Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»

Факультет радиотехники и кибернетики
Кафедра теоретической и прикладной информатики

Направление подготовки/специальность: 03.04.01 Прикладные математика и физика
Направленность (профиль) подготовки: Математические и информационные технологии
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения
___________/______________________
(подпись) *расшифровка подписи*
«____» ______________20____г.

ОБНАРУЖЕНИЕ ВРЕДОНОСНЫХ ПРОГРАММ МЕТОДАМИ МАШИННОГО ОБУЧЕНИЯ

(магистерская диссертация)

Обучающийся: Маракулин Роман Игоревич

____
(подпись обучающегося)

Научный руководитель: Кулага Андрей Александрович, к.ф.-м.н.

____
(подпись научного руководителя)

Москва 2018

Аннотация

Вредоносные программы (malware) становятся все более и более сложными. Современные вирусы имеют способность мутировать, изменяться в процессе жизнедеятельности, что приводит к росту количества вариантов вредоносных программ. Традиционные подходы, основанные на поиске сигнатур файлов перестают быть эффективными. На смену приходит автоматизация анализа файлов для поиска подозрительных файлов.

Данная работа посвящена исследованию методов машинного обучения для задачи обнаружения вредоносных программ PE (Portable Executable) формата для ОС Windows и построению движка машинного обучения, способного с высокой точностью детектировать вредоносный код до его исполнения. А именно, проводится:

1. Исследование существующих подходов к статическому анализу PE файлов
2. Отбор значимых для анализа атрибут PE файлов
3. Сравнение алгоритмов машинного обучения для данной задачи
4. Создание движка машинного обучения, способного детектировать вредоносные программы

Данные для обучения в размере 55432 файлов с вердиктами от 109 антивирусов были получены с помощью сервиса VirusTotal. Были выделены наиболее важные для анализа признаки (135 признаков в приложении А), выбран оптимальный с точки зрения качества работы и скорости алгоритм машинного обучения. Создан движок машинного обучения.

Полученные результаты демонстрируют, что данная задача успешно решается методами машинного обучения, достигается высокий уровень точности (accuracy) (96.5%), открываются возможности по детектированию ранее неизвестных вредоносных программ, защите от уязвимости нулевого дня. При реализации движка в программный продукт важна скорость его работы на машинах конечных пользователей, что также было исследовано в рамках данной работы. Созданный движок интегрирован в выпускаемый программный продукт и позволяет с высокой скоростью и хорошим качеством выносить вердикт об угрозе исследуемого файла.

Содержание

1 Введение **4**

2 Постановка задачи **8**

3 Обзор существующих решений **12**

4 PE формат **17**

5 Алгоритмы машинного обучения **22**

5.1 Дерево решений 22

5.2 Случайный лес 25

5.3 Градиентный бустинг 27

6 Эксперименты **30**

6.1 Получение данных 30

6.2 План экспериментов 33

6.3 Отбор признаков 34

6.4 Сравнение алгоритмов 36

7 Заключение **41**

Список литературы **43**

Приложение 1 (список используемых признаков) **46**

1 Введение

Сегодня компьютерные программы и приложения разрабатываются с большой скоростью. С развитием компьютерной техники и интернета, все больше людей начинают беспокоиться об их безопасности, безопасности их данных. Развитие вредоносного ПО приводит к развитию более современных механизмов защиты. По данным McAfee Lab в 4м квартале 2014 года появилось более 350 миллионов полностью уникальных вредоносных программ (рост на 17% относительно 3 квартала 2014 года) [1]. По исследованию Symantec более 44.5 миллионов вредоносных ПО было создано за май 2015 [2]. За 2017 год мировая общественность познакомилась с WannaCry, Petya, NotPetya. Анализ такого количества данных требует от антивирусных компаний все возрастающие усилия. Вирусы становится все более сложно обнаружить. Современное вредоносное ПО способно как внедряться и заражать чистые файлы системы и других программ, так и самостоятельно распространяться и менять свое тело исполнения в процессе жизнедеятельности. Применяются различные способы скрыть вредоносный код. Используются инструменты шифрования секций, кода, данных. Вредоносное ПО может не только украсть или повредить данные пользователя, вымогать деньги, замедлить работу компьютера, но и превратить компьютер пользователя в шпиона, завладеть его управлением.

Традиционный подход к обнаружению вредоносных программ основан на сопоставлении сигнатур исследуемых файлов [3]. Процедура заключается в следующем:

1. Новый вирус/вредоносное ПО начинает распространяться
2. Эксперты антивирусных компаний получают образцы для исследования поведения вируса
3. Эксперты присваивают вирусу уникальную сигнатуру, представляющую собой последовательность инструкций
4. Сигнатура добавляется в базу данных сигнатур вредоносных программ
5. Клиенты уведомляются об обновлении базы сигнатур

6. Клиенты обновляют их базы сигнатур, таким образом становясь защищенными от данного вида вредоносного ПО

Стоит отметить, что сигнатуры позволяют гарантированно определить тип вируса. Это позволяет дополнительно вносить в базу сигнатур способы лечения зараженных файлов, вирусов.

Данный подход при всей своей простоте обладает ключевыми недостатками:

1. Возможность защищать клиента только от известных вирусов. Нельзя получить сигнатуру совершенно нового вируса, не исследовав его. Здесь стоит оговориться, что сигнатуры, как правило, создаются так, чтобы покрывать не один, а как можно большее множество, семейство вирусов. Однако всегда существует такое изменение тела вируса, при котором сигнатура перестает обнаруживаться

2. Постоянный рост базы данных сигнатур. С увеличением количества вирусов, их семейств, а также способности вирусов изменяться увеличивается и скорость наполнения базы

3. При появлении вируса и до обновления базы сигнатур клиент уязвим для новой вредоносной программы. Только определив исследуемый файл как вирус можно получить его сигнатуру и добавить в базу

Более того, разработчики вирусов научились успешно обходить поиск сигнатур, обфусцируя тело вируса. Полиморфные и метаморфические вредоносные программы меняют свой внешний вид в процессе жизнедеятельности. Все это побудило антивирусные компании разрабатывать альтернативные способы защиты. А именно, можно выделить 2 крупных направления исследований [4]:

1. Статический анализ (анализирование структуры бинарного файла, его атрибутов, логических структур, потока исполнения и данных)

2. Динамический анализ (отслеживание действий программы при исполнении, построение её профиля)

Каждый из методов имеет свои достоинства и недостатки. Так, как правило, для лучшего детектирования вредоносных программ они используются одновременно. В каждом из этих методов существует вероятность ошибочно определить наличие в файле вируса, когда на самом деле файл чист. В таких случаях предполагаемое лечение может испортить файл, что повлечет к потере информации.

Динамический анализ позволяет обойти обфускацию бинарного файла. Так, например, вирусописатели широко используют системы упаковки, шифрование кода и данных, обфускацию функций и потока управления. Но те же методы используются и для создания приложений разработчиками, чтобы защитить интеллектуальную собственность, усложнить реверс инжиниринг. Метод выделяет несколько основных действий, таких, как удаление файла, запись в файл, общение с сетью, открытие порта на прослушивание, рассылка писем и другие. Этот профиль файла, его деятельности изучается экспертом или методами машинного обучения для вынесения вердикта о вредоносности образца. Тем не менее, динамический анализ возможен лишь при исполнении исследуемого кода, что делает операционную систему более уязвимой, а также, некоторые вирусы могут определять запускаемое окружение, маскируя себя, и, тем самым, ведут себя по-разному в тестовом и рабочем окружении. Таким образом, требуется создавать максимально схожие окружения, что представляется еще одной проблемой, требующей решения.

Статический анализ способен дополнять динамический, предоставляя информацию об атрибутах бинарного файла. Статический метод анализирует программу до исполнения, извлекает атрибуты из бинарного файла, подсчитывает статистики и на основе этой информации выносит вердикт об угрозе исследуемого файла. Такой подход безопасен — вердикт выносится еще до исполнения файла, но плохо работает на файлах с обфускациями, запакованными секциями. Также, с увеличением размера файла, время, требуемое для анализа, увеличивается. Помимо этого, для разработки качественного статического анализатора, требуется понимать работу загрузчика бинарного файла. Разницу между документацией и действительным поведением загрузчика. Вирусы могут использовать часть полей бинарного файла для своих нужд. Например в виде хранения данных или адреса исполнения вредоносного кода.

Далее, мы остановимся на статическом анализе PE (Portable Executable) файлов для операционной системы Windows. Выбор связан с огромной популярностью этой системы. PE формат же является стандартом исполняемых файлов и библиотек. Нужно понимать, что шаблон, алгоритм проектирования движка статического анализа вредоносного ПО применим и к другим платформам.

Задача состоит в создании движка машинного обучения для статического анализа PE файлов для операционной системы Windows с последующей интеграцией в программный продукт. Также, важно отметить, что движок машинного обучения должен обладать высокой скоростью работы. Для слабых персональных компьютеров время сканирования системного диска должно укладываться в несколько минут. Для решения данной задачи требуется:

1. исследовать существующие подходы
2. выявить наиболее важные признаки для определения вредоносности файла в условиях ограниченных ресурсов и времени
3. сравнить современные алгоритмы машинного обучения применительно к данной задаче
4. реализовать движок в рамках программного продукта

Данная работа отвечает на вопрос о возможности использования современных методов машинного обучения для эффективного детектирования вредоносного ПО в реальных условиях. Реализуется движок статического анализа, имеющий практическое применение и являющийся модулем системы защиты машин конечного пользователя.

2 Постановка задачи

Задача состоит в создании статического движка машинного обучения для анализа PE (Portable Executable) файлов, позволяющего с высокой точностью выносить вердикт о вредоносности файла.

Перед тем как решать задачу, сформулируем её более формально. А именно, обсудим требования и ограничения предъявляемые к решению, входные и выходные данные и метрики качества, по которым будет определяться наилучшее решение.

В работе ограничиваемся рассмотрением файлов для операционной системы Windows, а именно, файлов PE формата. Алгоритм действий для поиска лучшей модели для других файлов, файлов других систем будет тем же самым. Выбор сделан на основе популярности операционной системы Windows среди других систем. А все исполняемые файлы Windows имеют PE формат. Статический анализ файла основан только на изучении структуры его файла, в нашем случае, на изучении PE структуры и извлечении полезных для детекции признаков. Исполнение кода с какой-либо целью недопустимо. При запуске исследуемой программы надежность системы понижается. Доля ложно положительных срабатываний должна быть как можно меньше. Движок как можно реже должен ошибаться на определении чистых файлов как вредоносных. Это может привести к потере или повреждению данных пользователя. Будем считать количество ошибок в 3% приемлемым. Наиболее сложной характеристикой требований для определения является точность (accuracy). Точность алгоритма должна быть как можно выше, с другой стороны нам также важна скорость алгоритма. Пользователь не должен испытывать неудобств при использовании антивирусной защиты. А значит нам нужно найти приемлимые параметры для качества и скорости. Также, довольно сложно сопоставить хотя бы примерно результаты большинства исследований. Нет общедоступной большой базы файлов, которую можно было бы считать эталоном. Каждое исследование проводится со своими данными. Также нужно понимать, что не имеет смысла сравнивать алгоритмы машинного обучения и сигнатурный метод на одних и тех же данных. С большой вероятностью файлы устарели и их сигнатуры уже есть в базах. Выберем порог качества в 95% как минимально допустимый, при ко-

тором алгоритм даже не будет рассматриваться. Скорость движка подберем из расчета полного сканирования системного жесткого диска HDD. На новой системе насчитывается около 20-30 тысяч PE-файлов. Поэтому выберем приемлемую среднюю скорость работы алгоритма равной 30 файлам в секунду. Теперь мы можем сформулировать общие требования:

1. Источником данных для модели служит PE файл
2. Возможна лишь работа с исходным бинарным файлом без его исполнения
3. Точность (accuracy) предсказания выше 95%
4. Доля ложно положительных срабатываний менее 3%
5. Средняя скорость работы алгоритма — 30 файлов в секунду

Движок на вход принимает PE-файл и возвращает вердикт о вредоносности файла:
Вход: PE-файл
Выход: вердикт (0 - файл чистый, 1 - вредоносный)

Движок представляет из себя pipeline. Входящие данные проходят несколько этапов обработки и на выходе получаем вердикт о файле.

Выделим компоненты из которых состоит движок:

1. PE-парсер, позволяющий получить значимые для анализа атрибуты
2. Классификатор, принимающий на вход атрибуты исследуемого PE-файла и относящего его к одному из классов (0 — для чистых файлов и 1 — для вредоносных)

Хоть в данной работе и не рассматривается вопрос создания PE-парсера, он неразрывно связан с классификатором и также является компонентой движка. Классификатор диктует набор значимых для анализа атрибут. Схема движка представлена на Рис. 1. Вообще говоря, не все поступающие на вход PE-парсера файлы могут являться PE-файлами. Файлы могут быть битыми или не являться файлами формата Portable Executable. В реальных условиях PE-парсер сообщает об ошибке и признаки таких файлов не передаются в классификатор. Поэтому для простоты будем считать, что все передаваемые на вход

PE-парсера файлы имеют исполняемый Portable Executable формат и их атрибуты передаются в Attribute Selector.

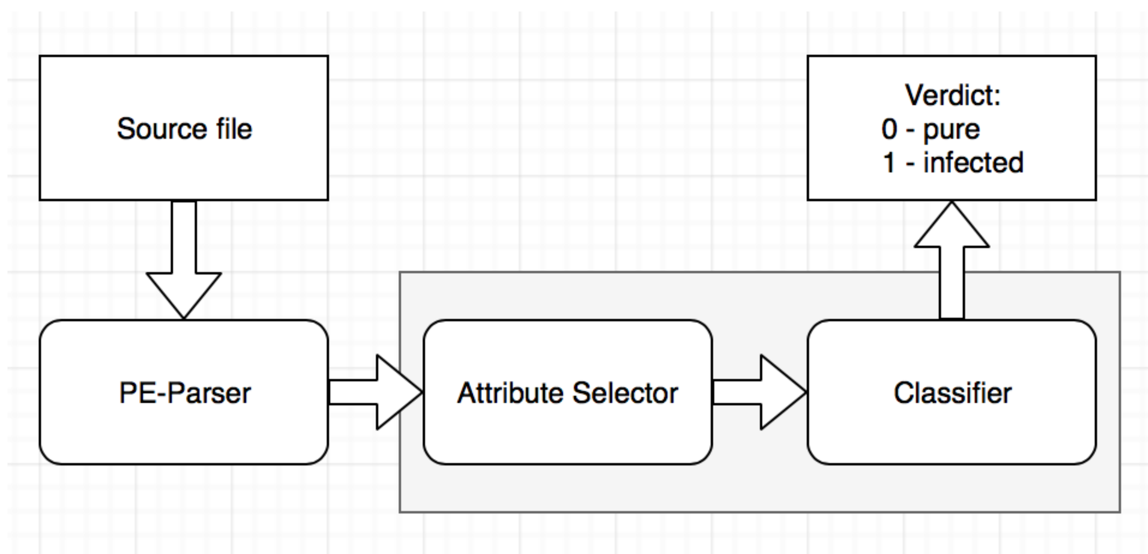

Рисунок 1 – Общая архитектура движка по анализу PE файлов. Серая область — фокус исследований в данной работе

Как было упомянуто. качество модели оцениваем метрикой accuracy — отношение количества верно классифицированных файлов ко всем файлам. Это будет нашей основной метрикой. Также, при оценке и сравнении качества моделей стоит обращать внимание и на false positive rate. Метрика accuracy при неравномерной выборке может не отражать действительную способность классификации алгоритма, тогда как вместе с false positive rate дает более полную картину. Метрика accuracy основная в том смысле, что она является определяющей при выборе среди алгоритмов у которых false positive rate меньше порогового значения.

Требование на скорость движка сформулировано таким образом, что можно говорить лишь о средней его скорости работы и среднем времени обработки файла. Это связано с зависимостью времени работы модуля от размера файла. Требование высокой скорости работы движка и, одновременно с этим, хорошей точности предсказания вынуждает разумно подходить к выбору атрибутов и их количества для извлечения из PE файла, а также, к выбору алгоритма машинного обучения, поэтому принципиально не будут рассматриваться нейронные сети и методы глубинного обучения. Это требует, с одной стороны, считывания большего количества байт файла, а с другой, использования более сложных мо-

делей, перемножения матриц, вычисления нелинейностей, что будет на порядок медленнее подходов, представленных в данной работе.

3 Обзор существующих решений

Большинство статических анализаторов основываются на извлечении структур PE (Portable Executable) файла, как в древовидном виде, так и в виде совокупности полей. Используются как признаки, получаемые из машинного кода, вызовов функций, так и признаки, получаемые из рассмотрения файла в виде последовательности байт, определения статистик, энтропий, подсчет n-грам.

Цель работы [5] состояла в создании отказоустойчивого PE парсера — PortEX, способного также выявлять аномалии в структурах бинарных файлов. Было проведено исследование стратегий заражения файлов, выделены наиболее вероятные атрибуты чистых файлов. Также проводилась оценка зависимости энтропии секций PE файла и вероятности того, что файл вредоносен. Для получения вероятности того, что файл вредоносен применялись эвристики. Были определены:

1. Бустеры (шаблоны, которые указывают на поведение или внешний вид вредоносного ПО). Увеличивают вероятность того, что файл будет идентифицирован как вредоносный
2. Стопперы (шаблоны, которые указывают на поведение или внешний вид, который является нетипичным для вредоносного ПО). Уменьшают вероятность того, что файл будет определен как вредоносный

Вычислялись суммарный бустер и суммарный стоппер на основе эвристик и статистики по собранным файлам и на основе получившегося значения выдавалась вероятность вредоносности файла. Энтропия секций PE файла помогла определять вредоносность файла. Границы и определение бустеров и стопперов представляют собой эвристики, основанные на статистике по многим файлам, которые можно улучшить методами машинного обучения.

В [3] анализ строится на извлечении 197 атрибутов из бинарного файла. Атрибуты представляют собой либо характер присутствия (DLLs referred, APIs referred), либо значения полей структур файла. Выделение значимых атрибут происходит на основе статистических тестов (t-тест, χ^2-тест). Используя отобранные 19-20 признаков обучают модели бустинга, случайного леса. Отсутствует более глубокий анализ PE-формата, рассмотрение участков файла в

виде последовательности байт. Также нет информации о скорости работы алгоритма.

Достаточно полная картина выбора важных атрибут PE файла приведена в [6]. Приведена таблица (Таблица 1) используемых признаков в предыдущих работах на данную тематику. Признаки — признаки, извлеченные из PE файла. Либо из заголовка, либо из тела. В колонке «Структура» показаны дополнительные признаки, собираемые с файла. Дополнительно введены сокращения следующие сокращения: API: Application Programming Interface, BYT: Byte code, FC: Function Call, STC: Structural features, OP: Operation code

Таблица 1 Методы статического анализа с типами извлекаемых признаков

Год	Статья	Тип	Признаки заголовка	Признаки тела	Структура
2008	Ye et al. [7]	детекция	API	—	Itemset
2009	PE-Miner [8]	детекция	STC	STC	—
2009	Tabish et al. [9]	детекция	BYT	BYT	N-gram
2009	Griffin et al. [10]	детекция	BYT	BYT	Sequence
2009	Hu et al. [11]	детекция	—	FC	Graph
2010	Sami et al. [12]	детекция	API	—	Itemset
2011	Natara j et al. [13]	классификация	BYT	BYT	—
2012	Jacob et al. [14]	классификация	STC	BYT	N-gram
2013	Santos et al. [15]	детекция	—	OP	Sequence
2014	Nissim et al. [16]	детекция	BYT	BYT	N-gram
2015	DLLMiner [17]	детекция	DLL	—	Tree

Задача стояла в классификации вредоносных программ на типы (malware family). Статья подготовлена на основе конкурса kaggle, где участникам Microsoft предоставила 21741 файлов без заголовка PE файла (PE header) для классификации по 9 типам. В исследовании основной упор был сделан на выборе лучшего набора атрибут. Выбор проводился как на основе полей структур PE файла, так и рассматривая файл в виде последовательности байт. Более конкретно, представим классификацию признаков, рассматриваемых в статьях:

1. Hex dump-based features (Признаки, извлеченные из последовательности байт файла)

   (a) N-грам. Последовательность N байт. 1-gram — частота байтов и была выбрана в статье

   (b) Метаданные. Например, размер файла

   (c) Энтропия. Подсчет энтропии всего файла или методом скользящего окна

   (d) Представление файла как изображения [13]

   (e) Длины строк. Извлечение длин возможных строк ASCII символов из файла

2. Features extracted from disassembled files (Извлечение признаков из структур PE файла)

   (a) Метаданные. Например, количество строк в файле

   (b) Частота специальных символов. Частота символов: $-, +, *,], [, ?, @$

   (c) Коды операций. Подсчет статистики по ассемблерным инструкциям

   (d) Регистры. Подсчет статистики по используемым регистрам

   (e) Application Programming Interface. Проверка присутствия/отсутствия определенного набора API вызовов

   (f) Секции. Статистика по секциям

   (g) Установка байт. Статистика по db, dw и dd инструкциям

Такая классификация позволяет в полной мере увидеть исследования в данной области по извлечению признаков и текущие результаты. Заметим, что часть этих признаков не доступна в нашем исследовании. Подсчет статистики по длинной последовательности байт представляет собой дорогостоящую операцию. А именно, подсчет статистики по всему файлу или представления файла как изображения будем избегать.

Также были оригинальные идеи из других статей, а именно, представление файла в виде изображения [13]. В статье приведен график важности при-

знаков для определения класса вредоносного ПО. В качестве алгоритма использован бустинг на решающих деревьях (XGBoost) [18], а также, для получения лучшего результата в конкурсе бэггинг (bagging) [19]. В реальных же условиях у нас доступна информация из PE header. Также, интересно исследовать значимость атрибут в задаче определения вредоносного ПО.

Для полноты картины приведем краткое описание оригинального исследования [13], в котором PE файл представлялся в виде черно-белого изображения. Задача стояла в определении типа (malware family) вредоносного ПО. Последовательность байт представляли в виде матрицы (0: черный, 1: белый). Ширина изображения фиксировалась в зависимости от размера файла (32 — <10kB, 64, 128, ..., 1024 — >1000kB). На основе изображений были определены общие паттерны, текстуры для 25 типов вредоносных ПО. Для анализа текстуры применялся фильтр Габора. Признаки, полученные из изображений использовали в классификаторе — метод k-ближайших соседей с метрика эвклида. Хоть исследование и представляет интерес, но в силу наших требований, а именно, строгого ограничения времени работы движка, мы вынуждены отказаться от этого вида признаков.

Упомянем о существовании библиотек, нацеленных на борьбу с деобфускацией кода. Один из таких инструментов FLOSS. FLOSS комбинирует в себе несколько подходов статического анализа для поиска обфусцированных ASCII строк в анализируемом файле. А именно, анализирует поток управления для разбора файла на функции, базовые блоки. Использует эвристики для поиска декодирующих функций. Эмулирует поведение функций декодирования для извлечения читаемых ASCII строк. Такой инструментарий призван помочь статическому анализу с зашифрованными файлами и получить больше информации о них. Используя FLOSS, к примеру, можно получить список импортируемых api и библиотек. В данной работе мы не будем использовать этот инструментарий. Задача состоит в создании быстрого и качественного алгоритма машинного обучения. Для более глубокого анализа стоит исследовать отказоустойчивость подобных библиотек и сравнить существующие инструменты деобфускации. В данной работе этот вопрос не освещается.

Таким образом, был проведен общий обзор методик и подходов к задаче статического анализа вредоносного ПО. Более подробные работы, представля-

ющие собой полный обзор методов можно найти в [20], [21].

В отличии от предыдущих исследований, цель данной работы состоит в:

(a) обобщении полученных результатов исследований

(b) получении наиболее важных атрибутов для задачи выявления вредоносного ПО на типичном для пользователей наборе файлов в условиях ограниченных ресурсов и времени

(c) реализации движка в программном продукте с использованием современных алгоритмов машинного обучения

4 PE формат

Для анализа исследуемого файла классификатор получает информацию от PE-парсера. Знание о PE (Portable Executable) формате необходимо для оценки полезности извлекаемой информации и понимания загрузки файла в память для выделения наиболее значимых атрибут.

Далее приводится краткое описание PE формата, его структуры, как для 32 битных, так и 64 битных систем.

Полная спецификация доступна на сайте Microsoft [22], тем не менее, в документации есть двусмысленности. Некоторые моменты покрыты статьей Криса Касперски [23]. Нужно понимать, что спецификация, приведенная на сайте Microsoft, имеет, скорее, обзорный характер. Как в действительности поступают загрузчики можно наблюдать лишь экспериментально. В особо хитрых случаях один и тот же файл может быть запущен одним загрузчиком, но, запустив другим, может привести всю систему к перезагрузке. Стоит помнить, что загрузчик при запуске бинарного файла может его изменять.

«Portable Executable (PE, переносимый исполняемый) — формат исполняемых файлов, объектного кода и динамических библиотек, используемый в 32- и 64-разрядных версиях операционной системы Microsoft Windows. Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую PE-загрузчику для отображения файла в память» [24]

Все PE файлы можно разделить на файлы с расширением EXE и DLL. DLL файлы предназначены для экспортирования функций или данных для использования другими программами. То есть, они обычно могут быть запущены в контексте других программ. Такие файлы имеют расширения .dll, .sys, .ocx, .cpl, .fon, .drv [25]. EXE файлы запускаются в своих собственных процессах. Они обычно имеют расширение .exe и не экспортируют символы. Несмотря на такое условное разделение, формат не запрещает создавать гибриды. К примеру, EXE файлы может экспортировать символы.

Структура PE файла представлена на Рис. 2. Атрибуты этой структуры, а также статистика по последовательностям байт этой структуры используется для получения признаков.

Любой PE файл начинается с MS-DOS Stub’a. Оставлен для совместимо-

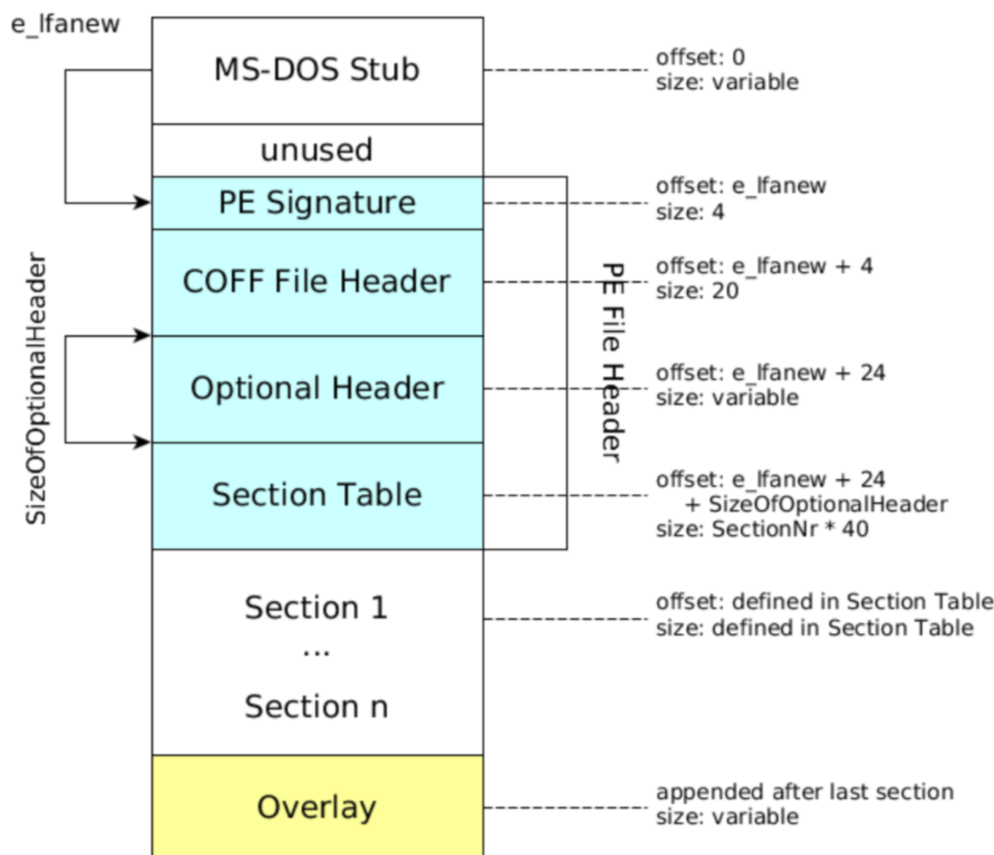

Рисунок 2 – Структура PE файла

сти и представляет собой заглушку. Первые два байта которой обязаны быть равными "MZ". Атрибут e_lfanew, представляет собой смещение PE File заголовка относительно начала файла и указывает на сигнатуру "PEx0x0". Не обязательно, чтобы e_lfanew указывало на область сразу после MS-DOS Stub, что также отражено на Рис. 2 — неиспользованная область. Секции располагаются сразу после PE File заголовка.

После PE сигнатуры "PEx0x0"в PE File Header располагается COFF File Header. Имеет фиксированный размер. Содержит общую информацию о файле, такую как: тип целевой машины (x64, ARM, MIPS и так далее), количество секций — NumberOfSections, дата создания файла, размер опционального заголовка — SizeOfOptionalHeader. Атрибуты всего файла — Characteristics (именно здесь содержатся флаги — является ли файл EXE или DLL). Практически все атрибуты этой структуры стоит рассматривать при детектировании вирусов, заражения файла. Исключим из рассмотрения дату создания файла — он не должен влиять на вероятность заражения, но выборка может оказаться сме-

щенной по этому параметру.

Далее идет опциональный заголовок (Optional Header). Хоть эта структура и имеет такое название, её присутствие обязательно для загрузки файла. Структура содержит уточняющую информацию о файле. Такую, например, как 32х или 64х битное адресное пространство. Суммарный размер секций кода, инициализированных и неинициализированных данных (согласно [23] эти поля никем не проверяются также как и относительные базовые адреса кодовой секции и секции данных). У некоторых антивирусов есть эвристики на значение адреса точки входа — AddressOfEntryPoint. Предполагается, что точка входа располагается в первой секции файла (как правило .text секция). Базовый адрес загрузки программы (ImageBase) должен быть кратен 64кб. Также, для анализа полезны выравнивания — FileAlignment, SectionAlignment, а вернее поля PE файла, от которых требуется выравнивание. Также, в опциональном заголовке содержится каталог данных (DataDirectory), количество элементов которой —
NumberOfRvaAndSizes.

В каталоге данных (DataDirectory) каждый элемент — структура, состоящая из указателя и размера. Каждая из записей имеет определенную роль. Так, IMAGE_DIRECTORY_ENTRY_EXPORT — указатель на таблицу экспортируемых функций и данных (встречается в основном в DLL). Указатель IMAGE_DIRECTORY_ENTRY_SECURITY представляет особый интерес. Он указывает на Certificate Table. Таблицу, находящуюся на диске. При
IMAGE_DIRECTORY_ENTRY_SECURITY != 0 практически исключена вероятность того, что файл является вредоносным. Также, если
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR != 0, то файл представляет собой .NET приложение, состоящее из байт-кода, что также снижает вероятности вредоносности файла.

После опционального заголовка идет таблица секций (Section Table), содержащая NumberOfSections секций. Как правило, секции имеют следующий порядок. Сначала описана секция с кодом, после — несколько секций инициализированных данных, а после — секция неинициализированных данных. Каждая секция имеет имя (не играет никакой роли при загрузке файла), адрес начала секции в памяти и в файле (выравненные), виртуальную и физическую

длину секции (VirtualSize и SizeOfRawData). Виртуальные адреса секций должны идти подряд, не накладываясь и не образуя пропусков. Поле Characteristics представляет собой права доступа к секции и особенности ее загрузки. Стоит рассмотреть как секции, так и их атрибуты. Для детекции вирусов и анализа также полезны поля таблицы экспорта, импорта.

Таблица экспорта требуется для связи экспортируемых функций с их адресами. Содержит указатели на 3 таблицы: таблица имен, ординалов, адресов.

Таблица импорта соотносит вызовы функции динамических библиотек с их адресами.Существует 3 режима импорта: стандартный, связывающий (bound import), отложенный (delay import). Для анализа PE файла стоит анализировать названия библиотек и функций (api). Здесь возникает вопрос о способе представления этой информации. Вектор признаков должен быть фиксирован, что накладывает определенные ограничения. С одной стороны мы должны выбирать часто встречающиеся api, с другой — те, которые статистически значимы при отделении вредоносного ПО от чистых файлов.

Изложив общую картину структур PE файлов, мы можем приступить к формированию атрибутов и их отбору. Становятся видны следующие свойства получаемых признаков. Признаки строго делятся по роли и значению структуры из которой были извлечены:

1. численный признак (количество секций, символов, размер опционального заголовка, адрес точки входа)
2. флаг присутствия (секции, api таблицы импорта)
3. значение функции от последовательности байт (энтропия секций)

Здесь мы не рассматриваем функции от сравнительно длинных последовательностей байт, так, например, энтропию всего файла, распределение определенных символов в файле или представление файла в виде изображения. Движок должен быстро работать даже на слабых машинах, поэтому задача состоит в отборе наиболее ценных с точки зрения определения вредоносного ПО признаков.

Здесь становятся видны и главные проблемы статического анализа. Одна из основных — зашифрованные, упакованные секции. Вирусы шифруют свой

код исполнения так, что он больше оказывается не виден с точки зрения статического анализатора. Для решения этой проблемы либо добавляют дешифратор, либо основываются на статистике по этой секции. В первом случае задача предстает отдельным исследованием, которое не будет рассмотрено здесь. Большинство дешифраторов используют эвристики и перебирают известные методы шифрования, а есть те, которые дешифруют при исполнении бинарного файла. При подсчете энтропий мы основываемся на предположении, что хорошее шифрование выравнивает частоту байт секций, тем самым увеличивая ее энтропию.

5 Алгоритмы машинного обучения

Далее будут рассмотрены алгоритмы машинного обучения для задачи классификации, применение которых наиболее оправдано в исследуемой задаче. А именно, мы должны учитывать разнородную природу признаков, получаемых из бинарных файлов. Часть признаков представляет собой флаги присутствия, часть — статистики. Такие, например, как энтропии секций, а часть — поля PE структур. Также, мы берем во внимание скорость работы алгоритма. В таких условиях стоит прежде всего рассмотреть алгоритмы, основанные на решающих деревьях (случайный лес, градиентный бустинг). Существующие исследования в области детектирования вредоносного программного обеспечения также подтверждают это предположение. Опишем рассматриваемые алгоритмы.

5.1 Дерево решений

Алгоритм «решающее дерево» (decision tree) здесь упомянем лишь потому, что на нем основываются более сложные, представленные здесь алгоритмы, и, следует понимать принцип построения базовых алгоритмов для использования их в системе.

Дерево решений, в общем случае, является способом представления правил в иерархической последовательной структуре, где каждому объекту соответствует единственный узел, дающий решение.

Как правило рассматривают бинарное решающее дерево.

Процесс поиска узла представлен на Рис. 3 с соответствующим листингом — Алгоритм 1.

Введем обозначения:

V — множество узлов в дереве. β_v — предикат, функция в вершине v, возвращающая $\{0, 1\}$. Y — множество классов, в нашем случае — 2 (pure — чистый файл, infected — вредоносный).

Выбор класса выполняется следующим образом:

$\forall v \in V_{\text{внутр}} \rightarrow$ предикат $\beta_v : X \rightarrow \{0, 1\}, \beta \in B$

$\forall v \in V_{\text{лист}} \rightarrow$ имя класса $\mathrm{c}_v \in Y$

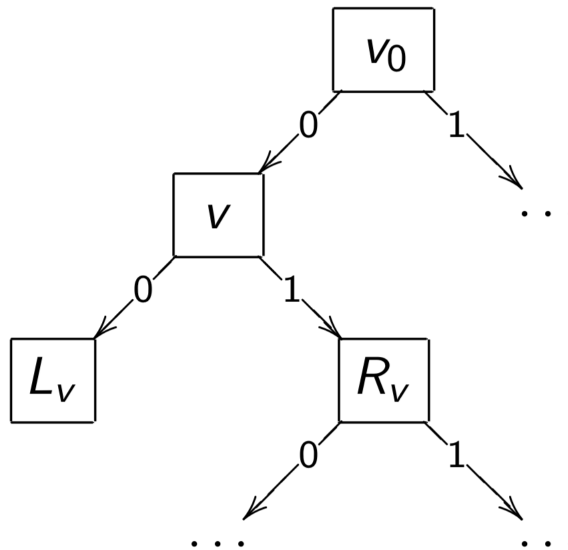

Рисунок 3 – Решающее дерево

while $v \in V_{\text{внутр}}$ **do**
    if $\beta_v(x) = 1$ **then**
        go to right;
        $v := R_v$;
    else
        go to left;
        $v := L_v$;
    end
    return c_v
end

Алгоритм 1: Поиск узла

На практике используются одномерные предикаты, которые сравнивают значение одного из признаков с порогом.

Для построения дерева существует множество алгоритмов. Конкретный метод построения решающего дерева определяется:

1. видом предикатов в вершинах
2. функционалом качества (для принятия решения о разбиении обучающей выборки в рассматриваемой вершине. Как правило используется критерий Джини или энтропийный критерий)

3. критерием останова

4. методом обработки пропущенных значений

5. методом стрижки (удаление некоторых вершин с целью понижения сложности и повышения обобщающей способности)

Возможность обрабатывать пропущенные значения полезна и в нашей задаче. Выбранные имена секций или импортируемые названия библиотек, функций могут быть не найдены. В таком случае, для такого файла часть значений, признаков будет пропущена. Решающее дерево может быть обучено и на таких данных. А именно (здесь, U — множество объектов обучения, $Gain$ — некоторый выбранный критерий для максимизации в каждой вершине, S_v — функция, при при $0 - L_v$, $1 - R_v$)

На стадии обучения:
$b_v(x_i)$ не определено => x_i исключается из U для $Gain(b_v, U)$
$q_{vk} = \frac{|U_k|}{|U|}$ — оценка вероятности k-й ветви, $v \in V_{\text{внутр}}$
$P(y|x, v) = \frac{1}{|U|} \sum_{x_i \in U} [y_i = y]$ для всех $v \in V_{\text{лист}}$

На стадии классификации:
$b_v(x)$ определено => из дочерней $s = S_v(b_v(x))$ взять $P(y|x, v) = P(y|x, s)$
$b_v(x)$ не определено => пропорциональное распределение:
$P(y|x, v) = \sum_{k \in D_v} q_{vk} P(y|x, S_v(k))$
Окончательное решение — наиболее вероятный класс:
$a(x) = arg \max_{y \in Y} P(y|x, v_0)$

Для построения дерева, как правило, используется жадный алгоритм с оптимизацией в каждом узле заданного функционала качества (обычно рассматривают критерий Джини, энтропийный критерий). Выбор функционала качества также сильно влияет на конечный результат.

Как готовый алгоритм, решающее дерево сегодня не используется. Алгоритм склонен к переобучению, сильное влияние на шумы. Также, не всегда дерево строит наилучшее разделение (знаменитый пример с задачей построения XOR функции). На Рис. 4 соответствующий пример.

Большое внимание уделяется ансамблям деревьев решений. Каждое из деревьев способно к переобучению, сильному влиянию на шумовые объекты, но

использование их ансамблей (бустинг, бэггинг) превращает в один из наиболее успешных и применимых на практике инструментов машинного обучения.

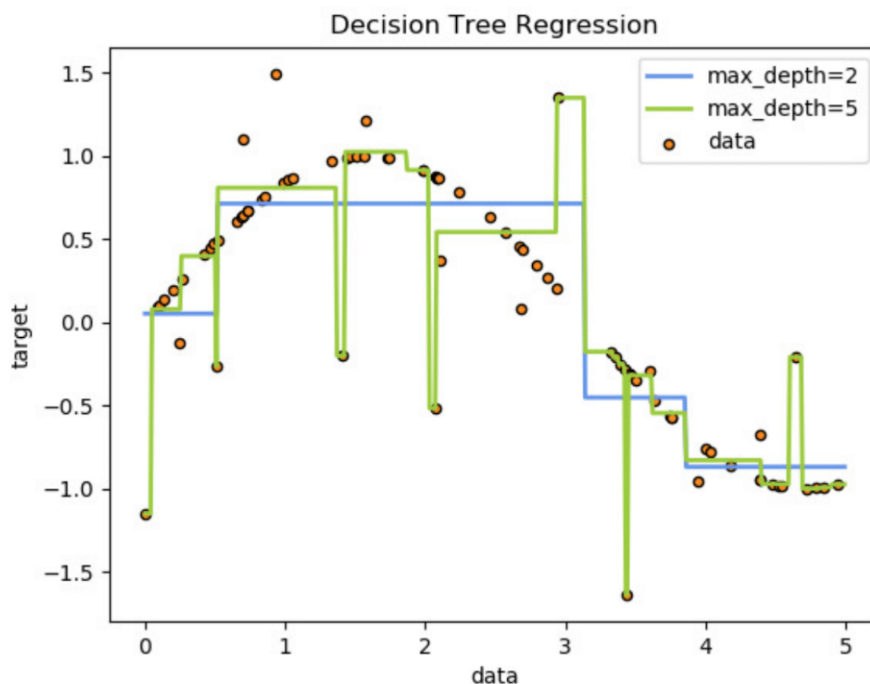

Рисунок 4 – Склонность к переобучению решающего дерева

Побочный эффект от построения дерева решений состоит в возможности оценки важности признаков. Важность признака рассчитывается на основании того, насколько улучшается выбранная метрика при разделении выборки по признаку в очередном узле. Это может нам дать представление какие признаки для построения дерева оказались наиболее решающими, а какие не были использованы. Нужно понимать, что важность признаков оценочна.

Далее будут приведены алгоритмы, основанные на деревьях решений. Многие свойства, указанные здесь, верны и для ансамблей. К примеру, мы по-прежнему можем получить важность признаков через усреднение по каждому дереву.

5.2 Случайный лес

Случайный лес (Random forest) представляет собой ансамбль деревьев решений, которые обучаются независимо [26]. Каждый из решающих деревьев, как правило, делают простым. Для принятия окончательного решения выбирается значение, за которое проголосовало большинство деревьев. Каждое дерево

в отдельности дает низкое качество, но за счет их большого числа результат получается хорошим. На Рис. 5 можно видеть пример случайного леса из двух деревьев.

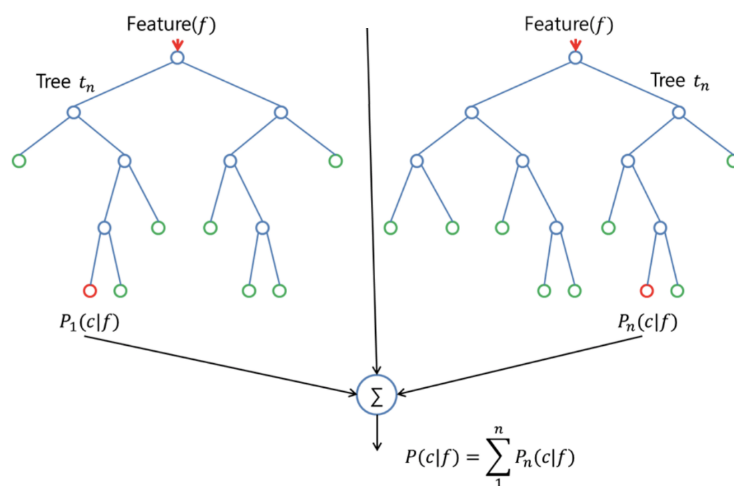

Рисунок 5 – Случайный лес из 2 деревьев

В построении деревьев на данных есть свои особенности. Укажем традиционный подход в построении дерева. Пусть N — количество примеров на обучении. Тогда:

1. выбирается подвыборка обучающей выборки размера N (возможно, с возвращением)

2. строится дерево. При поиске разбиения при построении каждого узла рассматриваются не все признаки, а только часть из них (sklearn реализация — корень из количества признаков)

3. дерево строится до полного исчерпания подвыборки, либо на основе каких-либо других критериев

Каждое из деревьев стараются сделать как можно проще. При принятии решения выбирается класс, за который проголосовало большинство деревьев решений.

Случайный лес дает сравнительно плохие результаты при разделении выборки в метрическом пространстве. В том случае, когда ожидается некоторый «правильный» вид разделяющей гиперплоскости. Тем не менее, у случайного

леса есть большое число достоинств, которые могут использоваться в рассматриваемой задаче определения вредоносности файла. Алгоритм не оперирует метриками, что позволяет свободно работать с признаками разной природы. Возможно обрабатывать данные с пропущенными значениями признаков. Высокая скорость работы алгоритма, независимое построение решающих деревьев.

5.3 Градиентный бустинг

При построении бустинга, базовые алгоритмы строятся последовательно, а не параллельно, компенсируя ошибку на предыдущей итерации [27]. В общем виде алгоритм обучения градиентного бустинга представлен ниже (Алгоритм 2). Здесь $L(a, y)$ — произвольная функция потерь, b_i — i-ый базовый алгоритм. f_i представляет собой i-ое приближение.

Имеем:

Линейную комбинацию базовых алгоритмов: $a(x) = \sum_{t=1}^{T} \alpha_t b_t(x)$. Это и есть алгоритм градиентного бустинга. Представляет линейную комбинацию базовых алгоритмов. Весь фокус состоит в том, как построить такой набор алгоритмов и подобрать веса.

Функционал качества с произвольной функцией потерь $L(a, y)$:

$$Q(\alpha, b; X^l) = \sum_{i=1}^{l} L\Big(\underbrace{\underbrace{\sum_{t=1}^{T-1} \alpha_t b_t(x_i)}_{f_{T-1,i}} + \alpha b(x_i)}_{f_{T,i}}, y_i\Big) \to \min_{\alpha, b}$$

$f_{T-1} = (f_{T-1,i})_{i=1}^{l}$ — текущее приближение

$f_T = (f_{T,i})_{i=1}^{l}$ — следующее приближение

На каждой итерации, при поиске очередного базового алгоритма, минимизируется данный функционал $Q(\alpha, b; X^l)$.

Data: обучающая выборка X^l, параметр T;

Result: базовые алгоритмы и их веса $\alpha_t b_t$, $t = 1, ..., T$;

инициализация: $f_i := 0, i = 1, ..., l$;

for $t \leftarrow 1$ **to** T **do**

базовый алгоритм, приближающий градиент:

$b_t := arg \min\limits_b \sum_{i=1}^{l} (b(x_i) + L'(f_i, y_i))^2$;

задача одномерной минимизации:

$\alpha_t := arg \min\limits_{\alpha > 0} \sum_{i=1}^{l} L(f_i + \alpha b_t(x_i), y_i)$;

обновление вектора значений на объектах выборки:

$f_i := f_i + \alpha_t b_t(x_i)$;

$i = 1, ..., l$;

end

Алгоритм 2: Алгоритм градиентного бустинга

В случае градиентного бустинга на решающих деревьях, очевидно, базовые алгоритмы представляют собой деревья решений. Параметр T — количество выстраиваемых деревьев.

Наиболее популярные реализации градиентного бустинга на решающих деревьях: XGBoost [18], LightGBM [28], CatBoost [29].

Xgboost представляет лишь наиболее удачную реализацию идей, предложенных ранее.

LightGBM алгоритм реализовывался с целью ускорить существующие реализации бустинга, а именно, XGBoost.

Основное время на построение алгоритма уходит на поиск лучшей точки разбиения тренировочных данных в узле по некоторому признаку. В XGBoost реализованы 2 механизма:

1. для нахождения лучшего разбиения признаки пресортируются и находится лучшее разбиение

2. алгоритм на основе гистограммы. признаки объектов разбиваются на группы

LightGBM предлагает 2 техники для многократного ускорения скорости работы алгоритма:

1. Gradient-based One-Side Sampling (GOSS). Для построения очередного дерева используются не все объекты. Удаляются из рассмотрения объекты с малыми градиентами

2. Exclusive Feature Bundling (EFB). Упаковка признаков. При большом количестве признаков данные с высокой вероятностью разрежены и есть возможность уменьшить количество эффективных признаков

В статье о LightGBM было проведено сравнение алгоритма с XGBoost как по скорости работы, так и по качеству на нескольких наборах данных. Разбирались задачи классификации и ранжирования. Было продемонстрировано от 2 - 20 кратное увеличение скорости работы алгоритма бустинга при той же точности на отложенной выборке.

CatBoost ставит своей задачей эффективную работу с категориальными признаками (catboost == categorial boosting), а также предлагает новую схему для подсчета значений в узлах, что позволяет уменьшить переобучение. Еще одно нововведение в том, что при построении очередного дерева разделение в узле может происходить по совокупности двух категориальных признаков. Поддержка GPU. В статье проводится сравнение с XGBoost, LightGBM, H2O реализациями. Показано, что CatBoost даже с параметрами по умолчанию позволяет добиться лучшего качества (Logloss) на 8 рассматриваемых наборах данных. Показаны результаты сравнения времен предсказания алгоритмов CatBoost, XGBoost, LightGBM. CatBoost оказался на порядок быстрее как в однопоточном режиме, так и в многопоточном. CatBoost находится в свободном доступе и активно развивается.

В силу того, что в нашей задаче присутствуют категориальные признаки, целесообразно исследовать также и эту реализацию.

6 Эксперименты

6.1 Получение данных

Для того, чтобы обучить классификатор требуется иметь размеченные данные. Для нашей задачи требуется получить набор чистых и вредоносных файлов PE формата. Проблемы получения чистых файлов нет. Можно использовать файлы системы Windows сразу после ее установки. Вредоносные же файлы возможно получить из сторонних источников. Был выбран сервис VirusTotal. Сервис позволяет получить вердикты 109 (на сегодняшний день) антивирусов по исследуемому файлу. Стоит отметить, что для исследуемого файла не по всем антивирусам может быть получен вердикт. Это происходит по нескольким причинам. К примеру, файл оказался в базе совсем не давно и еще не был обработан всеми антивирусами. Сервис предоставляет возможность как загрузить свой файл, так и получить ранее загруженные файлы с анализом антивирусов, используя платный API. На Рис. 6 показана статистика по загруженным файлам. Кроме PE формата, возможно загружать текст в разных форматах, архивы. На сегодняшний день в базе содержится более миллиона файлов PE формата.

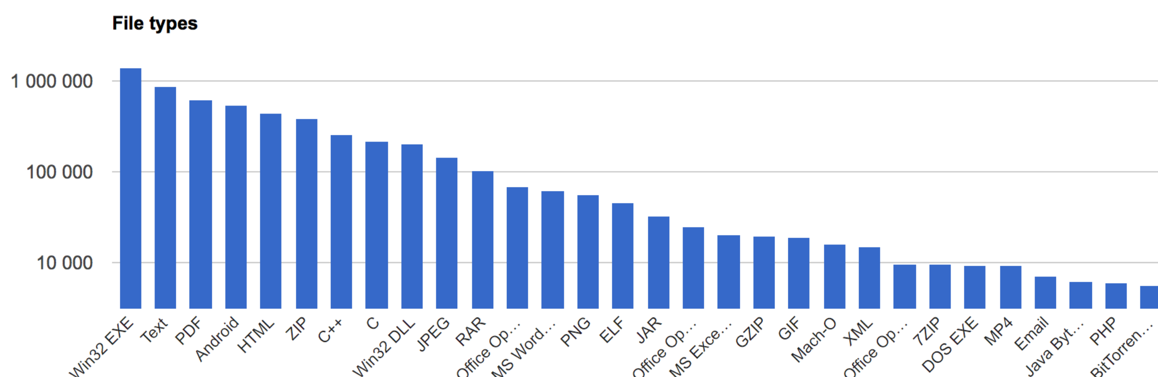

Рисунок 6 – Статистика сервиса VirusTotal по загруженным файлам

Было отмечено, что для исследуемого файла не всегда есть вердикты по всем антивирусам. Поэтому введем понятие «индекс вредоносности». «Индексом вредоносности» будем называть отношение числа антивирусов проголосовавших за вредоносность файла к количеству проголосовавших антивирусов по этому файлу. Таким образом, «индекс вредоносности» принимает значения от 0 до 1, где 1 означает, что все антивирусы, анализируемые файл, считают его

вирусом.

С помощью VirusTotal было получено 55432 PE-файлов (54299 exe файла, 1133 dll). Для каждого из них получены вердикты по антивирусам. На Рис. 7 показано распределение голосов антивирусов для полученных файлов. Для каждого файла было подсчитан «индекс вредоносности». Распределение файлов по индексу вредоносности мы видим на гистограмме.

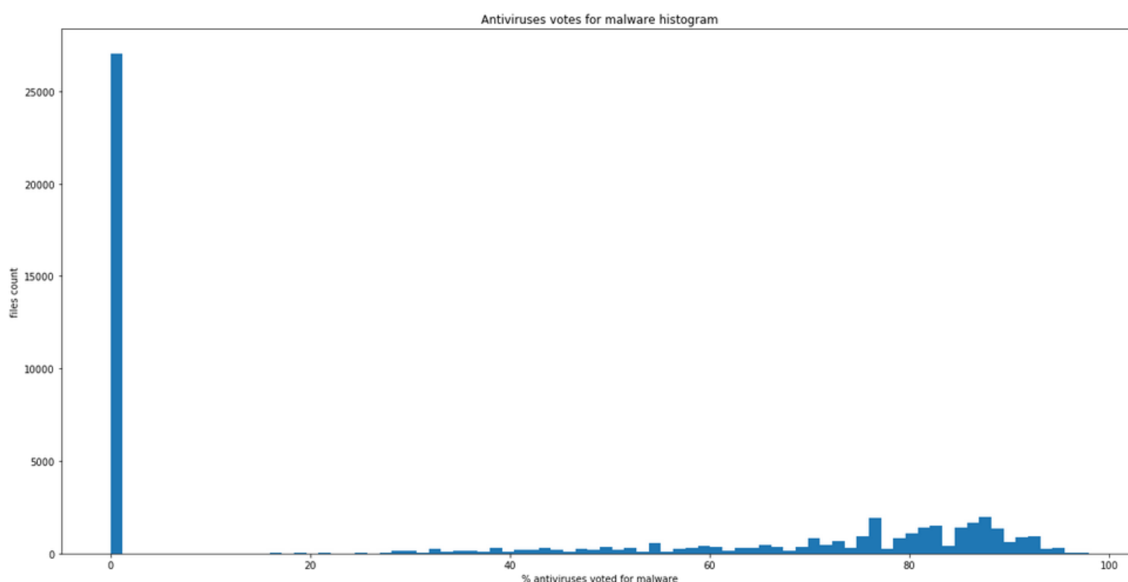

Рисунок 7 – Распределение файлов по голосам от всех антивирусов. По оси X отношение числа антивирусов проголосовавших за вредоносность файла ко всем голосовавшим антивирусам

Антивирусы устроены по-разному и выдают не всегда согласующиеся вердикты на исследуемых файлах, что видно на Рис. 7. То есть, с одной стороны, есть антивирусы, которые ошиблись, считая чистые файлы вирусами (false positive), и, существуют те, кто ошибся, считая вирус чистым файлом (false negative). При идеальных детектирующих алгоритмах антивирусов на гистограмме не было бы «хвоста» правее 100%. Выделяется проблема — какие файлы считать вредоносными при обучении алгоритма машинного обучения.

Для наиболее точного предсказания были выбраны топ-8 популярных и надежных антивирусов (Paloalto, SentinelOne, McAfee, BitDefender, Kaspersky, CrowdStrike, Avast, Symantec). Индекс вредоносности подсчитывается только для этих антивирусов. Соответствующая гистограмма на Рис. 8. При таком подходе около 5000 файлов по-прежнему остаются сомнительными (серая зона) — около половины из надежных антивирусов проголосовали за вредонос-

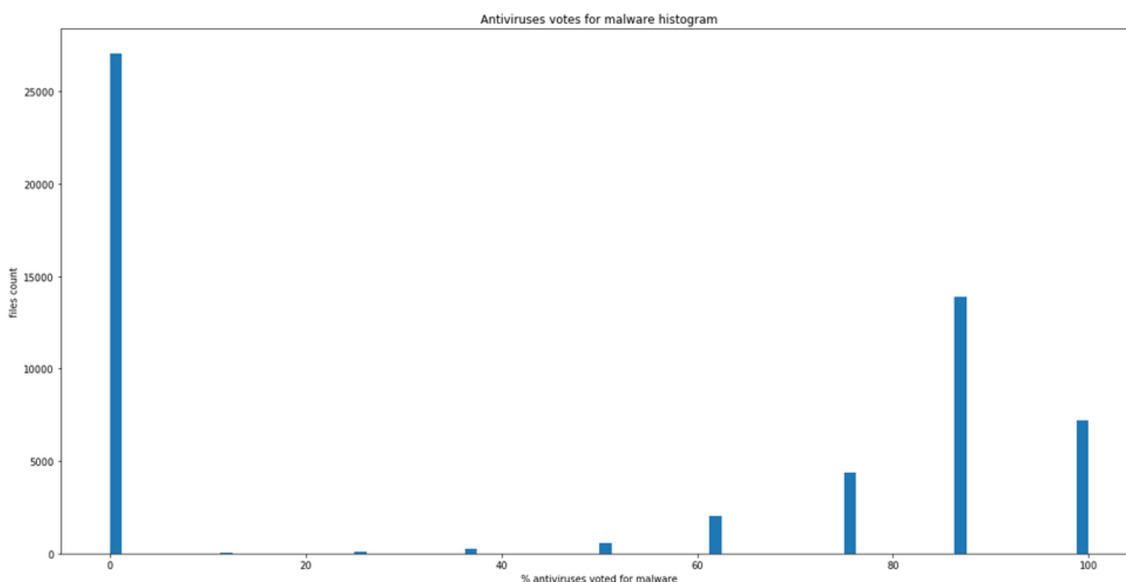

Рисунок 8 – Распределение файлов по голосам от надежных (топ-8) антивирусов. По оси X отношение числа антивирусов проголосовавших за вредоносность файла ко всем голосовавшим антивирусам

ность. Определим файл вредоносным, если хотя бы один из топ-8 антивирусов определил его таковым. При таком подходе мы пытаемся максимально усилить детектирование вирусов, уменьшая количество ложно отрицательных срабатываний. Заметим, что большинство алгоритмов классификации поддерживает возможность помимо индекса предсказанного класса выдавать вероятность отнесения объекта к классу, то есть, мы имеем еще один инструмент воздействия на предсказания уже после обучения алгоритма (считать вредоносным файлом, если вероятность больше 30%, а не больше 50%).

Таким образом, мы получаем следующие данные (размер подобран так, чтобы данные были сбалансированы) в таблице 2.

Таблица 2 Статистика по собранным файлам

Все файлы	55432
Чистые файлы	27049
Вредоносные файлы	28383

На Рис. 9 приведена гистограмма распределения размеров файлов скачанных из VirusTotal как для чистых, так и для вредоносных файлов. Видим,

что в основном файлы имеют размер в 500-1000 килобайт и очень мало файлов имеют размер порядка 10 мегабайт. Также, стоит отметить, что вредоносное ПО, как правило, имеет меньший размер файлов.

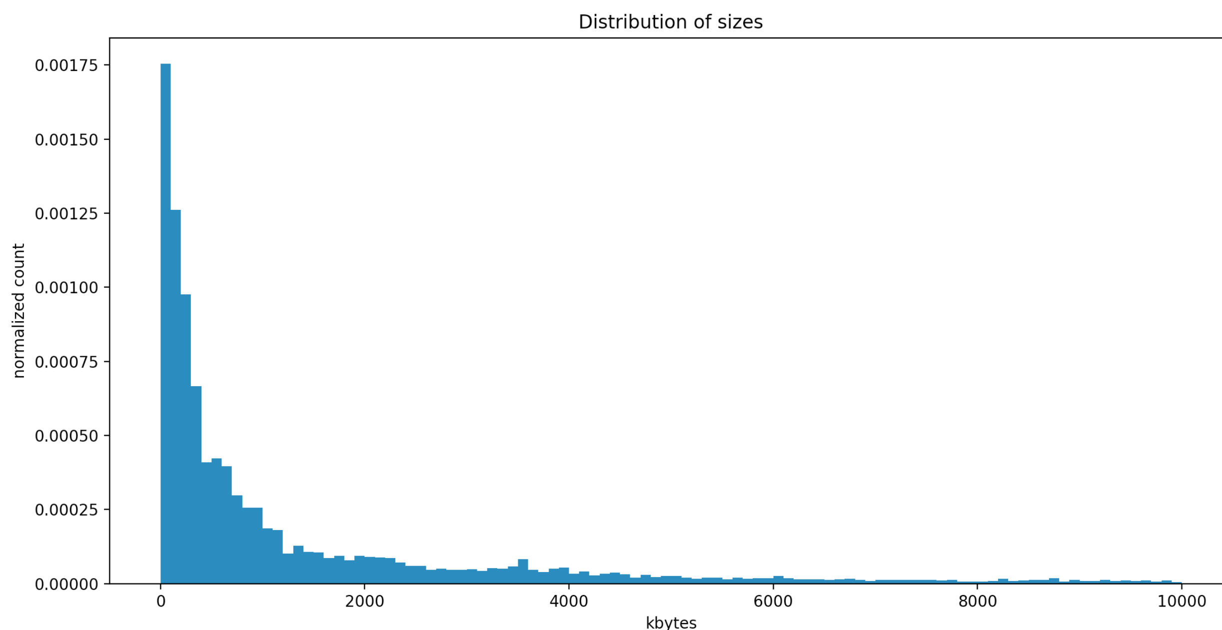

Рисунок 9 – Гистограмма распределения размеров полученных файлов. По оси X размер файла в килобайтах

6.2 План экспериментов

Опишем процесс проведения экспериментов. Перед проведением экспериментов разобьем данные (файлы) на обучающую выборку и тестовую в отношении 5 к 1 сбалансированно (с одинаковым отношением количества вредоносных файлов к чистым). Подбор признаков, гиперпараметров будем выполнять на обучающей выборке. Сравнение алгоритмов машинного обучения будем проводить на результатах с тестовой выборки, что позволит наиболее честно провести эксперимент, избегая переобучения на тестовых данных.

Любой алгоритм машинного обучения содержит гиперпараметры — некоторый набор параметров, который влияет на модель, и, в конечном счете, качество обучения. Так, например, для случайного леса это количество деревьев в лесу, глубина деревьев, количество признаков, рассматриваемых при разбиении выборки в очередном узле дерева и другие. Для поиска гиперпараметров воспользуемся 3-fold кросс-валидацией на обучающей выборке. То есть, разобьем

всю обучающую выборку на 3 сбалансированные части, трижды обучим алгоритм на 2 частях и вычислим значение метрик на одной оставшейся. Усредним полученные значения. Пример для k = 10 представлен на Рис. 10. Для каждого алгоритма с наилучшим набором гиперпараметров будем вычислять accuracy (точность) и false positive rate (долю ложно положительных срабатываний) на тестовой (отложенной) выборке для сравнения моделей между собой. В силу того, что нам удалось получить сбалансированные данные, accuracy представляется удачной метрикой для оценки качества алгоритмов. Иначе, стоило бы выбрать иные способы оценки, так, например, f1 меру или площадь под AUC кривой. Мотивация расчета false positive rate (FPR) состоит в том, что нам важно знать количество чистых файлов, которые по ошибке отнесены к вредоносным. Таких файлов должно быть как можно меньше. Такое ложное определение может привести к порче файлов, потере данных.

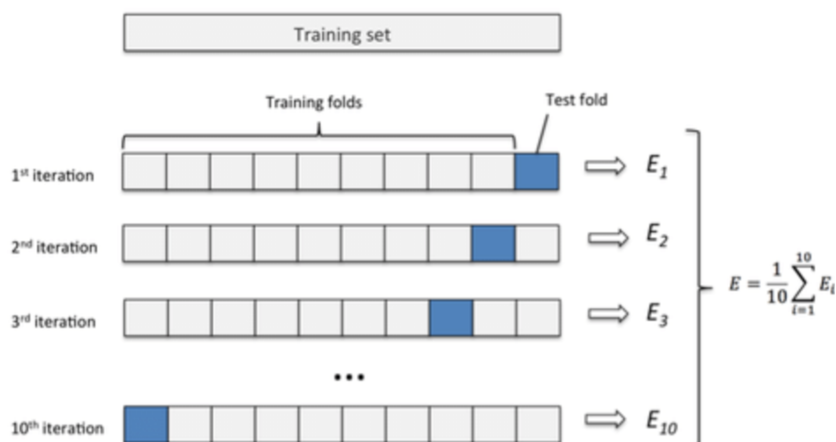

Рисунок 10 – Пример k-fold кросс-валидации при k = 10

6.3 Отбор признаков

Ранее (PE формат) мы определили каким образом и какого типа атрибуты будем извлекать из PE файла. Извлечем максимально возможное число признаков из файла, а затем отбор будем проводить жадным алгоритмом. То есть, на каждом шаге будем добавлять к уже имеющимся тот атрибут, который дает наибольший прирост точности (accuracy) модели/классификатору. Так будем делать до тех пор, пока качество перестанет расти. Выполняем отбор на обучающей выборке 3-fold кросс-валидацией. Для отбора признаков в качестве алгоритма будем использовать случайный лес.

Дополнительно к атрибутам будем вычислять энтропию секций.

Энтропия рассчитывается следующим образом (энтропия Шеннона) [30]:

$$H = -\sum_{i=1}^{n} p_i \log p_i$$

Основание логарифма берем равным 256 — число возможных значений байта. Таким образом, энтропия H будет принимать значения от 0 до 1.

Высокая энтропия секции может сигнализировать о сжатии секции, использовании упаковщиков, шифровальщиков для скрытия вредоносного кода.

Собранные атрибуты с файлов:

1. численные значения атрибут, взятые из полей PE файла напрямую
2. флаги присутствия api, секций и их характеристики. Значения: $\{0, 1\}$
3. энтропия секций

После работы жадного алгоритма мы получили набор из 135 признаков, на основе которых возможно детектировать вредоносное ПО.

Полный перечень собранных атрибут файлов представлен в Приложении А. Собираем прямые атрибуты структуры PE файла с MS-DOS заглушки, coff заголовка, опционального заголовка. Для фиксированного количества dll библиотек и импортируемых api были выделены предложенные в приложении названия. Было решено выделить 5 секций с названиями: text, data, rsrc, rdata, reloc и получить характеристики — права секции (shared, execute, read, write) и их энтропию. Ранее оговаривалось, что названия секций для загрузчика не имеют значения, однако данные, «стандартные» секции встречаются у большинства файлов и содержат, как правило то, на что и указывают. Такие отклонения движок будет отлавливать. Вместе с напрямую извлеченными параметрами будет собирать общую информацию по самим структурам. К примеру, количество импортируемых библиотек, названий функций, секций, символов. Назовем категорию таких признаков — general.

Можно выделить группы атрибутов в зависимости от структуры их извлечения. Отдельно будем рассматривать характеристики секций и их энтропии. После выделения таких групп возможно построить график их важности. Важность группы считается как сумма важностей признаков, входящих в группу.

На Рис. 11 представлена значимость групп признаков. По графику видно, что энтропия секций вместе с признаками опционального заголовка играет решающую роль в детекции вируса, заражения файла. Наименее важна информация о DOS заголовке, coff заголовке и характеристиках секций.

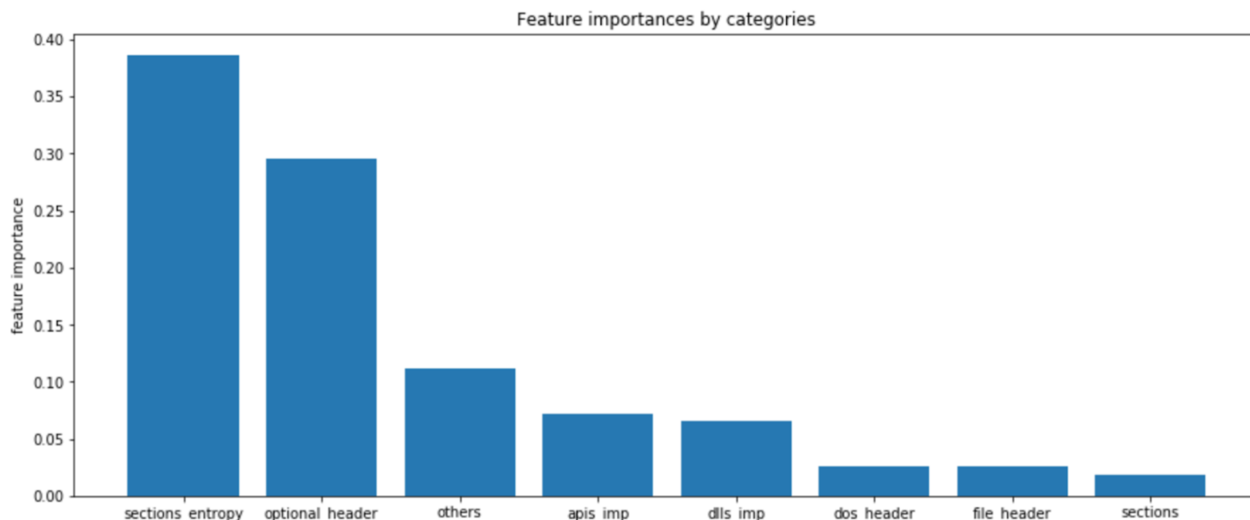

Рисунок 11 – Значимость признаков, рассчитанная по случайному лесу

6.4 Сравнение алгоритмов

После того, как найдены определяющие признаки и зафиксированы данные требуется найти наилучшую модель, лучший алгоритм классификации. Будем рассматривать 4 выбранных модели: случайный лес (Random Forest), XGBoost, LightGBM, CatBoost. Все эксперименты проводим на языке Python в среде ipython notebook с соответствующими инструментами и библиотеками. Для каждой модели поиск гиперпараметров осуществляем по 3-fold кросс валидации. После подбора гиперпараметров обучаем модель на всей предоставленной тренировочной выборке и вычисляем accuracy и false positive rate на тестовых данных. Получившиеся результаты приведены в таблице 3.

В дополнение к сводной таблице качества алгоритмов приведем зависимость качества алгоритмов от размера обучающей выборки — Рис. 12. По кривым можно заметить (аппроксимация), что увеличение обучаемых данных может улучшить качество детектирования вирусов (кривые не достигли насыщения).

Лучшим из рассмотренных алгоритмов в применении к данной задаче

Таблица 3 Сравнение алгоритмов

Алгоритм	Accuracy %	False positive rate %
Случайный лес	95.9	1.9
XGBoost	96.3	1.4
LightGBM	96.5	1.4
CatBoost	96.3	1.5

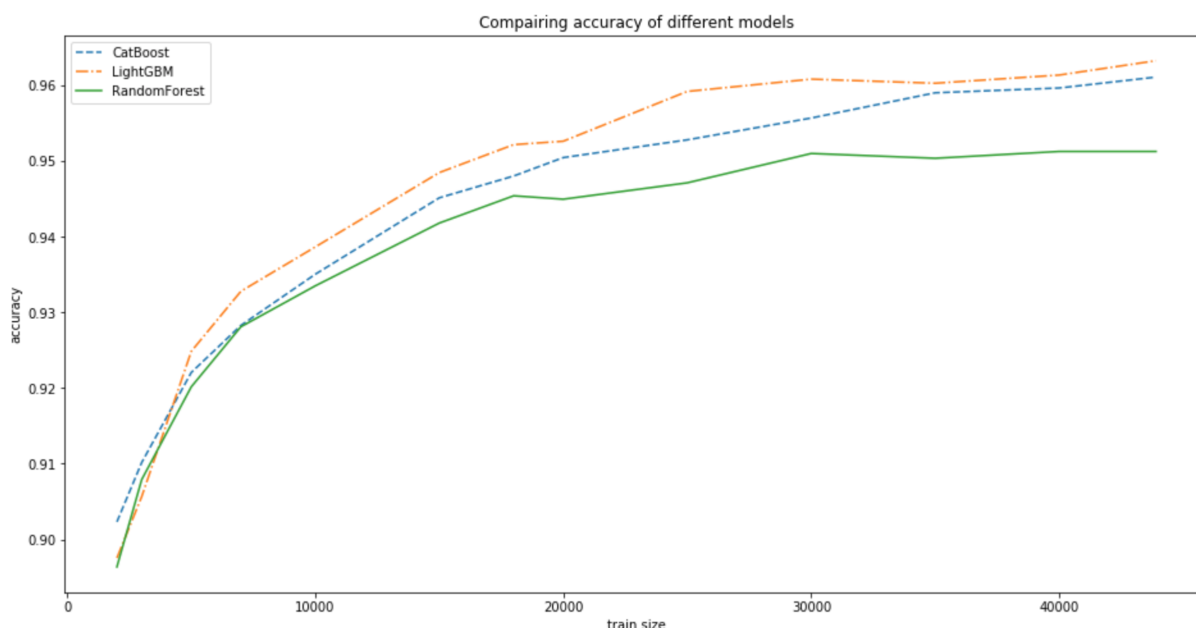

Рисунок 12 – Зависимость качества алгоритма от размера данных

оказался градиентный бустинг с LightGBM реализацией. Удалось добиться лучшей, чем требуемая точности (accuracy) и хорошей false positive rate. CatBoost даже с подбором гиперпараметров не оказался лучшим.

Взяв за основу алгоритм градиентного бустинга с LightGBM реализацией был реализован движок на C++ и интегрирован в продукт. Помимо качества работы алгоритма другая важная характеристика — скорость его работы. Была измерена скорость работы движка для проверки удовлетворения требованиям в реальных условиях. Результаты представлены в таблице 4. Скорость вычислялась на 50 тысячах файлах, расположенных на HDD диске. При запуске на SSD диске скорость работы классификатора не изменилась, что логично, а PE-парсер стал работать в среднем за 5 мс на одном файле.

На Рис. 13 представлено распределение времени обработки движком файлов. Видим, что, в основном, файл обрабатывается менее чем за 20 мс. Также проанализируем долю файлов, обрабатываемых за требуемое время. А именно, долю файлов, обрабатываемых за время менее 30 мс, если бы ограничение на время было жестким и применялось к каждому файлу. В этом случае будут обработаны 90% всех файлов, а 10% — проигнорированы. Стоит понимать, что в реальных условиях ограничения вводятся не на один файл, а на несколько, исходя из среднего времени, а также из того, что большинство вирусов, инфицированных файлов имеют малый размер файла и, соответственно, малое время работы алгоритма. Среднее время обработки файла — 21 мс, что является хорошим показателем и удовлетворяет требованию.

На Рис. 14 представлена зависимость скорости работы движка от размера файла. Видим, что зависимость времени обработки файла от его размера линейна. При извлечении атрибутов из PE файла мы должны пройтись по таблице импорта, по секциям для получения энтропий. Логично предполагать, что чем размер файла больше, тем больше становятся секции и таблицы импорта, содержащие названия dll и api. Имея размер исследуемых файлов, можно приблизительно оценить время их обработки. Так, на 100 gb данных требуется порядка 10 минут.

Таблица 4 Среднее время работы составных частей движка

	Время работы, ms
PE-парсер	16.0
Классификатор	0.5
Общее время	16.5

Получившиеся результаты оптимистичны и показывают применимость движка на практике. Также интересно посмотреть за такой характеристикой, как false negative rate. Доля вирусов, которые движок не смог обнаружить. Ранее считалось, что файл является вирусом, если за это проголосовал хотя бы один из надежных антивирусов. Теперь построим распределение индекса вредоносности от всех антивирусов для false negative файлов. Гистограмма приведена

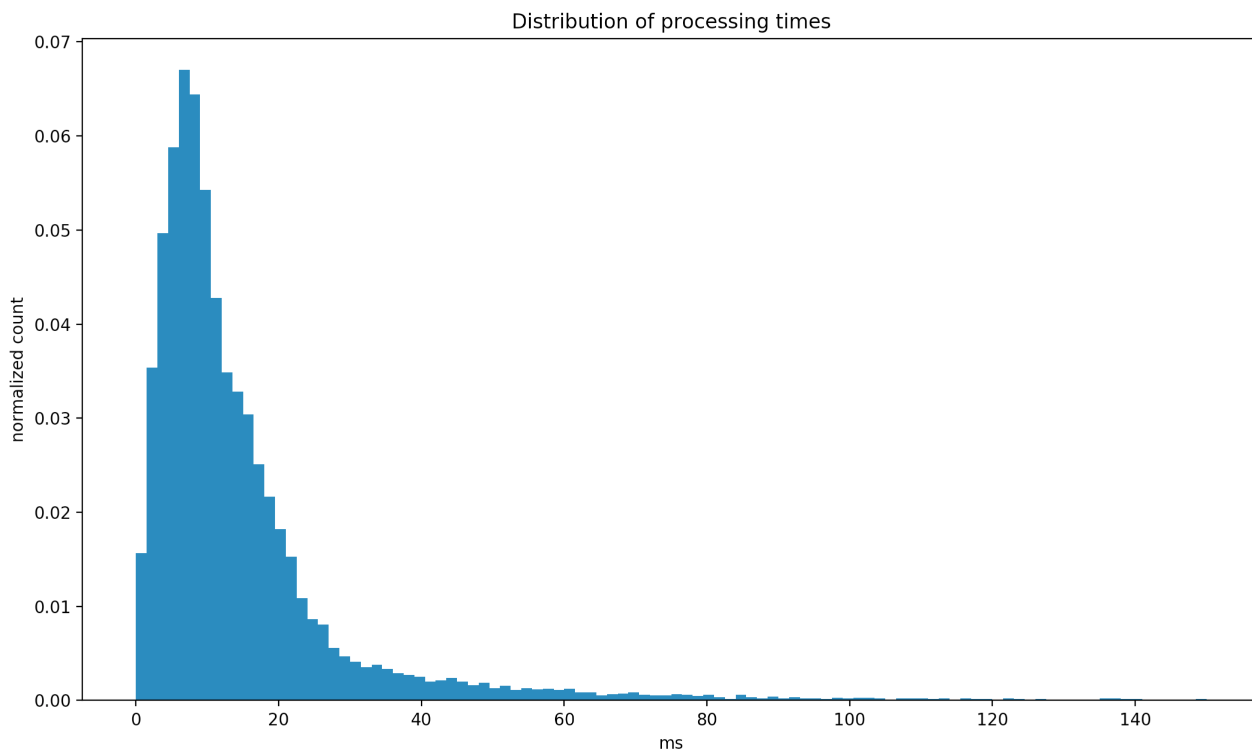

Рисунок 13 – Распределение времени работы движка для файлов различного размера

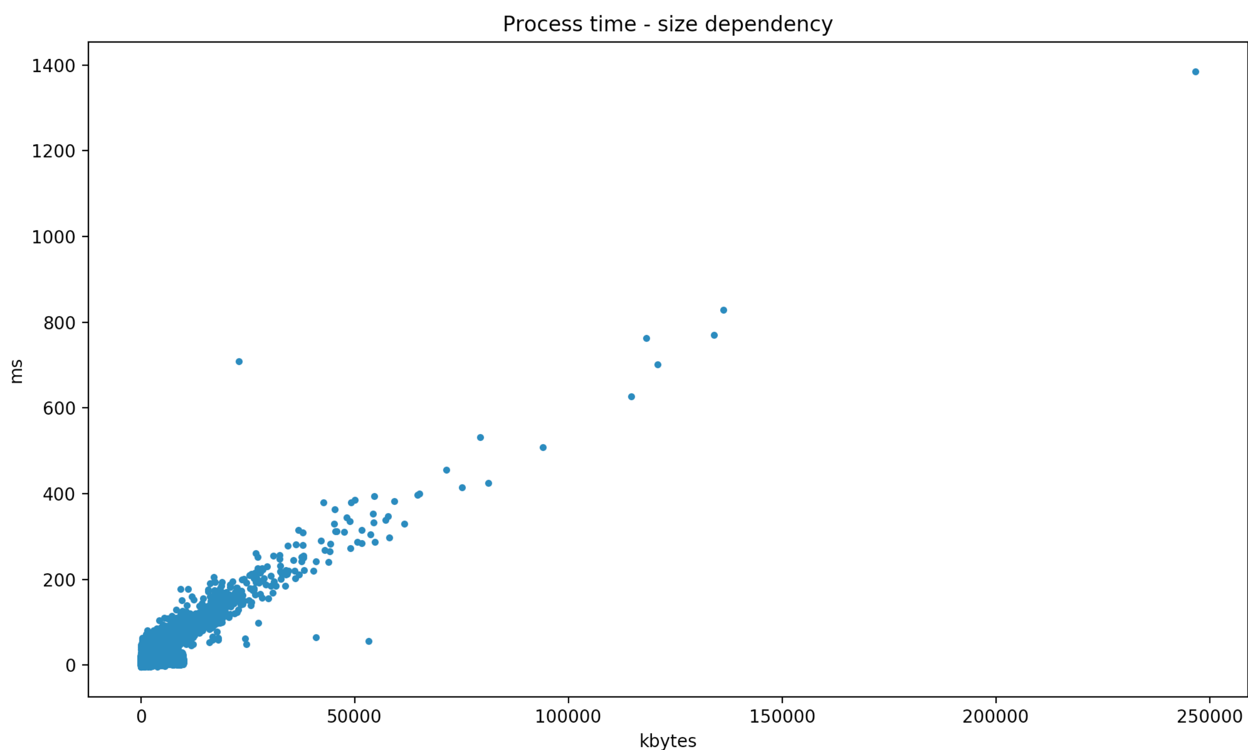

Рисунок 14 – Зависимость скорости работы движка от размера файла

на Рис. 15. Можно сравнить данное распределение с представленным ранее для всех файлов (Рис. 7). На Рис. 15 распределение более равномерно. несмотря

на общее смещение вправо (центр масс примерно у 0.6), многие антивирусы оказались неспособны верно определить эти случаи. Для улучшения качества алгоритма возможно сконцентрироваться на таких серых файлах и определить их особенности.

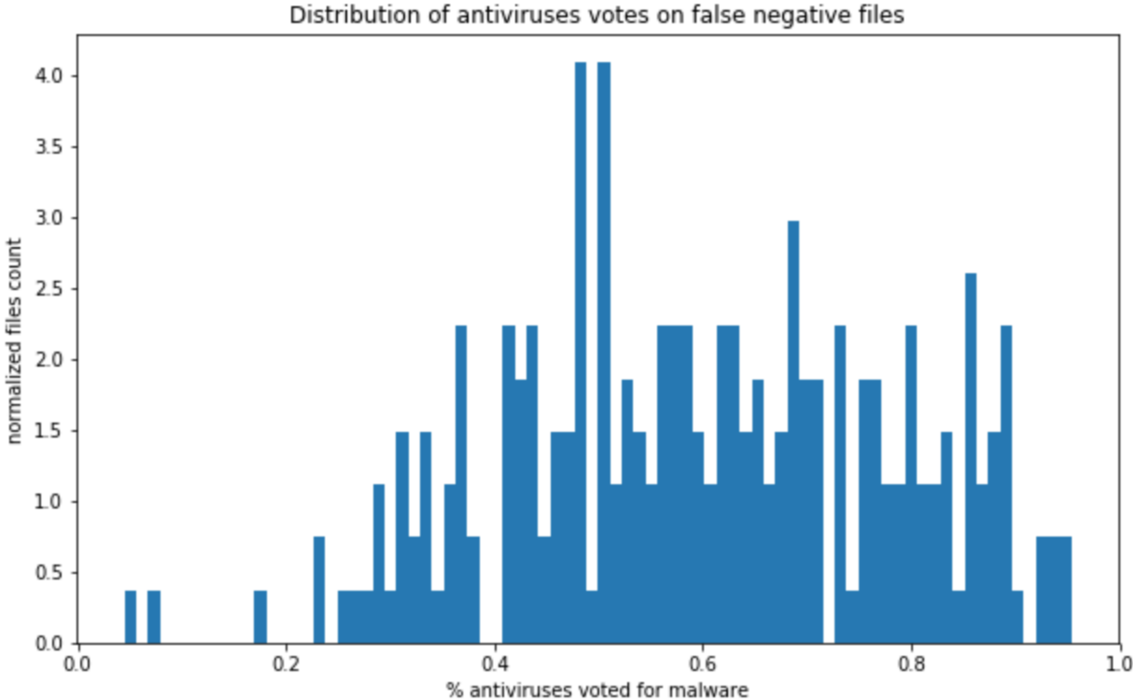

Рисунок 15 – Распределение голосов антивирусов по false negative файлам

7 Заключение

В ходе работы был получен ряд результатов:

1. Выполнен обзор применяемых методов статического анализа определения вредоносных программ. Область является активно развивающейся в данное время, происходит переход от эвристик к применению методов машинного обучения. Выделены основные принципы построения статического анализа. Существует большое число работ по созданию эвристик для детекции вредоносного кода. Во многих работах не освещается практическое применение алгоритмов машинного обучения

2. Получены данные для анализа (более 50000 файлов), построения движка статического анализа. Представлен метод разбиения файлов на чистые и вредоносные

3. Произведен отбор наиболее значимых признаков. С полным списком полученных признаков можно ознакомиться в Приложении А

4. Произведено сравнение современных алгоритмов машинного обучения в рамках данной задачи в условиях ограниченных ресурсов и времени. Выбран наилучший алгоритм по метрике точности (accuracy), удовлетворяющий выдвинутым требованиям качества

5. Реализован движок детектирования вредоносного кода в рамках программного продукта для конечных пользователей, удовлетворяющий предъявляемым требованиям скорости работы

Ценным практическим результатом работы стало создание эффективного по скорости и качеству движка машинного обучения.

Проектирование и создание движка выполнялось на основе результатов проведенных исследований по отбору наиболее значимых признаков, выбору подходящего алгоритма. Требования к скорости работы модели накладывали ограничения на выбор алгоритма, количество извлекаемых признаков. В ходе работы было получено представление о значимости отдельных признаков и групп признаков для детектирования вредоносного ПО.

Полученную экспертизу можно использовать для построения более сложных систем детектирования вирусов. Представим здесь лишь небольшой перечень возможных дальнейших шагов по развитию системы статического анализа по защите данных пользователя:

1. Переход к модели клиент-сервер. Модель предназначена для переноса всей сложной работы с данными на сервер. Пользователь отправляет на сервер набор признаков для расчета угрозы файла. Это позволяет снять ограничения на выбор алгоритма

2. Добавление возможности расшифровки файлов. Вредоносный код может быть зашифрован. Возможность получить исходный код приведет к улучшению детекции моделью вирусов

3. Применение глубинных методов машинного обучения. В условиях клиент-серверной архитектуры становятся доступны более сложные методы анализа файлов. Некоторые исследования уже были выполнены в этом направлении. Исследование возможности применения сверточных, рекуррентных сетей для детектирования вредоносного ПО

Список литературы

[1] Mcafee labs threats report. — 2015. — URL: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2014.pdf

[2] Symantec intelligent report. — 2015. — URL: https://www.symantec.com/content/en/us/enterprise/other_resources/intelligence_report_05-2015.en-us.pdf

[3] *Jinrong Bai, Junfeng Wang, Guozhong Zou.* A Malware Detection Scheme Based on Mining Format Information // The Scientific World Journal. — 2014. — Vol. 2014.

[4] Eureka: A Framework for Enabling Static Malware Analysis / Sharif M. [et al.] // Recent Advances in Intrusion Detection, Lecture Notes in Computer Science. — 2008. — Vol. 5283. — Pp. 481-500.

[5] *Hahn K.* Robust Static Analysis of Portable Executable Malware // HTWK Leipzig. — 2014.

[6] Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification / Ahmadi M. [et al.] // In Proceedings of the 6 ACM Conference on Data and Application Security and Privacy / ACM. — 2016. — Pp. 183-194.

[7] An intelligent PE-malware detection system based on association mining / Ye Y. [et al.] // Journal in Computer Virology. — 2008. — Vol. 4, no. 4. — Pp. 323-334.

[8] PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime / Shafiq M. Z. [et al.] // Recent Advances in Intrusion Detection, Lecture Notes in Computer Science — 2009 — Vol. 5758 — Pp. 121-141.

[9] *Tabish S. M., Shafiq M. Z., Farooq M.* Malware detection using statistical analysis of byte-level file content // In Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics / ACM. — 2009. — Pp. 23-31.

[10] Automatic generation of string signatures for malware detection / Griffin K. [et al.] // In Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection. — 2009. — Pp. 101-120.

[11] *Hu X., Chiueh T.-c., Shin K. G.* Large-scale malware indexing using function-call graphs // In Proceedings of the 16th ACM Conference on Computer and Communications Security / ACM. — 2009. — Pp. 611-620.

[12] Malware detection based on mining api calls / Sami A. [et al.] // In Proceedings of the 2010 ACM Symposium on Applied Computing / ACM. — 2010. — Pp. 1020-1025.

[13] Manjunath. Malware images: Visualization and automatic classification / Nataraj L. [et al.] // In Proceedings of the 8th International Symposium on Visualization for Cyber Security / ACM. — 2011. — Pp. 41-47.

[14] A static, packer-agnostic filter to detect similar malware samples / Jacob G. [et al.] // In Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. — 2013. — Pp. 102-122.

[15] Opcode sequences as representation of executables for data-mining-based unknown malware detection / Santos I. [et al.] // Information Sciences. — 2013. — Vol. 231 — Pp. 64-82.

[16] Novel active learning methods for enhanced PC malware detection in windows OS / Nissim N. [et al.] // Expert Systems with Applications. — 2014. — Vol. 41, no. 13. — Pp. 5843-5857.

[17] DLLMiner: structural mining for malware detection / Narouei M. [et al.] // Security and communication networks. — 2015. — Vol. 8, no.18. — Pp. 3311-3322.

[18] *Chen T., C. Guestrin C.* XGBoost: A Scalable Tree Boosting System // Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. — 2016. — Pp. 785-794.

[19] *Breiman L.* Bagging predictors // Machine Learning. — 1996. — Vol. 24, no. 2 — Pp. 123-140.

[20] A survey on Heuristic Malware Detection Techniques / Bazrafshan Z. [et al.] // 5th Conference on Information and Knowledge Technology. — 2013. — Pp. 113-120.

[21] *Ucci D., Aniello L., Baldoni R.* Survey on the Usage of Machine Learning Techniques for Malware Analysis // arXiv preprint arXiv:1710.08189. — 2018.

[22] Microsoft Corporation. PE Format specification. — URL: `https://msdn.microsoft.com/library/windows/desktop/ms680547(v=vs.85).aspx?id=19509`

[23] *Касперски К.* Путь воина – внедрение в pe/coff-файлы. — 2004 — URL: `http://samag.ru/archive/article/297`

[24] Portable Executable Wiki. — URL: `https://en.wikipedia.org/wiki/Portable_Executable`

[25] Microsoft Corporation. What is a DLL? — 2007 — URL: `https://support.microsoft.com/kb/815065/EN-US`

[26] *Воронцов К.* Курс лекций. — URL: `http://www.machinelearning.ru/wiki/images/3/3e/Voron-ML-Logic.pdf`

[27] *Воронцов К.* Курс лекций. Презентации. — URL: `http://www.machinelearning.ru/wiki/images/0/0d/Voron-ML-Compositions.pdf`

[28] LightGBM: A Highly Efficient Gradient Boosting Decision Tree / Ke G. [et al.] // Advances in Neural Information Processing Systems 30. — 2017.

[29] *Dorogush A. V., Ershov V., Gulin A.* CatBoost: gradient boosting with categorical features support. — 2017.

[30] *Shannon C. E.* A Mathematical Theory of Communication // Bell System Technical Journal. — 1948. — Vol. 27. — Pp. 379-423.

Приложение 1 (список используемых признаков)

1. MS-DOS Stub/e_lfanew
2. MS-DOS Stub/e_crlc
3. Coff Header/NumberOfSections
4. Coff Header/NumberOfSymbols
5. Coff Header/PointerToSymbolTable
6. Coff Header/SizeOfOptionalHeader
7. Coff Header/Characteristics
8. Optional Header/SizeOfInitializedData
9. Optional Header/SizeOfUninitializedData
10. Optional Header/BaseOfCode
11. Optional Header/DllCharacteristics
12. Optional Header/SizeOfStackReserve
13. Optional Header/SizeOfHeapReserve
14. Optional Header/SectionAlignment
15. Optional Header/SizeOfHeapCommit
16. Optional Header/FileAlignment
17. Optional Header/SizeOfImage
18. Optional Header/NumberOfRvaAndSizes
19. Optional Header/SizeOfHeaders
20. Optional Header/SizeOfStackCommit
21. DLL Referred/Gdiplus.dll

22. DLL Referred/Rpcrt4.dll
23. DLL Referred/Comdlg32.dll
24. DLL Referred/Wtsapi32.dll
25. DLL Referred/Imm32.dll
26. DLL Referred/Mscoree.dll
27. DLL Referred/Gdi32.dll
28. DLL Referred/Oleacc.dll
29. DLL Referred/Version.dll
30. DLL Referred/Userenv.dll
31. DLL Referred/Msvcrt.dll
32. DLL Referred/Shell32.dll
33. DLL Referred/Wintrust.dll
34. DLL Referred/Uxtheme.dll
35. DLL Referred/Wininet.dll
36. DLL Referred/Iphlpapi.dll
37. DLL Referred/Winspool.drv
38. DLL Referred/Winmm.dll
39. DLL Referred/Setupapi.dll
40. DLL Referred/Advapi32.dll
41. DLL Referred/Ole32.dll
42. DLL Referred/Shlwapi.dll
43. DLL Referred/Ntdll.dll

44. DLL Referred/Mpr.dll
45. DLL Referred/Netapi32.dll
46. DLL Referred/User32.dll
47. DLL Referred/Oleaut32.dll
48. DLL Referred/Urlmon.dll
49. DLL Referred/Comctl32.dll
50. DLL Referred/Crypt32.dll
51. DLL Referred/Ws2_32.dll
52. DLL Referred/Psapi.dll
53. DLL Referred/Msimg32.dll
54. API Referred/Createfilew
55. API Referred/Createfilea
56. API Referred/Getmodulehandlew
57. API Referred/Getmodulehandlea
58. API Referred/Getmodulehandleexw
59. API Referred/Heapsetinformation
60. API Referred/Loadlibrarya
61. API Referred/Rtllookupfunctionentry
62. API Referred/Virtualalloc
63. API Referred/Openprocesstoken
64. API Referred/Openprocess
65. API Referred/Rtlcapturecontext

66. API Referred/Setunhandledexceptionfilter
67. API Referred/Getenv
68. API Referred/Createprocessasusera
69. API Referred/Createprocessasuserw
70. API Referred/Duplicatehandle
71. API Referred/Loadicona
72. API Referred/Virtualprotect
73. API Referred/Virtualprotectex
74. API Referred/Writeprocessmemory
75. API Referred/Createdirectorya
76. API Referred/Createdirectoryw
77. API Referred/Rtlvirtualunwind
78. API Referred/Openfile
79. API Referred/Reopenfile
80. API Referred/Ntcreatefile
81. API Referred/Ntsetinformationfile
82. API Referred/Setfileinformationbyhandle
83. API Referred/Movefilea
84. API Referred/Movefilew
85. API Referred/Movefileexa
86. API Referred/Movefileexw
87. API Referred/Copyfilea

88. API Referred/Copyfilew
89. API Referred/Copyfileexw
90. API Referred/Copyfileexa
91. API Referred/Ntopenfile
92. API Referred/Deletefilew
93. API Referred/Deletefilea
94. API Referred/Findfirstfilea
95. API Referred/Findfirstfilew
96. API Referred/Findfirstfileexa
97. API Referred/Findfirstfileexw
98. API Referred/Replacefilea
99. API Referred/Replacefilew
100. API Referred/Writefile
101. API Referred/Writefileex
102. API Referred/Suspendthread
103. API Referred/Ntsuspendprocess
104. API Referred/Ntsuspendthread
105. Section/text : характеристики * 4, энтропия
106. Section/data : характеристики * 4, энтропия
107. Section/rsrc : характеристики * 4, энтропия
108. Section/rdata : характеристики * 4, энтропия
109. Section/reloc : характеристики * 4, энтропия

110. General/NumberOfReferredDLLs
111. General/NumberOfReferredAPIs
112. General/NumberOfReferredAPIOrdinals
113. General/NumberOfSections
114. General/NumberOfExportTableSymbols
115. General/NumberOfRelocSectionItems