НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

«На правах рукопису»
УДК

«До захисту допущено»
Завідувач кафедри

_____ Дмитро Ланде
(підпис)

“_____” _____ 2022 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-науковою програмою «Системи, технології та математичні методи кібербезпеки»

на тему: «Моделі і методи протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання»

Виконав: студент 6 курсу, групи ФБ-01мн
Войцеховський Андрій Валерійович _____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник к.т.н.,доц. Ільїн М.І. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ - 2022 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра інформаційної безпеки

Рівень вищої освіти – другий (магістерський)

Спеціальність – 125 Кібербезпека («Системи, технології та математичні методи кібербезпеки)

Освітньо-наукова програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Дмитро Ланде
(підпис)

“_____” _____ 2022 р.

ЗАВДАННЯ

на магістерську дисертацію здобувачу ступеня магістра

Войцеховському Андрію Валерійовичу
(підпис)

1) Тема дисертації: Моделі і методи протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання

2) Термін подання студентом роботи: 12 червня 2022р.

3) Об’єкт дослідження : методи класифікації шкідливого програмного забезпечення на основі машинного навчання

4) Предмет дослідження: моделі шкідливого програмного забезпечення

5) Перелік завдань, які потрібно розробити : виконано дослідження методів

протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання

6) Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) - презентація

7) Дата видачі завдання

Календарний план

No з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Формування теми дипломної роботи, визначення мети та постановка задач	30.11.2021 – 13.12.2021	виконано
2	Узгодження дипломної роботи з керівником	14.12.2021 – 15.12.2021	виконано
3	Визначення структури дипломної роботи	14.03.2022 – 15.03-2022	виконано
4	Робота над першим розділом; аналіз літературних джерел	27.03.2022 – 10.05.2022	виконано
5	Робота над другим розділом; реалізація програмної моделі	12.05.2022 – 30.05.2022	виконано
6	Робота над третім розділом; Аналіз отриманих результатів	31.05.2022 – 30.05.2022	виконано
8	Попередній захист дипломної роботи		виконано
9	Підготовка ілюстративного матеріалу; оформлення текстової і графічної частини		

Студент _____ (підпис) _____ (ініціали, прізвище)

Керівник роботи _____ (підпис) _____ (ініціали, прізвище)

РЕФЕРАТ

Дипломна робота: 72 с., 18 рис., 9 табл., 14 джерел.

В роботі проаналізовано методи класифікації шкідливих виконуваних файлів, особливостей відбору ознак для створення моделей класифікаторів. На основі проведеного аналізу, сформуловано задачу дослідження, та виділено основі класифікаційні ознаки, що використовуються для виявлення шкідливого програмного забезпечення, які можна змінити за допомогою проведення мутації виконувального файлу.

Використовуючи дані ознаки, сформульовано алгоритми протидії на основі елементарних мутації, та використання методів обфускації за допомогою Tigress та OLLVM обфускатора. Отримані результати можуть бути використані для протидії існуючим системам антивірусного захисту, систем виявлення вторгненя, та в якості основи для подальших досліджень методів протидії класифікаторам шкідливого програмного забезпечення та систем захисту на основі машинного навчання. Запропоновані моделі та методи дозволяють вдосконалювати класифікатори на основі машинного навчання.

ПРОТИДІЯ ДЕТЕКТУВАННЮ, ЗАСОБИ ПРОТИДІЇ АНАЛІЗУ, ШКІДЛИВЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

ABSTRACT

The paper analyzes the methods of classification of malicious executable files, features of attribution selection for the creation of classifier models. Based on the analysis, the research task is formulated, and the classification features used to detect malicious software, which can be changed by mutating the executable file, are identified.

Using these features, algorithms for counteraction based on elementary mutations and the use of obfuscation methods using tigress and llvm obfuscator are formulated. The obtained results can be used to counteract existing anti-virus protection systems, intrusion detection systems, and as a basis for further research on methods of counteracting malware classifiers and protection systems based on machine learning. The proposed models and methods allow to identify weaknesses of classifiers based on machine learning, and proves that machine learning cannot be applied by itself, but must be combined with other detection methods, such as behavioral and heuristic methods.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів

ШПЗ – Шкідливе програмне забезпечення

АВ – антивірусне програмне забезпечення

EDR – аббревіатура з англійської “Endpoint Detection & Response”: система виявлення цільових атак на кінцевих точках мережі

Шелкод – двійковий виконуваний код, зазвичай як частина експлойта, що забезпечує зловмиснику доступ до командної оболонки

syscall – з англійської “system call”: системний виклик

ВСТУП

Безумовно, моделі машинного навчання для класифікації ШПЗ мають більшу ефективність, аніж звичайні підходи на основі сигнатур. На основі відібраних даних, машинне навчання узагальнює зловмисні зразки, що дозволяє виявляти частково змінені та поліморфні зразки. Водночас, мають низькі показники помилок першого та другого роду. Незважаючи на велику кількість переваг, існують способи протидії моделям класифікації ШПЗ на основі машинного навчання. Тому, в даній роботі будуть розглянуті вразливості цих моделей та способи ухилення від класифікації.

Актуальність роботи - підвищення ефективності систем антивірусного захисту, та методів на основі штучного інтелекту

Мета і завдання дослідження - створення моделі протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання з подальшим вдосконаленям методів класифікації

Об'єкт дослідження - методи класифікації шкідливого програмного забезпечення на основі машинного навчання

Предмет дослідження - моделі шкідливого програмного забезпечення

Практичне значення одержаних результатів - Запропоновано програмну модель протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання. Можливіть використання даної моделі для підвищення ефективності антивірусного захисту

1 СУЧАСНІ МЕТОДИ КЛАСИФІКАЦІЇ ШПЗ НА ОСНОВІ МАШИННОГО НАВЧАННЯ

1.1 Використання нечіткого хешування та глибинного навчання для боротьби з методами ухилення від виявлення шкідливих програм

З появою нових загроз в інформаційній безпеці, з'являються і нові методи ухилення від детектування. Використовується поліморфне ШПЗ, оскільки незначна зміна дозволяє уникнути виявлення. Існує багато можливостей, для розгортатання унікальних зловмисних програми, маскувати їх під легітимне програмне забезпечення, використовувати легальні інструменти для зловмисних задач. Тому якість сучасних систем в інформаційній безпеці визначаєть не кількістю шкідливих програм, які вони можуть виявити, а рішеннями, що дозволяють виявляти різноманітні зразки відповідно одного класу та їхні майбутні модифікації.

Одним із нових методів є поєднання глибинного навчання з нечітким хешуванням [1]. В цьому підході, спочатку використовуються нечіткі хеші, як вхідні дані для виявлення подібності між зразками та визначення, чи є відповідні зразки ПЗ шкідливими. Зібравши вхідні дані, далі використовується метод глибинного навчання, зазснована на нейролінгвістичному програмуванні (NLP). Таким чином краще визначається схожість, що значно покращує якість виявлення.

На рисунку 1.1, видно що два варіанти сімейства одного зразка мають подібні нечіткі хеші TLSH і SSDEP. Що дозволяє виявляти майбутні подібні зразки з незначними змінами.2

MD5	6a6c0394ecee0fbbc081efcdcfb4ef0e
SHA-1	e4287ed1457c18ab820347840cfbbd21164c8192
SHA-256	124cca040d99fdbeaf252dac5cded0f01c6cf1add58c68ed2cacbd06f3077bec
Vhash	036046651d6550b8z201cpz31zd025z
Authentihash	22291b6cdd660c06cf0993e215e3e89f58872cac5c6e9ddbf4f32459bcb909e8
Imphash	9ecee117164e0b870a53dd187cdd7174
Rich PE header hash	09c088bc95bf88e6f4df4d6ca904611b
SSDEEP	49152:2nAQqMSPbcBVQej/1INRx+TSqTdX1HkQo6SAA:yDqPoBhz1aRxcSUDk36SA
TLSH	T14406AD05E2E51BA0D6F25FF6226B9710427A3E45C99BA25E1621A04F1C77F0C
File type	Win32 EXE
Magic	PE32 executable for MS Windows (GUI) Intel 80386 32-bit
TrID	Win32 Executable MS Visual C++ (generic) (38.8%)
TrID	Microsoft Visual C++ compiled executable (generic) (20.5%)
TrID	Win64 Executable (generic) (13%)
TrID	Win32 Dynamic Link Library (generic) (8.1%)
TrID	Win16 NE executable (generic) (6.2%)
File size	3.55 MB (3723264 bytes)
PEiD packer	Microsoft Visual C++

MD5	71d66432e40bf042bd52f3b76fd2c460
SHA-1	61299dbee0cbcde62db028caa720460122deb5d0
SHA-256	65c89d07d44261b9cc5c4953fe979de96ee00eac63e5a48cabe504da76ed1a8a
Vhash	036046651d6550b8z201cpz31zd025z
Authentihash	99ddbfce53986fd347a9b79310a84b03daf1354859ae9bdb78747baa104e98b3
Imphash	9ecee117164e0b870a53dd187cdd7174
Rich PE header hash	09c088bc95bf88e6f4df4d6ca904611b
SSDEEP	49152:VnAQrMSPbcBVQej/1INRx+TSqTdX1HkQo6SAARdhnv:ZDrPoBhz1aRxcSUDk
TLSH	T145063359713CD2FCC10619B414A7CA63E7B37C5A16FE6A0F8F408A661D53B59
File type	Win32 EXE
Magic	PE32 executable for MS Windows (GUI) Intel 80386 32-bit

Рисунок 1.1 — Порівняння властивостей двох варіантів одного зразка Trojan.WannaCry.Win32.1

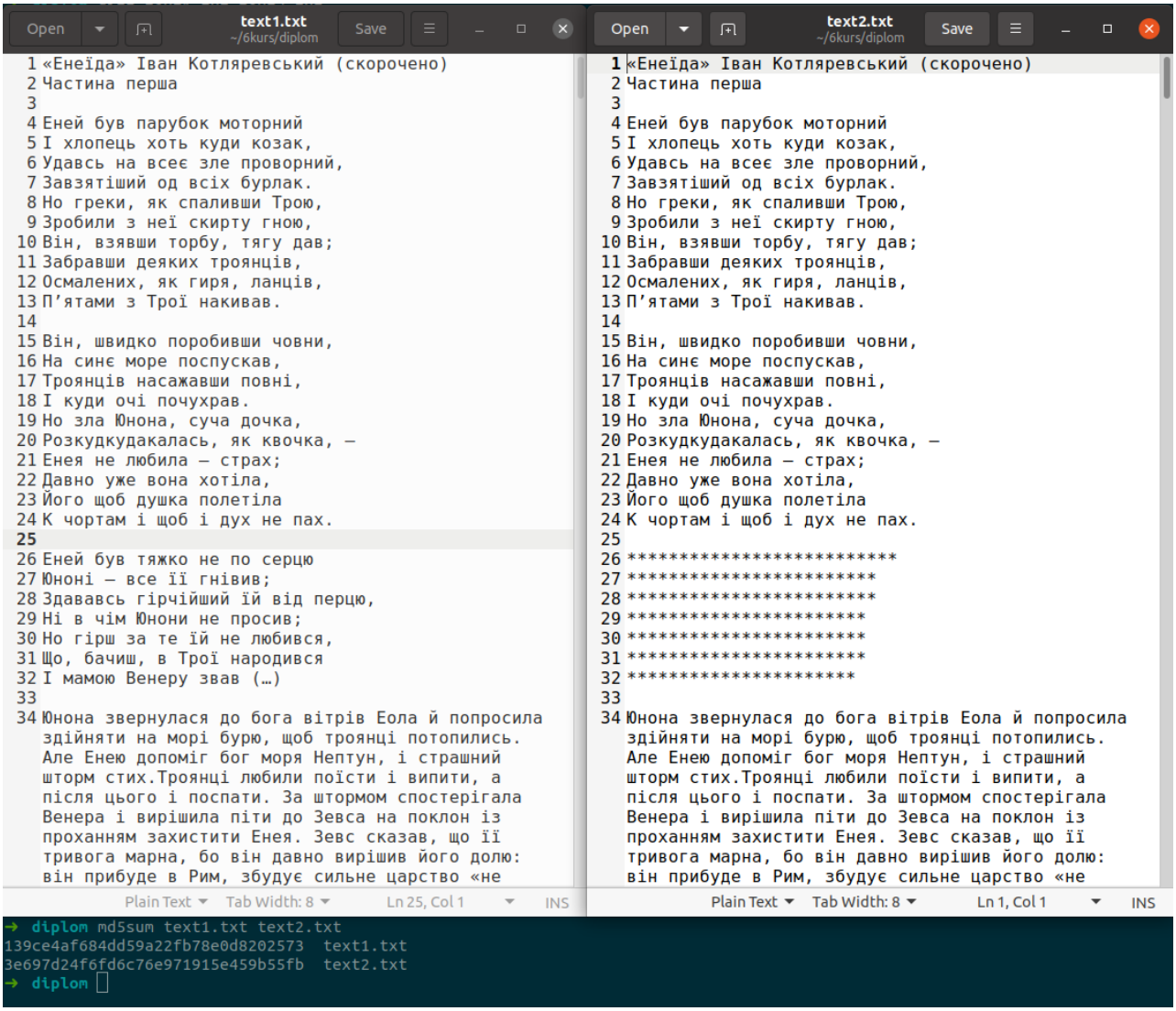

Рисунок 1.2 — Приклад криптографічного хешування

Хешування є важливою технікою у роботі зі шкідливими програмами, адже за допомогою хешу виконувального файлу та відповідної баз з хешами різних файлі, можна визначити чи є даний зарзок шкідливим.

Однак хешування є односторонньою функцію, що приховує звязок між вхідним файлом та отриманим хешем, і зовсім незначна зміна може суттєво змінити сам хеш, яки видно на рисунку 1.2.

Нечітке хешування має такі ж властивості, але при цьому видає подібні хеші, при роботі з подібними вхідними даними.Існують різні алгоритми нечіткого хешування такі як Nilsimsa, TLSH, SSDEEP або sdhash. На рисун-

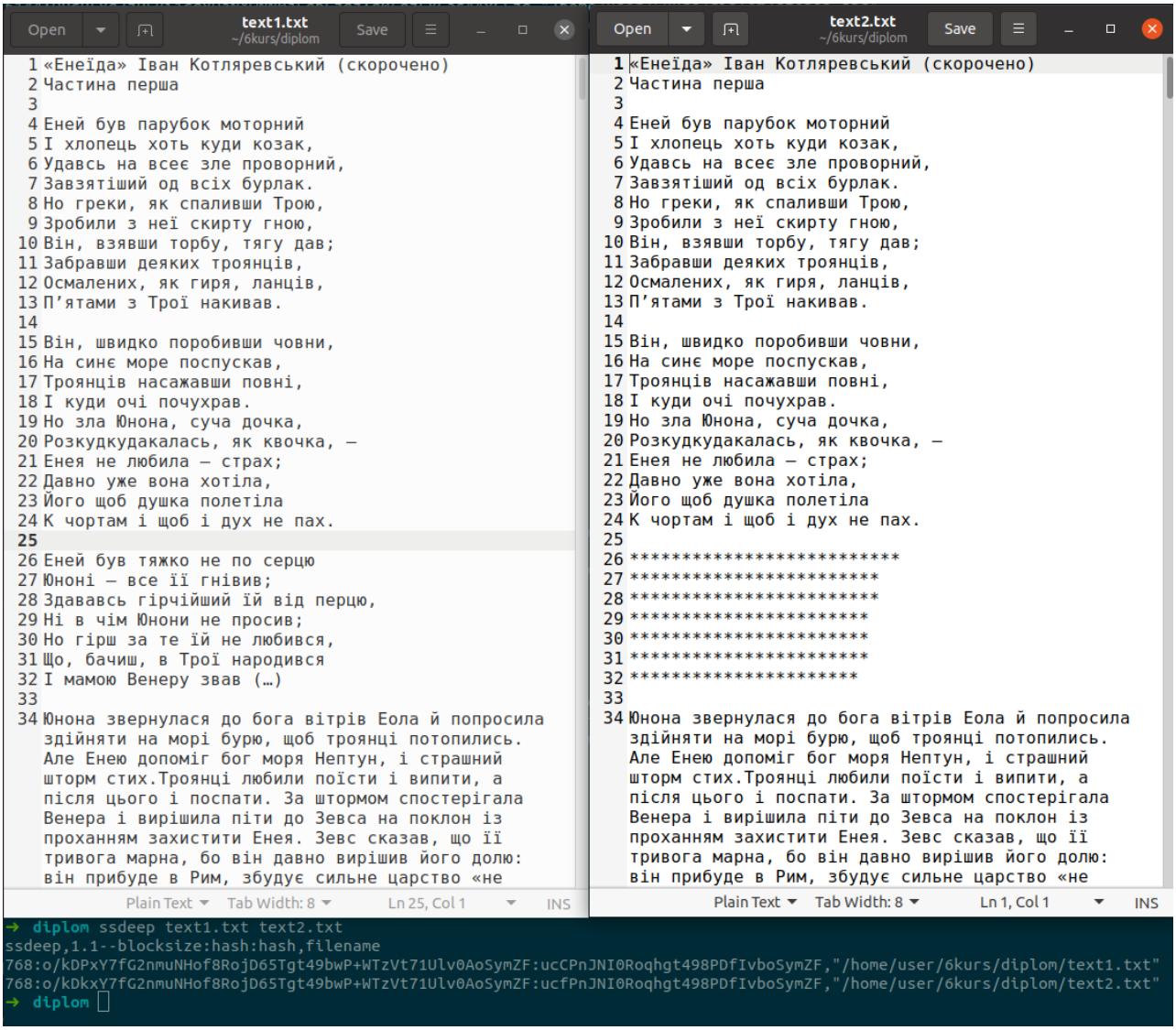

Рисунок 1.3 — Приклад нечіткого хешування

ку 1.3 наведено приклад нечіткого хешування за допомогою ssdeep.

Глибинне навчанння досить ефективно застосовується в моделях обробки природньої мови, в згорткових та рекурсивних архітектурах, GRU або LSTM, і є найсучаснішим інструментом у вирішенні завдань людської мови, таких як аналіз настроїв, відповіді на запитання чи машинний переклад. Такий чином подібний метод можна застосовувати разом з нечітким хешування, адже його окремі елементи можуть розглядатись як слова, та використовуватись для виявлення зловмисного програмного забезпечення.(рисунок 1.4)

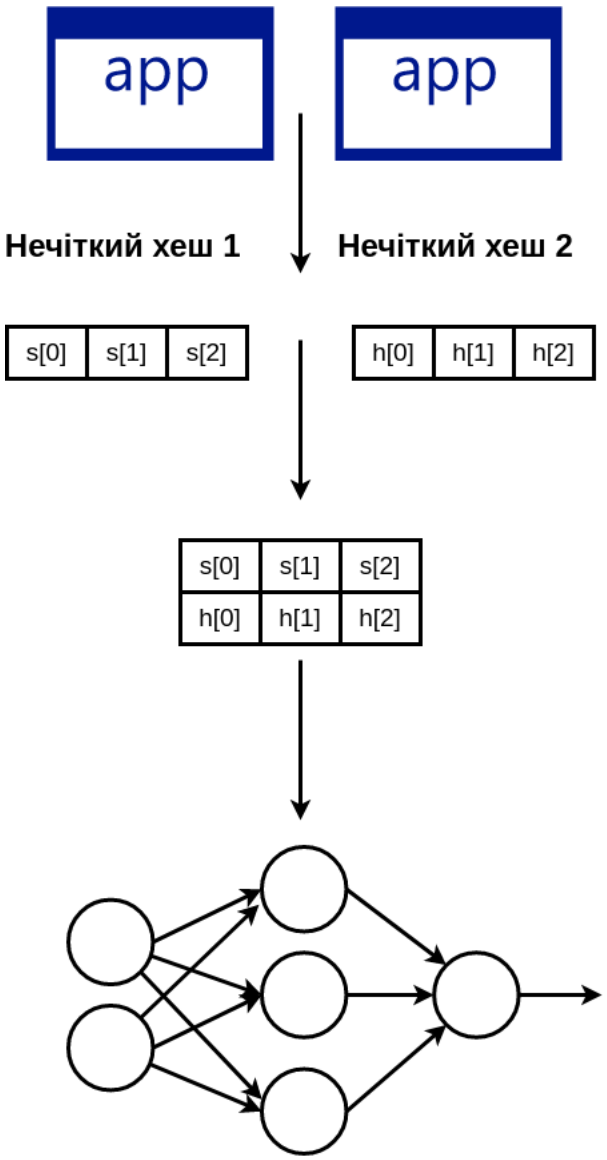

Рисунок 1.4 — Огляд архітектури моделі глибинного навчання з використанням нечітких хечів

1.2 Відкритий набір даних Ember

Набір даних Ember [2] - це колекція атрибутів, отриманих з PE файлів, що служать еталонним набором даних для проведення досліджень. Існують набори за 2017 та 2018 рік, де кожного року було відскановано 1.1млн файлів. Дана колекція також дозволяє легко відтворювати тренування моделей, розширювати набір атрибутів, та класифікувати нові файли.

Для вилучення атрибутів з PE файлі, використовується інструмент LIEF.

Список атрибутів файлу які беруться до уваги:

- Гістограма байтів
- Ентропія
- Інформація по секціям
- Таблиця імпортованих бібліотек
- Функції
- Загальна інформація
- Інформація заголовків
- Друковані рядки

Атрибути файлів видобуваються в json формат.

"sha256": "000185977be72c8b007ac347b73ceb1ba3e5e4dae4fe98d4f2ea92250f7f580e",
"appeared": "2017−01",
"label": −1,
"general": {
"file_size": 33334,
"vsize": 45056,

"has_debug": 0,
"exports": 0,
"imports": 41,
"has_relocations": 1,
"has_resources": 0,
"has_signature": 0,
"has_tls": 0,
"symbols": 0
},
"header": {
"coff": {
"timestamp": 1365446976,
"machine": "I386",
"characteristics": ["LARGE_ADDRESS_AWARE", ..., "EXECUTABLE_IMAGE"]
},
"optional": {
"subsystem": "WINDOWS_CUI",
"dll_characteristics": ["DYNAMIC_BASE", ..., "TERMINAL_SERVER_AWARE"],
"magic": "PE32",
"major_image_version": 1,
"minor_image_version": 2,
"major_linker_version": 11,
"minor_linker_version": 0,
"major_operating_system_version": 6,
"minor_operating_system_version": 0,
"major_subsystem_version": 6,
"minor_subsystem_version": 0,
"sizeof_code": 3584,
"sizeof_headers": 1024,
"sizeof_heap_commit": 4096
}
},
"imports": {

"KERNEL32.dll": ["GetTickCount"],
...
},
"exports": []
"section": {
"entry": ".text",
"sections": [
{
"name": ".text",
"size": 3584,
"entropy": 6.368472139761825,
"vsize": 3270,
"props": ["CNT_CODE", "MEM_EXECUTE", "MEM_READ"]
},
...
]
},
"histogram": [3818, 155, ..., 377],
"byteentropy": [0, 0, ... 2943],
"strings": {
"numstrings": 170,
"avlength": 8.170588235294117,
"printabledist": [15, ... 6],
"printables": 1389,
"entropy": 6.259255409240723,
"paths": 0,
"urls": 0,
"registry": 0,
"MZ": 1
},
}

Далі зібрані атрибути можуть бути конвертованими в csv або інші фор-

мати набору даних для машинного навчання.

1.2.1 Приклад використання

.

Для навчання моделі, необроблені дані конвертуються в векторну форму, створючи додаткові файли. Після створення, дані можна зчитати в зручний формат, відповідною функцією.

```
ember.create_vectorized_features("/data/ember2018/")
ember.create_metadata("/data/ember2018/")
X_train, y_train, X_test, y_test = ember.read_vectorized_features("/data/ember2018/")
metadata_dataframe = ember.read_metadata("/data/ember2018/")
ember.create_vectorized_features("/data/ember2018/")
lgbm_model = ember.train_model("/data/ember2018/")
```

Завершивши навчання моделі, модуль ember можна використовувати для класифікації будь якого PE файлу.

```
lgbm_model = lgb.Booster(model_file="/data/ember2018/ember_model_2018.txt")
putty_data = open("~/putty.exe", "rb").read()
print(ember.predict_sample(lgbm_model, putty_data))
```

1.3 Модель MalConv

Malconv [3] - модель нейронної мережі для виявленння шкідливих програм, що приймає на вхід послідовність байтів PE файлу. Тобто зразки для класифікації передаються у вигляді послідовності цілих чисел, що представляють байти файлу від (0 до 255). Перший шар нейронної мережі відображає кожен байт у вектор, далі послідовність векторів використовується наступними шарами. У результаті модель виводить два числа, що визначають ймовір-

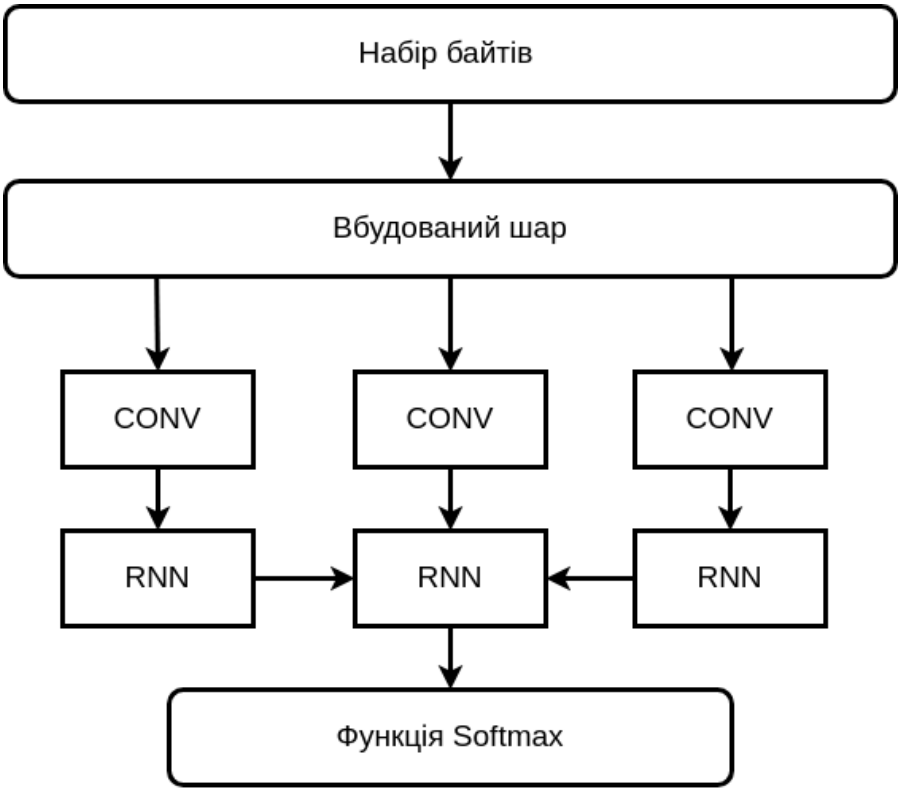

Рисунок 1.5 — Модель Malconv

ності того, що виконувальний файл є доброякісним та шкідливим. Загальна архітектура моделі зображена на рисунку 1.5

```
x = self.embd(x.long())
x = torch.transpose(x,−1,−2)
cnn_value = self.conv_1(x)
gating_weight = torch.sigmoid(self.conv_2(x))
x = cnn_value * gating_weight
x = self.pooling(x)
#Flatten
x = x.view(x.size(0), −1)
x = F.relu(self.fc_1(x))
x = self.fc_2(x)
```

Значним недоліком цієї моделі є те, що для протидії достатньо додати в кінець файлу такий набір байтів, що подібний до набору байтів доброякісної програми. Дана частина не буде виконуватись та зовсім не впливає на роботу

додатка. Якщо частина доброякісних байтів переважає частину шкідливих, то файл не буде визничений як ШПЗ. Більше того, маючи в наявності цілюву модель, можна використати метод грубої сили та швидко підібрати набір байтів для протидії класифікації.

1.4 Модель Nongen-MalConv

Дана модель, побудована на базі попередньої malconv, але дещо обмежений в навчанні, тим що має додатні матриці ваг.[4] Дана відмінність дозволяє запобігти елементарним методам протідії, по типу додавання нових елементів в виконувальний файл. Відповідно обмежені бінарні класифікатори можуть ідентифікувати лише ознаки, що пов'язані з позитивним класом під час тестування. Таким чином не приділяється увага додатим ознакам, і протидіяти моделі можна лише, якщо видалити певні ознаки пов'язані з цим класом. Тому цей метод є досить популярним в сфері інформаційної безпеки, під час виявлення шкідливого програмного забезпечення. Однак, також, існують певні реалізації даної моделі. На виході роботи класифікатора отримуємо два значення: ймовірність шкідливого зразка та ймовірність доброякісного. Далі ймовірність для кожного класу перетворюється за допомогою нормованої експоненціальної функції. Додаткові дані все ще можуть підвищити ймовірність доброякісної ймовірності і в міру підвищення, нормована експоненціальна функція, знизить показник ймовірності шкідливості, незважаючи на таку ж наявність шкідливого вмісту. Відповідно тут можна застосувати ті самі методи протидії, що і використовувались проти звичайного malconv класифікатора.

1.5 Модель класифікації ШПЗ представленого як граф потоку виконання, використовуючи згорткову нейронну мережу

Для класифікації шкідливого програмного забезпечення, виконувальний файл представляється у вигляді спеціального графа потоку виконання (CFG)[5][6]. Даний метод активно використовується і існує багато інструментів для аналізу потоку виконання. Mamadroid - будує графіки викликів з файлів apk, які представляютья за допомогою вузлів і ребер в об'єкті графіка. Hindroid - аналізує код, створюючи 4 різні графіка, кожний з яких представлений у вигляді матриці. Значення всередині матриці відповідають ребрам графа. Metapath2Vec - за допомогою визначених меташляхів, проводить обхід по графу відповідно до типу вузлів, далі отримані зані представляються у вигляді природньої мови.

CFG, як і звичайний граф, складається з вузлів та ребер, де потік виконання це порядок, в якому вионуються окремі інструкціх та виклики функції програми. Кожен вузол представляє базовий блок коду, де немає стрибків чи переходів. Ребро відповідно є самим переходом, зміною потоку виконання в іншу частину коду.

За результатами аналізу літературии існує багато різноманітних методів для класифікації шкідливого програмного забезпечення. Всі розглянуті методи мають обмеження щодо виду вхідних даних, не вроховують поліморфні, метаморфні засоби обфускації. Це зумовлює вдосконалення засобів протидії за допомогою GAN, що в подальшому призводить до обходу засобів класифікації.

Отримавши велику кількість графів з великої кількості виконувальних файлі, створюється загальний граф, який звязує попередні, спільні за певною ознакою. Спочатку вибираються всі ребра, вузли та ваги окремого графа, а

потім ці елементи об'єднуються в загальний граф.

ВИСНОВКИ З РОЗДІЛУ 1

В даному розділі виконано критичний огляд існуючих методів класифікації шкідливого програмного забезпечення на основі машинного навчання, особливостей виділення атрибутів та унікальних ознак виконувальних файлів. Проведений аналіз дозволяє сформулювати актуальну тему дослідження, створення методів протидії на основі GAN та методів машинного навчання.

2 МЕТОДИ ПРОТИДІЇ КЛАСИФІКАТОРАМ ШПЗ НА ОСНОВІ МАШИННОГО НАВЧАННЯ

Для подальшого аналізу та перевірки методів протидії, необхідно підготувати програмний код, використовуючи моделі та методи класифікації розглянуті в 1 розділі. Для початку підготуємо класи відповідних моделей.

```
class MalConvModel(object):
    def __init__(self, model_path, thresh=0.5, name='malconv'):
        self.model = MalConv(channels=256, window_size=512, embd_size=8).train()
        weights = torch.load(model_path,map_location='cpu')
        self.model.load_state_dict( weights['model_state_dict'])
        self.thresh = thresh
        self.__name__ = name
    def predict(self, bytez):
        _inp = torch.from_numpy( np.frombuffer(bytez,dtype=np.uint8)[np.newaxis,:] )
        with torch.no_grad():
            outputs = F.softmax( self.model(_inp), dim=-1)
        return outputs.detach().numpy()[0,1] > self.thresh


class EmberModel(object):
    # ember_threshold = 0.8336 # resulting in 1% FPR
    def __init__(self, model_path=EMBER_MODEL_PATH, thresh=0.8336, name='
        ember'):
        # load lightgbm model
        self.model = lgb.Booster(model_file=model_path)
        self.thresh = thresh
        self.__name__ = 'ember'
    def predict(self,bytez):
        return predict_sample(self.model, bytez) > self.thresh
```

Далі використаємо ці класи для проведення самої класифікації майдутніх зразків.

```
models = [malconv,ember]
for m in models:
        if m.predict(bytez):
            print("Detected")
        else:
            print("Passed")
```

2.1 Генеративні змагальні мережі та навчання з підкріпленням

Запропонуємо модель на базі платформи проміжного програмного забезпечення Pesidious. Pesidious [7] - інструмент, що викоиствує штучний інтелект, для покращення мутації шкідливого програмного зразка, зберігаючи при цьому його функціональність. Дане рішення використовує комбінацію глибинного навчання з підкріпленням [8] та генеративної змагальної мережі [9]. Загальний процес мутації файлу зображений на рисунку 2.1

Для того щоб додати власну мутацію необхідно внести зміни в gym_malware/envs/controls/manipulate2.py.

де self.bytes - це набір байтів файлу, який потрібно змінити

Приклад функції, що додає додаткові байти, не змінюючи при цьому функціонал додатку.

```
def overlay_append(self, seed=None):
        random.seed(seed)
        L = self.__random_length()
        upper = random.randrange(256)
        return self.bytez + bytes([random.randint(0, upper) for _ in range(L)])
```

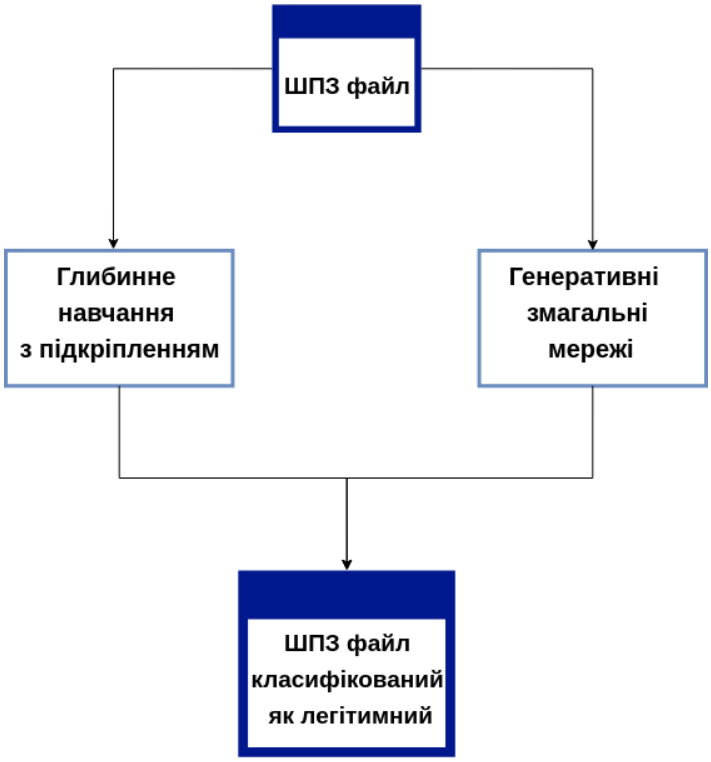

Рисунок 2.1 — Алгоритм Pesidious

2.1.1 Генеративні змагальні мережі

.

Модель навчання генеративної змагальної мережі може бути розбита на дві компоненти: [10] [9]

Вилучення атрибутів

.

1 Особливості - основні складові виконуваного файлу такі як назви розділів, імпортовані бібліотеки та функції

2 Векторне відображення особливостей - кожному атрибуту присвоюється індекс

3 Векторне відображення бінарного файлу - кожен файл представляєть-

ся у бінарному вигляді, використовуючи векторне відображення атрибутів, де 1 це наявність особливостей, та 0 це їх відсутність

Генерація протилежних ознак

.

Після створення простішого представлення бінарних файлів можна перейти до навчання GAN, що буде складатись з 3 компонентів

1 Генератор - створює функції, що виглядають безпечно, намагаючись зберегти шкідливі функції

2 Детектор - виявляє чи є віхід генератора шкідливим, чи доброякісним. Покращує наступні ітерації генератора

3 Детектор Blackbox - навчається з моделями машинного навчання і допомагає у прийняті рішень

2.1.2 Навчання з підкріпленням

.

Модуль навчання з підкріпленням, включає наступні елементи: [10] [9]

1 Середовище ШПЗ - середовище, що допоможе агенту додавати мутації, класифікувати та оцінювати мутовані ознаки

2 Агент - модель нейронної мережі, що вивчає послідовність мутацій для зразків ШПЗ, що може протидіяти класифікаторам

3 Мутації

3.1 Додавання імпортованих функцій

3.2 Додавання секцій

3.3 Додавання байтів до секцій

3.4 Перейменування секцій

3.5 Пакування

3.6 Розпакування

3.7 Додавання та вилудення сигнатур

3.8 Додавання випадкової кількості байтів

4 Класифікатор - приймає двійковий файл та класифікує його

2.2 Методи протидії класифікаторам ШПЗ

Проаналізувавши методи класифікації ШПЗ у розділі 1 та атрибути, що використовуються для класифікації, реалізувати певний набір мутацій, не змінюючи функціональність додатку.

Для того щоб створити найпростішу мутацію, достатньо додати нову секцію, та заповнити її довільними байтами.

```
def add_section_constant(binary, name, constant, size):
    # create a section
    section = lief.PE.Section(name)

    # fill it with our constant
    section.content = [constant] * size

    # add the section
    binary.add_section(section, lief.PE.SECTION_TYPES.DATA)

    # build and reparse
```

```
    builder = get_builder(binary)
    builder.build()
    builder.write('/tmp/asdf')
    binary = get_binary('/tmp/asdf')

    return binary
```

Другим кроком, можна створити секцію з даними та помістити туди друковані тексти з офіційних програм.

```
 def get_binary(file):
    return lief.parse(file)
def get_builder(binary):
    return lief.PE.Builder(binary)
def add_section_strings(binary, name, string_file):
    # grab strings
    with open(string_file, 'rb') as fd:
        data = fd.read()
    # create new section
    section = lief.PE.Section(name)
    # convert characters to decimal representation
    section.content = [c for c in data]
    # add section to binary
    binary.add_section(section, lief.PE.SECTION_TYPES.DATA)
    # build and reparse
    builder = get_builder(binary)
    builder.build()
    builder.write('/tmp/asdf')
    binary = get_binary('/tmp/asdf')

    return binary
```

Після того як в програмі з'явиться набір байтів, що визначається як доброякісний, збільшиться ймовірність класифікації ШПЗ як доброякісного додат-

ку.

Дані кроки є дієвими, але не достатньо ефективними, якщо ми не знаємо які саме дані будуть давати найкращу протидію. Тому оскільки в нас є задані моделі, ми можемо додавати кожного разу різні байти, та вибрати найкращий результат.

```
with open(file, 'rb') as fd:
        bytez = fd.read()
    # original score
    best = predict_sample(model, bytez)
    # greedy attack
    done = False
    while not done:
        done = True
        # iterate over possible bytes
        for b in range(256):
            # see if adding this byte lowers the score
            score = predict_sample(model, bytez + bytearray([b] * size))
            if score < best:
                # if it does add more until it stops working
                while score < best:
                    best = score
                    bytez = bytez + bytearray([b] * size)
                    score = predict_sample(model, bytez + bytearray([b] * size))
                    done = False
                    if best < thresh:
                        done = True
                        break
```

2.3 Перетворення на основі Tigress обфускатора

Tigress — це диверсифікуючий віртуалізатор/обфускатор для мови C, який підтримує багато засобів захисту від статичного та динамічного аналізу. Tigress приймає початковий програмний код на вихід, і на вході видає, також початковий код, але вже обфускований. Нижче наведено приклад обфускації за допомогою tigress. [11]

```
tigress --Verbosity=1 --Environment=x86_64:Linux:Gcc:4.6 --Seed=42 \
  --Transform=Flatten \
    --Functions=bar \
    --FlattenDispatch=switch \
    --FlattenObfuscateNext=false \
    --FlattenRandomizeBlocks=true \
    --FlattenConditionalKinds=branch,compute,flag \
    --FlattenImplicitFlowNext=true \
  --Transform=Virtualize \
    --Functions=bar \
    --VirtualizeDispatch=direct \
  --Transform=CleanUp \
     --CleanUpKinds=annotations,constants,names \
  --out=foo_out.c foo.c
```

На рисунку 2.2 бачимо приклад коду до обфускації, та на рисунку 2.3 вже після обфускації за допомогою Tigress.

Віртуалізація

.

Трансформація перетворює функцію в інтерпритатор, з таким набором байт-коду, що залежить від самої функції [12]. Трансформація розроблена для того, щоб викликати як можна більше різноманітних результатів, тобто

```c
#include "tigress.h"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void fac(int n) {
  int s=1;
  int i;
  for (i=2; i<=n; i++) {
    s *= i;
  }
  printf("fac(%i)=%i\n",n,s);
}

void fib(int n) {
  int a=0;
  int b=1;
  int s=1;

  int i;
  for (i=1; i<n; i++) {
    s=a+b;
    a=b;
    b=s;
  }
  printf("fib(%i)=%i\n",n,s);
}

int main (int argc, char** argv) {
  fac(1);
  fib(1);
  fac(5);
  fib(5);
  fac(10);
  fib(10);
  return 0;
}
```

Рисунок 2.2 — До обфускації за допомогою Tigress

Рисунок 2.3 — Після обфускації за допомогою Tigress

кожне прийняте рішення залежить від початкової точки рандомізації. Різноманітність може бути як статичною так і динамічною, тобто кожний варіант інтерпритатора відрізняється структурою свого коду, а також схемою виконання. Згенерований інтерпритаток складається з віртуального набору інструкці, спеціальної функції введення, масиву байт-кодів, віртуального програмного лічильника, вказівника на віртуальний стек, обробник для кожної віруальної інструкції. Нижче наведено приклад інтерпритатора побудованого за допомогою Tigress

```
 enum ops {Locals = 116, Plus = 135, Load = 60, Goto = 231,
           Const = 3, Store = 122, Return = 72};

unsigned char bytecode[41] = {
 Locals,24,0,0,0,Const,1,0,0,0,Locals,24,0,0,0,Load,Plus,Store,Locals,
 28,0,0,0,Const,0,0,0,0,Store,Goto,4,0,0,0,Locals,28,0,0,0,Load,Return};

int main() {
  while (1) {
    switch (*pc) {
    case Const: pc++; (sp+1)->_int = *((int *)pc); sp++; pc+=4; break;
    case Load: pc++; sp->_int = *((int *)sp->_vs); break;
    case Goto: pc++; pc+= *((int *)pc); break;
    case Plus: pc++; (sp+-1)->_int=sp->_int+(sp+-1)->_int; sp--; break;
    case Return: pc++; return (sp->_int); break;
    case Store: pc++; *((int *)(sp+-1)->_vs)=sp->_int; sp+=-2; break;
    case Locals: pc++; (sp+1)->_vs=(void *)(vars+*((int *)pc));
                 sp++; pc+=4; break;
}}
```

Для цієї трансформації Tigress спочатку конструює абстрактне синтаксичне дерево з початкового C коду, з якого він генероє графіки потоку керування. Далі вибирає архітектуру випадкового набору інструкції(ISA), і ге-

нерує спеціальний байт код, для вхідної функції. Вибирає випадковий метод динамічнох диспетчеризації та створює вихідну програму.

Метод динамічної рекомпіляції

.

Дане трасформація перетворює функцію F в нову функцію F', що складається з такої послідовності проміжних інструкції, що при виконанні F', код F буде динамічно скомпільовано в машинний код. Тобто це приклад генерації коду під час виконання програми, своєчасної компіляції, або динамічного розпаковування. Перші декілька рядків F' функції можуть виглядати наступним чином [13].

```
int fac(int x ) {
  ...
   p3 = jit_init();
   jit_enable_optimization(p3, 3L);
   label4 = jit_get_label(p3);
   jit_add_prolog(p3, & _3_obf_int_binary_I___foo, 0);
   localSize6 = jit_allocai(p3, 8L);
   jit_add_op(p3, JIT_DECL_ARG, ((0 << 4) | (2 << 2)) | 2, 0, 4, 0L, 0L, 0);
   jit_add_op(p3, JIT_DECL_ARG, ((0 << 4) | (2 << 2)) | 2, 0, 4, 0L, 0L, 0);
   jit_add_op(p3, JIT_ADD | 2, ((2 << 4) | (1 << 2)) | 3, 0 | ((0 << 1) |
                    (1 << 4)), 0 | ((2 << 1) | (0 << 4)), (jit_value )(localSize6 + 4), 0L, 0)
                       ;
   jit_add_op(p3, JIT_GETARG, ((0 << 4) | (2 << 2)) | 3, 0 | ((0 << 1) |
                     (2 << 4)), 1L, 0L, 0L, 0);
   jit_add_op(p3, JIT_ST | 1, ((0 << 4) | (1 << 2)) | 1, 0 | ((0 << 1) |
                     (1 << 4)), 0 | ((0 << 1) | (2 << 4)), 0L, 4, 0);
   jit_add_op(p3, JIT_ADD | 2, ((2 << 4) | (1 << 2)) | 3, 0 | ((0 << 1) |
                     (3 << 4)), 0 | ((2 << 1) | (0 << 4)), (jit_value )(localSize6 + 0), 0L,
                        0);
```

```
    jit_add_op(p3, JIT_GETARG, ((0 << 4) | (2 << 2)) | 3, 0 | ((0 << 1) |
                        (4 << 4)), 0L, 0L, 0L, 0);
    jit_add_op(p3, JIT_ST | 1, ((0 << 4) | (1 << 2)) | 1, 0 | ((0 << 1) |
                        (3 << 4)), 0 | ((0 << 1) | (4 << 4)), 0L, 4, 0);
    ...
    jit_generate_code(p6);
    ...
    result58 = (*fac_foo)(x);
    return (result58);
}
```

Оператори проміжного коду ($JIT\_MOV, JIT\_EQ,$), рандомізуються при кожному виклику обфускатора. Але в динамічно згенерованому коді відсутня різноманітність, тобто при кожному запуску програми виконується один і той самий код.

Згладжування потоку керування

.

Класичне перетворення потоку керування, що видаляє початковий структурований потік [14]. Даний метод корисно використовувати у якості бази для інших методів, оскільки перетворення вирівнює функцію перед заміною прям розгалуджень, оскільки це створює більше можливостей для кодування. Окрім того, перетворення може виріанювати функції пере їх виконанням, таким чином ретельніше заплутувати код. Нижче наведено чотири види основних блоків.

1 switch dispatch - традиційна реалізація де кожний блок поміщається в структуру switch, а далі сам перемикач поміщається в нескінченний цикл.

2 goto - використовуєтсья goto для переходу між блоками

3 непрямий - для перемикання між блоками використовуються непрямі переходи через таблицю переходів

4 виклики - кожен блок стає незалежною функцією, і для переходу між блоками використовуються непрямі виклики через спеціальну таблицю переходів.

Розподіл

.

Розподіляє частини функції в окрему власну функцію, це перетворення може бути корисне для розбиття великої віртуальної функції на менші, та менш помітні частини. Існують чотири різні методи розподілу. Порядок їх використання може вплинути на кінцевий результат отриманого коду.

1 поверхневий - оператори верхнього рівня розподіляються на дві функції

2 блоковий - базовий блок розподіляється на дві функції

3 глибинний - владена структура розбивається щонайменше на 2 функції

4 рекурсивний - так само як і блоковий, але також дозволяється розподіляти виклики до розподіленої функції

Приклад використання наведено нижче:

```
tigress --Seed=0 \
  --Transform=split \
      --SplitKinds=deep,block,top \
      --SplitCount=100 \
      --Functions=foo \
  --Transform=Split \
      --SplitKinds=block \
```

```
    --SplitCount=100 \
    --Functions=/.\*foo_split.\*/ \
--out=foo prog.c
```

Злиття

.

Об'єднання кількох функції в одну. Присутній додатковий формальний аргумент, щоб можна було викликати будь яку функцію. Дана трансформація корисна для використання перед трансформацією віртуалізації. Якщо потрібно віртуалізувати дві функції, то їх можна спочатку об'єднати. Перетворення об'єднує локальні аргументи та змінні функцій, таким чином пов'язуючи їх разом. Злиття залежить від трансформації розподілу, і має спільні з нею характеристики.

Непрозорий предикат

.

Розбиває блоки коду, додаючи непрозорі предикати. Для виконання даної трансформації необхідно використати InitOpaque та бажано декілька разів UpdateOpaque

Існує декілька типів таких блоків

1 call

```
if expr=false then
 call to random existing function
```

2 fake

```
if expr=false then
    call to non-existing function
```

3 true

```
if expr=true then
    existing statement
```

4 bug

```
if expr=true then
    existing statement
else
    buggified version of the statement
```

5 question

```
if expr=true || false then
    existing statement
else
    obfuscated version of the statement
```

6 junk

```
if expr=false then
    asm(".byte RandomBytes")
```

2.3.1 Кодування літералів

.

Шифруфрує числа та рядки. Цілі числа можна замінити різними виразами, що вимагає попереднього виконання перетворення InitOpaque. Текстові рядки можна замінити функцією, що буде генерувати їх під час виконання.

Застосувати дане перетворення можна до частини функції додавши заголовковий файл, а потім закріпити регіони, які необхідно трансформувати, застосовуючи макроси наступним чином.

```
#include "tigress.h"

void foo () {
    ...

    ENCODE_INTEGER_BEGIN(obfuscateThis);

    int x = 1234567;

    ENCODE_INTEGER_END(obfuscateThis);

    ...

    ENCODE_INTEGER_BEGIN(obfuscateThat);

    int y = 7654321;

    ENCODE_INTEGER_END(obfuscateThat);

    ...
}
```

Кодування даних

.

Кодує цілочисленні змінні так, щоб вони мали нестандартне представлення даних. Мета полягає в тому, щоб дійсне значення змінної та виразів, що використовується для їх обчислення, ніколи не було виявлене, доки воно не буде виходити з самої програми. Наприклад цілу змінну v можна замінити на v' = a*v+b

```
int main () {
```

```
  int arg1 = ...
  int arg2 = ...
  int a = arg1;
  int b = arg2;
  int x = a*b;
  printf("x=%i\n",x);
}
a = 1789355803 * arg1 + 1391591831;
b = 1789355803 * arg2 + 1391591831;
x = ((3537017619 * (a * b) - 3670706997 * a) - 3670706997 * b) + 3171898074;
printf("x=%i\n", -757949677 * x - 3670706997);
```

Важливо зазначити, що усі змінні мовинні бути цілими числами, масивами цілих чисел або їх комбінаціями.

Кодування арифметичних виразів

.

Цілі арифметичні вирази змінюються на більш складні. Наприклад закодувати звичайне додавання можна наступним чином.

```
z = x + y + w
z = (((x ^ y) + ((x & y) << 1)) | w) +
    (((x ^ y) + ((x & y) << 1)) & w);
```

Для кожного оператора існує багато можливих кодувань, і під час перетворення вони вибираються випадковим чином.

Кодування зовнішніх атрибутів

.

Метою цього перетворення є приховати виклики зовнішніх функцій, таких як системні виклики або виклики стандартних бібліотечних функцій.

Прямі виклики замінюються непрямими і завантажується адреса функцій за допомогою dlsym(). Наприклад, якщо є така програма з двома системними викликами getpid та gettimeofdat.

```
 #include<stdio.h>
#include<unistd.h>
#include<sys/time.h>

void tigress_init() {}

int main () {
    tigress_init();
    int x = getpid();
    printf("%i\n", x);
    struct timeval tv;
    int y = gettimeofday(&tv, NULL);
    printf ("%ld.%06ld\n", tv.tv_sec, tv.tv_usec);
}
```

То її можна обфускувати наступним чином:

```
 void *_externalFunctionPtrArray[2];

void tigress_init(void) {
  STRINGENCODER(0, encodeStrings_litStr0);
  _externalFunctionPtrArray[0] = dlsym((void *)-3, encodeStrings_litStr0);
  STRINGENCODER(1, encodeStrings_litStr1);
  _externalFunctionPtrArray[1] = dlsym((void *)-3, encodeStrings_litStr1);
}

void STRINGENCODER(int n, char str[] ) {
  STRINGENCODER_$sp[0] = STRINGENCODER_$stack[0];
  STRINGENCODER_$pc[0] = STRINGENCODER_$array[0];
  while (1) {
```

```
    switch (*(STRINGENCODER_$pc[0])) {
    case STRINGENCODER__store_char$left_STA_0$right_STA_1:
    (STRINGENCODER_$pc[0]) ++;
    *((char *)(STRINGENCODER_$sp[0] + 0)->_void_star) = (
        STRINGENCODER_$sp[0] + -1)->_char;
    STRINGENCODER_$sp[0] += -2;
    break;
    ...
}

int main( ) {
  int x,y;
  struct timeval tv ;
  ...
  tigress_init();
  x = ((pid_t (*)(void))_externalFunctionPtrArray[1])();
  printf((char const *)"%i\n", x);
  y = ((int (*)(struct timeval * __restrict , void * __restrict ))
                  _externalFunctionPtrArray[0])((struct timeval *)(& tv),
                   (void *)((void *)0));
  printf((char const *)"%ld.%06ld\n", tv.tv_sec, tv.tv_usec);
}
```

Кодування переходів

.

Мета цих перетворень – ускладнити роботу автоматичним інструментам аналізу (наприклад, дизассемблерам) Тут не використовуются хеш таблиця, оскільки це значно ускладнює перетворення. Зміщення переходу передається як аргумент до самої функції переходу. Згенерований код виглядає наступним чином, де викликається функція bf, результат якої є насправді стрибком

у частину коду lab2.

```
 void bf(unsigned long offset) {
   __asm__ volatile ("addq %0, 8(%%rbp)": : "r" (offset));
}

int main() {
    bf((unsigned long)(&& lab2) − (unsigned long)(&& lab3));
    lab3:
        __asm__ volatile (".byte 0x76,0x9b,0x8e,0x1b,0x4d":);
    ...
    lab2: ...;
}
```

За замовчуванням функція вирівнюється до того, як прямі переходи замінюються викликами функції розгалуження. Це створює більше прямих переходів і, отже, більше можливостей для застосування перетворення функції розгалуження. Функція розгалуження не прихована і її можливо знайти, використовуючи тривіальні методи. Тому доцільно об'єднати дане перетворення з іншимим методами приховування функції.

Протидія аналізу псевдонімів

.

Метою цієї трансформації є протидія інструментам статичного аналізу, що використовують міжпроцедурний аналіз псевдонімів. Виклик x = foo(n) перетворюється на:

```
 void *arr[] = {..., & foo, ... };

int main () {
    int x = ((int (*)(int n ))arr[expr=42])(n);
}
```

Щоб зробити аналіз складнішим, можна ще вставити фальшиві вирази.

```
int main () {
  int x;
  arr[expr=6] = arr[expr=42];
  arr[expr=7] = &x;
}
```

Генерування ентропії

.

Деяким перетворення необхідно мати випадкові значення під час виконання. Для цього ми можемо додати оператори, які будуть збирати випадкові значення зі змінних самої програми, що залежать від вхідних даних які надходять до програми. Існує два основних способи збору випадкових даних. Можна вставити в програму такі оператори які будуть збирати значення, залежні від введених змінних. Або розгорнути фоновий потік який буде періодично робити це автоматично. Приклад використання наведено нижче:

```
tigress --Seed=0 --Verbosity=1 --Environment=... \
  --Transform=InitEntropy \
    --InitEntropyThreadName=ENTROPYTHREAD \
    --InitEntropyThreadSleep=1000000\
    --InitEntropyKinds=vars,thread \
    --InitEntropyTrace=true \
  --Transform=UpdateEntropy \
    --Functions=inputData \
    --UpdateEntropyKinds=vars \
    --UpdateEntropyTrace=true \
    --UpdateEntropyVars=x,y,z \
  --Transform=UpdateEntropy \
    --Functions=acceptNetworkPacket \
    --UpdateEntropyKinds=vars \
```

```
    --UpdateEntropyTrace=true \
    --UpdateEntropyVars=packet \
--Transform=UpdateEntropy \
    --Functions=random \
    --UpdateEntropyKinds=vars \
    --UpdateEntropyTrace=true \
    --UpdateEntropyVars=p\* \
--Transform=UpdateEntropy \
    --Functions=tigress_init \
    --UpdateEntropyKinds=thread \
```

Неявний потік виконання

.

Кілька трансформацій, зокрема AntiTaintAnalysis, використовують неявний потік як базовий блок для побудови. Перш ніж ви зможете використовувати їх, потрібно в перелічити, які з неявних варіантів потоків ви збираєтеся використовувати пізніше. Нижче наведено види неявних потоків

1 counter_int -скопіювати змінну шляхом підрахунку до її значення

```
  int i,a = 0;
for(i=0;i<b;i++)
   a++;
```

2 counter_float те саме, але за допомогою float

3 counter_signal - скопіювати змінну побітово за допомогою обробника сигналів

4 bitcopy_unrolled - копіювати порозрядно де кожен біт перевіряється оператором if

5 bitcopy_loop - обрати біти та скопіювати кожен біт шляхом перевірки оператором if

6 bitcopy_signal - скопіювати кожен біт в обробних сигналів

```
  unsigned int value;
int bitNo;
void handler(int sig){
  value |= 1 << bitNo;
}

{
    value=0;
    signal(31, handler);
    for(i in 0..(bits in b)−1]) {
        if ((i:th bit of b)==1) {
              bitNo=i;
              raise(31);
      }
    }
    signal(31, 1);
  a = value;
}
```

7 file_write - скопіювати змінну побайтно, записавши у файл і знову прочитавши його.

8 trivial_clock - скопіювати змінну змінну, визначивши тривіальний цикл.

9 trivial_thread_1 -Те саме, що і trivial_clock, але з використанням потоків для визначення часу.

10 trivial_thread_2 - Те саме, що і trivial_thread_1, але використовує два потоки замість одного.

11 file_cache_time - скопіювати змінну, визначивши час читання файлу з увімкненим або вимкненим кешуванням

```
  process<(param,nocache):
   posix_memalign(&buf,pagesize,
                  pagesize);
   fd=open("/tmp/file.txt", writing);
   fcntl(fd, F_NOCACHE, nocache);
   for(i=0; ireading);
   fcntl(fd, F_NOCACHE, nocache);
   start = time();
   for(i=0; islow process(param):
   process(param,1):
fast process(param):
   process(param,0):
```

12 file_cache_thread_1 - Те саме, що і file_cache_time, але з використанням потоків для визначення часу.

13 file_cache_thread_2 - Те саме, що і file_cache_thread_1, але використовує два потоки замість одного.

14 mem_cache_time - скопіювати змінну, визначивши час зчитування кешу даних ЦП з кешуванням або без нього

```
  posix_memalign(&buf1,pagesize,64);
posix_memalign(&buf2,pagesize,64);
slow process(param):
   for(i=0; i< param;i++){
      asm ("mfence\n"
      "clflush (%0)\n"::"r"(buf1));
      sum += *((long *)buf1);
      *((long *)buf1) = sum;
   }
```

```
fast process(param):
    for(i=0;i < param;i++){
        asm ("mfence\n"
        "clflush (%0)\n"::"r"(buf1));
        sum += *((long *)buf2);
        *((long *)buf2) = sum;
    }
```

15 mem_cache_thread_1 - Те саме, що і mem_cache_time, але з використанням потоків для визначення часу.

16 mem_cache_thread_2: Те саме, що і mem_cache_thread_1, але використовує два потоки замість одного.

17 time_jit - Скопіюйвати змінну, визначивши час для jitted функції

```
  int freq=0;
void foo(input,output) {
    static void (*foop)(...,...);
    if (freq==0) {
        foop = JIT(bytecodes)
        freq++;
    }
    (*foop)(input,output);
}
slow process(param):
    freq=0;
    start=time();
    foo(...,...);
    time=time()−start;


fast process(param);
    freq=0;
    foo(...,...);
```

```
start=time();
foo(...,...);
time=time()−start;
```

Самомодифікація

.

Мета цих перетворень - зробити функції самомодифікованими. Приклад виклику Tigress.

```
> tigress −−Environment=x86_64:Darwin:Clang:5.1 \
        −−Transform=Virtualize\
          −−Functions=add \
          −−VirtualizeDispatch=direct \
        −−Transform=SelfModify \
          −−Functions=add \
          −−SelfModifyFraction=%100 \
          −−SelfModifySubExpressions=false \
          −−SelfModifyOperators=\* \
          −−SelfModifyKinds=\* \
          −−SelfModifyBogusInstructions=0 \
  inc.c −−out=obf.c
```

Перетворення вбудовує шаблон двійкового коду у верхній частині функції. Повинен бути один шаблон для кожного типу (int/long), один для арифметичних операцій, один для порівнянь і один для гілок. Виглядають приблизно так:

```
addrPtr10 = (unsigned long *)((unsigned long )(&& Lab_1) + 5);
 *addrPtr10 = (unsigned long )(& A);
 addrPtr10 = (unsigned long *)((unsigned long )(&& Lab_1) + 18);
 *addrPtr10 = (unsigned long )(& B);
 addrPtr10 = (unsigned long *)((unsigned long )(&& Lab_1) + 41);
 *addrPtr10 = (unsigned long )(& C);
```

```
addrPtr10 = (unsigned long *)((unsigned long )(&& Lab_1) + 61);
*addrPtr10 = (unsigned long )(&& Lab_2);
Lab_1:
__asm__ volatile (
    ".byte 0x50;\n" // push %eax/rax
    ".byte 0x51;\n" // push %ecx/rcx
    ".byte 0x56;\n" // push %rsi/esi
    ".byte 0x48, 0xb8,0xc,0x7b,0x21,0x3d,0x54,0x2d,0xc0,0x1;\n" //movabs &A,%rax
    ".byte 0x8b, 0x00;\n" // movl (%rax),%eax
    ".byte 0x48, 0xb9,0x4f,0x40,0x2f,0xa3,0xd0,0xdc,0x24,0x42;\n" //movabs &B,%rcx
    ".byte 0x8b, 0x09;\n" // (%rcx),%ecx
    ".byte 0x31, 0xf6;\n" // xorl %rsi,%rsi
    ".byte 0x39, 0xc8;\n" // cmpl %rcx, %rax
    ".byte 0x40, 0x0f, 0x9c, 0xc6;\n" // setl %sil
    ".byte 0x48, 0xb9,0x7f,0x91,0x1c,0xaf,0xab,0x5f,0x2d,0x6a;\n" // movabs &C,%rcx
    ".byte 0x67, 0x89, 0x31;\n" // movl %esi,(%rcx)
    ".byte 0x5e;\n" // popq %rsi/esi
    ".byte 0x59;\n" // popq %ecx/rcx
    ".byte 0x58;\n" // popq %eax/rax
    ".byte 0xff, 0x25, 00, 00, 00, 00, 0xba,0xee,0x19,0x51,0x13,0x4b,0x12,0xdb;\n" // jmp
        (%rip)
:);
Lab_2:
```

Потім для кожної двійкової операції в решті функції вставляється наступний код:

```
A = i;
  B = n;
  addrPtr10 = (unsigned long *)((unsigned long )(&& Lab_3) + 6);
  *addrPtr10 = (unsigned long )(&& Lab_1);
  addrPtr10 = (unsigned long *)((unsigned long )(&& Lab_1) + 61);
  *addrPtr10 = (unsigned long )(&& Lab_4);
  opPtr11 = (int *)((unsigned long )(&& Lab_1) + 35);
```

```
*opPtr11 = 0xc69c0f40L; // setl %sil
Lab_3:
__asm__ volatile (
  ".byte 0xff, 0x25, 00, 00, 00, 00, 0x3b,0x21,0x2e,0xd7,0xc5,0x44,0xf0,0xdd;\n" //jmp
      (%rip)
:);
Lab_4:
```

Використання віртуалізації та самозаміни особливо корисна після перетворень, які вводять непрямі команди переходу, наприклад віртуалізація та згладжування з прямою або непрямою диспетчеризацією. Нижче наведено приклад:

```
void add(void) {
 ...

 //------------------ Instruction Handler
     ----------------------
 Lab_1: _1_add__load_int$left_STA_0$result_STA_0:
 (_1_add_$pc[0]) ++;
 (_1_add_$sp[0] + 0)->_int = *((int *)(_1_add_$sp[0] + 0)->_void_star);

 // BEGIN indirect branch to the next instruction starts here
 branchAddr10 = (unsigned long *)((unsigned long )(&& Lab_3) + 6);
 *branchAddr10 = *(_1_add_$pc[0]);
 Lab_3: /* CIL Label */
 __asm__ volatile (".byte 0xff, 0x25, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00":);
 // END

 //------------------ Instruction Handler
     ----------------------
 _1_add__returnVoid$:
 (_1_add_$pc[0]) ++;
```

```
  return;
  ...
}
```

Кожний непрямий перехід перетворюється на наступний код, де послідовність байтів відповідає 8-байтовому прямому переходу X86:

```
addr = (unsigned long *)((unsigned long )(&& Lab) + 6);
 *addr = address-to-jump-to;
 Lab: asm volatile(".byte 0xff, 0x25, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00":);
```

2.4 Перетворення на основі ollvm обфускатора

Інструмент для обфускації програмного коду на базі універсальної системи llvm, що не залежить від мови та переважно цільової архітектури, підримує заміни основних інструкції, вставки додаткових конструкції до потоку керування, злиття процедур, алгоритм захисту від коду від несанкціонованого доступу, перевірку контрольних сум. Зазвичай є потуєни інструментом для захисту від зворотньої розробки, та несакціонованого доступу, але також є досить актуальним для проведення змін з метою протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання.

На рисунку 2.4, використовуючи дизасемьлер Ida Free ми бачимо звичайне, не обфусковане представлення ассемблерового коду та потоку виконання простої програми.

2.4.1 Алгоритм Instructions substitution

.

Заміна інструкції є одним з найпростіших форм обфускації. Полягає в заміні стандартних двійкових операторів, таких як арифметичні або булеві,

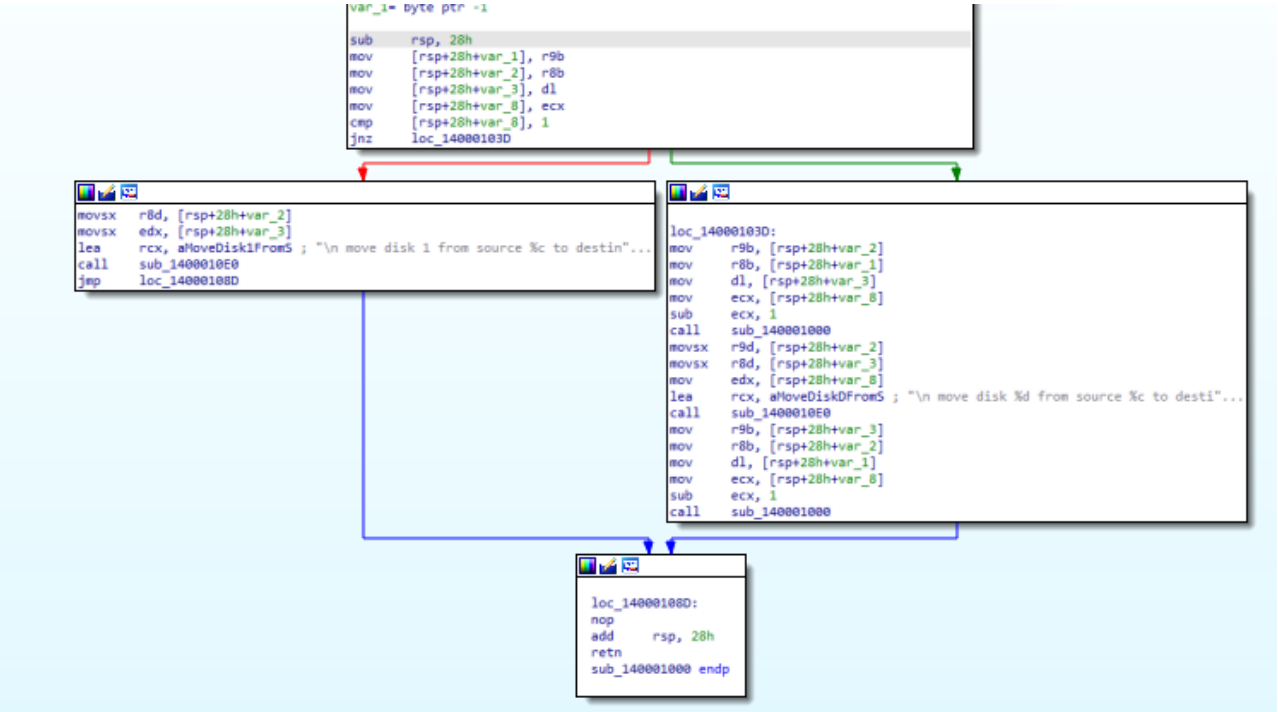

Рисунок 2.4 — Необфусковане представлення програми в Ida Pro

на функціонально еквівалентні але більш складні послідовності інструкції. Підтримується додавання, віднімання та такі логічні оператори як "or "and"та "xor". Розмір самої заміни є контрольованим, адже від цього залежить, результуючий розмір машинного коду, і відповідно самого виконувального файлу. Підміна деяких операторів не підтримується адже це може негативно вплинути на точність обчислень. Даний метод змінює такі ознаки як частоту інструкції, певні сигнатури послідовності байт, та власне саму частоту байтів, що може впливати на класифікацію. На рисунку 2.5 бачимо всього лише одну арифметичну операцію за допомогою інструкції sub ecx,1, а вже на наступному рисунку 2.6 багато різних арифметичних та булевих операції, які в результаті можна спростити до єдиної початкової операції віднімання.

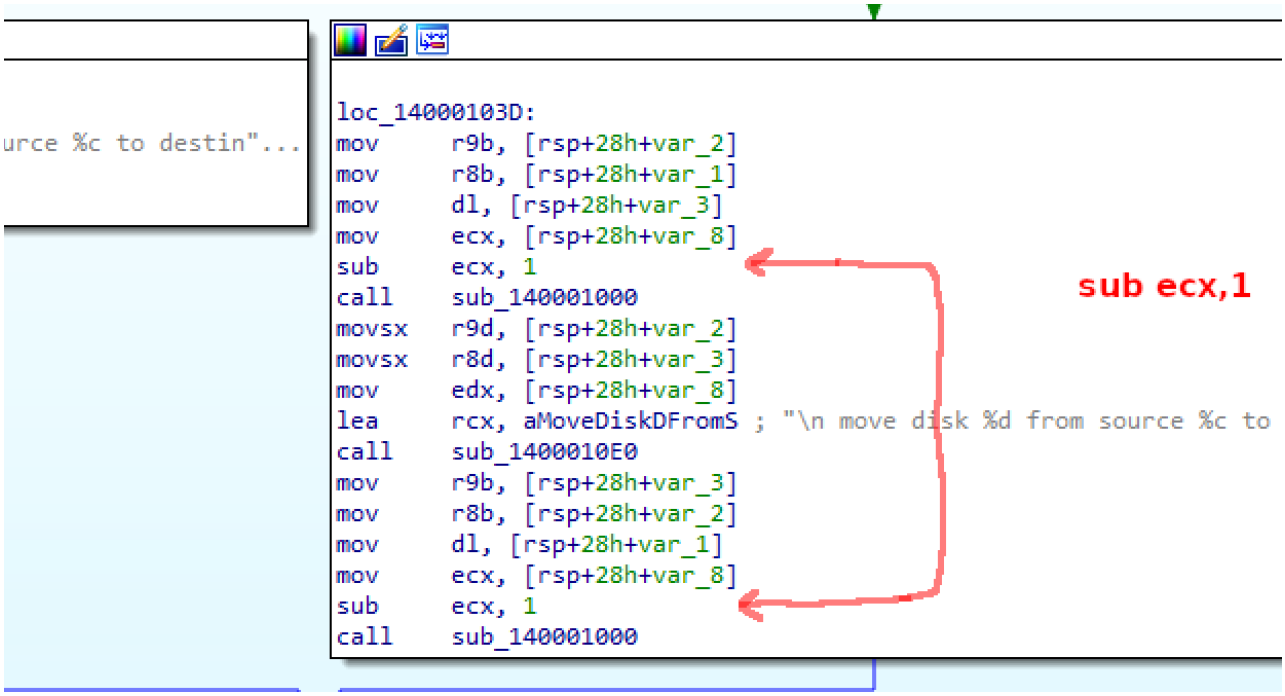

Рисунок 2.5 — Початковий код з операцією віднімання представлений єдиною інструкцією

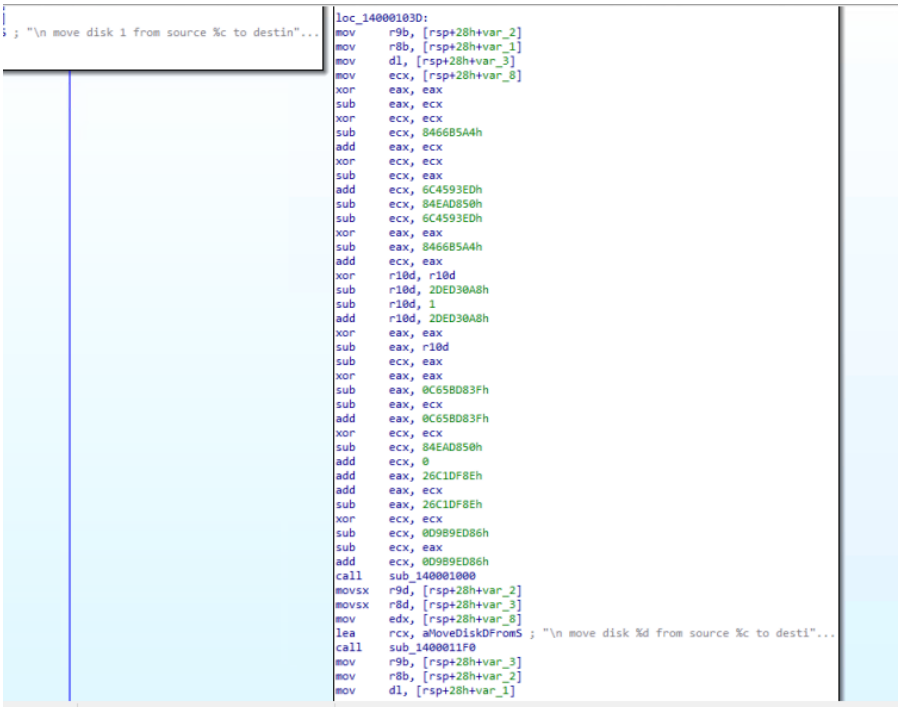

Рисунок 2.6 — обфускований код який також виконує віднімання

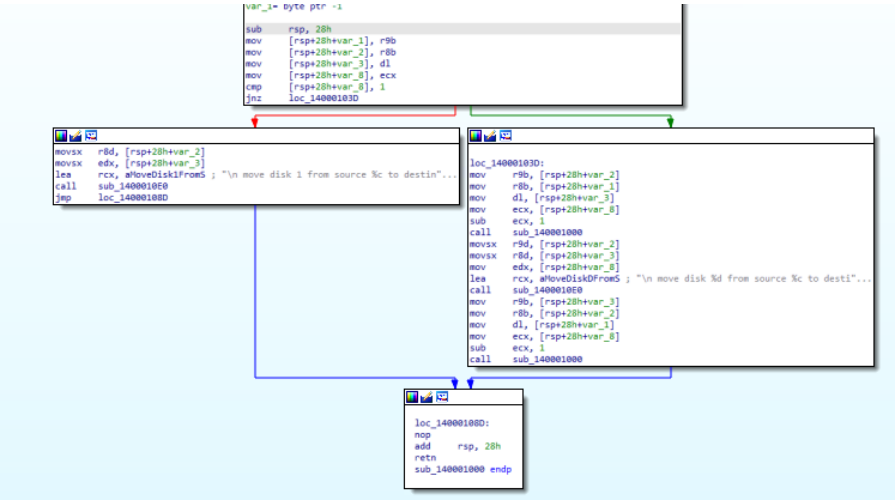

Рисунок 2.7 — Початковий набір блоків

2.4.2 Алгоритм Bogus Control Flow Insertion

.

Даний алгоритм модифікації потоку керування полягає у зміні функції графіка потоку керування шляхом додавання конструкції умовного переходу, яка вказує або на оригінальний базовий блок або на підроблений базовий блок, що повертається до оригінального. Під час роботи програми виконується лише початковий базовий блок. Навідмінно від попереднього алгоритму, замініінструкцій, оптимізатор не спростити результуючий графік викликів шляхом визначення мертвого коду. Алгоритм можна налаштувати різними способами, включаючи щільність вставок, кількість ітерацій тощо. Результуючий графік програмного коду є досить складний незважаючи на те що його частина є підробленою і ніколи не буде виконана.

На рисунку 2.7 зображений стандартний граф з 4 ма вузлами, які також присусті на наступному рисунку 2.8, але це важко помітити, адже до та пісоя них додана велика кількість блоків.

Даний алгоритм подібний до попереднього, але набагато більше заплутує сам потік виконання. Спочатку виділяються основні блоки функції та

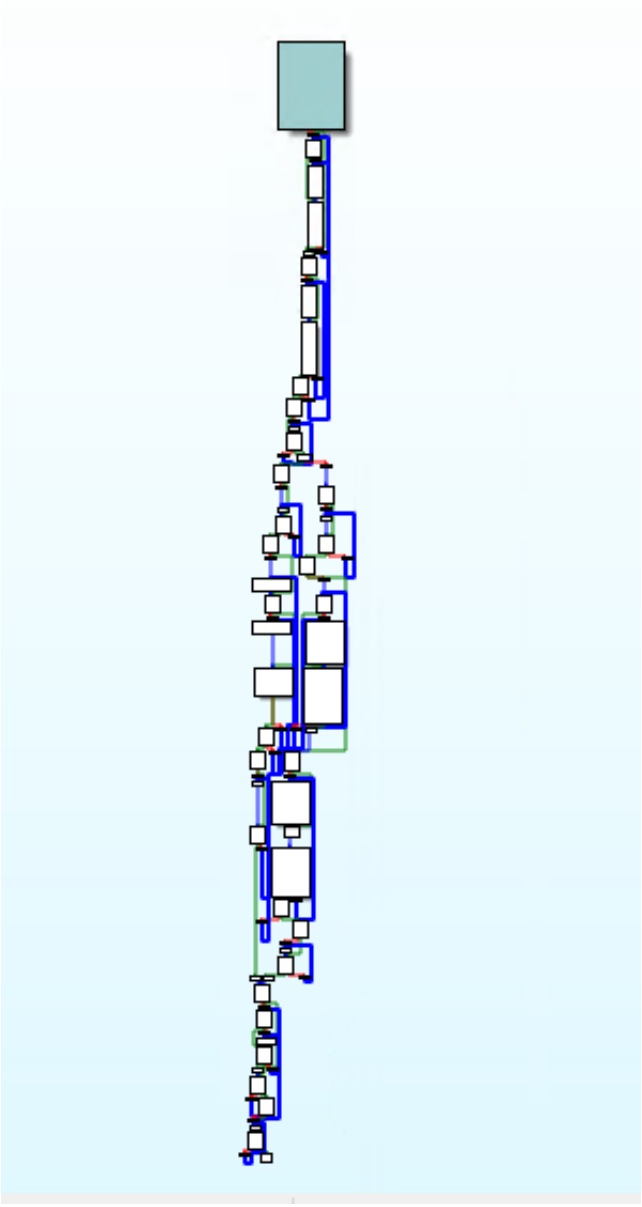

Рисунок 2.8 — Вигляд графу після обфускації алгоритмом Bogus Control Fwow

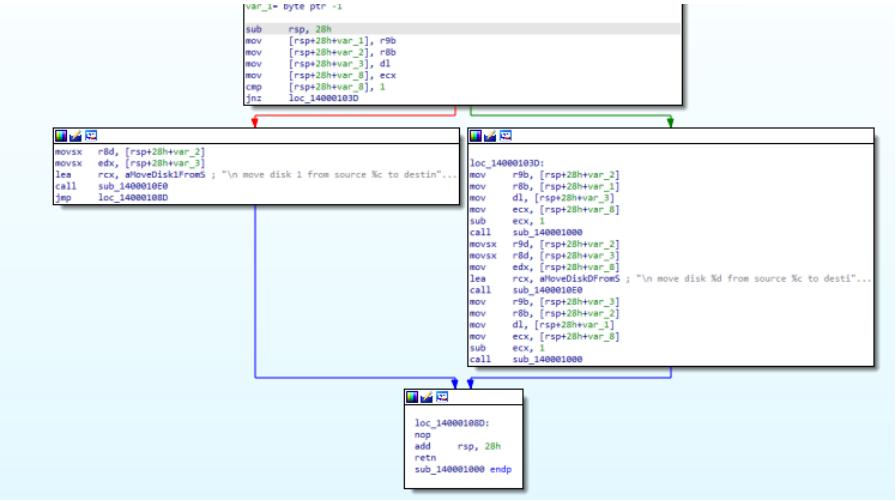

Рисунок 2.9 — Початковий набір блоків

виставляються на один рівень. Далі блоки поміщаються в певні труктури множинного вибору (наприклад оператор switch) та повторно поміщаються в цикли. На виході отримуємо графік заплутаного коду де всі блоки знаходять на одному рівні, приховубчи оригінальний потіх виконання.

Зазначимо мінімальний параметр обфускації для візуальної можливості розуміння алгоритму. На рисунку 2.9 зображений стандартний граф з 4 ма блокамию На рисунку 2.10, можна помітити ці 4 базові блоки на одному рівні, але програма заплутана додаванням інших блоків та розгалужень. При збільшенні параметру обфускації аналі стає вже неможливим.

2.4.3 Алгоритм String obfuscation

.

Складні алгоритми, що вимагають імпортування додаткових бібліотек, можуть збільшити ймовірність детектування засобами захисту. Тому краще використовувати такі алгоритми як XOR, RC4 або модифікації даних алгоритмів. На рисунку 2.11 зображений код з видимими текстовими данимим На рисунку 2.12, даний текст вже зашифрований, та додатково згенеровані інструкції для розшифрування.

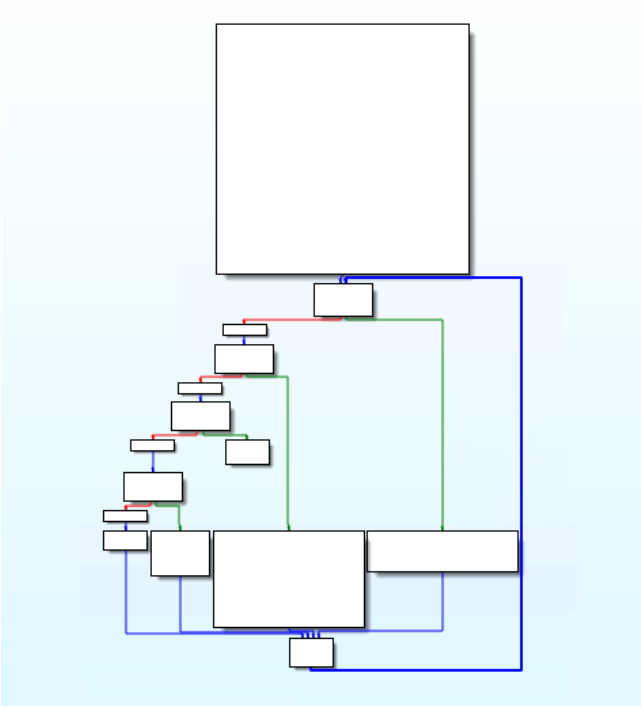

Рисунок 2.10 — Вигляд графу після обфускації алгоритмом Control Flow Flattening

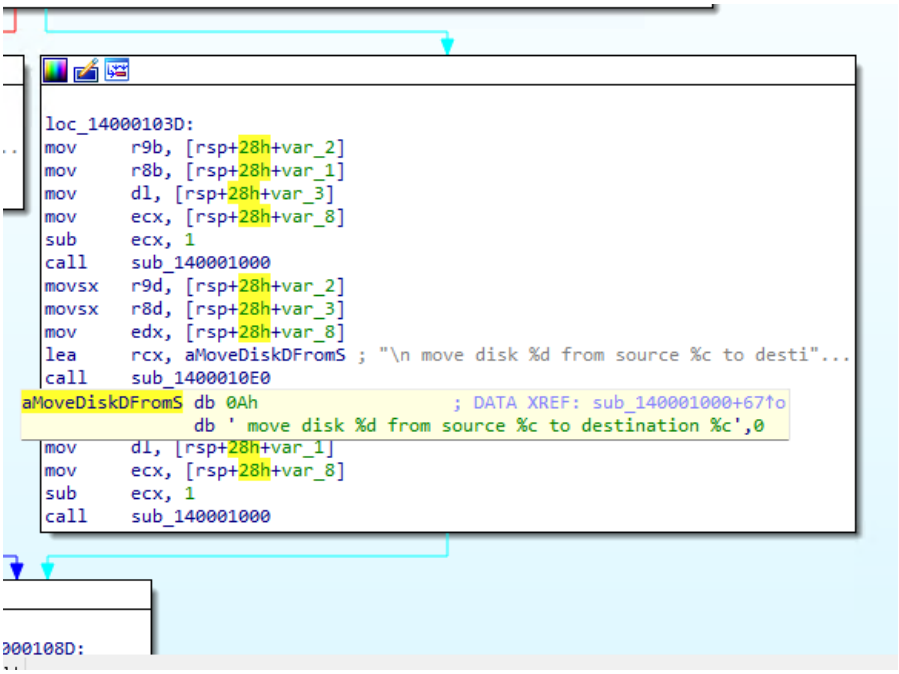

Рисунок 2.11 — Початковий текст

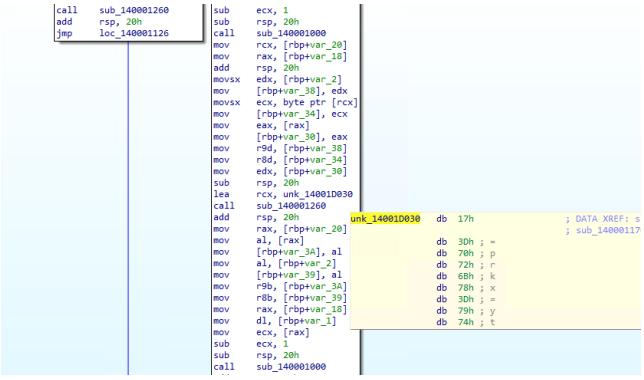

Рисунок 2.12 — Зашифрований вигляд

2.5 Загальна модель

Проаналізовані алгоритми дозволяють змінювати ознаки, що вибираються для побудови алгоритмів класифікації, розглянутик в розділі 1. Таким чином протидіяти сучасним методам класифікації шкідливого програмного забезпечення. Алгоритми протидії застосовуються на всіх рівнях програмного коду. Обфускація може використовуватись як на початковому коді так і на проміжному поданні, а мутації з додаванням байтів, зміною структури шкідливого файлу зостосовуються безпосередньо до виконувального файлу.

Перетворення застосовуються на всіх етапах створення програми. Далі описано, які саме перетворення виконуються на кожному з етапів.

1 Перетворення початкового коду за допомогою обфускатора Tigress

1.1 Virtualize

1.2 Jit

1.3 JitDynamic

1.4 Flatten

1.5 Split

1.6 Merge

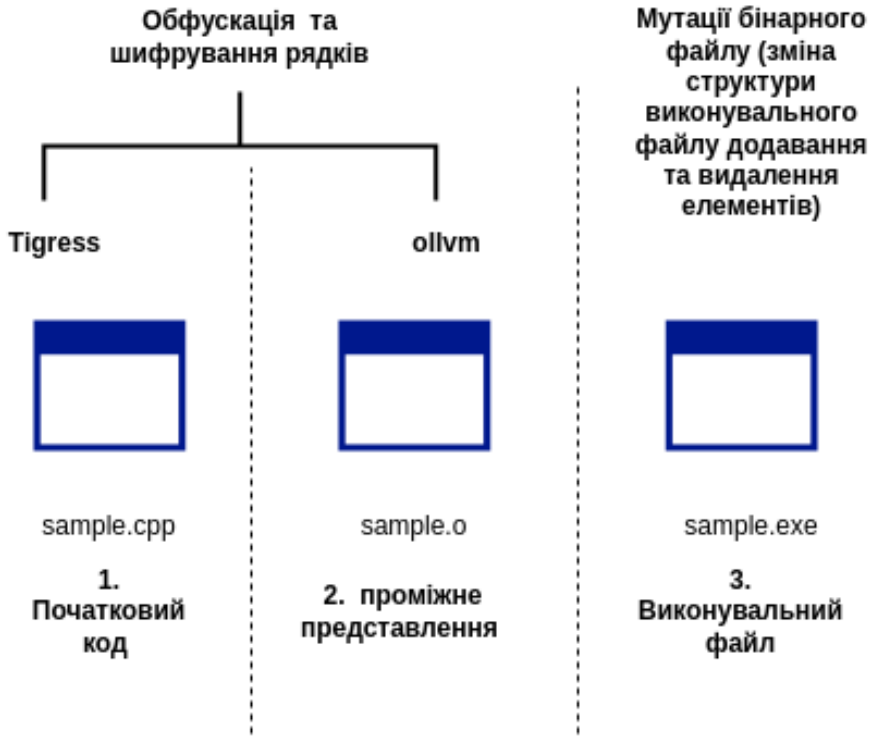

Рисунок 2.13 — Загальна модель протидії

2.3 Перебудова потоку виконання

2.4 Шифрування текстових рядків

3 Мутації над виконувальним файлом

3.1 Додавання імпортованих функцій

3.2 Створення нової секції

3.3 Перейменування секцій

3.4 Додавання до існуючої секції

3.5 Додавання випадкових байтів

3.6 Додавання/видалення сигнатури

3.7 Пакування

Загальна ілюстрація програмної моделі представлена на рисунку 2.13

ВИСНОВКИ З РОЗДІЛУ 2

В розділі запропоновано моделі протидії класифікаторам шкідливого програмного забезпечення на основі машинного навчання. Виділено основні способи перетворення виконувального файлу на всіх рівнях з метою зміни атрибутів, що відбираються для класифікації методами машинного навчання розглянутими в попередньому розділі. Запропоновано моделі протидії класифікаторам на основі алгоритмів обфускації та протидії зворотній розробці програмного коду, пронаналізовано вплив даних алгоритмів на атрибути шкідливого рпограмного забезпечення, що використовується для класифікації.

3 ЕКСПЕРЕМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

3.1 Базові статичні модифікації в PE файлі

В таблиці 3.1 зображено результати класицікації 50-ти зразків за допомогою malconv. Ті які класифікувались як шкідливе програмне забезпечення позначкні цифрою один, а нулем ті, які вдалось класифікувати як доброякісні, після мутацій згаданих в розділі 2. В таблиці 3.2 відсоткове відношення результатів класифікації мутованих зразків за допомогою моделі malconv.

Таблиця 3.1 — Результати класифікації malconv після базових статичних модифікацій (1- ШПЗ , 0 - доброякісний)

	1	2	3	4	5	6	7	8	9	10
Malconv	0	1	0	1	1	0	0	0	0	1
	11	12	13	14	15	16	17	18	19	20
Malconv	1	1	0	1	1	1	0	0	0	1
	21	22	23	24	25	26	27	28	29	30
Malconv	0	0	0	0	0	0	1	0	0	0
	31	32	33	34	35	36	37	38	39	40
Malconv	0	1	1	1	0	1	0	0	0	1
	41	42	43	44	45	46	47	48	49	50
Malconv	0	0	0	1	0	1	1	0	0	0

Таблиця 3.2 — Відстокове відношення результатів класифікації malconv

ШПЗ	Доброякісні
38%	62%

В таблиці 3.3 зображено результати класицікації за допомогою ember. В таблиці 3.4 відсоткове відношення результатів класифікації мутованих зразків за допомогою моделі ember.

Таблиця 3.3 — Результати класифікації ember після базових статичних модифікацій (1- ШПЗ , 0 - доброякісний)

	1	2	3	4	5	6	7	8	9	10
Ember	0	0	0	0	0	0	0	0	0	0
	11	12	13	14	15	16	17	18	19	20
Ember	0	0	0	1	0	0	0	0	0	0
	21	22	23	24	25	26	27	28	29	30
Ember	0	0	0	0	0	0	0	0	0	0
	31	32	33	34	35	36	37	38	39	40
Ember	0	0	0	0	0	1	0	0	0	0
	41	42	43	44	45	46	47	48	49	50
Ember	0	0	0	0	0	0	0	0	0	0

Таблиця 3.4 — Відстокове відношення результатів класифікації ember

ШПЗ	Доброякісні
4%	96%

В таблиці 3.5 зображено результати класицікації за допомогою Nonneg_malcon

. В таблиці 3.6 відсоткове відношення результатів класифікації мутованих зразків за допомогою моделі ember.

Таблиця 3.5 — Результати класифікації Nonneg_malconv після базових статичних модифікацій (1- ШПЗ , 0 - доброякісний)

	1	2	3	4	5	6	7	8	9	10
Nonneg_malconv	0	0	0	0	0	0	0	0	0	0
	11	12	13	14	15	16	17	18	19	20
Nonneg_malconv	0	0	0	0	0	0	0	0	0	0
	21	22	23	24	25	26	27	28	29	30
Nonneg_malconv	0	0	0	0	0	0	0	0	0	0
	31	32	33	34	35	36	37	38	39	40
Nonneg_malconv	0	0	0	0	0	1	0	0	0	0
	41	42	43	44	45	46	47	48	49	50
Nonneg_malconv	0	0	0	0	0	0	0	0	0	0

Таблиця 3.6 — Відстокове відношення результатів класифікації Nonneg_malconv

ШПЗ	Доброякісні
2%	98%

3.2 Перетворення на основі обфускатора

Обфускація проводиться у два етапи. Спочатку над початковим кодом за допомогою tigress. Нижче наведено приклад F1 - назва обфускованого файлу, F1,F2 функціїї над якими виконувались перетворення, такі як шифрування змінних, віртуалізація та перебудова функцій

F1=client_obf

FN1=encrypt,is_zero,main,randombytes,to_hex

FN2=A,add,add1305,car25519,Ch,core,crypto_box_curve25519xsalsa20poly1305_tweet,
crypto_box_curve25519xsalsa20poly1305_tweet_afternm,
crypto_box_curve25519xsalsa20poly1305_tweet_beforenm,
crypto_box_curve25519xsalsa20poly1305_tweet_keypair,
crypto_box_curve25519xsalsa20poly1305_tweet_open,
crypto_box_curve25519xsalsa20poly1305_tweet_open_afternm,
crypto_core_hsalsa20_tweet,crypto_core_salsa20_tweet,
crypto_hashblocks_sha512_tweet,crypto_hash_sha512_tweet,
crypto_onetimeauth_poly1305_tweet,crypto_onetimeauth_poly1305_tweet_verify,
crypto_scalarmult_curve25519_tweet,crypto_scalarmult_curve25519_tweet_base,
crypto_secretbox_xsalsa20poly1305_tweet,
crypto_secretbox_xsalsa20poly1305_tweet_open,crypto_sign_ed25519_tweet,
crypto_sign_ed25519_tweet_keypair,crypto_sign_ed25519_tweet_open,
crypto_stream_salsa20_tweet,crypto_stream_salsa20_tweet_xor,
crypto_stream_xsalsa20_tweet,crypto_stream_xsalsa20_tweet_xor,
crypto_verify_16_tweet,crypto_verify_32_tweet,cswap,dl64,inv25519,L32,ld32,M,Maj,
modL,neq25519,pack,pack25519,par25519,pow2523,R,reduce,S,scalarbase,scalarmult,
sel25519,set25519,sigma0,Sigma0,sigma1,Sigma1,st32,ts64,unpack25519,unpackneg,vn,Z

```
FN=$FN1,$FN2

tigress --Transform=EncodeLiterals --Functions=$FN --Transform=EncodeArithmetic
    --Functions=$FN --Transform=Flatten --Functions=$FN --Transform=
    Virtualize --Functions=$FN --Environment=x86_64:Darwin:Clang:5.1 --out=$F1
    -tigress.c $F1.c -o $F1-tigress.bin
```

Далі, сформувавши обфускований початковий код, відбувається додаткова обфускація на етапі проміжного коду.

```
clang-cl /MT -D__CUDACC__ -
    D_ALLOW_COMPILER_AND_STL_VERSION_MISMATCH -mllvm -bcf -
    mllvm -bcf_prob=10 -mllvm -bcf_loop=2 -mllvm -sub -mllvm -sub_loop=3 -
    mllvm -split_num=9 -mllvm -sobf -mllvm -split -mllvm -aesSeed=
    DEADBEEFDEADCODEDEADBEEFDEADCODE client_obf-tigress.c -o
    client_obf-tigress.exe
```

Варто зазначити, що перетворинння, які призводять до протидії класифікаторам шкідливого забезпечення, не повинні шкодити роботі самої програми, сповільнювати її роботу, та занадто сильно збільшувати у розмірі (рекомендований розмір ШПЗ до 5МБ).

Тому було вибрано найоптимальніші значення параметрів ollvm.

```
1 -mllvm -bcf -mllvm -bcf_prob=10 -mllvm -bcf_loop=2

2 -mllvm -sub -mllvm -sub_loop=3

3 -mllvm -fla -mllvm -split_num=9

4 -mllvm -sobf -mllvm -split
```

Для того щоб кожний раз генерувати різні перетворення, використовувався також параметр -aesSeed=

Для проведення даного експерименту було відібрано, зразки що мали вихідні коди на ресурсі https://github.com/vxunderground/MalwareSourceCode. Зауважимо, що деякі з них не класифікувались як ШПЗ ще до початку проведених перетворень(Таблиця 3.7).

Таблиця 3.7 — Результати класифікації до проведення перетворень на основі обфускації (1- ШПЗ , 0 - доброякісний)

	1	2	3	4	5	6	7	8	9	10
Nonneg_malconv	1	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9	10
Ember	0	1	0	1	1	0	1	1	1	0
	1	2	3	4	5	6	7	8	9	10
Malconv	0	1	1	1	1	0	1	1	0	1

Результати класифікації після проведення перетворень на основі обфускації зображені на таблиці 3.8

Таблиця 3.8 — Результати класифікації після проведення перетворень на основі обфускації (1- ШПЗ , 0 - доброякісний)

	1	2	3	4	5	6	7	8	9	10
Nonneg_malconv	0	0	1	1	0	0	1	0	0	0
	1	2	3	4	5	6	7	8	9	10
Ember	0	0	0	1	0	0	1	0	0	0
	1	2	3	4	5	6	7	8	9	10
Malconv	0	1	0	0	0	0	1	0	0	1

На таблиці 3.9 бачимо відсоток зразків, класифікованих як легітимне програмне забезпучення до та після перетворень. Одночасно виконавши перетворення над виконувальним файлом та обфускацію над початковим і проміжним кодом, вдається повністю протидіяти використаним класифікаторам.

Таблиця 3.9 — Відсоток зразків, класифікованих як легітимне програмне забезпучення до та після перетворень

	Nonneg_malconv	Ember	Malconv
До	40 %	50 %	30 %
Після	70 %	80 %	80 %

ВИСНОВКИ З РОЗДІЛУ 3

В даному розділі проведено експерементальні дослідження та визначено, що за допомогою мутацій двійкового файлу можна збільшити ефективність протидії до 62%-98%, та використовуючи перетворення початкового та проміжного коду за допомогою обфускатора, збільшити ефективність протидії від 30-40% до 70-80%. Отримані результати можуть бути використані для протидії існуючим системам антивірусного захисту, систем виявлення вторгненя, та в якості основи для подальших досліджень методів протидії класифікаторам шкідливого програмного забезпечення та систем захисту на основі машинного навчання. Запропоновані моделі та методи дозволяють виявити слабкі місця класифікаторів на основі машинного навчання, та показує, що дослідженні методи не можуть застосовуватись самі по собі, а повинні комбінуватись з іншими методами виявлення, такими як поведінковий та евристичний методи.

ВИСНОВКИ

В першому розділі виконано критичний огляд існуючих методів класифікації шкідливого програмного забезпечення на основі машинног навчання, особливостей виділення атрибутів. Проведений аналіз дозволяє сформолювати актуальну тему дослідження, створення методів протидії класифікації ШПЗ на основі машинног навчання. В другому розділі запропоновано моделі протидії класифікаторам ШПЗ на основі машинного навчання. Виділено основні способи перетворення виконувального файлу на всіх рівнях з метою зміни атрибутів, що вибираються для класифікації. Запропоновані моделі протидії класифікаторам на основі алгоритмів обфускації, проаналізовано вплив алгоритмів на атибути ШПЗ, що використовуються для класифікації. В третьому розділі проведено експерементальні дослідження та визначено, що за допомогою мутацій двійкового файлу можна збільшити ефективність протидії до 62%-98%, та використовуючи перетворення початкового та проміжного коду за допомогою обфускатора, збільшити ефективність протидії від 30-40% до 70-80%. Отримані результати можуть бути використані для протидії існуючим системам антивірусного захисту, систем виявлення вторгненя, та в якості основи для подальших досліджень методів протидії класифікаторам шкідливого програмного забезпечення та систем захисту на основі машинного навчання. Запропоновані моделі та методи дозволяють виявити слабкі місця класифікаторів на основі машинного навчання, та показує, що дослідженні методи не можуть застосовуватись самі по собі, а повинні комбінуватись з іншими методами виявлення, такими як поведінковий та евристичний методи. Запропоновано програмну модель протидії класифікаторам

шкідливого програмного забезпечення на основі машинного навчання. Можливіть використання даної моделі для підвищення ефективності антивірусного захисту та методів на основі штучного інтелекту.

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАНЬ

1. Combing through the fuzz: Using fuzzy hashing and deep learning to counter malware detection evasion techniques - Microsoft 365 Defender Research Team. — Режим доступу: https://www.microsoft.com/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-counter-malware-detection-evasion-techniques/.

2. Hyrum S. Anderson Phil Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models.

3. Edward Raff Jon Barker Jared Sylvester Robert Brandon Bryan Catanzaro Charles Nicholas. Malware Detection by Eating a Whole EXE.

4. William Fleshman Edward Raff Jared Sylvester Steven Forsyth Mark McLean. Non-Negative Networks Against Adversarial Attacks.

5. Jiaqi Yan Guanhua Yan Dong Jin. Classifying Malware Represented as Contro Flow Graphs using Deep Graph Convolutional Neural Network.

6. Edwin Huang Sabrina Ho. Malware Detecting using Control Flow Graphs.

7. Pesidious -Create Mutated Malware Using Artificial Intelligence.

8. Anderson, H., Kharkar, A., Filar, B., Evans, D. and Roth, P. (2018). Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. — Режим доступу: https://arxiv.org/abs/1801.08917.

9. Hu W., Tan Y. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN.

10. Pesidious. — Режим доступу: https://vaya97chandni.gitbook.io/pesidious/.

11. the tigress c obfuscator. — Режим доступу: https://tigress.wtf/index.html.

12. Kinder Johannes. Towards Static Analysis of Virtualization-Obfuscated Binaries, WCRE'12.

13. Taylor Claire, Collberg Christian. Getting RevEngE: A System for Analyzing Reverse Engineering Behavior,.

14. Laszlo, Kiss. Obfuscating C++ Programs via Control Flow Flattening.