2}$$

6.3 Cost function

The cost function expresses the two objectives we specified above – whether the desired accuracy is achieved and the throughput. Given a fully specified decision system, with classifiers $C_{1...5}$, decision thresholds $t_1^1, t_1^2, t_4^1, t_4^2, t_5$, and a set of files S, the cost,c incurred by the system is evaluated as:

$$c = -\gamma * g(\hat{Y}, Y) + (1-\gamma) * \frac{1}{|S|} \sum_i r_i \tag{6.3}$$

where $\hat{Y}$ is the predicted labels, Y are the corresponding true labels, $g(.)$ measures the accuracy of the predicted labels, and $\frac{1}{|S|}\sum_i r_i$ is the average classification time taken to make these decisions based on the subset of models used for each file in the set per sample, and γ is a weight associated with each of the factors.

6.3.1 $g(.)$ function

The $g(.)$ function describes the accuracy of a system. We provide two methods of characterizing system accuracy.

Algorithm 1 ActiveDefender.Decide(S,$t_1^1, t_1^2, t_2^1, t_2^2, t_3^1, t_3^2, t_4^1, t_4^2, t_5, C_1, C_2, C_3, C_4, C_5$)

$\hat{Y} \leftarrow \{\}$
$P_{final} \leftarrow \{\}$
for $s \in S$ **do**
  1. $p_1 \leftarrow C_1(s)$
  if $p_1 < t_1^1$ **then**
    $P_{final} \leftarrow (t_1^1 - p_1) * \frac{1}{3}$
    $\hat{Y} \leftarrow \{\hat{Y}|False\}$
  else if $p_1 > t_1^2$ **then**
    $P_{final} \leftarrow (p_1 - t_1^2) * \frac{1}{3}$
    $\hat{Y} \leftarrow \{\hat{Y}|True\}$
  else
    2. $p_2 \leftarrow C_2(s)$
    3. $p_4 \leftarrow C_4(p_1, p_2)$
    if $p_4 < t_4^1$ **then**
      $P_{final} \leftarrow (t_4^1 - p_4) * \frac{2}{3}$
      $\hat{Y} \leftarrow \{\hat{Y}|False\}$
    else if $p_4 > t_4^2$ **then**
      $P_{final} \leftarrow (p_4 - t_4^2) * \frac{2}{3}$
      $\hat{Y} \leftarrow \{\hat{Y}|True\}$
    else
      4. $p_3 \leftarrow C_3(s)$
      5. $p_5 \leftarrow C_5(p_1, p_2, p_3)$
      if $p_5 < t_5^1$ **then**
        $P_{final} \leftarrow (p_5 - t_5) * \frac{3}{3}$
        $\hat{Y} \leftarrow \{\hat{Y}|False\}$
      else
        $P_{final} \leftarrow (p_5 - t_5) * \frac{3}{3}$
        $\hat{Y} \leftarrow \{\hat{Y}|True\}$
      end if
    end if
  end if
end for
return $< \hat{Y}, P_{final} >$

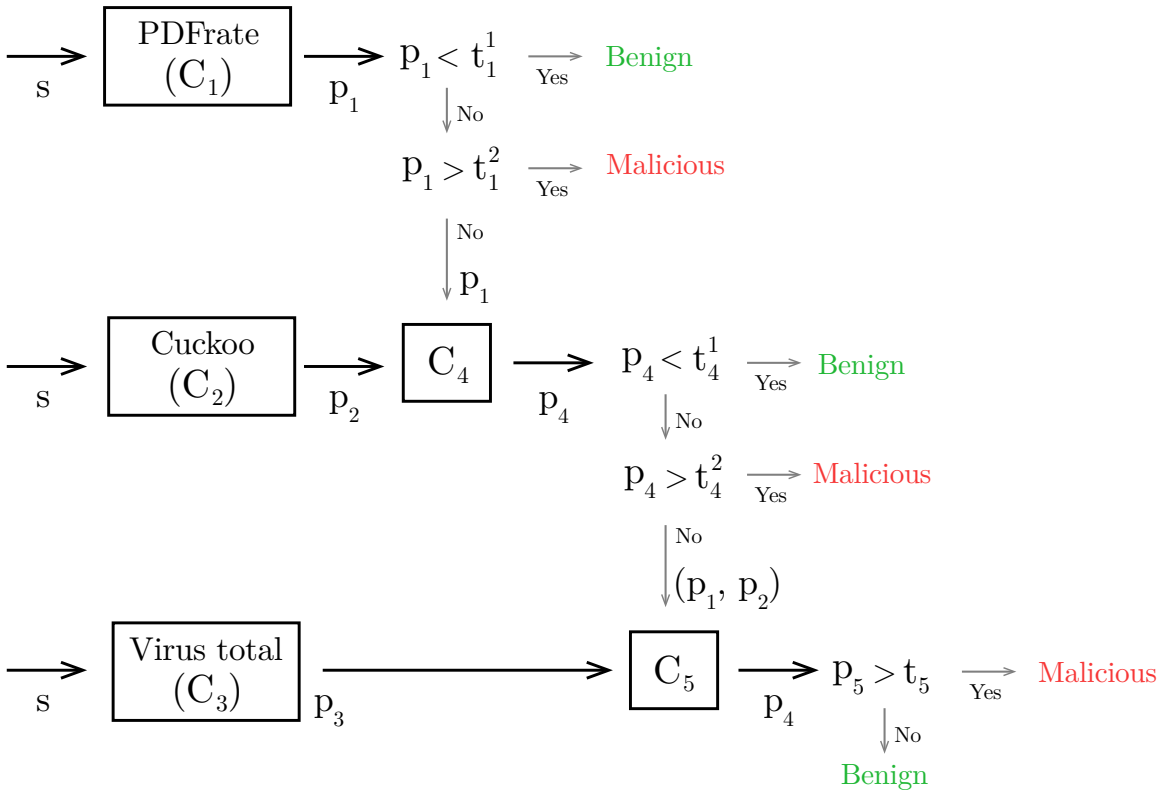

Figure 6-2: Active Defender Hierarchical Decision Algorithm: A PDF is first sent to the PDFRate classifier (C_1). Based on the output of PDFrate, p_1, a decision is made whether to return a result or send the file to the Cuckoo classifier (C_2). If the file is sent to the Cuckoo classifier, the results from PDFRate (p_1), and Cuckoo (p_2) are sent to the secondary classifier C_4 and a decision is made as to whether to return a result or sent the file to VirusTotal (C_3). If the file is sent to the VirusTotal classifier, classification scores from the PDFRate (p_1), Cuckoo (p_2), and VirusTotal (p_3) classifiers are sent to the C_5 secondary classifier and a final decision is made.

In $g_1(.)$ the f1 score is optimized to improve precision and recall equally.

$$g_1(.) = f_1(predicted, true\_labels) \tag{6.4}$$

In $g_2(.)$ the function requires a minimal threshold of precision and then optimizes for recall. This function is especially applicable for malware detection as allowing an additional malicious file to enter the system can be very costly, but is required to

keep false rejection of benign files below a certain specified rate for user happiness.

$$g_2 = \begin{cases} \text{recall}(\hat{Y}, Y) & \text{if precision}(\hat{Y}, Y) \geq 0.9 \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

6.4 Tuning algorithm

The tuning algorithm uses additional data to optimize the decision function using a cost function. Since the *active defender* system utilizes a set of thresholds to determine the decision for an input sample as shown in Algorithm 1, *tune* optimizes these thresholds based on their effect on a cost function.

Tune comprises two main steps. First, the tune algorithm enumerates an initial set of classifier thresholds using an enumeration function $e()$ to generate a set of thresholds T, and scores them with the cost function c. Enumerating a large threshold set is important in systems with complex costs functions such as $g_2(.)$ which are not monotonic. If too few initial thresholds are enumerated, optimization can result in thresholds that find a local rather than global minimum cost function value.

Second, *tune* uses a maximum of $n_{iterations}$ of Bayesian hyper-parameter tuning to propose an additional candidate threshold, evaluate it using c, add it to the threshold set T, and find the thresholds that minimize the cost function c. In iterative tuning, ϵ specifies the minimum distance between successive minimum scores to stop optimization [33, 10].

Bayesian optimization allows for the optimization of a black-box cost function using a set of tunable parameters. In our system, the tunable parameters for the decision system are the lower thresholds in each set and the difference between the lower and upper thresholds (which is fixed to 0 for the last threshold set).

6.4.1 $e()$

The enumeration function $e()$ produces a list of threshold sets necessary to minimize the cost function. This is done in two steps.

- First, we produce a 4 list of possible threshold pairs for each pair of thresholds: ℓ_{t_1},ℓ_{t_2},ℓ_{t_3},ℓ_{t_4} (t_1^1,t_1^2),(t_2^1,t_2^2),(t_3^1,t_3^2), and (t_4^1,t_4^2) respectively.

- For the last threshold t_5 we produce a single list of possible thresholds ℓ_{t_5}.

- Finally, we create ℓ_T using all possible combinations of threshold pairs across lists ℓ_{t_1},ℓ_{t_2},ℓ_{t_3},ℓ_{t_4} and ℓ_{t_5}.

We propose a simple enumeration function $e_1()$. The enumeration function $e_1()$ produces threshold pairs using the 0%, 20%, 40%, 60%, 80%, and 100% percentile values of previous classification scores for that classifier. For example, if previous PDFRate classification scores p_1 were observed between 0.0 and 0.5, then: $\ell_{t_1} \equiv \{(0.0, 0.1), (0.1, 0.2), (0.3, 0.4), (0.4, 0.5)\}$. The last threshold list is (ℓ_{t_5}) a list of the 0% , 20%, 40%, 60%, 80%, and 100% percentiles for respective score p_5.

More complex enumeration functions can be developed to capture a more expressive range of thresholds.

Name	Description
S	Samples
$\|S\|$	Number of samples
s	Individual sample
C	Classifier
Y	True Labels
$\hat{Y}$	Predicted Labels
P	Set of probabilities
p	Individual probabilistic score
C_1	PDFRate (primary) classifier
p_1	Output of C_1
C_2	Cuckoo (primary) classifier
p_2	Output of C_2
C_3	Virus Total (primary) classifier
p_4	Output of C_3
C_4	Secondary classifier using p_1 and p_2 as inputs
p_4	Output of C_4
C_5	Secondary classifier using p_1, p_2, and p_3 as inputs
p_5	Output of C_5
p_{last}	Output of the last classifier used in making a decision
D_i	Decision function
t_i^1	Lower threshold for probability score i
t_i^2	Upper threshold for probability score i
P_{final}	Output probability score of *active defender* system
t_{last}	Last threshold used in making a decision
N_{pc_used}	Number of primary classifiers used in classifying a file
N_{pc_total}	Number of total classifiers used
γ	value of accuracy score vs. time
$\sum_i r_i$	Total time taken to make decision per file
$\frac{1}{\|S\|}\sum_i r_i$	Average time taken to make decision per file
recall()	Recall
precision()	Precision
f_1	function that computes the f1 score
β	$precision\_recall\_weight$ – weight of precision vs recall
$g(.)$	Function describing accuracy of a system
g_1	Specific evaluation function maximizing f_1 score.
g_2	Specific evaluation function maximizing recall given precision above 0.9
c	Cost function
$e()$	Enumeration function that generates initial threshold sets to evaluate
ϵ	Small value specifying distance in successive g() scores after which to stop optimizing thresholds
$n_{iterations}$	maximum number of iterations to run in each tuning step

Table 6.1: Notation and Definitions used in the *Decide* algorithms (1/2)

Name	Description
$n_{iterations}$	maximum number of iterations to run in each tuning step
ℓ_{t_1}	list of threshold combinations for (t_1^1,t_1^2)
ℓ_{t_2}	list of threshold combinations for (t_2^1,t_2^2)
ℓ_{t_3}	list of threshold combinations for (t_3^1,t_3^2)
ℓ_{t_4}	list of threshold combinations for (t_4^1,t_4^2)
ℓ_{t_5}	list of threshold combinations for (t_5)
T	set of thresholds $\{t_1^1,t_1^2,t_2^1,t_2^2,t_3^1,t_3^2,t_4^1,t_4^2,t_5\}$
ℓ_T	list of threshold sets
$e_1()$	simple enumeration function

Table 6.2: Notation and Definitions used in the *Decide* algorithms (2/2)

Chapter 7

Adapting over time

In the *active defender* system one of the important aspects is to incorporate adaptation of the entire system over time. A lot of work has been done to show how motivated attackers can build evasive variants. It is natural to ask how would the defense mechanisms adapt as new variants are produced. Known as active learning, this adaptation can happen over time by simply adding training examples whose labels are *verified*.

In [37], Veeramachaneni and Arnaldo study the use of Active Learning in a human in the loop detection system. Using multiple outlier detection systems to send suspicious data to analysts the system is able to improve the machine learning model - over time. Building on the idea of sending data that is classified with some uncertainty by a faster more cost-effective model to a more expensive, but accurate analyst, we expand this method to use a variety of possible ways to generate more training data. Our new training examples come from the following ways:

- higher accuracy classifiers: we can incorporate predictions from Virus Total as possible source of truth and incorporate them as training examples.
- human analysts: we can send some examples to humans to get their analysis. This is an expensive mechanism but still doable.
- synthetically generated evasive variants: From time to time, we can create evasive variants using the machine learning methods we described in Chapter 4.

For these evasive variants we know the ground truth and they can provide training examples for our system.

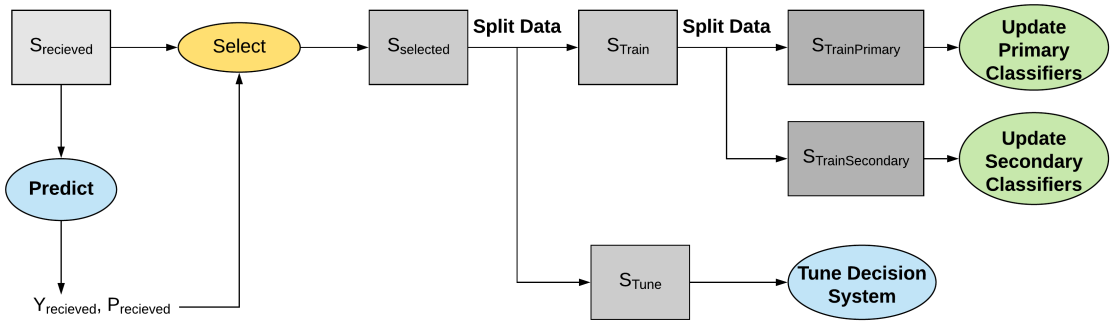

Figure 7-1: Diagram of the adapt system. 1) Input data $S_{received}$ sent through the decision system to produce predicted labels $Y_{received}$ and probabilities probablities . 2) Samples are selected in using probabilities $P_{received}$ 3) The selected data $S_{selected}$ is split into S_{train} and S_{tune} 4) The training data S_{train} is split into $S_{primary}$ used to train the update the primary classifiers and $S_{secondary}$ used to update the secondary classifiers 5) The tuning data S_{tune} is used to Tune the decision system

7.1 Adapt in Active Defender

In the active experiment, we use additional data to *update* the models and *tune* the decision system. The system can be adapted using synthetic data or unlabeled data. In the case of unlabeled data, the system generates labels and final probabilities using the predictions from the previous learned models and the decision system. The adapt algorithm uses the following steps as shown in Figure 7-1.

– Select: chooses the data that is is above a set minimum probability threshold (α) to be used to update the system

– Update: uses a fraction of the selected data specified by (λ) to learned model. This data is split again into data used to train the primary and secondary classifiers, specified by parameter μ. In the *active defender* system, the $PDFRate$ classifier (C_1) is the only one of the primary classifiers that can be retrained to utilize additional data.

The secondary training data is appended to additional secondary training data and the secondary classifiers, C_4, C_5 are updated using the new predictions of PDFRate for the labeled data.

– Tune: uses the remaining data to tune the decision function according to a specified enumeration function,$e()$, maximum number of tuning iterations ($n_{iterations}$), and difference between successive minimum scores ϵ.

Name	Description
α	Minimum probability used in selecting data
λ	weight between data used for training and tuning
μ	weight between training data used for primary and secondary classifiers
$S_{received}$	Samples received by decision system
$P_{received}$	predicted probability for received samples produced by decision function
$Y_{received}$	predicted labels for received samples produced by decision function
$S_{selected}$	Samples selected for updated the system
S_{Train}	Samples selected for updated the system
$S_{selected}$	Samples selected for updated the system
S_{Train}	Samples used to train classifiers
$S_{TrainPrimary}$	Samples used to train primary classifiers
$S_{TrainSecondary}$	Samples used to train secondary classifiers
S_{Tune}	Samples used to tune the decision function
C	Classifier
p	Individual probabilistic score
C_1	PDFRate (primary) classifier
p_1	Output of C_1
C_2	Cuckoo (primary) classifier
p_2	Output of C_2
C_3	Virus Total (primary) classifier
p_3	Output of C_3
C_4	Secondary classifier using p_1 and p_2 as inputs
p_4	Output of C_4
C_5	Secondary classifier using p_1, p_2, and p_3 as inputs
c	Cost function
$e()$	Enumeration function that generates initial threshold sets to evaluate
ϵ	Difference in successive g() scores after which to stop optimizing thresholds
$n_{iterations}$	maximum number of iterations to run in each tuning step

Table 7.1: Notation and Definitions used in the *Adapt* algorithms.

Chapter 8

Experimental Results

In order to understand the performance of the *active defender* system, we analyze its accuracy and resource usage as it adapts.

In the experimental design, we first split the data into two data sets, as shown in Figure 8-1. D_1 corresponds to data used to train the system, and D_2 is data received by the system after it is deployed.

Training Data:

D_1 is the training data available to the system before it is deployed. In our experimental setup, D_1 consists of the 10,597 Contagio malware files and 10,597 benign PDFs randomly selected from the 44,680 benign files discussed in Section 2.3. This training data consists of malicious files collected by security analysts and a corpus of collected benign PDF files.

Adaptation Data:

The adaptation data, D_2, consists of the evasively generated malware and remaining benign PDF files Figure 8-1. As shown in Figure 8-1, this data is split into subsets $q1$ through $q5$ and is sent to the decision system across 5 stages or time periods.

8.1 Experimental setup

In setting up the experiment, we perform 25 random trials. For both D_1 and D_2, the order of the files is randomized across trials, so splitting gives different subsets q_1 ...

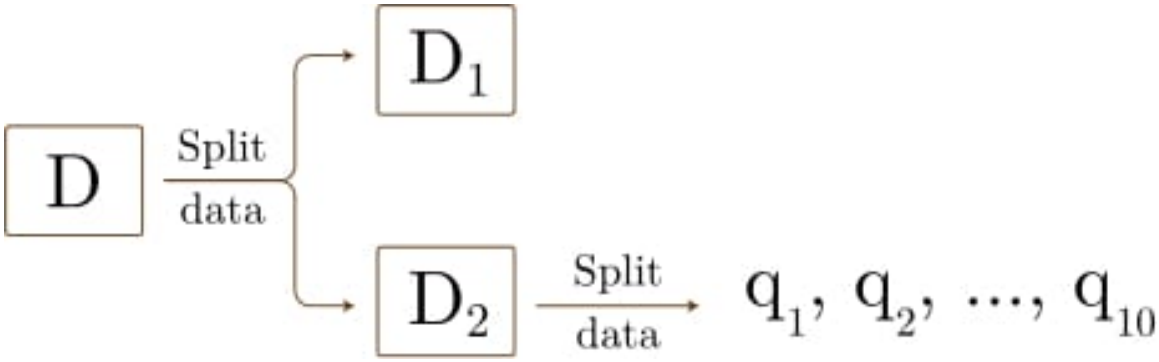

Figure 8-1: Splitting Experimental Data. In the following experiment the data is split into data sets D_1 and D_2. D_1 is used to initialize the decision system. D_2 represents data received by the system after it is deployed. D_2 is split into subsets q_i, representing the files received in each successive stage.

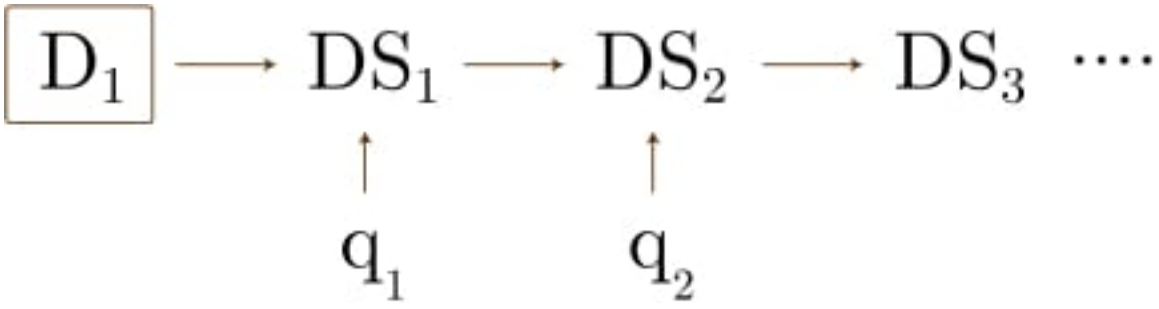

Figure 8-2: Updating the decision system. In the experiment, training data D_1 is used to initialize the decision system. After the system is deployed, it received additional data. After each additional received dataset q_i, the decision system *adapts*.

q_5.

In setting up the decision system, we set the following tuning parameters, as described in Chapter 6 and in Chapter 7.

The cost function is set up as described in Chapter 6, using the $g_1()$ function and a γ value of 0.9 to prioritize accuracy over resource constraints.

The tuning parameters use $e_1()$ as the threshold enumeration function and an epsilon value of $\epsilon \equiv 0.1$ specifying successive minimum cost scores.

8.2 Experimental Results

Overall, we see that the system is able to adapt to achieve high accuracy in the presence of evasive adversaries, and to reduce resource usage over time.

Accuracy

As shown in Figure 8-3 and Table 8.1, we observe the performance of the decision system on classifying successive sets of received files. We characterize accuracy by observing the f_1 score, comparing truth versus labeled data. As evasive variants are introduced in stage 1, we observe a low f_1 score. However, as stages progress, we observe that the system is able to adapt to improve accuracy over time.

Resource Usage

In this experiment, we characterize the resource usage by studying the average time used to classify each file. As shown in Figure 8-4 and Table 8.1, when the system is initialized, classification time is relatively low, at around 1(s) per file. However, we observe that the classification time continues to decrease over time, indicating that the PDFRate static classifier is improving and being utilized. In calculating the estimated classification time, we model the PDFRate as taking 1 second, Cuckoo as taking 25 seconds and VirusTotal as taking 90 seconds. Notably, the standard deviation in classification time is to small to observe using four decimals of precision. This is likely due to the majority of files being classified by the static classifier and our estimation function limiting the variability in time.

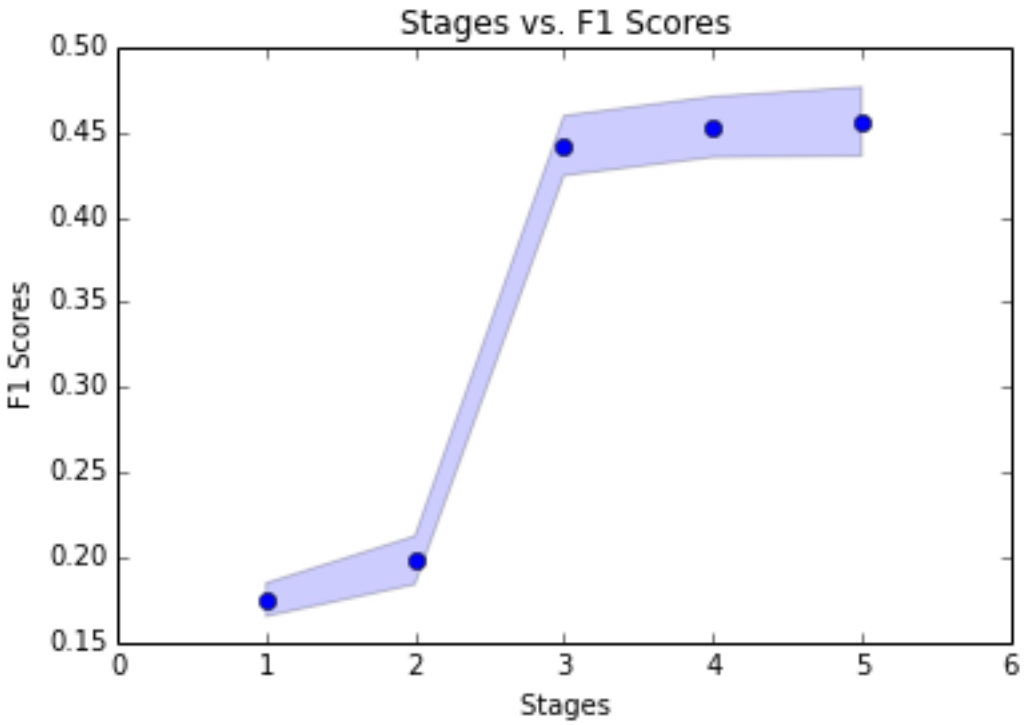

Figure 8-3: Active Defender Accuracy over Adaptation. In this figure, we observe the f_1 score vs. the experimental stage over time. We plot the mean f_1 score as points and show the standard deviation in the surrounding band. In this experiment, we observe the experiment achieving poor results in Stage 1 when evasive samples are introduced. Over time, we observe that the f_1 score increase over time as the system adapts to evasive samples.

Stage	μ_{f1}	σ_{f1}	$\mu_{TimeperFile}$	$\sigma_{TimeperFile}$
1	0.17535	0.01003	1.16908	<0.0001
2	0.19852	0.01459	1.16908	<0.0001
3	0.44201	0.01804	1.10766	<0.0001
4	0.45301	0.01829	1.10208	<0.0001
5	0.4562	0.02082	1.09649	<0.0001

Table 8.1: Experimental data 25 trials the Active Defender System performance over 5 stages. Column μ_{F1} corresponds to the average f_1 score across all trials. Column σ_{f1} corresponds to the standard deviation in f_1 score across all trials. Column $\mu_{TimeperFile}$ corresponds to the average estimated classification time per file. Column $\sigma_{TimeperFile}$ corresponds to the standard deviation in approximated average classification time per file.

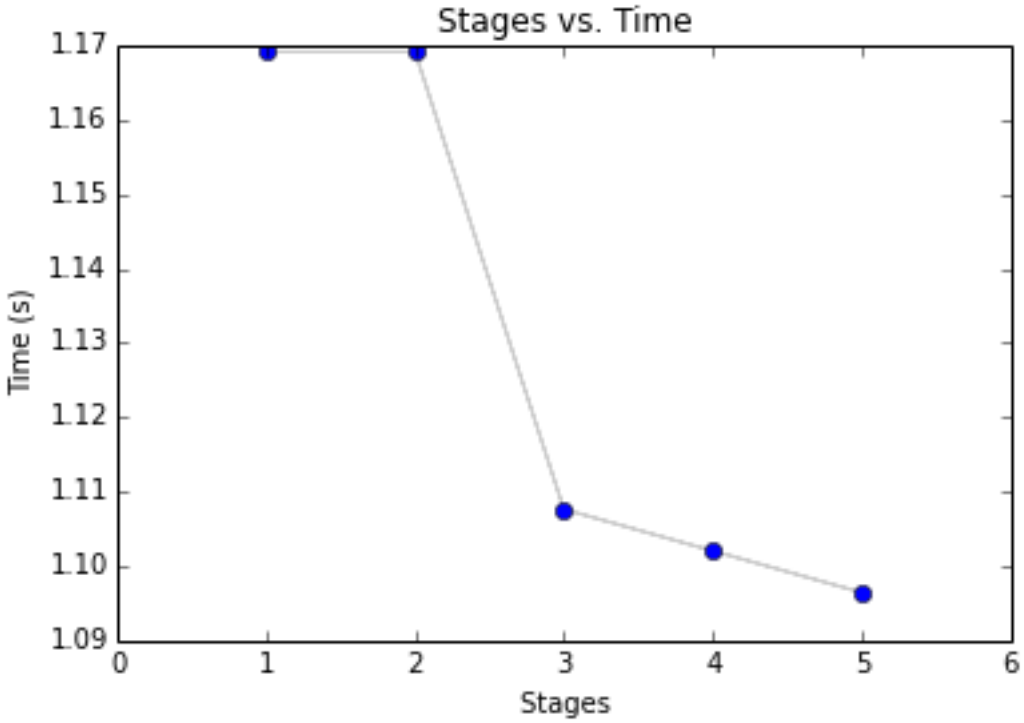

Figure 8-4: Active Defender Average Classification Time over Adaptation. In this figure we observe the estimated average classification time per file at each stage. We plot the mean time score as points and show the standard deviation in the surrounding band. Here we see that the average classification time is pretty low – around 1 second – throughout the course of the experiment, and decreases over time. In addition, the deviation in time is small across successive stages, and is not observable due to the estimation function.

Chapter 9

Discussion and Future Work

Through this project, we were able to make four contributions. First, we developed a method to use machine learning in cybersecurity in a resource-constrained environment

Second, we developed algorithms that use active learning to improve fast classifiers in the presence of adversaries.

Third, we provide an extensible framework to facilitate building, evaluating and deploying decision systems in an adversarial and resource-constrained environment.

Fourth, we provide a simple evasive algorithm that was shown to confuse automated classifiers.

Through studying the adversarial and resource-constrained problem of detecting evasive PDF malware and building these solutions, we identified a few takeaways that motivate future work.

9.1 Evasion

In studying the available classifiers, it was surprising to see that *max-diff* algorithm was effective in causing confusion in the Virus Total classifier. Virus Total is a powerful classification system that has been acquired by Google and was considered

to be one of the best products of 2007[1]. If this genetic programming-based algorithm can cause confusion in malicious and benign files, it suggests that adversaries are more than capable of deploying their own evasive algorithms to evade automated classifiers. This motivates the need for human-in-the-loop systems and systems that adapt over time.

9.2 Active Defender

9.2.1 Decision System

In studying the behaviour of the *active defender* decision system, we identified interesting aspects of the threshold decision method that impact performance, and identified areas of the decision system that future work could explore.

We saw that the choice of both the evaluation function and the number of classifiers used affect modeling performance.

In the choice of the decision function, we observed that non-continuous or complex evaluation functions need more initial thresholds to be enumerated, and thus require more time to adapt the decision system. When deploying these systems, defenders should explore tradeoffs between choice in evaluation function and tuning time.

Threshold / Adaptation scaling:
In the current implementation of the decision system, for n primary classifiers, there are $2n$ -1 thresholds. However, in tuning the thresholds, we enumerate the initial threshold set combinatorially before turning with Bayesian optimization. Future work could focus on reducing the set of initial thresholds necessary to tune in proportion to the number of classifiers used.

Using additional primary classifiers:
In analyzing methods to improve this system, we started with the primary classifiers used. In this thesis we focused on three primary methods of classification: static, dynamic, and API-based. Studying the integration of additional primary classifiers

[1]https://www.google.com/search?q=best+products+2007+virus+totaloq=best+products+2007+virus+totalaqs=chrome..69i57j69i64.3734j0j4sourceid=chromeie=UTF-8

(such as humans) or unsupervised methods would be an interesting next step. Alternatively, unsupervised anomaly detection could provide valuable insight into a changing dataset, and thus further improve the system.

Another possibility for improving the decision systems lies with randomization. Randomly selecting a small number of files to be sent to the most accurate classifiers can strengthen the system against files that can completely evade simple classifiers.

9.2.2 Adaptation

In this thesis, we discussed simple adaptation methods. However, additional methods can be used to improve adaptation. First, as the system adapts and identifies files that confuse fast classifiers and require the use of expensive ones, it could synthesize variants similar to these files, using automated evasive algorithms to retrain and improve static models. Alternatively, in adaptation a system could use optimization methods more targeted to specific use cases in an effort to improve performance.

9.2.3 Resource-Constrained Classification Systems

Thanks to the increasing amounts of data collected by enterprises, machine learning can be an asset to cybersecurity. However, each company or institution looking to defend their system will have different limitations on the amount of resources they can devote to analyzing data. The *active defender* system can be tailored to different resource limitations and different environments, using a variety of evaluation or cost functions.

9.2.4 Using Active Learning to Improve Against Evasive Samples

In adversarial environments that require cybersecurity, it is essential to be able to update to a changing data distribution. We showed how the Active Defender system provides a mechanism for updating results using higher accuracy as the adversaries evolve over time.

9.3 SMDA Framework

The SMDA framework provides a platform for bringing together innovative algorithms made by research scientists with developers who have the data to deploy these algorithms. Future work directions include improving the SMDA framework to generalize to adversarial use in the cybersecurity space as well as in other areas, such as detecting malicious bots on social media.

9.4 Conclusion

As motivated attackers use more and more computational resources and state-of-the-art algorithms to persistently attack smaller corporations, it is necessary to figure out how to automate detection in a resource-constrained environment. In this thesis, we built the SMDA framwork and the Active Defender classification system which perform well when faced with the hard problem of detecting evasive malware. Furthermore, we believe that this software framework and algorithms can generalize beyond PDF malware detection, enabling researchers and corporations to work together to secure systems against powerful and evolving adversaries.

Appendix A

PDFRate Classifier Features

Feature	Name	Feature	Name
pdfrate 0	author_dot	pdfrate 18	count_box_overlap
pdfrate 1	author_lc	pdfrate 19	count_endobj
pdfrate 2	author_len	pdfrate 20	count_endstream
pdfrate 3	author_mismatch	pdfrate 21	count_eof
pdfrate 4	author_num	pdfrate 22	count_font
pdfrate 5	author_oth	pdfrate 23	count_font_obs
pdfrate 6	author_uc	pdfrate 24	count_image_large
pdfrate 7	box_nonother_types	pdfrate 25	count_image_med
pdfrate 8	box_other_only	pdfrate 26	count_image_small
pdfrate 9	company_mismatch	pdfrate 27	count_image_total
pdfrate 10	count_acroform	pdfrate 28	count_image_xlarge
pdfrate 11	count_acroform_obs	pdfrate 29	count_image_xsmall
pdfrate 12	count_action	pdfrate 30	count_javascript
pdfrate 13	count_action_obs	pdfrate 31	count_javascript_obs
pdfrate 14	count_box_a4	pdfrate 32	count_js
pdfrate 15	count_box_legal	pdfrate 33	count_js_obs
pdfrate 16	count_box_letter	pdfrate 34	count_obj
pdfrate 17	count_box_other	pdfrate 35	count_objstm

Feature	Name
pdfrate 36	count_objstm_obs
pdfrate 37	count_page
pdfrate 38	count_page_obs
pdfrate 39	count_startxref
pdfrate 40	count_stream
pdfrate 41	count_stream_diff
pdfrate 42	count_trailer
pdfrate 43	count_xref
pdfrate 44	createdate_dot
pdfrate 45	createdate_mismatch
pdfrate 46	createdate_ts
pdfrate 47	createdate_tz
pdfrate 48	createdate_version_ratio
pdfrate 49	creator_dot
pdfrate 50	creator_lc
pdfrate 51	creator_len
pdfrate 52	creator_mismatch
pdfrate 53	creator_num
pdfrate 54	creator_oth
pdfrate 55	creator_uc
pdfrate 56	delta_ts
pdfrate 57	delta_tz
pdfrate 58	image_mismatch
pdfrate 59	image_totalpx
pdfrate 60	keywords_dot
pdfrate 61	keywords_lc
pdfrate 62	keywords_len
pdfrate 63	keywords_mismatch

Feature	Name
pdfrate 64	keywords_num
pdfrate 65	keywords_oth
pdfrate 66	keywords_uc
pdfrate 67	len_obj_avg
pdfrate 68	len_obj_max
pdfrate 69	len_obj_min
pdfrate 70	len_stream_avg
pdfrate 71	len_stream_max
pdfrate 72	len_stream_min
pdfrate 73	moddate_dot
pdfrate 74	moddate_mismatch
pdfrate 75	moddate_ts
pdfrate 76	moddate_tz
pdfrate 77	moddate_version_ratio
pdfrate 78	pdfid0_dot
pdfrate 79	pdfid0_lc
pdfrate 80	pdfid0_len
pdfrate 81	pdfid0_mismatch
pdfrate 82	pdfid0_num
pdfrate 83	pdfid0_oth
pdfrate 84	pdfid0_uc
pdfrate 85	pdfid1_dot
pdfrate 86	pdfid1_lc
pdfrate 87	pdfid1_len
pdfrate 88	pdfid1_mismatch
pdfrate 89	pdfid1_num
pdfrate 90	pdfid1_oth
pdfrate 91	pdfid1_uc

Feature	Name
pdfrate 92	pdfid_mismatch
pdfrate 93	pos_acroform_avg
pdfrate 94	pos_acroform_max
pdfrate 95	pos_acroform_min
pdfrate 96	pos_box_avg
pdfrate 97	pos_box_max
pdfrate 98	pos_box_min
pdfrate 99	pos_eof_avg
pdfrate 100	pos_eof_max
pdfrate 101	pos_eof_min
pdfrate 102	pos_image_avg
pdfrate 103	pos_image_max
pdfrate 104	pos_image_min
pdfrate 105	pos_page_avg
pdfrate 106	pos_page_max
pdfrate 107	pos_page_min
pdfrate 108	producer_dot
pdfrate 109	producer_lc
pdfrate 110	producer_len
pdfrate 111	producer_mismatch
pdfrate 112	producer_num
pdfrate 113	producer_oth

Feature	Name
pdfrate 114	producer_uc
pdfrate 115	ratio_imagepx_size
pdfrate 116	ratio_size_obj
pdfrate 117	ratio_size_page
pdfrate 118	ratio_size_stream
pdfrate 119	size
pdfrate 120	subject_dot
pdfrate 121	subject_lc
pdfrate 122	subject_len
pdfrate 123	subject_mismatch
pdfrate 124	subject_num
pdfrate 125	subject_oth
pdfrate 126	subject_uc
pdfrate 127	title_dot
pdfrate 128	title_lc
pdfrate 129	title_len
pdfrate 130	title_mismatch
pdfrate 131	title_num
pdfrate 132	title_oth
pdfrate 133	title_uc
pdfrate 134	version

Appendix B

Virus Total Classifiers

The following classifiers and information used in Virus Total to analyze a submitted file[1].

- AegisLab (AegisLab)
- Agnitum
- AhnLab (AhnLab V3)
- Antiy Labs (Antiy-AVL)
- Androguard
- Aladdin (eSafe)
- ALWIL (Avast! Antivirus)
- AVG Technologies (AVG)
- Avira
- BluePex (AVware)
- Baidu (Baidu-International)
- BitDefender GmbH (BitDefender)

[1] https://www.virustotal.com/en/about/credits/

- Bkav Corporation (Bkav)
- ByteHero Information Security Technology Team (ByteHero)
- Cat Computer Services (Quick Heal)
- CarbonBlack
- Cuckoo Sandbox
- CMC InfoSec (CMC Antivirus)
- CYREN
- ClamAV
- Comodo (Comodo)
- CrowdStrike
- Doctor Web Ltd. (Dr.Web)
- Emsi Software GmbH (Emsisoft)
- Endgame
- Eset Software (ESET NOD32)
- Exif Tool
- Fortinet
- FRISK Software (F-Prot)
- F-Secure
- G Data Software (G Data)
- Hacksoft (The Hacker)
- Hauri (ViRobot)

- IKARUS Security Software (IKARUS)
- INCA Internet (nProtect)
- Invincea (Invincea, acquired by Sophos)
- Intel Security (McAfee)
- Jiangmin
- K7 Computing (K7AntiVirus, K7GW)
- Kaspersky Lab (Kaspersky Anti-Virus)
- Kingsoft
- Malwarebytes Corporation (Malwarebytes' Anti-Malware)
- Magic Descriptor
- Microsoft (Malware Protection)
- Microworld (eScan)
- Nano Security (Nano Antivirus)
- Norman (Norman Antivirus)
- NSRL
- Panda Security (Panda Platinum)
- PDFiD
- Pefile
- PEiD
- Qihoo 360
- Rising Antivirus (Rising)

- Sigcheck
- Snort
- Sophos (SAV)
- SUPERAntiSpyware
- Suricata
- ssdeep
- Symantec Corporation (Symantec)
- Taggant packer information tool
- TrID
- Tencent
- ThreatTrack Security (VIPRE Antivirus)
- TotalDefense
- Trend Micro (TrendMicro, TrendMicro-HouseCall)
- UEFI Firmware parser
- VirusBlokAda (VBA32)
- Webroot
- Wireshark
- WhiteArmor
- Zemana behaviour
- Zillya! (Zillya)
- Zoner Software (Zoner Antivirus)

Bibliography

[1] Contagio dump, http://contagiodump.blogspot.com (accessed on 2016.11.11).

[2] Cybersecurity ceo says beware of north korean hackers 'building a cache of bitcoin'.

[3] Multiple vulnerabilities in adobe acrobat and adobe reader could allow for remote code execution (apsb17-36).

[4] The rise of document-based malware. https://www.sophos.com/en-us/security-news-trends/security-trends/the-rise-of-document-based-malware.aspx.

[5] The rise of machine learning (ml) in cybersecurity. https://www.crowdstrike.com/resources/white-papers/rise-machine-learning-ml-cybersecurity/.

[6] Mimicus framweork. https://github.com/srndic/mimicus, 2017.

[7] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection.

[8] George Argyros, Ioannis Stais, Suman Jana, Angelos D Keromytis, and Aggelos Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1690–1701. ACM, 2016.

[9] George Argyros, Ioannis Stais, Aggelos Kiayias, and Angelos D Keromytis. Back in black: towards formal, black box analysis of sanitizers and filters. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 91–109. IEEE, 2016.

[10] HDI Project Bennet Cyphers, Kalyan Veeramachaneni. Btb. https://github.com/HDI-Project/BTB, 2017.

[11] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 387–402. Springer, 2013.

[12] Thomas P. Bossert. It's official: North korea is behind wannacry, Dec 2017.

[13] Yizheng Chen, Yacin Nadji, Athanasios Kountouras, Fabian Monrose, Roberto Perdisci, Manos Antonakakis, and Nikolaos Vasiloglou. Practical attacks against graph-based clustering. *arXiv preprint arXiv:1708.09056*, 2017.

[14] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. 2017.

[15] Demidova Gudkova, Vergelis. Kaspersky security bulletin. spam and phishing in 2016 q3, 2016.

[16] Hossein Hosseini, Baicen Xiao, Andrew Clark, and Radha Poovendran. Attacking automatic video analysis algorithms: A case study of google cloud video intelligence api. *arXiv preprint arXiv:1708.04301*, 2017.

[17] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*, 2017.

[18] Alex Kantchelian, JD Tygar, and Anthony Joseph. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning*, pages 2387–2396, 2016.

[19] Pavel Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[20] Wei-Jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D Keromytis. A study of malcode-bearing documents. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 231–250. Springer, 2007.

[21] Matthew V Mahoney and Philip K Chan. Phad: Packet header anomaly detection for identifying hostile network traffic. 2001.

[22] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 119–130. ACM, 2013.

[23] Nir Nissim, Aviad Cohen, and Yuval Elovici. Aldocx: Detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. *IEEE Transactions on Information Forensics and Security*, 2016.

[24] Michael Riley, Jordan Robertson, and Anita Sharpe. The equifax hack has the hallmarks of state-sponsored pros, Sep 2017.

[25] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against rnns and other api calls based malware classifiers. *arXiv preprint arXiv:1707.05970*, 2017.

[26] Offensive Security. Client side exploits. https://www.offensive-security.com/metasploit-unleashed/client-side-exploits/, 2017.

[27] Tegjyot Singh Sethi and Mehmed Kantardzic. Data driven exploratory attacks on black box classifiers in adversarial domains. *arXiv preprint arXiv:1703.07909*, 2017.

[28] Tegjyot Singh Sethi, Mehmed Kantardzic, and Joung Woo Ryu. security theater: On the vulnerability of classifiers to exploratory attacks. In *Pacific-Asia Workshop on Intelligence and Security Informatics*, pages 49–63. Springer, 2017.

[29] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 239–248. ACM, 2012.

[30] Charles Smutz and Angelos Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. NDSS, 2016.

[31] Nedim Šrndic and Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[32] Nedim Šrndić and Pavel Laskov. Hidost: a static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security*, 2016(1):22, 2016.

[33] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. Atm: A distributed, collaborative, scalable system for automated machine learning.

[34] @threatintel. Pdf malware writers keep targeting vulnerability. https://www.symantec.com/connect/blogs/pdf-malware-writers-keep-targeting-vulnerability.

[35] Liang Tong, Bo Li, Chen Hajaj, and Yevgeniy Vorobeychik. Feature conservation in adversarial classifier evasion: A case study. *arXiv preprint arXiv:1708.08327*, 2017.

[36] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, page 4. ACM, 2011.

[37] Kalyan Veeramachaneni, Ignacio Arnaldo, Vamsi Korrapati, Constantinos Bassias, and Ke Li. Aiˆ 2: training a big data machine to defend. In *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), 2016 IEEE 2nd International Conference on*, pages 49–54. IEEE, 2016.

[38] Beilun Wang, Ji Gao, and Yanjun Qi. A theoretical framework for robustness of (deep) classifiers under adversarial noise. *arXiv preprint arXiv:1612.00334*, 2016.

[39] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

Appendix C

SMDA Software

C.1 Overview

In previous chapters we discussed the logical abstractions and classes of algorithms in the SMDA framework. Our goal in developing the software is to create an open source platform that can generalize building decision systems in a variety of adversarial environments. We aim to make it easy for developers to improve upon existing algorithms and easy for developers to deploy a detection system. By providing this software system we can connect algorithms researchers with end-users and provide a system for building an evaluating detection systems for developers.

In this chapter, we discuss the design goals in implementing the software framework, the current implementation, and future directions.

C.2 Design Goals

In this section we discuss our primary design goals in software system design. In priority order, they are Usability, Extensibility, and Scalability.

Usability

Our first goal in designing the system is make the software easy to use by developers trying to deploy a detection system in an adversarial environment. To this effort, we want it to be as easy as possible for developers to input their data, select their system

configuration, and evaluate system performance.

Extensibility

Our second goal is to make the system easily extendable. While we've developed algorithms that perform well for PDF malware detection in a resource constrained environment, we recognize other researches may have developed new algorithms to synthesize data, model data, deploy decision functions, or adapt systems. We hope to make a collaborative environment so researchers can benchmark their algorithms against existing algorithms and data sets and facilitate algorithms to be deployed by end users.

To accomplish the goal of enabling extensible we utilize a highly modular design. Making modules independent as possible and automating testing allows for components to be improved independents of the rest of the system.

Scalability

We aim to implement the SMDA framework so that it can scale to adding additional classes. Using a hierarchical model, we limit scale of testing.

C.3 Modules

The initial version of the system has the following modules. This may be subject to change as we deploy with beta-testers and gain a better understanding of the best interface for developers and researchers.

For each of the four classes of algorithms we have corresponding abstract class. *Synthesize* is done in the *SampleGenerator*, and corresponds to the generating data. *Learn* is implemented in the *Model* class. *Decide* and *Adapt* are implemented in the *PredictionPipeline* class as they can have dependencies on the models used.

In addition to the three core abstract classes and corresponding packages, we implement packages for the necessary static functions needed to deploy the model. The static function packages are *FeatureExtraction* and *Evaluation*.

C.3.1 Sample Generator

SampleGenerator creates labeled data to be used in model training, experimental evaluation, or adaptation. The abstract class requires the *generate*(*n_samples*) method to be implemented to return a tuple of an array of nsamples and nlabels.

Creating subclasses of SampleGenerator enables specifc dataset creation. For example, in the simple case samples can be created from a CSV file. More advanced implementations would be to use evasive algorithms to create evasive files from a set of known malicious and benign files.

C.3.2 Model

Model allows for fitting the labeled data to a known distribution. The abstract model class requires that the sample_type be set to indicate the acceptable inputs for a file. Model.fit() operates on extracted features from samples and the corresponding labels to model the data. Model.predict() returns a tuple of categorical predictions and corresponding confidence.

C.3.3 PredictionPipeline

PredicitonPipleine is the class used to facilitate deploying a detection model. PredictionPipelines requires an input model or models, feature extraction functions, and initial thresholds and hyper parameters used for training and tuning.

This abstract skeleton for a PredictionPipeline contains the following methods:

- __init__
- extract_features : feature extraction function
- train : train the model(s) used in the system
- @public: predict
- tune: optimize decision system

- @public: adapt
- adapt_unlabled

This simple skeleton where the required public methods are predict and adapt. Other functions follow the suggested logical breakdown for system models and may be very simple depending on model(s) used and requirements.

MultiModelThresholdPredictionPipeline is an implementation of pipeline that implements the threshold decision making necessary for a system with multiple classifiers in a resource constrained environment using bayesian hyper parameter tuning to optmize decision thresholds. This can generalize for any set of samples and classifiers where the following parameters are set:

- primary_models : base classifiers
- primary_model_names : name for base classifiers
- secondary_models : machine learning models to use to model output of base classifiers
- training_data : initial training daata for primary classifiers
- feature_fns : functions to extract features for each base classifier
- thresholds : initial system thresholds
- eval_fn : function to maximize when tuning system.
  E.g. (1-pct of maximum time taken) + precision when recall is ≥ 90
- split_train_tune_fn : function to split data between training model and tuning decision function
- split_train_primary_secondary_fn : function to split data between training primary and secondary classifiers
- max_tune_iterations : maximum number of iterations to run per round of tuning

- tune_eps : acceptable difference between best evaluation function scores to determine end of tuning

- enumerate_options_fn : Enumerate initial threshold options

- conf_threshold : Value that specifies minimum confidence in data / labels used to adapt system

C.3.4 FeatureExtraction

FeatureExtraction corresponds to the package of static functions used to extract features from a sample. Examples of feature extraction include selecting columns from a dataframe or extracting the PDFrate features from the path to a PDF file.

C.3.5 Evaluation

Correctly specifying function is essential to providing a good system. The evaluation function is a negative cost function which is maximized as the system is tuned. A simple evaluation function could be the f_1 score for the aggregate decision system. However in some cases, there are other meta information of the method of classification that impact the score. In the case of the active defender system the evaluation as the negative of the cost functions described in Chapter 7. This requires the predicted value, the true value, and classifier used as meta data.

Other evaluation functions could maximize accuracy as long as average classifying time is less than a specified threshold.

C.4 Testing

We designed the system to be as module are possible to enable independent testing and evaluation of system modules. We implemented unit tests usint the Nose python testing framework and enabled continuous integration using Circle.ci.

C.5 Documentation

To facilitate ease of understanding we use function, class, and package level documentation using the Sphinx documentation package.