Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

1. —
2. **Институт кибербезопасности и защиты информации**

**КУРСОВАЯ РАБОТА**

1. «Выявление полиморфного и метаморфного кода»
2. по дисциплине «Технологии реверс-инжиниринга программного обеспечения»
3. Выполнил
4. студент гр. 4851001/00201 Милютин Н.А.

<*подпись*>

1. Преподаватель Жуковский Е.В.

<*подпись*>

ОГЛАВЛЕНИЕ

Введение3

ГЛАВА 1. МЕТОДЫ ЗАЩИТЫ ВРЕДОНОСНЫХ ПРОГРАММ2

1.1. Эволюция полиморфных вирусов4  
1.1.1. Простые вирусы4  
1.1.2. Зашифрованные вирусы5  
1.1.3. Полиморфные вирусы.7

1.2. Пермутация.8

1.3. Метаморфные вирусы.8

1.4. Используемые методы трансформации кода.10

ГЛАВА 2. МЕТОДЫ ВЫЯВЛЕНИЯ ПОЛИМОРФНОГО И МЕТАМОРФНОГО КОДА12

2.1. Развитие идей12

2.2. Метод эмуляции13  
 2.3. Эвристический метод14

ГЛАВА 3. АНАЛИЗ БИНАРНЫХ ФАЙЛОВ16  
 3.1. BinDiff 16  
 3.2. BinHunt17  
 3.3. PatchDiff18  
 3.4. TurboDiff18  
 3.5. IdaCompare19  
 3.6. Результаты сравнение программ20

ГЛАВА 4. ПРАКТИЧЕСКАЯ ЧАСТЬ22  
 4.1. Методы сравнения применяемые при разработке программы 22  
 4.1.1. Классические сравнения22  
 4.1.2. Построение графа потока управления22  
 4.1.3. Control Flow Graph23  
 4.1.4. Annotated Control Flow Graph 25  
 4.2.Реализация программы 28

4.3. Сравнение реализованной программы с BinDiff 32

**ЗАКЛЮЧЕНИЕ 36**

**СПИСОК ЛИТЕРАТУРЫ37**

**ВВЕДЕНИЕ**

Изучение методов защиты от вредоносного программного обеспечения – одна из ключевых сфер в кибербезопасности. В современных реалиях очень просто заразить свой компьютер вирусом, «червем» или же фишинговым ПО.

Способы обнаружения вредоносного ПО бывают разными, но главным помощником в борьбе против заражения являются антивирусы. В свою очередь вирмейкеры, пытаясь обойти существующие методы защиты, создают новые механизмы для сокрытия функциональности вирусов, такие как полиморфизм, метаморфизм, упаковщики и крипторы. Эти методы придают вредоносному приложению дополнительную функциональность, которая позволяет ему изменять свой код во время распространения или работы для того, чтобы избежать обнаружения. Всё вышеперечисленное говорит о важности и актуальности проведения исследований в данной теме.

В курсовой работе будут рассмотрены методы обнаружения полиморфного и метаморфного ПО, а также будет реализована программа для выявления полиморфного кода на основе сравнения исполняемых файлов.

**ГЛАВА 1. МЕТОДЫ ЗАЩИТЫ ВРЕДОНОСНЫХ ПРОГРАММ**

**1.1. Эволюция полиморфных вирусов.**

Полиморфные компьютерные вирусы [1] являются наиболее сложными и трудно обнаруживаемыми вирусами, часто требующими от разработчиков антивирусных программ большого количества времени на создание процедур обнаружения, необходимых для поимки одного полиморфного вируса.

**1.1.1. Простые вирусы**

Компьютерный вирус - это самовоспроизводящаяся компьютерная программа, которая работает без согласия пользователя. Он распространяется путем присоединения своей копии к какой-либо части программного файла, например, электронной таблицы или текстового редактора. Вирусы также атакуют главную загрузочную запись (MBR), которая содержит информацию, используемую компьютером для запуска.

Большинство вирусов просто реплицируются. Некоторые выводят сообщения. Некоторые, конечно, передают полезную нагрузку - часть вирусной программы, которая предназначена для повреждения программ, удаления файлов, переформатирования жесткого диска или сбоя корпоративной сети, что, собственно, и приследует злоумышленник.

Простой вирус, который просто воспроизводит себя, легче всего обнаружить. Если пользователь запускает зараженную программу, вирус получает контроль над компьютером и присоединяет свою копию к другому программному файлу. После распространения вирус передает управление обратно программе-хозяину, которая продолжает нормально функционировать. Однако сколько бы раз простой вирус ни заражал, например, новый файл или диск, он всегда создает точную копию самого себя. Антивирусу в свою очерь необходимо лишь найти в вирусе "сигнатурную" последовательность байтов, известную как сигнатура, чтобы найти и сам вирус.

**1.1.2. Зашифрованные вирусы.**

В ответ на то, что антивиру так просто обнаруживает вирусов-репликантов, авторы вирусов начали шифровать вирусы. Идея заключалась в том, чтобы скрыть фиксированную сигнатуру, зашифровав вирус и сделав его неузнаваемым для антивирусного сканера.

На рисунке 1 изображена структура вируса, использующего шифрование для сокрытия полезной нагрузки. «Вирус-шифровальщик» всегда распространяется, используя одну и ту же процедуру расшифровки, от заражения к заражению меняется только ключ.

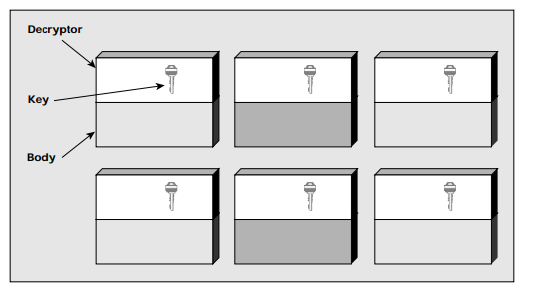


Рисунок 1 - Структура зашифрованных вирусов

Зашифрованный вирус состоит из программы расшифровки и зашифрованного тела. При запуске пользователем зараженной программы, сначала над компьютером получает контроль программа расшифровки, затем происходит сама расшифровка, и после этого программа расшифровки передает управление компьютером расшифрованному вирусу. «Вирус-шифровальщик» заражает программы и файлы, как и любой другой простой вирус. Заражая новую программу, он делает копию своего тела и связанной с ним процедуры дешифровки и прикрепляет их к цели. Для шифрования используется ключ, который вирус запрограммирован менять от заражения к заражению. По мере изменения этого ключа меняется и шифрование, в результате чего от инфекции к инфекции вирус выглядит по-разному. Это и придает сложности для поиска его сигнатуры.

Однако процедуры дешифровки в разных поколениях вируса остаются неизменными – это и есть слабое место, которым быстро воспользовались разработчики антивирусного ПО. Так же вместо того, чтобы сканировать только сигнатуры вирусов, антивирусные сканеры были модифицированы для поиска сигнатур конкретных процедур расшифровки.

**1.1.3. Полиморфные вирусы.**

В отместку, вирмейкеры создали полиморфные вирусы [2]. Так же как и зашифрованные вирусы, полиморфные содержат в себе зашифрованное тело и процедуры расшифровки.

Однако, полиморфные вирусы добавляют к двум компонентам зашифрованных вирусов третий – полиморфный движок (mutation engine), который создает произвольные процедуры расшифровки каждый раз, когда вирус инфицирует новую программу.

В полиморфном вирусе шифруется не только тело, но и сам движок. Когда пользователь запускает инфицированную программу, расшифровщик расшифровывает и тело, и движок и далее передает контроль над компьютером самому вирусу, который в свою очередь ищет очередную жертву для заражения.

В этот момент вирус создает копию как себя, так и механизма мутации в RAM. Далее вызывается механизм мутации, которые генерирует новую программу дешифрования, способную так же расшифровать вирус, но при этом практически она не имеет сходств с предыдущей программой расшифровки. Затем происходит шифровка тела вируса и мутационного механизма. И, наконец, происходит заражение программы, казалось бы, абсолютно новым вирусом.

В результате мы имеем: полностью зашифрованное тело вируса, как и в случае ранее и полностью новый расшифровщик. Это и должно сбивать с толку антивирус, который ищет сигнатуру идентифицирующую программу расшифровки.

**1.3. Пермутация**

Само слово «пермутация» [3],[4] означает: перестановка, подстановка, изменение, перемещение и т.д. В случае метода защиты вируса от обнаружения антивирусом будем понимать под термином процесс пермутации – процесс изменения вируса на уровне ассемблерных инструкций. Или же создание новой копии вируса, состоящей из перемещенных в другое место и, возможно, измененных инструкций, используя в качестве источника имеющуюся копию.

В чем же заключается пермутация? Базовым алгоритмом пермутирующих движков является дизассемблирование кода с последующей его мутацией и ассемблированием.

Три основных этапа работы движка:

1. Дизассемблируются по длинам все инструкции нашего кода и отмечаются условные и безусловные переходы.
2. Изменение списка инструкций (мутация)
3. Ассеблирование нового кода (трансляция обратно в машинный код).

**1.3. Метаморфный код**

Игорь Муттик объяснил метаморфные вирусы [5][6] самым кратким образом: «Метаморфические вирусы – это полиморфизм тела». Техника метаморфизма генерирует не только новый декриптор, но и полностью новое тело вредоносного приложения при сохранении его поведения. В отличие от пермутации, в данном случае код не просто переставляет блоки внутри себя, а реально меняет своё содержание.

Основой для формирования следующего поколения является код в некотором его представлении, доступном для анализа и трансформации (зачастую это некоторое промежуточное представление). Данный код содержит всю необходимую последовательность действий и транслируется в исполняемый двоичный код. При этом процесс трансляции включает в себя некоторую неопределенность, которая позволяет генерировать различные сигнатуры.

Подытожив, могу сказать, что к метаморфизму относятся два типа мутаций:

* Когда весь код, вместе с движком разбивается на блоки и затем производится перестановка данных блоков и их связь.
* Когда весь код вируса представлен в виде псевдо кода и хранится в зашифрованном виде. И на базе этого псевдо кода, генерируется каждый раз новый код.

Данные техники являются очень сложными и не очень приятными по следующим причинам:

* И в первом, и втором случаях достаточно сложно написать универсальный движок. По этой причине каждый раз эту технику необходимо реализовывать заново.
* В первом случае получается ограниченное количество мутаций.
* Во втором случае возможно детектирование по зашифрованной части кода.

**1.4. Используемые методы трансформации кода**

В течении предыдущих пунктов говорилось об преобразованиях или мутациях кода [7], пришло время рассказать о конкретных примерах этих самых преобразований.

**Диверсифицирующие и обфусцирующие трансформации.** Диверсифицирующие и обфусцирующие трансформации напоминают компиляторные по своему устройству, но преследуют другие цели. Цель оптимизирующих преобразований – увеличить скорость работы программы или уменьшить занимаемое место. Цель диверсифицирующих преобразований – изменить код приложения, не изменяя его функциональности. Целью обфусцирующих преобразований является запутывание приложения таким образом, чтобы осложнить понимание принципов работы алгоритмов приложения и структур данных.

**Вставка недостижимого кода.** Недостижимый код – это код, который не выполняется в программе ни при каких входных данных. Вставка недостижимого кода может быть реализована различными способами. Недостижимый код вставляется в граф потока управления в виде нового базового блока.

**Вставка бесполезных инструкций.** Данное преобразование привносит в код программы инструкции, которые, в отличии от недостижимого кода выполняются, но никак не влияют выходные данные или наблюдаемое поведение приложения. Количество привносимых инструкций может быть любым, как и их сложность, поэтому данной трансформацией можно добиться континуального множетсва вариантов программы.

**Перестановка базовых блоков.** Базовый блок – линейная последовательность инструкций, оканчивающаяся единственной инструкцией передачи управления следующему базовому блоку. Последовательность раскладки базовых блоков в образе программы можно менять, что позволяет менять сигнатуру приложения. Перестановка базовых блоков в случайном порядке в пределах одной функции была реализована внутри инструмента Syzygy.

**Перестановка местами функций в модуле.** Перестановка местами функций дает дополнительную возможность по изменению сигнатуры программы на более крупнозернистом уровне по сравнению с перестановкой базовых блоков.

**Шифрование константных буферов.** Константные буферы, хранящиеся в памяти процесса, могут быть источником характерных сигнатур и причиной обнаружения. Преобразование, маскирующее такие буферы, предназначено для изменения их сигнатуры.

**ГЛАВА 2. ВЫЯВЛЕНИЕ ПОЛИМОРФНОГО И МЕТАМОРФНОГО КОДА**

**2.1. Развитие идей**

Разработчики антивирусных программ сначала боролись с ними, создавая отслеживающий механизм отдельно для каждого полиморфного вируса. Вручную, строка за строкой, они писали специальные программы, предназначенные для обнаружения различных последовательностей компьютерного кода, которые, как известно, используются данным мутационным механизмом для расшифровки тела вируса.

Такой подход оказался по своей сути непрактичным, трудоемким и дорогостоящим. Каждый новый полиморф требует своей собственной программы обнаружения. Кроме того, мутационный движок производит на первый взгляд случайные программы, любая из которых может правильно выполнить расшифровку - а некоторые мутационные движки генерируют миллиарды и миллиарды вариаций.

Собственно это и заставило разработчиков антивирусного программного обеспечения задуматься об универсальном методе расшифровки.

**2.1. Метод эмуляции**

Итак, что предполагает общая расшифровка:

* Тело полиморфного вируса зашифровано, чтобы избежать обнаружения.
* Полиморфный вирус должен расшифровать программу, прежде чем она сможет нормально выполняться.
* Как только зараженная программа начинает выполняться, полиморфный вирус должен немедленно перехватить контроль над компьютером, чтобы расшифровать тело вируса, а затем передать контроль над компьютером расшифрованному вирусу.

Антивирусный сканер, использующий общую расшифровку, полагается именно на такое поведение для обнаружения полиморфных вирусов. Каждый раз, когда сканер сканирует новый программный файл, он загружает его в что-то вроде виртуальной машины. Внутри этой виртуальной машины программные файлы выполняются так, как если бы они работали на реальном компьютере.

Сканер отслеживает и контролирует программный файл в процессе его выполнения в виртуальном компьютере. Полиморфный вирус, запущенный в виртуальной машине, не может причинить вреда, поскольку он изолирован от реального компьютера.

Когда сканер загружает файл, зараженный полиморфным вирусом, в этот виртуальный компьютер, выполняется процедура расшифровки вируса, которая расшифровывает зашифрованное тело вируса. Это открывает тело вируса для сканера, который затем может искать в теле вируса сигнатуры, точно идентифицирующие штамм вируса.

Если сканер загружает файл, который не заражен, то нет вируса для обнаружения и мониторинга. В ответ на невирусное поведение сканер быстро прекращает запуск файла внутри виртуального компьютера, удаляет файл из виртуального компьютера и переходит к сканированию следующего файла.

К сожалению, данный подход будет занимать слишком много времени и ресурсов для проверки файлов таким образом. Не говоря уже о том, что современные вирусы имеют защиту от выполнения на виртуальных машинах.

**2.2. Эвристический метод**

Эвристический анализатор – это код, который проверяет файл на некоторые характерные признаки заражения, что-то вроде:

* Некорректный (испорченный вирусом, но работоспособный) заголовок файла
* Инструкции JMP прямо в точке входа
* «rwx» на секции кода
* Точка входа в секции открытой для записи(rwx). Открытая для записи, исполняемая секция, в которую сразу передается управление, с большой вероятностью свидетельствует о наличии самомодифицирующегося кода, такие секции используются в подавляющем большинстве случаев вирусами и программными защитами.
* Инструкция перехода в точке входа. Особого смысла в размещении инструкции перехода в точке входа нет и такой признак указывает на наличие самомодифицирующегося кода в файле.
* Точка входа во второй половине секции. Вирусы, использующие расширение секции, в большинстве случаев располагаются в конце секции. Это нетипично для нормальных файлов, поэтому такая ситуация является подозрительной.
* Поломки в заголовке. Некоторые модификации заголовка после инфицирования оставляют файл работоспособным, но сам заголовок при этом содержит ошибки, которые линкер бы не допустил. Это тоже подозрительно.
* Нестандартный формат некоторых служебных секций. В исполняемых файлах есть служебные секции, такие как, например, .ctors, .dtors, .fini и т.п. Особенности этих секций могут использоваться вирусами для заражения файла. Нарушение формата такой секции также является подозрительным.

Данный подход позволяет исключать не-вирусы, и, если файл вызывает подозрения выполнить ресурсоемкую проверку, например применить подход описанный выше.

Данный метод работает несколько хуже, в плане обнаружения вирусов, но он позволяет сильно сократить вычислительные ресурсы. Так же есть ещё одна отдельная, небольшая проблема – т.к. данный метод реагирует на всё подозрительное, коммерческие защиты помечаются им также, как и вирус, но эта проблема решается добавлением в список «белых» сигнатур.

**ГЛАВА 3. АНАЛИЗ БИНАРНЫХ ФАЙЛОВ**

В данной главе будут рассмотрены основные подходы для нахождения идентичности блоков программ такие как: Bindiff, BinHunt, PatchDiff, TurboDiff.

**3.1. Bindiff**

Абстрактно-Двоичное сравнение – это процесс обнаружения различий и сходств в функциональности между двумя двоичными программами. Именно эту проблему решает компонент Bindiff [8].

На начальном этапе следует дизассемблировать бинарный файлы. Для этого используется алгоритм Diaphora – эвристический метод, основанный на одном или нескольких признаках функции, а также хешировании. Алгоритм сначала находит точные совпадения, чтобы уменьшить пространство поиска функции, а затем находит частичные совпадения. Однако только этого недостаточно, что довольно понятно иллюстрирует рисунок 2.

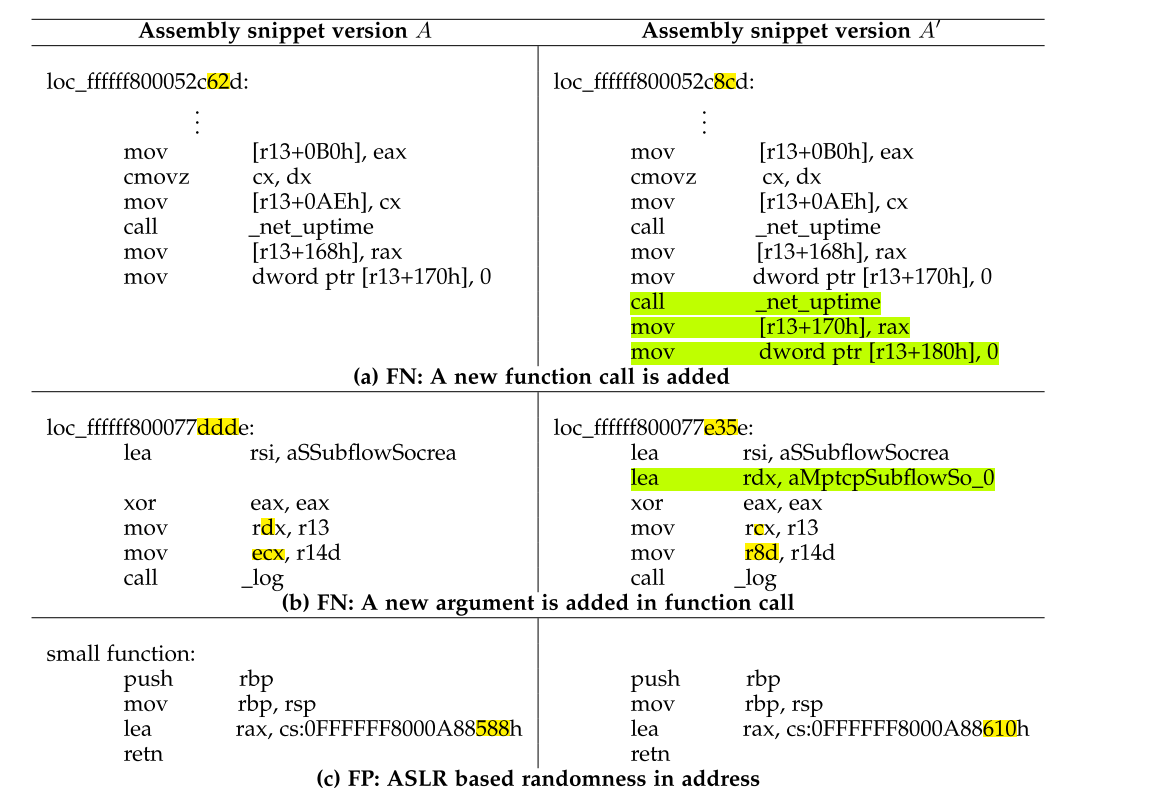


Рисунок 2 – Ложноотрицательные и ложноположительные случаи

Далее, для моделирования бинарного различия применяется двухэтапный процесс, включающий этап сопоставления 1:1 и этап классификации, который основан на использовании семантической нейронной сети. В то же время, следует отметить, что для анализа бинарных программ также применяются подходы изоморфизма графов потока управления и графов вызова, однако основной акцент сделан на использовании методов глубокого обучения. Кроме того, для этого процесса используются различные статистические параметры, такие как количество вершин, ребер, базовых блоков, вызывающих и вызываемых объектов, а также их различные комбинации.

**3.2. BinHunt**

BinHunt - это метод [9], который использует символьное выполнение для точного сравнения двух базовых блоков. Кроме того, BinHunt использует метод изоморфизма графов, который применяется к потоку управления и графам вызовов двух двоичных файлов, для нахождения частичных соответствий между функциями и базовыми блоками.

Однако, чтобы проверить изоморфизм графов бинарных программ, необходимо сначала найти поток управления программ и выполнить дизассемблирование бинарных файлов, используя, например, IDA Pro. Затем инструкции на языке ассемблера преобразуются в язык IR для упрощения анализа, стоит отметить, что данное преобразование может увеличить вероятность ошибок.

После этого строятся графы потока управления и вызова, чтобы выявить идентичные участки кода. Однако различия могут быть внутри функций, поэтому необходимо найти максимальный общий подграф для двух совпадающих функций, чтобы определить бинарные различия. Если изоморфные графы функций не одинаковы, используется символьное выполнение для проверки идентичности выходных регистров и переменных базовых блоков.

Общее представление алгоритма BinHunt представлено на рисунке 3.

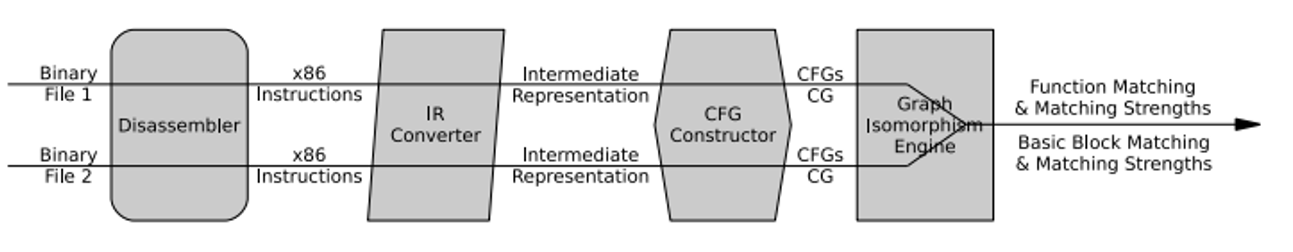


Рисунок 3 – схематическое изображение алгоритма BinHunt

**3.3. PatchDiff**

Плагин PatchDiff для IDA[10]. С помощью этого плагина можно осуществлять декомпиляцию бинарных файлов, а затем проводить анализ функций, включая подсчет контрольных сумм и построение графа потока управления.

Результаты анализа можно использовать для определения идентичных и совпадающих функций, а также для выявления функций, которые не удалось сопоставить (с использованием CRC). Кроме того, плагин может отображать блок-схему для идентичных и совпадающих функций.

Основная цель PatchDiff состоит в том, чтобы быстро и точно анализируя файлы помочь разработчикам в работе с патчем безопасности или исправлением.

**3.4. TurboDiff**

TurboDiff [11] является плагином для IDA, который выполняет декомпиляцию бинарных файлов, вычисление контрольных сумм функций, подсчет количества инструкций и построение графа потока управления. Затем, на основе статистических параметров и графа, происходит анализ для выявления идентичных блоков инструкций.

Категоризация функций в TurboDiff происходит следующим образом:

1. **Идентичный.**

Тот же график функции. Одинаковая контрольная сумма в каждом базовом блоке. Одинаковое количество инструкций в каждом базовом блоке.

1. **Подозрительный+.**

Тот же график функции. Разные контрольные суммы в каждом базовом блоке. Одинаковое количество инструкций в каждом базовом блоке.

1. **Подозрительный++.**

Тот же график функции. Разная контрольная сумма хотя бы в одном базовом блоке. Разное количество инструкций хотя бы в одном базовом блоке.

1. **Измененный.**

Ни один из вышеперечисленных. Связаны эвристикой.

1. **Бесподобный:**

Ни один из вышеперечисленных.

**3.5. IdaCompare**

Статья [12] о данной утилите рассказывает о принципе работы IDACompare, а также описывает структуру и функциональность каждого файла, из которых он состоит. Плагины IDACompare являются небольшими приложениями, которые экспортируют дизассемблированный код из базы данных IDB в базу данных доступа. Затем плагин WinMerge берет на себя задачу сравнения на уровне блоков x86 asm. IDACompare поддерживает только x32 версию IDA.

На рисунке 4 показано, как работает этот плагин.

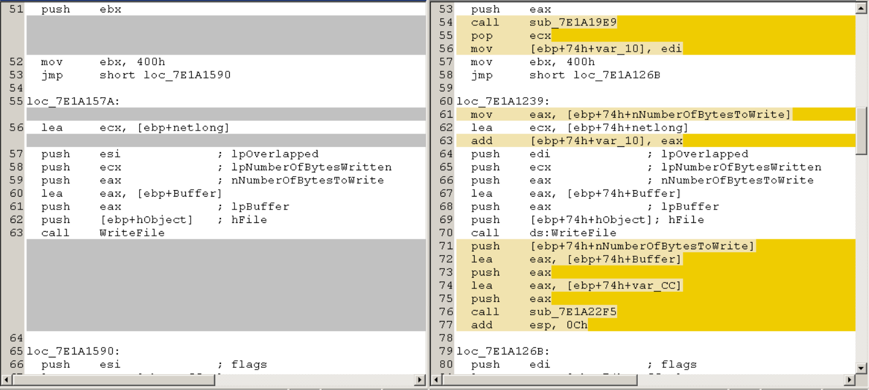


Рисунок 4 – работа плагина IdaCompare

**3.6. Результаты сравнения программ**

Результаты тестирования были взяты со статьи [13]. При описании в этой статье влияния оптимизаций компиляторов на анализ бинарных файлов были рассмотрены несколько типов изменений для сокрытия идентичности блоков кода. Таблица 2 с результатами сравнения некоторых рассмотренных программ представлена ниже.

Таблица 2 – Результаты тестирования некоторых программ

|  |  |  |
| --- | --- | --- |
| Программа | Тип | Результат |
| IDACompare | Тип 1 | 100% |
| Тип 2 | 92.55% |
| Тип 3 | 93.75% |
| Тип 4 | 92.50% |
| PatchDiff | Тип 1 | 100% |
| Тип 2 | 97.97% |
| Тип 3 | 96.92% |
| Тип 4 | 97.95% |
| TurboDiff | Тип 1 | 95.32% |
| Тип 2 | 96.95% |
| Тип 3 | 95.92% |
| Тип 4 | 96.94% |

Из данных таблицы можно сделать вывод, что лучший результат показал плагин PatchDiff, худший – TurboDiff. Однако расхождения незначительны. Как оказалось, большинство программ-анализаторов не восприимчивы к типу клонов 1.

Теперь соберем всю информацию о просмотренных программах в единую таблицу 3.

Таблица 3 – Информация о программах

|  |  |
| --- | --- |
| **Программа** | **Методы** |
| BinHunt | Количество инструкций  CFG  CG  Символьное выполнение  Convert to IR language |
| BinDiff | Количество инструкций  CFG  CG  CRC функций  Сравнение Hash  Diaphora |
| IDACompare | Количество инструкций  CFG  CRC функций |
| PatchDiff | Количество инструкций  CFG  CRC функций |
| TurboDiff | Количество инструкций  CFG  CRC функций |

**ГЛАВА 4. ПРАКТИЧЕСКАЯ ЧАСТЬ**

**4.1. Методы сравнения применяемые при разработке программы**

В данном разделе будут представлены методы сравнения исполняемых файлов, которые использовались для реализации программы. А именно, подсчет CRC функций, построение графа потока управления CFG для каждой функции и подсчет количества инструкций.

**4.1.1. Классические сравнения**

Эти методы сравнения двух или более бинарных программ являются неэффективными и недостаточными. Например, подсчет контрольной суммы программ, таких как CRC, может использоваться только для выявления полной идентичности программы. Используют, чтобы сразу отсечь абсолютно идентичные файла на раннем этапе, чтобы сократить время работы программы сравнения. Этот метод нельзя рассматривать как состоятельный т.к. он не может дать полную информацию о совпадении программы.

Построчное сравнение инструкций ассемблера также является неэффективным из-за оптимизации компилятора, которая может изменить инструкции ассемблера, что приводит к ложным результатам.

Сравнение размера файлов может использоваться только для нахождения идентичных файлов, но не дает информации о том, какие участки кода в этих файлах совпадают, что как раз и является более важной информацией для анализа программы.

**4.1.2. Построение графа потока управления**

Один из наиболее важных способов сравнения двух или более бинарных файлов - анализ потока управления (CFA), который основан на создании графа потока управления (CFG). Более подробное описание данного метода можно найти в источнике [14].

Граф потока управления представляет все возможные пути выполнения программы в виде графа, где узлами являются вызовы функций (в ассемблере: call) и переходы (в ассемблере: jmp, jnz, jz и т.д.). CFG программы фактически содержит все пути, которые могут быть выбраны во время ее выполнения. После создания графов потока управления происходит проверка на изоморфизм графов или подграфов. Рисунок 5 показывает примеры изоморфных графов.

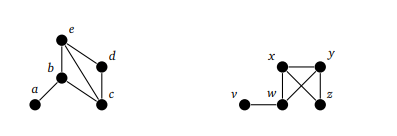


Рисунок 5 – изоморфные графы

Два графа называются изоморфными если любые две вершины одного графа смежны тогда и только тогда, когда эти же вершины смежны в другом графе. Это определение относится к неориентированным графам без взвешенных вершин и ребер. Если говорить об ориентированных или взвешенных графах, то на сохранение направления дуг и значений весов налагаются дополнительные ограничения. Однако, в случае построения графа управляющего потока, граф будет ориентированным, но не взвешенным.

**4.1.3. Control Flow Graph (CFG)**

Фактически смысл его работы описан в предыдущем пункте. Построение модели графа потока управления [15]:

1. Дизассемблируется .exe-файл с помощью стороннего дизассемблера.
2. Удаляются все лишние инструкции, по мимо вызовов функций/процедур и переходов.
3. Выбираются все пути, которые могут быть выбраны во время выполнения программы.
4. На основе этого составляется граф, где вершинами являются call или переходы.

Однако, при использовании данного метода возникает весомый недостаток, особенно когда рассматриваются графы большого размера. В статье [14] отмечается, что время сравнения файлов может колебаться от нескольких секунд при 100 инструкциях ассемблера до 250 секунд и более при 10 000 инструкциях ассемблера. Тем не менее, данный метод является более эффективным по сравнению с классическим сравнением, так как количество ложных срабатываний здесь меньше. Для повышения эффективности был разработан более сложный метод для обнаружения переиспользования кода, о чем в следующем разделе.

**4.1.4 Annotated Control Flow Graph (ACFG)**

Данный подход является улучшенной версией предыдущего метода (CFG). Его преимущества:

* Фиксация семантики
* Обеспечивает более быстрое сравнение ACFG и может обрабатывать вредоносные программы с меньшими CFG, чем другие аналогичные методы, при сохранении высокой точности.
* Содержит больше информации, что обеспечивает более высокую точность по сравнению с CFG.

Различие данного метода заключается в том, что для начала программа разбивается на блоки (функции) и уже для каждого блока создается ACFG а далее анализируется. В процессе создания ACFG используется так называемый язык инструкций MAIL (Malware Analysis Intermediate Language). Его суть заключается в том, что каждая hex-строка переводится в инструкцию из списка таблицы инструкций MAIL (представлена в таблице 1).

Таблица 1 – Инструкции MAIL

|  |  |
| --- | --- |
| **ASSIGN** | Оператор присваивания, например, EAX=EAX+ECX |
| **ASSIGN\_CONSTANT:** | Оператор присваивания, включающий константу, например, EAX=EAX+0x01 |
| **CONTROL** | Оператор управления, в котором цель перехода неизвестна, например, if (ZF == 1) JMP [EAX+ECX+0x10] |
| **CONTROL\_CONSTANT** | Управляющий оператор, в котором известна цель перехода. Например, если (ZF == 1) JMP 0x400567 |
| **CALL** | Оператор вызова, в котором цель вызова неизвестна, например, CALL EBX |
| **CALL\_CONSTANT** | Оператор вызова, для которого известна цель вызова, например, CALL 0x603248 |
| **FLAG** | Оператор, включающий флаг, например, CF = 1 |
| **FLAG\_STACK** | Оператор, включающий флаговый регистр со стеком, например, EFLAGS = [SP=SP–0x1] |
| **HALT** | Оператор остановки, например HALT |
| **JUMP** | Оператор перехода, где цель перехода неизвестна, например, JMP [EAX+ECX+0x10] |
| **JUMP\_CONSTANT** | Оператор перехода, в котором цель перехода известна, например, JMP 0x680376 |
| **JUMP\_STACK** | Обратный прыжок, например, JMP [SP=SP-0x8] |
| **LIBCALL** | Библиотечный вызов, например, compare(EAX, ECX) |
| **LIBCALL\_CONSTANT** | :Вызов библиотеки, включающий константу, например, compare(EAX, 0x10) |
| **LOCK** | Оператор блокировки, например lock |
| **STACK** | Оператор стека, например, EAX = [SP=SP–0x1] |
| **STACK\_CONSTANT** | Оператор стека, включающий константу, например [SP=SP+0x1] = 0x432516 |
| **TEST** | Тестовый оператор, например, EAX и ECX |
| **TEST\_CONSTANT** | Тестовый оператор, включающий константу, например, EAX и 0x10 |
| **UNKNOWN** | Любая неизвестная инструкция по сборке, которую нельзя перевести |
| **NOTDEFINED** | Шаблон по умолчанию, например, всем новым операторам при создании присваивается это значение по умолчанию. |
| ***Примечание***: **EAX, EBX, ECX** — регистры общего назначения, **ZF** и **CF** — флаги нуля и переноса соответственно, а **SP** — указатель стека. | |

Сопоставление ACFG графов осуществляется путем сравнения ACFG графа программы P с ACFG графами известных образцов вредоносного ПО из базы данных. Если P`, являющийся подграфом P, изоморфен M, ACFG графу вредоносного ПО, то программа считается зараженной.

В случае, когда P` меньше всех шаблонов в базе данных, для успешного совпадения требуется, чтобы все утверждения в соответствующих узлах имели одинаковые аннотации, но блоки инструкций могут отличаться.

Из-за большого объема графов и количества шаблонов для эффективного анализа требуются оптимизации, такие как распараллеливание и сокращение ACFG графов, подробно описанные можно найти в статье [14].

**4.2. Реализация программы**

Далее представлено рассмотрение работы программы [17],[18],[19] по шагам:

Шаг номер 1. В самом начале выполнения программы создается директория, где будут храниться графы, а так же вызывается ida для дизассемблирования всех бинарных файлов в указанной директории, для этого было использована надстройка на IDA Pro IDAHunt [16](рисунок 6).

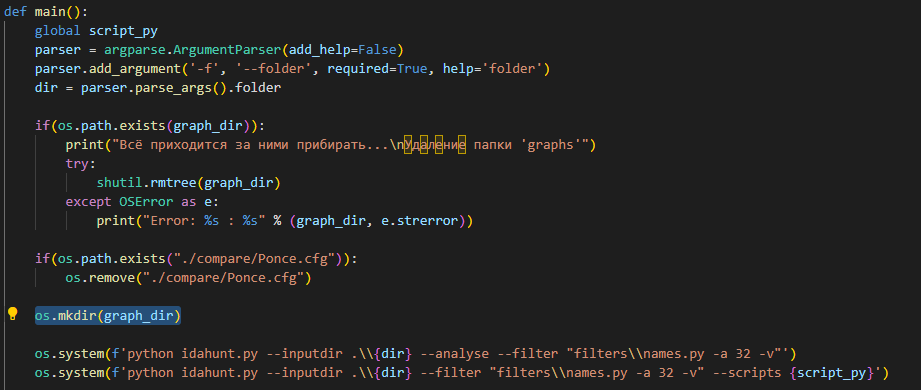


Рисунок 6 – main

Шаг номер 2. На этом шаге происходит выполнения скрипта для каждого idb файла. После этого происходит запись статистических данных в текстовый файл (CRC функций, кол-ва инструкций). А также построение графа потока управления для каждой функции с последующей выгрузкой в отдельную папку, названную именем файла (рисунок 7).

Подробнее про то, как происходит сбор статистики. Выполнение происходит в цикле перебора функций и их инструкций, с последующим их записью в словарь (рисунок 8).

Так же на этом этапе происходит построение графов с последующей записью в файлы (рисунок 9).

Шаг номер 3. Происходит вычисление процента сходства количества инструкций. После этого вычисляется процент совпадения CRC найденных функций.

Шаг номер 4. Парсинг данных графа (рисунок 10) и их последующий анализ (рисунок 11).

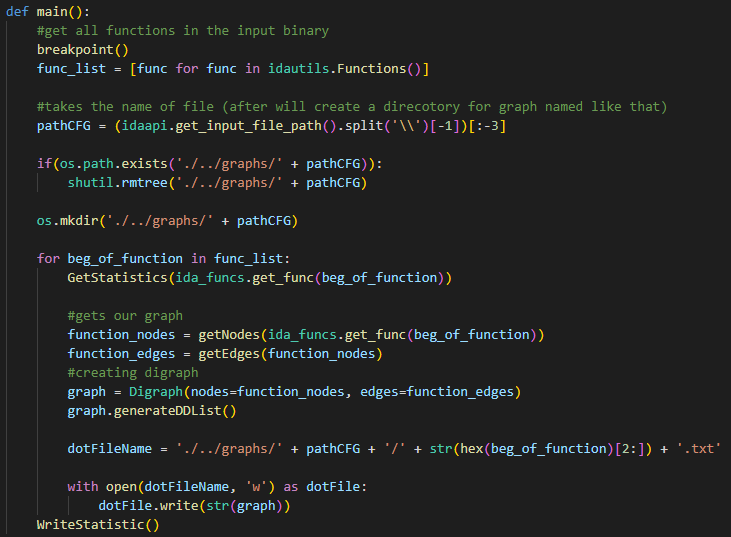


Рисунок 7 – Основной код скрипта

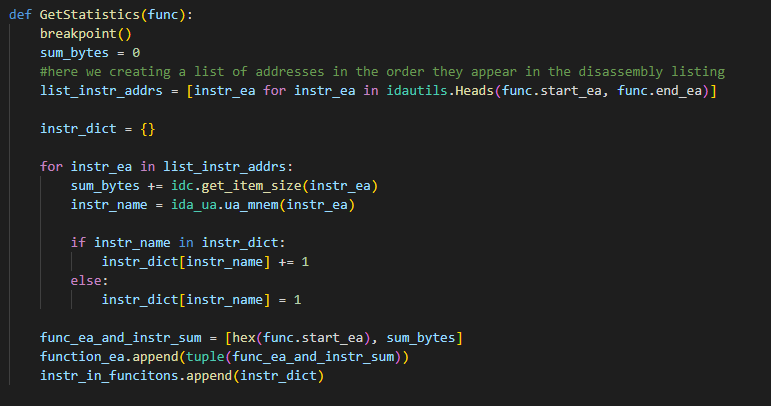


Рисунок 8 – Сбор статистики

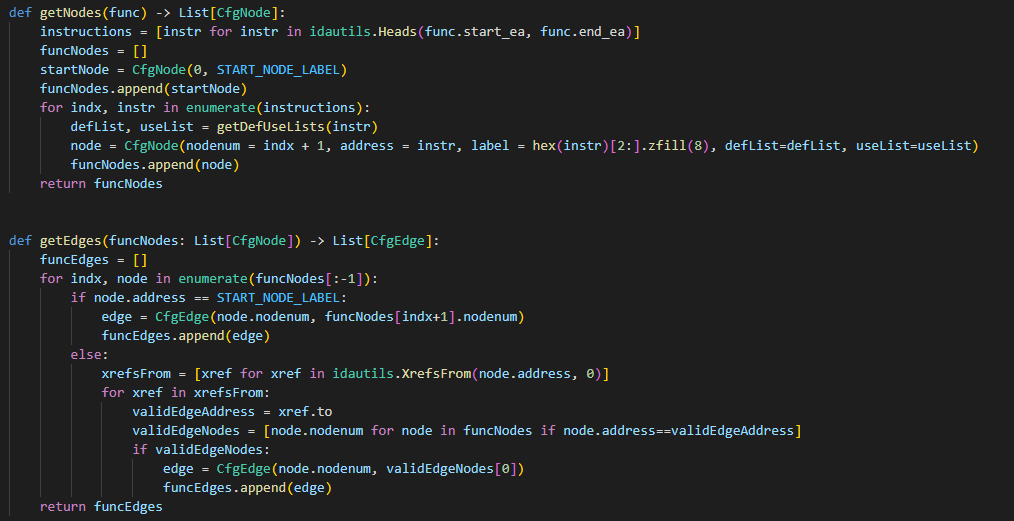


Рисунок 9 – Построение графов

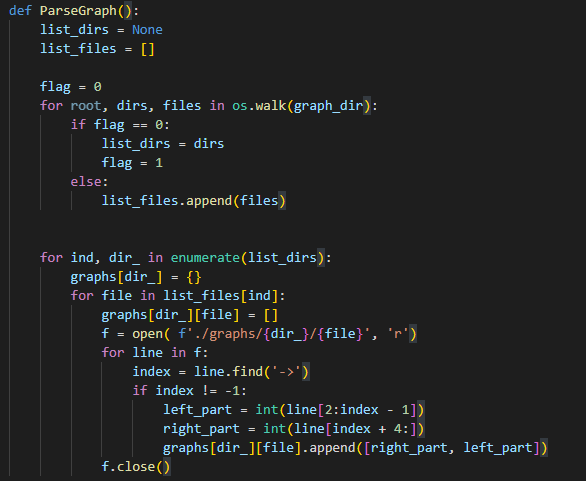


Рисунок 10 – Парсинг графов

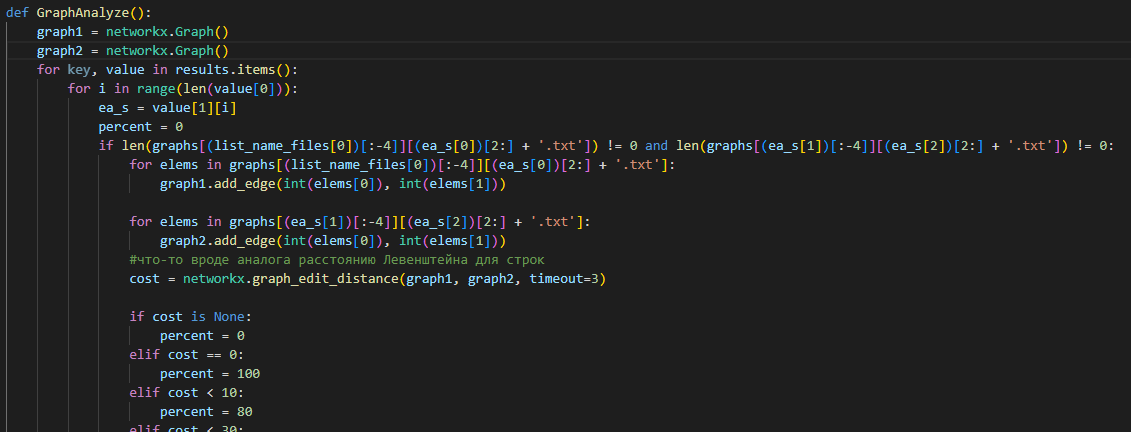


Рисунок 11 – Сравнение схожести графов

**4.3. Сравнение реализованной программы с BinDiff**

Для тестирования разработанной программы была использована ассемблерная программа (рисунок 13) и её полиморфная копия. Для мутации программы используется полиморфный движок SPE32[20].

SPE32 имеет в своем вооружении простой механизм шифрования, в котором он шифрует некоторые случайно выбранные инструкции случайными ключами, создает декриптор для конкретных мест отдельно и создает мусорный код. Важно отметить, что он не шифрует тело программы целиком, почему я и использую для тестирования данный движок. Пример того, как выглядит тело основной функции после мутации представлен на рисунке 12 (слева изображение до мутации, справа после).

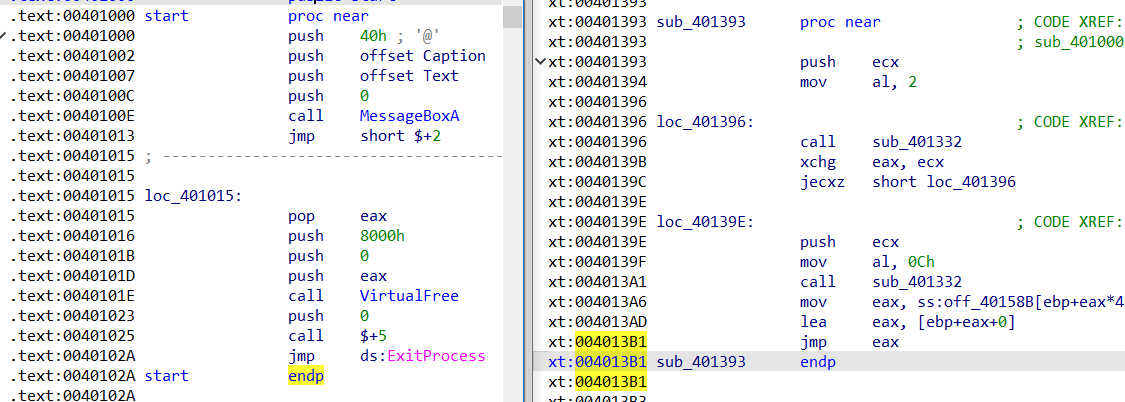


Рисунок 12 – Сравнение основных тел программ

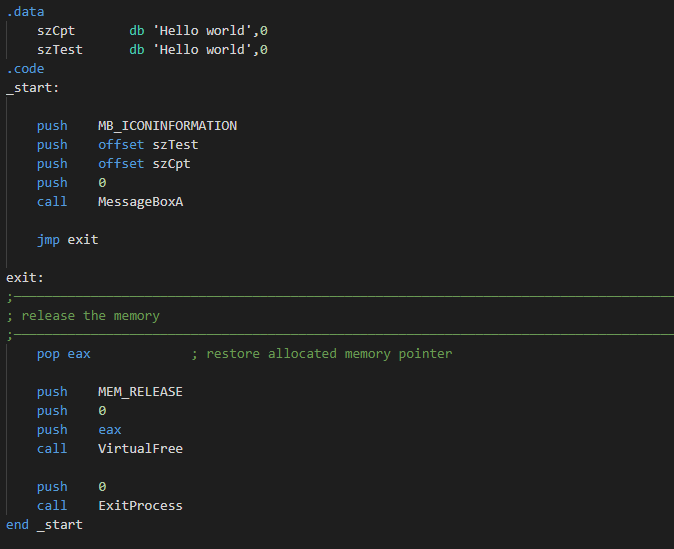


Рисунок 13 – Исходный код программы для тестов

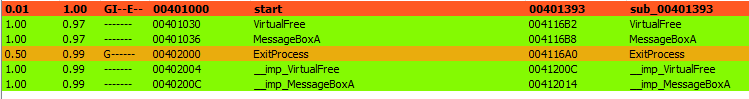


Рисунок 13 – результат сравнения BinDiff

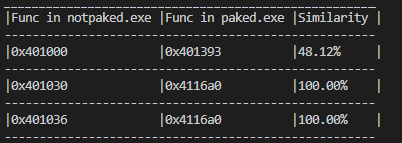


Рисунок 14 – Результат сравнения в реализованной программе

Метрики в реализованной программе:

* Подсчет количества инструкций в каждой функции и сравнение
* CRC функций
* Сравнение CFG c помощью расстояния редактирования графа. (Расстояние редактирования графа - это мера сходства графов, аналогичная расстоянию Левенштейна для строк. Оно определяется как минимальная стоимость пути редактирования (последовательность операций редактирования вершин и ребер), преобразующего граф G1 в граф, изоморфный G2.)

Некоторые метрики, которые использует BinDiff:

* Сопоставление на уровне базовых блоков: BinDiff сопоставляет базовые блоки между двумя двоичными файлами. Базовые блоки - это последовательности инструкций, которые начинаются в точке входа в подпрограмму или в цели инструкции перехода или ветвления и заканчиваются в точке, где поток управления может покинуть подпрограмму.
* Сопоставление графиков: BinDiff строит графы потока управления для двух двоичных файлов и сравнивает их для выявления совпадающих функций. Граф потока управления - это граф, который представляет поток управления между основными блоками в функции.
* Сопоставление последовательности инструкций: BinDiff сравнивает последовательности инструкций в двух двоичных файлах для выявления сходства.
* Сравнение сигнатур функций: BinDiff сравнивает сигнатуры функций в двух двоичных файлах. Подпись функции включает имя функции, количество и типы ее аргументов, а также тип возвращаемого значения.
* Сопоставление потоков данных: BinDiff сопоставляет потоки данных между основными блоками в двух двоичных файлах. Поток данных относится к перемещению данных между переменными в программе.
* Сопоставление сходства кода: BinDiff использует алгоритмы сходства кода для определения схожих фрагментов кода в двух двоичных файлах.

В данном случае можно сказать, что программа, реализованная мною, справилась лучше, чем BinDiff. В защиту BinDiff можно сказать, что SPE32, генерируя мусорный код, усложняет анализ последовательностей инструкций, который и используется в BinDiff в качестве одной из метрик.

**ЗАКЛЮЧЕНИЕ**

В заключение следует отметить, что угроза, исходящая от компьютерных вирусов, остается серьезной проблемой как для специалистов по компьютерной безопасности, так и для конечных пользователей. Полиморфные и метаморфные вирусы - два типа вредоносных программ, которые скрываются от антивирусов, изменяя свой код и поведение. При появлении новых представителей обнаружение данных видов вирусов требует больших затрат ресурсов, как при разработке антивирусного программного обеспечения, так и при проведении сканирования на вычислительных системах.

Одним из таких методов является использование инструментов анализа сходства исполняемых файлов. Если реализованной программе подать на вход пример тела вируса и тело исследуемого файла, то, если исследуемый файл вирус, программа с большой вероятностью обнаружит сходства в их функционале.

**СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Understanding and Managing Polymorphic Viruses [Электронный ресурс]. Режим доступа: <https://docs.broadcom.com/doc/understanding-and-managing-polymorphic-viruses-96-en>

2. Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey [Электронный ресурс]. Режим доступа: <https://arxiv.org/ftp/arxiv/papers/1406/1406.7061.pdf>

3. О пермутации [Электронный ресурс]. Режим доступа: <http://z0mbie.daemonlab.org/permut.html>

4. Полиморфизм. Новые техники [Электронный ресурс]. Режим доступа: <https://www.allasm.ru/vir_10.php>

5. EXECUTABLE CODE METAMORTHIC TRANSFORMATION MODEL [Электронный ресурс]. Режим доступа: <https://dspace.susu.ru/xmlui/bitstream/handle/0001.74/767/12.pdf?sequence=1>

6. Camouflage in Malware: From Encryption to Metamorphism [Электронный ресурс]. Режим доступа: <https://www.researchgate.net/publication/235641122_Camouflage_In_Malware_From_Encryption_To_Metamorphism>

7. Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода [Электронный ресурс]. Режим доступа: <https://cyberleninka.ru/article/n/primenenie-diversifitsiruyuschih-i-obfustsiruyuschih-preobrazovaniy-dlya-izmeneniya-signatury-programmnogo-koda>

8. Bindiff: Распределенное обучение Представление сборки для надежного двоичного файла Отличия от семантических различий [Электронный ресурс]. – Режим доступа: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9470904>.

9. BinHunt: автоматический поиск семантических различий в двоичных программах [Электронный ресурс]. – Режим доступа: <https://link.springer.com/chapter/10.1007/978-3-540-88625-9_16>.

10. PatchDiff: Анализатор двоичных файлов для IDA [Электронный ресурс]. – Режим доступа: <https://github.com/filcab/patchdiff2>.

11. TurboDiff: Инструмент для сравнения двоичных файлов, разработанный как плагин для IDA [Электронный ресурс]. – Режим доступа: <https://github.com/helpsystems/turbodiff>.

12. IDACompare: Анализатор двоичных файлов для IDA [Электронный ресурс]. – Режим доступа: <http://sandsprite.com/iDef/IDACompare/>.

13. Влияние оптимизации компилятора на обнаружение подобия двоичных файлов [Электронный ресурс]. – Режим доступа: <https://www.atlantis-press.com/proceedings/icetis-13/8070>.

14. Аннотированный график потока управления для обнаружения метаморфических вредоносных программ [Электронный ресурс]. – Режим доступа: <https://www.researchgate.net/profile/Shahid-Alam-3/publication/282287967_Annotated_Control_Flow_Graph_for_Metamorphic_Malware_Detection/links/562e095c08ae518e34825a05/Annotated-Control-Flow-Graph-for-Metamorphic-Malware-Detection.pdf>.

15. Анализ потока управления [Электронный ресурс]. – Режим доступа: <https://dl.acm.org/doi/pdf/10.1145/390013.808479>.

16. IDAHunt [Электронный ресурс]. – Режим доступа: https://github.com/nccgroup/idahunt

17. IDA Python Documentation [Электронный ресурс]. – Режим доступа: https://www.hex-rays.com/products/ida/support/idapython\_docs/.

18. NetworkX Documentation [Электронный ресурс]. – Режим доступа: <https://networkx.org/documentation/stable/tutorial.html#analyzing-graphs>.

19. Static analyze plalgin [Электронный ресурс]. – Режим доступа: <https://github.com/TheComputeGuy/static-analysis-plugin>

20. Simple Polymorphic Engine — SPE32 [Электронный ресурс]. – Режим доступа: <https://github.com/PELock/Simple-Polymorphic-Engine-SPE32>