МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

Кваліфікаційна робота магістра

«ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ КІБЕРЗАХИСТУ МОБІЛЬНИХ ПЛАТФОРМ»

Здобувач освіти гр. ІН.м – 13 Андрій МАЛЬЦЕВ

Науковий керівник,
доцент, к.т.н В'ячеслав МОСКАЛЕНКО

Завідувач кафедри
доцент, к.т.н Ігор ШЕЛЕХОВ

Суми 2022

Сумський державний університет

(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«122 - Комп'ютерні науки»*

Затверджую:

В.о.зав.кафедри ________________________

____

"________" ______________________ 20___р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Мальцеву Андрію Віталійовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна технологія кіберзахисту мобільних платформ*
затверджую наказом по інституту від "____" _________________ 20 ___ р. № _____________

2. Термін здачі студентом закінченого проекту (роботи)_________________________

3. Вхідні данні до проекту (роботи) ____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Огляд літератури та аналіз існуючих рішень 2) Постановка завдання й формування завдань дослідження; 3) Вибір методу класифікації для інтелектуальної системи; 4) Розробка інформаційної технології розпізнавання; 5) Аналіз результатів.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) ________

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання ____

Керівник ________________________
(підпис)

Завдання прийняв до виконання ________________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	Аналіз проблеми. Постановка мети, об’єкту та предмету дослідження	19.10–25.10	
2.	Огляд літератури та аналіз існуючих рішень	26.10–06.11	
3.	Вибір методу класифікації для інтелектуальної системи	07.11–17.11	
4.	Розробка інформаційної технології розпізнавання, проведення навчання моделі та аналіз результатів	18.11–04.12	
5.	Оформлення пояснювальної записки до дипломної роботи	04.12–11.12	

Студент – дипломник ________________________
(підпис)

Керівник проекту ________________________
(підпис)

РЕФЕРАТ

Записка: 72 стор., 18 рис., 1 додаток, 25 джерел.

Об'єкт дослідження – кіберзахист на мобільних платформах

Предмет дослідження – інформаційна технологія кіберзахисту для мобільних платформ

Мета дослідження – розроблення ефективної системи кіберзахисту для мобільних платформ

Результати — реалізована інтелектуальна система кіберзахисту мобільних платформ на основі класифікатора SVM і системі пояснень на основі градієнтного методу. Була сформована навчальна вибірка даних для класифікації та проведене тренування моделі. Результати роботи системи показали високу точність класифікації шкідливих зразків ПЗ. Програмна реалізація здійснена з використанням мови Python, бібліотеки SecML та ApkTool.

МАШИННЕ НАВЧАННЯ, КІБЕРЗАХИСТ, МОБІЛЬНІ ПЛАТФОРМИ, ШКІДЛИВЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, PYTHON, SECML, APKTOOL, MANIFEST, ANDROID, DALVIK, LINUX, ANDROIDMANIFEST.XML, API

ЗМІСТ

РЕФЕРАТ..4
ВСТУП..7
1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ 9
1.1 Аналіз існуючих підходів до виявлення шкідливих програм на мобільних платформах..9
1.2 Аналіз існуючих моделей і методів класифікаційного аналізу даних 14
1.2.1 Статичний аналіз на основі маніфесту за допомогою ML 15
1.2.2 Статичний аналіз на основі коду за допомогою ML..............................26
1.2.3 Статичний аналіз на основі маніфесту і коду 31
1.2.4 Динамічний метод...35
1.2.5 Гібридний метод 41
1.2.6 Виявлення на основі емуляції 43
1.3 Формалізована постановка задачі 44
2 ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗПІЗНАВАННЯ ШКІДЛИВИХ ПРОГРАМ НА МОБІЛЬНИХ ПЛАТФОРМАХ..45
2.1 Модель екстракції ознакового опису спостережень 45
2.2 Модель і метод навчання класифікатора шкідливих програм......................46
2.2.1 Визначення навчальних векторів 47
2.2.2 Оцінка навчальних векторів...50
2.2.3 Пояснення класифікації додатків за допомогою процесів Гауса 52
2.3 Критерії оцінювання ефективності моделі розпізнавання шкідливих програм. 53

3 ПРОГРАМНА РЕАЛІЗАЦІЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ РОЗПІЗНАВАННЯ ШКІДЛИВИХ ПРОГРАМ НА МОБІЛЬНИХ ПЛАТФОРМАХ 54

3.1 Формування навчальних та тестових даних 54

3.1.1 Завантаження датасету 54

3.1.2 Навчання класифікатора...54

3.1.3 Пояснення рішень 54

3.1.4 Створення змагальних прикладів...56

3.1.5 Оцінка безпеки 57

3.2 Короткий опис програмного забезпечення..57

3.2.1 Apktool...57

3.2.2 Python...58

3.2.3 SecML 58

3.3 Результати машинного навчання 59

ВИСНОВКИ 63

ЛІТЕРАТУРА 64

ДОДАТОК А 66

ВСТУП

Останнім часом у зв'язку зі стрімким розвитком смартфонів платформа Android стає все більш популярною. Відповідно до згідно зі звітом IDC [1] , частка ринку Android сягає 85,3%. Ця платформа стає все більш і більш незамінною завдяки своєму відкритому коду та перевагам безкоштовності в нашому повсякденному житті. Однак, кількість шкідливого програмного забезпечення також стрімко зростає. Отже, виявлення шкідливого програмного забезпечення Android з високою точністю є гострою проблемою.

Централізація ринку в додатках для мобільних платформ зробила виявлення зловмисного програмного забезпечення дуже виснажливим завданням для більшості методів виявлення, навіть для машинного навчання та методів штучного інтелекту.

Традиційно було розроблено численні інструменти виявлення зловмисного програмного забезпечення для Android, але деякі інструменти не можуть виявити новостворену програму зловмисного програмного забезпечення та програму зловмисного програмного забезпечення, інфіковану різними троянами, хробаками, шпигунським програмним забезпеченням тощо. Виявлення великої кількості шкідливих програм у мільйонах програм для Android все ще є складним завданням. Дослідження мобільної безпеки стали предметом занепокоєння. Низка досліджень у мобільних технологіях, починаючи від дизайну, уразливості, загроз і методів виявлення, триває. Багато галузей безпеки витрачають мільярди коштів на цю сферу.

Традиційний підхід виявлення на основі сигнатури широко використовується як у пристроях Android, так і на платформі ПК завдяки вилучення підпису з APK і порівняння зі зловмисним підписом у вірусній базі даних, однак цей підхід обмежений виявленням невідомих зловмисних програм, яких немає в вірусній базі даних. Щоб вирішити це питання, попередні дослідники виявили, що існують дві методики аналізу невідомого зловмисного програмного забезпечення Android: статичний аналіз та динамічний аналіз [2]. Статичний аналіз, який виконується до встановлення програми

Android, є технікою на основі перевірки вмісту APK [3] за допомогою зворотного проектування. На відміну від статичного аналізу, динамічний аналіз відстежує стан роботи програми Android у віртуальному середовищі. З розвитком технології машинного навчання, підходи машинного навчання також згадуються для класифікації отриманих спостереження як доброякісні чи зловмисні шляхом збору різних сигнальних характеристик і подій із програм і систем, однак необроблені функції можуть мати нерелевантні або зайві функції, які можуть призвести до неправильного результату, тому процес вибору функцій є таким важливим. Виділення функцій і вибір функцій є ключовими кроками, які визначають показник точності класифікатора. Після цього буде легше досягти результату виявлення за допомогою класифікатора.

Керуючись наведеними вище спостереженнями, ми пропонуємо систему виявлення для платформи Android на основі навчального класифікатора за допомогою штучного інтелекту, яка буде стійка до атак на неї шляхом додавання в зловмисний застосунок доброякісних функцій або видаленням з неї злоякісних.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз існуючих підходів до виявлення шкідливих програм на мобільних платформах

Сьогодні Android швидко оновлюється, хоча основна архітектура ОС Android залишилися незмінною. Архітектуру Android можна розділити на чотири рівня.

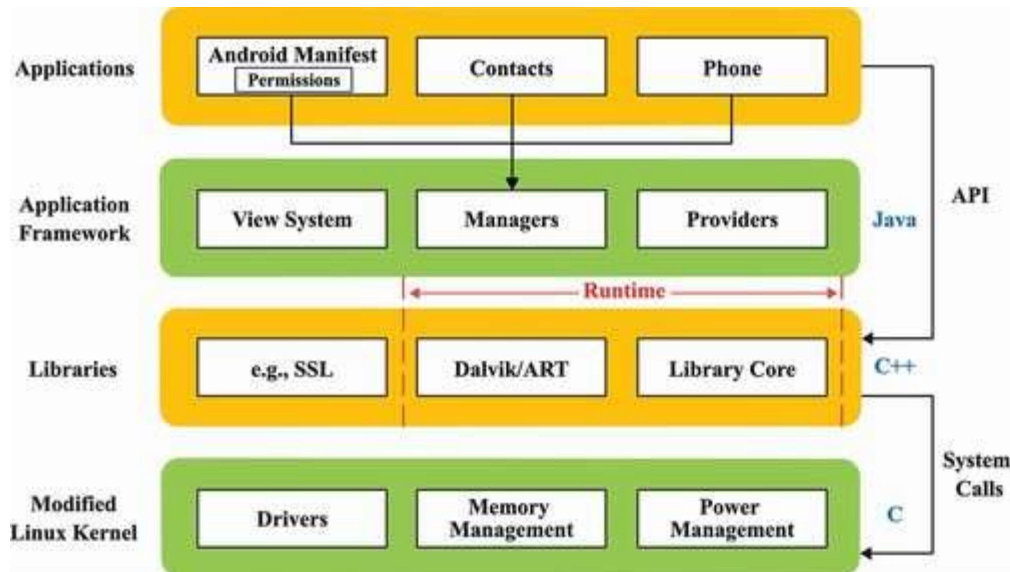


Рисунок 1.1 – Структура архітектури ОС Android

Як показано на рисунку 1.1.1, архітектуру Android можна розділити на модифікований рівень ядра Linux, бібліотеки, рівень бібліотеки середовища виконання системи та рівень інфраструктури додатків.

(1) Модифіковане ядро Linux. Основні системні служби, які надає Android, базуються на системі Linux, наприклад, безпека, керування живленням і драйвери. Діючи як рівень абстракції між апаратним і програмним забезпеченням, модифіковане ядро Linux приховує деталі апаратного рівня та надає послуги верхнім рівням для зменшення зв’язку.

(2) Бібліотеки. Механізм пісочниці процесів на базі Linux у бібліотечному рівні Android є одним із наріжних каменів усієї конструкції безпеки. Базуючись на модифікованому рівні ядра Linux для таких основних функцій, як керування потоками

та керування пам'яттю, віртуальна машина Dalvik оптимізована для ефективного запуску кількох екземплярів віртуальних машин одночасно в обмеженій пам'яті, і кожна програма Android виконується як процес Linux з екземпляром віртуальної машини Dalvik.

(3) Виконання системи. З точки зору загальної архітектури, розробник має активний контроль лише над системним рівнем виконання та структурами над ним, тому виявлення зловмисного програмного забезпечення Android також має зосереджуватися на тому самому місці. Його можна класифікувати на системну бібліотеку та середовище виконання Android. Основна бібліотека середовища виконання Android надає більшість API, таких як ОС Android, Android.net і Android.media.

(4) Рівень інфраструктури програми. Android має рівень інфраструктури додатків, який надає різноманітні API для розробки Android. Розробники можуть вільно використовувати ці API для створення своїх додатків з урахуванням обмежень безпеки реалізації фреймворку.

Чотири ключові компоненти Android: діяльність, служба, приймач трансляції та постачальник вмісту, які тісно пов'язані з поведінкою зловмисного програмного забезпечення Android.

(1) Діяльність. Активність надає користувачам графічне вікно для таких дій, як кнопки, текстові блоки, блоки введення тощо. Користувачі взаємодіють із програмою, торкаючись цих елементів. Діяльність зазвичай діє як проміжний рівень між користувачами та функціями програми, відповідальним за передачу намірів користувача.

(2) Сервіс. Сервіс часто використовується для трудомісткої логічної обробки у фоновому режимі, тому багато зловмисних дій пов'язано зі Сервісом через його невидимість для користувачів. Сервіс не працює в окремому процесі, а залежить від процесу програми, у якому було створено Сервіс.

(3) Трансляційний приймач (Broadcast Receiver). Будучи широко використовуваним механізмом для передачі інформації між програмами, Broadcast Receiver зазвичай використовується розробниками зловмисного програмного забезпечення для моніторингу різноманітних подій, пов'язаних із конфіденційною інформацією. Broadcast Receiver фільтрує, приймає та відповідає на вихідні трансляції. Broadcast Receiver дозволяє додаткам Android реагувати на зовнішню подію, наприклад увімкнення телефону, отримання текстового повідомлення чи телефонного дзвінка.

(4) Постачальник вмісту (Content Provider). Content Provider підтримує зберігання та зчитування даних у кількох програмах, виконуючи роль бази даних для програм, тому він забезпечує доступ до відкритих даних, таких як контактні книги та повідомлення для розробників шкідливих програм.

Необхідні дозволи та кожен із чотирьох компонентів, які використовуються в програмі Android, потрібно зареєструвати в AndroidManifest.xml. Таким чином, аналіз AndroidManifest.xml може дати загальне уявлення про функціональність і шкідливу поведінку програм. Файл AndroidManifest.xml зазвичай використовується як допоміжний індикатор для взаємодії з іншими методами аналізу для виявлення шкідливих програм.

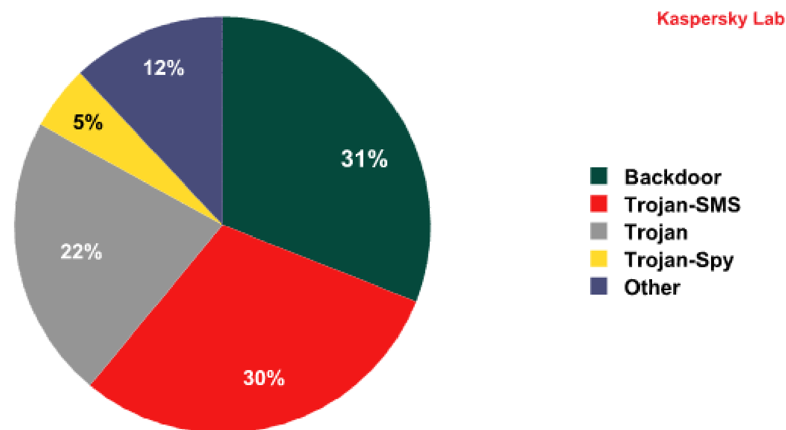


Рисунок 1.2 – доля різних шкідливих програм

Як показано на рисунку 1.2, найбільш поширені види шкідливих програм можна розділити на невелику кількість категорій. Далі ми розглянемо наступні:

- Бекдор — в комп'ютерній системі (криптосистемі або алгоритмі) — це метод обходу стандартних процедур автентифікації, несанкціонований віддалений доступ до комп'ютера, отримання доступу до відкритого тексту, і так далі, залишаючись при цьому непоміченим. Бекдор може приймати форму встановленої програми (наприклад, Back Orifice) або може проникнути у систему через руткіт;
- Троянські програми — різновид шкідницького програмного забезпечення, яке не здатне поширюватися самостійно (відтворювати себе) на відміну від вірусів та хробаків, тому розповсюджується людьми;
- Вірус: шкідлива комп'ютерна програма, котра розповсюджується шляхом вбудовування власного коду у код інших програм. Вірус може поширюватись від однієї програми до інших, також від одного комп'ютера до інших;
- Ransomware: програма – вимагач, котра блокує доступ до користувацьких даних на комп'ютері або в мережі шляхом шифрування файлів і подальшого повідомлення про необхідну матеріальну винагороду для зняття блокування;
- Spyware: шпигунське програмне забезпечення, котре встановлюється без відома користувача і використовується для збору персональної інформації про нього. Отримані дані надсилаються до зловмисників.

Як показано у вступному модулі, у виявленні Android існує два основних підходи до виявлення, які можна описати як статичний аналіз і динамічний аналіз.

Статичний аналіз, декомпіляція APK перед установкою програми, може забезпечити швидкий і безпечний результат для виявлення шкідливої програми. Дослідники зазвичай вибирають відстеження потоку даних або відповідну інформацію про атрибути з APK для того, щоб розрізняти шкідливі програми. Наприклад, Qin [4] аналізує небезпечні дозволи та зберігає їх у базі даних. Вони витягують дозволи з невідомої програми Android для того щоб після порівняння з небезпечним дозволом,

вони могли отримати висновок. Felt [5] пропонує інструмент під назвою Stowaway для виявлення додатків із надлишковими правами доступу або підозрілими викликами API. Yang [6] виявляє витік конфіденційної інформації на Android за допомогою статичного аналізу забруднення. Однак, статичний аналіз має обмеження для аналізу обфускованої програми [7] , тому результат може бути неправильним, якщо програма зашифрована.

Динамічний аналіз головним чином отримує функції, коли вже запущена програма Android, відстежуючи поведінку або стан конфіденційних даних, щоб шкідливе програмне забезпечення можна було виявити під час роботи програми. Qiao [8] представляє структуру під назвою CBM, яка витягує послідовності викликів API за допомогою інструменту динамічного аналізу поведінки, Tam [9] також розробляє автоматичну динаміку систем аналізу на основі VMI для ідентифікації шкідливих програм відповідно до динамічної поведінки. Проте динамічне виявлення часто не можуть точно визначити, коли і як запускати всі шкідливі дії [10] , більше того, динамічний аналіз вимагає багато часу для аналізу програми, що не підходить для смартфона.

Статичний аналіз чутливий до обфускації коду. Методи та результати виявлення динамічного аналізу можуть вплинути на пропуск ключових шляхів до виконуваних файлів. Ye та Cui та ін. проаналізував весь виконуваний файл шкідливого програмного забезпечення, який має тенденцію послабити шкідливі функції коду. Крім того, ці методи базуються на наявних зразках шкідливих програм і результатах у виявленні шкідливих програм має певний гістерезис.

Qiao [11] пропонує підхід машинного навчання для виявлення зловмисного програмного забезпечення шляхом аналізу шаблонів дозволів і API викликів функцій, отриманих та використаних програмами Android, вони встановлюють контакт між дозволом і API, однак, шкідливих зразків, які вони зібрали, недостатньо. Munoz [12] отримує метадані з Google Play, а потім вони розглядають метадані як функції для

застосування підходу машинного навчання для визначення найвищих прогнозних характеристик і виявлення шкідливих програм. Так само Peiravian [13] також пропонує структуру навчання на основі функцій, яка зосереджена на запитувані дозволи та поведінку викликів API, а також застосовує SVM, дерево рішень і алгоритми пакетування. Але вони лише витягують дозвіл та API як функцію, що призводить до низької точності [14] .

1.2 Аналіз існуючих моделей і методів класифікаційного аналізу даних

Такого моніторингу безпеки недостатньо через те, що мільйони розробників Android, пов'язаних із Google, чиї програми не перевіряються належним чином, перш ніж їх опублікують у Play Store. Це схоже на магазин телефонів і магазин додатків для Windows і Apple відповідно. Це призвело до того, що безпека мобільних пристроїв, особливо на базі Android, не була абсолютною на 100%. Розглянутий підхід до виявлення ґрунтувався головним чином на неправильному використанні та аномаліях із застосуванням, ненадійними даними та станом системи як основними цілями аналізу. Згідно з отриманим результатом, жоден із методів не забезпечив 100% виявлення зловмисного програмного забезпечення для мобільних пристроїв. Anusha динамічно порівнював поведінку зловмисних і мобільних програм за допомогою мобільного API для виявлення шкідливих програм. Розроблена система виявлення (WMMD) виявила затуманене мобільне шкідливе програмне забезпечення, яке не вдалося виявити антивірусному ПЗ. Підхід до виявлення використовував автомат кінцевого стану (FSA) для вибірки коду зловмисного програмного забезпечення та перевірки шаблонів. Використовуючи аналіз за часом виконання, модель виявила віруси як до, так і після пакування. Усі шість антивірусних програм, які використовували, жодне не змогло виявити ці мобільні віруси після запакування за допомогою UPX. Тоді стало вважатися, що більшість антивірусних програм не реагують на виявлення зловмисного програмного забезпечення, оскільки вони

базуються на сигнатурах. Цей результат базується на аналізі поведінки для виявлення зловмисного програмного забезпечення в програмах Android. Дослідження видобувало 216 і 278 окремо для звичайних і шкідливих програм Android. Різноманітність навчених математичних алгоритмів було застосовано як до безпечного, так і до шкідливого набору даних. Використовуючи кореляційний аналіз, дослідження показало точність виявлення 97,16%. Розглядаючи попередні артефакти зловмисного програмного забезпечення в мобільній пам'яті, дослідники виконали аналіз виявлення зловмисного програмного забезпечення в мобільній пам'яті за допомогою методології криміналістичної експертизи пам'яті. Дослідження з точністю 90% виявили троян, що самовідтворюється, з 20% некласифікованих зразків. Дослідження показало, що значна інформація про зловмисне програмне забезпечення може бути отримана з аналізу дампа пам'яті мобільних пристроїв Android. Приховані коди, які чекають вибуху за сприятливих умов, легко викриваються. Дослідження, однак, не змогло забезпечити жодного дослідження щодо пошуку атрибутів зловмисного програмного забезпечення, важливих для криміналістичного аналізу та аналізу безпеки. Була проведена детальна робота над методами виявлення, але не було досліджень, які б ідентифікували та перерахували обмеження та сильні сторони цих методів. Визначення як обмежень, так і переваг допоможе підвищити ефективність цих методів і покращити виявлення шкідливих програм на пристроях Android.

У цьому розділі буде розглянуто наступні методи виявлення зловмисного програмного забезпечення:

1.2.1 Статичний аналіз на основі маніфесту за допомогою ML

Система Android має жорсткий механізм управління дозволами для обмеження поведінки програм. ОС Android надає близько 75 дозволів із середнім контролем, високим рівнем інформації та низькою інтерактивністю. Деякі системні функції не можна викликати за замовчуванням, коли запущена програма Android, наприклад телефонні дзвінки, SMS, Bluetooth, WIFI тощо. Ці функції зазвичай відповідають

конкретним апаратним пристроям. Щоб використовувати ці функції, програмі необхідно надати відповідні дозволи за допомогою тегу в AndroidManifest.xml.

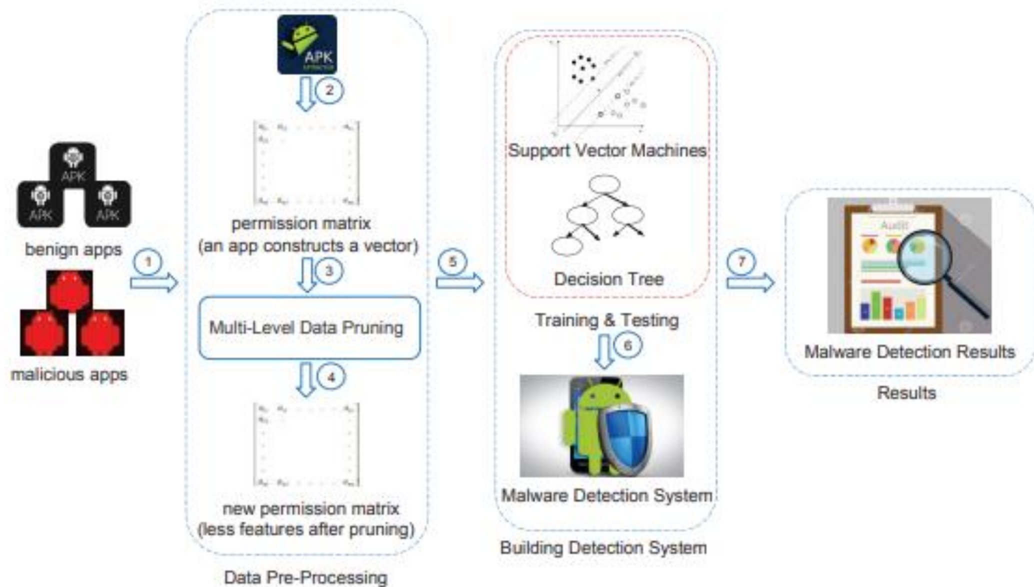


Рисунок 1.3 – Схема статичного аналізу на основі маніфесту

На рисунку 1.3 представлена схема статичного аналізу, яка зазвичай присутня у більшості моделей, побудованих на визначенні шкідливих програм на основі маніфесту. Перший компонент SIGPID (Significant Permission Identification) — це багаторівневий процес скорочення даних для визначення важливих дозволів, щоб усунути потребу розглядати всі доступні дозволи в Android. Жодна програма не запитує всі дозволи, а ті, які запитує програма, перераховані в пакеті програми Android (APK) як частину manifest.xml. Коли нам потрібно проаналізувати велику кількість програм (наприклад, кілька сотень тисяч), загальна кількість дозволів, запитуваних усіма програмами, може бути надзвичайно великою, що призводить до тривалого аналізу. Такі високі витрати на аналіз можуть негативно вплинути на ефективність виявлення зловмисного програмного забезпечення, оскільки це знижує продуктивність аналітика. Ми пропонуємо три рівні методів скорочення даних, щоб відфільтрувати дозволи, які мало впливають на ефективність виявлення зловмисного програмного

забезпечення. Таким чином, їх можна безпечно видалити без негативного впливу на точність виявлення шкідливих програм.

Потім ми опишемо кожен рівень у процесі обрізки. 1) Ранжування дозволів із негативним коефіцієнтом (PRNR): кожен дозвіл описує певну операцію, яку програмі дозволено виконувати. Наприклад, дозвіл INTERNET вказує, чи має програма доступ до Інтернету. Різні типи нешкідливих програм і шкідливих програм можуть запитувати різноманітні дозволи відповідно до їхніх операційних потреб. Для шкідливих програм ми припускаємо, що їхні потреби можуть мати загальні підмножини, і нам не потрібно аналізувати всі дозволи, щоб побудувати ефективну систему виявлення шкідливих програм. У результаті, з одного боку, ми більше зосереджуємося на дозволах, які створюють поверхні для атак із високим ризиком і які часто запитують зразки зловмисного програмного забезпечення. З іншого боку, дозволи, які рідко запитують зразки зловмисного програмного забезпечення, також є хорошими показниками для розрізнення шкідливих програм від нешкідливих. Таким чином, наша процедура скорочення визначає обидва типи дозволів, які сильно відрізняються, щоб ми могли використовувати цю інформацію для класифікації шкідливих і безпечних програм. У той же час ми виключаємо дозволи, які зазвичай використовуються як безпечними, так і шкідливими програмами, оскільки вони вносять неоднозначність у процес виявлення зловмисного програмного забезпечення. Наприклад, дозвіл ІНТЕРНЕТ часто запитують як зловмисне програмне забезпечення, так і безпечні програми, оскільки майже всі програми запитують доступ до Інтернету. Тому наш підхід обрізає дозвіл ІНТЕРНЕТ.

Щоб визначити ці два типи важливих дозволів, ми розробляємо схему ранжирування дозволів, щоб ранжувати дозволи на основі того, як вони використовуються шкідливими та безпечними програмами. Рейтинг не нова концепція. У попередніх роботах також використовувалася загальна стратегія ранжирування дозволів, така як взаємна інформація, щоб ідентифікувати дозволи з

високим ризиком. Однак їхні підходи, як правило, зосереджуються лише на дозволах із високим ризиком і ігнорують усі дозволи з низьким рівнем ризику, які в нашому підході визначаються як важливі дозволи. Причина того, що попередні роботи ігнорують дозволи з низьким рівнем ризику, полягає в тому, що вони зацікавлені в ідентифікації дозволів, які зловживає зловмисне програмне забезпечення, тоді як наша мета полягає в тому, щоб відрізнити зловмисне програмне забезпечення від безпечних програм. По суті, ризиковані дозволи зосереджуються лише на дозволах, які можуть допомогти виявити зловмисне програмне забезпечення, тоді як важливі дозволи не лише піклуються про ідентифікацію зловмисного програмного забезпечення, але й беруть до уваги, чи можна ідентифікувати безпечні програми чи ні. Наш підхід, який називають рейтингом дозволів із негативним коефіцієнтом або PRNR, забезпечує стислий рейтинг і зрозумілий результат. Підхід працює з двома матрицями, M і B. M представляє список дозволів, які використовуються зразками зловмисного програмного забезпечення, а B представляє список дозволів, які використовуються доброякісними програмами. Mij показує, чи запитує j-й дозвіл i-й зразок шкідливого програмного забезпечення, тоді як «1» означає «так», «0» означає «ні». Bij показує, чи запитує j-й дозвіл i-та доброякісна вибірка програми. Перед обчисленням підтримки дозволів з матриць M і B спочатку перевіряємо їхні розміри. Як правило, кількість безпечних програм (310 926 у цій статті) значно перевищує кількість шкідливих програм (5 494 зразки в цій статті); отже, розмір B набагато більший за розмір M. З нашою схемою ранжирування вважаємо за краще, щоб набір даних у двох матрицях був збалансованим. Навчання на незбалансованому наборі даних може призвести до спотворення моделей. Щоб збалансувати дві матриці, використовуємо рівняння 1 для розрахунку підтримки кожного дозволу у більшому наборі даних, а потім пропорційно зменшуємо відповідну підтримку, щоб відповідати меншому набору даних. У випадку, якщо кількість рядків B більша, ніж кількість рядків M, маємо:

$$S_B(P_j) = \frac{\sum_i Bij}{size(B_j)} * size(M_j) \tag{1.2.1}$$

де Pj позначає j-й дозвіл, а SB(Pj) представляє підтримку j-го дозволу в матриці B. PRNR може бути реалізований за допомогою рівняння 2:

$$R(P_j) = \frac{\sum_i M_{ij} - S_B(P_j)}{\sum_i M_{ij} + S_B(P_j)} \tag{1.2.2}$$

Алгоритм PRNR використовується для ранжування наших наборів даних. У наведеній вище формулі R(Pj) представляє швидкість j-го дозволу. Результат R(Pj) має значення в межах [-1, 1]. Якщо R(Pj) = 1, це означає, що дозвіл Pj використовується лише в шкідливому наборі даних, що є дозволом із високим ризиком. Якщо R(Pj) = -1, це означає, що дозвіл Pj використовується лише в безпечному наборі даних, який є дозволом із низьким ризиком. Якщо R(Pj) = 0, це означає, що Pj дуже мало впливає на ефективність виявлення зловмисного програмного забезпечення. Оскільки і -1, і 1 важливі, створюємо два ранжируваних списки: один список генерується в порядку зростання на основі значення R(Pj), а інший список генерується в порядку спадання. Далі розробляємо нову інкрементну систему дозволів (PIS), щоб включити дозволи на основі порядку в двох списках. Наприклад, вибираємо найкращий дозвіл у списку безпечних і найкращий дозвіл у списку зловмисних як наші вхідні функції для створення системи виявлення зловмисного програмного забезпечення. Потім оцінюємо виявлення зловмисного програмного забезпечення за такими показниками: частота справжніх позитивних результатів (TPR), частота хибних позитивних результатів (FPR), точність (P = TPR TPR+FPR), відкликання (R = TPR TPR+FNR), точність і F-міра (2P*R P+R). Потім вибираємо перші три дозволи в обох списках, щоб створити систему виявлення зловмисного програмного забезпечення. Ми повторюємо процес ще раз, збільшуючи кількість основних дозволів для виявлення зловмисного програмного забезпечення, доки показники виявлення не досягнуть плато. Основна мета полягає в тому, щоб знайти найменшу кількість дозволів, яка дає таку ж

ефективність виявлення зловмисного програмного забезпечення, як і використання всього набору даних. У нашому наборі даних оцінки з 5494 доброякісними програмами з 310926 загальна кількість окремих дозволів, запитуваних усіма програмами, становить 135. Після застосування PRNR виявили, що використання 95 дозволів працює майже так само добре, як використання повних 135 дозволів.

Рейтинг дозволів на основі підтримки (SPR): щоб ще більше зменшити кількість дозволів, зосереджуємося на підтримці кожного дозволу. Як правило, якщо підтримка дозволу занадто низька, це не має великого впливу на продуктивність виявлення зловмисного програмного забезпечення. Наприклад, знаходимо дозвіл BIND TEXT SERVICE лише в доброякісних програмах. У результаті можемо вважати будь-яку програму, яка використовує BIND TEXT SERVICE, безпечною. Однак ці дозволи фактично використовує лише одна програма з понад 310 926 доброякісних програм. Отже, покладатися лише на ранжований показник, наданий PRNR, може бути неточним. Завдяки підтримці дозволів можемо виключити дозволи з низькою підтримкою, щоб ще більше підвищити точність класифікації зловмисного програмного забезпечення. Подібно до PRNR, використовуємо PIS, щоб знайти найменшу кількість дозволів із високою підтримкою, забезпечуючи високу точність. У нашому наборі даних після застосування SPR можемо зменшити кількість значущих дозволів лише до 25 або 19% від загальної кількості дозволів.

Інтелектуальний аналіз дозволів із правилами асоціації (PMAR): після скорочення 110 із 135 дозволів за допомогою PRNR та SPR із PIS хочемо продовжити дослідження підходів, які можуть ще більше видалити невпливові дозволи. Перевіряючи скорочений список дозволів, який містить 25 важливих дозволів, знаходимо три пари дозволів, які завжди з’являються разом у програмі. Наприклад, дозвіл WRITE SMS і дозвіл READ SMS завжди використовуються разом. Вони також належать до списку «небезпечних» дозволів Google. Проте немає необхідності розглядати обидва дозволи, оскільки одного з них достатньо для характеристики

певної поведінки. В результаті можемо асоціювати того, хто має вищу підтримку, до свого партнера. У цьому прикладі можемо видалити дозвіл WRITE SMS. Щоб знайти дозволи, які зустрічаються разом, пропонуємо механізм аналізу дозволів із правилами асоціації (PMAR) з використанням алгоритму аналізу правил асоціації. Інтелектуальний аналіз правил асоціації використовувався для виявлення значущих зв’язків між змінними у великих базах даних. Наприклад, якщо події A та B завжди відбуваються одночасно, дуже ймовірно, що ці дві події пов’язані. У цьому документі розглядаємо лише правила з високою впевненістю, тому застосування PMAR створить лише невелику кількість правил. Ми використовуємо Apriori, широко використовуваний алгоритм видобутку асоціацій, щоб створити правила асоціації. Apriori використовує стратегію пошуку в ширину для підрахунку підтримки наборів елементів і використовує процес генерації кандидатів, який використовує властивість опори закривати вниз.

Зрештою, визначаємо 3 правила асоціації на основі нашого набору даних дозволів, що відповідають трьом парам пов’язаних дозволів, коли встановлюємо 96,5% мінімальної надійності та 10% мінімальної підтримки. Нарешті можемо видалити три додаткові дозволи, залишивши 22 дозволи, які вважаємо важливими.

Спочатку використовуємо SVM і невеликий набір даних, щоб перевірити запропоновану нами модель MLDP. SVM визначає гіперплощину, яка розділяє обидва класи з максимальним запасом на основі навчального набору даних, який включає доброякісні та шкідливі програми. У цьому випадку один клас пов’язаний зі зловмисним програмним забезпеченням, а інший – із безпечними програмами. Потім припускаємо, що дані тестування є невідомими програмами, які класифікуються шляхом відображення даних у векторному просторі, щоб визначити, чи є вони на зловмисній чи доброякісній стороні гіперплощини. Потім можемо порівняти всі результати аналізу з їхніми оригінальними записами, щоб оцінити правильність виявлення зловмисного програмного забезпечення запропонованої моделі за

допомогою SVM. Щоб продемонструвати застосовність і масштабованість MLDP, використовуємо 67 загальновідомих алгоритмів машинного навчання та розширюємо наш набір даних. Порівнюємо результати між частотою виявлення зловмисного програмного забезпечення з використанням усіх ідентифікованих 135 дозволів (базова лінія) та виявлення зловмисного програмного забезпечення за допомогою MLDP для кожного контрольованого алгоритму машинного навчання. Ми спостерігаємо, що, аналізуючи всі дозволи, алгоритми машинного навчання з деревовидною структурою зазвичай покращують виявлення зловмисного програмного забезпечення порівняно з іншими. Серед усіх алгоритмів машинного навчання з деревовидною структурою наш метод найкраще працює на дереві рішень, яке є дуже поширеним методом класифікації в машинному навчанні. Дерево рішень має використовувати навчальний набір даних для побудови класифікатора, щоб вирішити, яким класом повинні бути дані тестування, вимагає багато попередньої обробки перед тим, як буде створено класифікатор. Коли атрибутів навчального набору даних забагато, дерево рішень має низьку точність, а фаза навчання займає більше часу та пам’яті. Отже, скорочення дерева рішень є обов’язковим, що узгоджується з нашим MLDP.

Отже, більш вигідно використовувати MLDP для виявлення зловмисного програмного забезпечення, оскільки це може бути настільки ж ефективним, заощаджуючи час і пам’ять. Оскільки час і пам’ять у звичайних комп’ютерах обмежені, можемо передати завдання в хмару відповідним чином для підвищення ефективності. Оскільки дозволи можна ідеально записати за допомогою двійкового вектора, могли б видалити таблицю зіставлення та виконати певну перестановку перед аутсорсином, щоб зберегти конфіденційність даних.

Цей метод використовує алгоритм PCA (аналіз основних компонентів) для вибору функцій після отримання дозволів і застосовує методи SVM (Support Vector Machine) для класифікації зібраних даних як безпечних або шкідливих.

1) Модуль попередньої обробки: його роль полягає в розпакуванні кожного файлу пакета програми Android, отриманні файлу AndroidManifest.xml із видобутого вмісту, а потім його декомпіляції. Нарешті отримайте список дозволів для кожного файлу .apk із його декомпільованого AndroidManifest.xml. Усі ці вектори дозволів утворюють оригінальний набір функцій;

2) Вибір функцій: у цьому модулі вихідний набір функцій, отриманий від модуля попередньої обробки, буде оброблено алгоритмом PCA для виділення основного компонента;

3) Класифікатор SVM: під час фази навчання класифікатору надається навчальний набір, що складається з векторів функцій зразків зловмисного програмного забезпечення та безпечного програмного забезпечення. На етапі виявлення навчений класифікатор може класифікувати вектори вхідних ознак невідомого APK;

4) Набір даних про функції: цей модуль відповідає за зберігання та оновлення функцій, отриманих із зразків.

Автор провів експерименти з 234 доброякісними зразками додатків і 220 зразками зловмисного програмного забезпечення, а результати експериментального моделювання показують, що:

- Ця система може досягти високого рівня виявлення невідомих шкідливих програм, хоча бібліотека зразків обмежена;
- Порівняно з традиційним антивірусним програмним забезпеченням, воно може ефективно та негайно виявляти невідоме шкідливе програмне забезпечення;
- Це демонструє, що використання лише функцій дозволів із методами машинного навчання може досягти хороших результатів виявлення;
- Хоча змовні програми можуть уникати виявлення цією системою.

Ця модель базується на виборі ознак як першу фазу, створення моделі кластеризації K-Mean як другу фазу, класифікацією цього нового набору даних, який

генерується другою фазою, як третю фазу та, нарешті, оцінку продуктивності цієї моделі з точки зору точності та запам'ятовування.

Загальні кроки, які мають бути дотримані для кожної програми:

1. Завантажити та зребегти зловмисне та доброякісне програмне забезпечення з ринку програм;

2. Розпакувати програми, щоб витягнути вміст;

3. Витягнути функції запиту дозволу з кожної програми;

4. Створити набір даних у форматі файлу ARFF із витягнутими даними.

Ми розпаковуємо файл пакета програми Android, щоб витягнути вміст. Протягом перших трьох кроків було отримано інформацію з цього джерела. Потрібно обробити файл AndroidManifest.xml, щоб отримати ці дані.

Ця схема виявлення зловмисного програмного забезпечення Android базується на комбінаціях дозволів, заявлених у файлі маніфесту програми. Він пропонує ефективну схему безпеки Droid Detective для виявлення шкідливих програм Android. На рис. 5 показано процес генерації набору правил k-map. Вони отримали комбінації дозволів, які часто запитують зловмисне програмне забезпечення, але рідко доброякісні програми. Методи виявлення шкідливих програм на основі дозволів широко використовуються в літературі. Цзян і Чжоу з NCSU опублікували результати аналізу на основі 1260 зразків зловмисного програмного забезпечення для Android, зібраних з різних Android Market протягом більш ніж одного року. Вони порівняли 20 найпопулярніших дозволів, які запитують зразки зловмисного програмного забезпечення, і дозволи, які запитують безпечні програми на Google Android Market.

Результат аналізу:

1) Дозволи, які часто запитують у зловмисних і безпечних програмах:

- INTERNET;
- READ_PHONE_STATE;
- ACCESS_NETWORK_STATE;

- WRITE_EXTERNAL_STORAGE.

2) Дозволи переважно запитують шкідливі програми, але рідко доброякісні програми:

- READ_SMS;
- WRITE_SMS;
- SEND_SMS;
- RECEIVE_SMS.

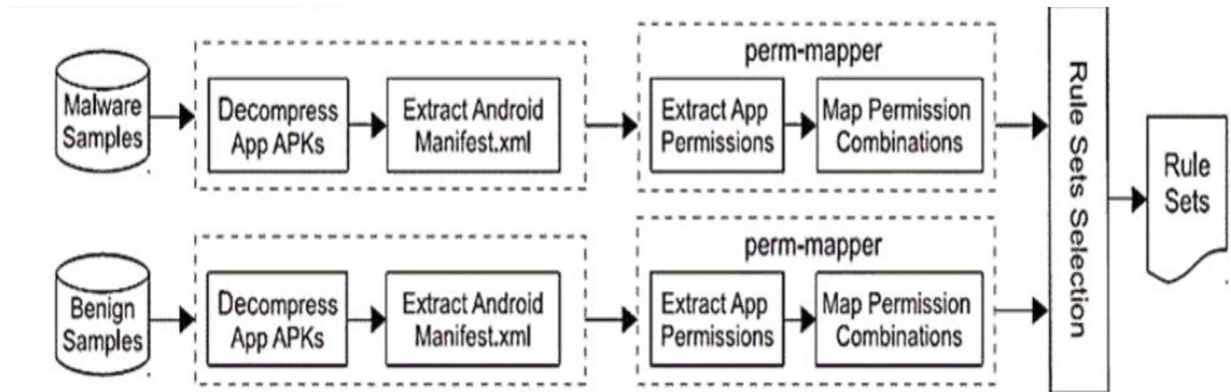


Рисунок 1.4 – Процес генерації сету дозволів

На рисунку 1.4 можна побачити схему, за якою генеруються два сету дозволів, для доброякісних і злоякісних програм відповідно. На основі комбінацій дозволів Android вони розробили та реалізували схему автоматичної генерації правил і ефективну схему виявлення шкідливих програм на основі правил – Droid Detective. Показано відповідність між створеними правилами та відповідними діями шкідливих програм. Були проведені експерименти з впровадження Droid Detective з реальними зразками шкідливого програмного забезпечення, які показали, що ця схема є дуже ефективною та ефективною. Хоча змовні програми також можуть уникнути виявлення за допомогою цієї схеми.

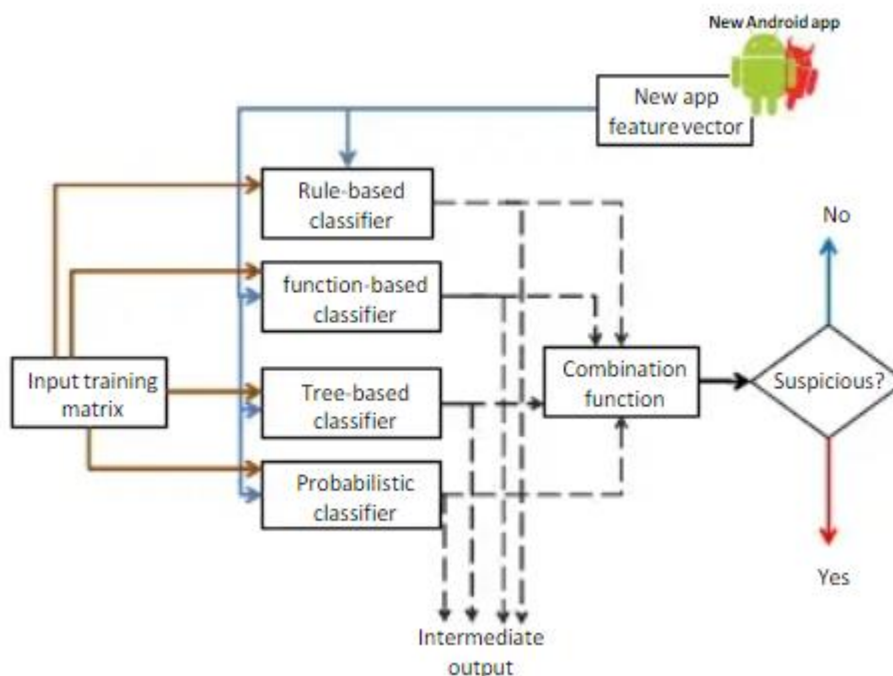


Рисунок 1.5 – Підхід з паралельним класифікатором

На рисунку 1.5 зображена схема роботи паралельного класифікатору програми Droid Detective. Під час перевірки, коли кількість комбінацій дозволів, запитуваних однією програмою, збільшується, показник звернень зменшується (оскільки потрібно задовольнити більше дозволів, щоб бути оголошеним зловмисним програмним забезпеченням), але коефіцієнт проходження доброякісних програм зростає.

1.2.2 Статичний аналіз на основі коду за допомогою ML

MaMaDroid створює модель послідовності викликів API у вигляді ланцюгів Маркова, які, у свою чергу, використовуються для вилучення функцій для алгоритмів машинного навчання, щоб класифікувати програми як доброякісні чи шкідливі. Абстракція. MaMaDroid фактично не використовує необроблені виклики API, а абстрагує кожен виклик до свого сімейства, пакету чи класу.

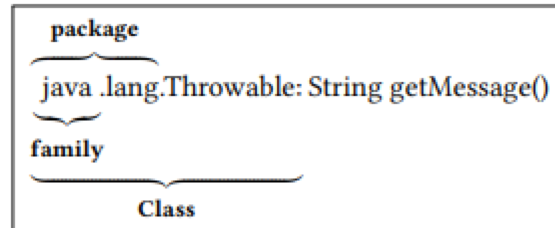


Рисунок 1.6 – Приклад виклику API, його родини, ім’я пакета і клас

Наприклад, на рисунку 1.6 продемонстрований виклик API getMessage(), в якому розбирається відповідно його компоненти java, java.lang і java.lang.Throwable.

Маючи три різні типи абстракцій, MaMaDroid працює в одному з трьох режимів, у кожному з яких використовується один із типів абстракції. Природно, очікується, що чим вище абстракція, тим легша система, хоча, можливо, менш точна. Будівельні блоки. Робота MaMaDroid проходить чотири етапи. Спочатку витягуємо графік викликів із кожної програми за допомогою статичного аналізу (1), потім отримуємо послідовності викликів API, використовуючи всі унікальні вузли, після чого абстрагуємо кожен виклик до класу, пакету або родини (2). Далі моделюємо поведінку кожної програми, будуючи ланцюжки Маркова з послідовностей абстрактних викликів API для програми (3), з ймовірностями переходу, які використовуються як вектор ознак, щоб класифікувати програму як безпечну або шкідливу програму за допомогою класифікатора машинного навчання. (4). У решті цього розділу докладно обговоримо кожен із цих кроків.

Першим кроком у MaMaDroid є вилучення графіка викликів програми. Робиться це шляхом статичного аналізу apk програми, тобто стандартного формату архівного файлу Android, який містить усі файли, включаючи байт-код Java, з яких складається програма. Ми використовуємо фреймворк оптимізації та аналізу Java, Soot [15] , щоб видобувати графіки викликів, і FlowDroid [16] , щоб забезпечити збереження контекстів і потоків. Зокрема, використовуємо FlowDroid, який базується на Soot, для створення фіктивного основного методу, який служить основною точкою входу в аналізовану програму (AUA). Робиться це тому, що програми Android мають кілька

точок входу, через які їх можна запускати або викликати. Незважаючи на те, що програми мають засіб запуску активності, який служить основною точкою входу, їх впровадження не є обов’язковим (наприклад, програми, які працюють як сервіс), отже, створення єдиної точки входу дозволяє нам надійно проходити AUA. FlowDroid також дозволяє моделювати потік інформації від джерел і приймачів, використовуючи ті, що надаються SuSi [17] , а також зворотні виклики

. На рис. 1.2.4 наведено клас, витягнутий із декомпільованого apk зловмисного програмного забезпечення, замаскованого під програму підсилення пам’яті (з назвою пакета com.g.o.speed.memboost), яка виконує команди (rm, chmod тощо) від імені root. Щоб спростити презентацію, зосереджуємося на частині коду, що виконується в блоці try/catch. Для простоти опустимо виклики для ініціалізації об’єкта, типів і параметрів, що повертаються, а також неявні виклики в методі. Додаткові виклики, які викликаються під час виклику getShell(true), не показані, за винятком методу add(), який безпосередньо викликається програмним кодом.

Вилучення послідовності. Оскільки MaMaDroid використовує статичний аналіз, графік, отриманий від Soot, представляє послідовність функцій, які потенційно можуть бути викликані програмою. Однак кожне виконання програми може займати певну гілку графіка та виконувати лише підмножину викликів. Таким чином, на цьому етапі MaMaDroid працює наступним чином. По-перше, він ідентифікує набір вузлів входу в графі викликів, тобто вузлів без вхідних ребер. Потім він перераховує шляхи, доступні з кожного вузла входу. Набори всіх шляхів, ідентифікованих під час цієї фази, становлять послідовності викликів API, які використовуватимуться для побудови поведінкової моделі ланцюга Маркова та вилучення функцій.

Абстракція виклику API. Замість того, щоб аналізувати необроблені виклики API з послідовності викликів, створюємо MaMaDroid для роботи на вищому рівні та роботи в одному з трьох режимів, абстрагуючи кожен виклик до його сімейства, пакету або класу. Інтуїція полягає в тому, щоб зробити MaMaDroid стійким до змін API і

досягти масштабованості. Насправді наші експерименти показують, що з набору даних із 44 тисяч програм витягуємо понад 10 мільйонів унікальних викликів API, які, залежно від підходу до моделювання, який використовується для моделювання кожної програми, можуть призвести до того, що вектори функцій будуть дуже рідкісні. Хоча пакунок і клас уже є існуючими назвами для цих рівнів абстракції, використовуємо «сімейство», щоб вказати ще вищий рівень абстракції, який наразі не існує. Наше використання «сімейства» відноситься до «кореневих» імен пакетів API, а не до «сімейств зловмисного програмного забезпечення», оскільки не намагаємося позначити кожен зразок зловмисного програмного забезпечення його сімейством. Під час роботи в сімейному режимі абстрагуємо виклик API до одного з дев'яти «кореневих» імен пакетів Android, тобто android, google, java, javax, xml, apache, junit, json, dom, які відповідають android.* пакети , com.google.*, java.*, javax.*, org.xml.*, org.apache.*, junit.*, org.json і org.w3c.dom.*. У той час як у режимі пакета абстрагуємо виклик до назви його пакета, використовуючи список пакетів Android із документації3, що складається з 243 пакетів на рівні API 24 (версія на вересень 2016 року), а також 95 з API Google.4 У режимі класу абстрагуємо кожен виклик до його назви класу, використовуючи білий список усіх імен класів в API Android і Google, який складається відповідно з 4855 і 1116 класів.5 У всіх режимах абстрагуємо визначені розробником (наприклад, com. stericson.roottools) і обфускованих (наприклад, com.fa.a.b.d) викликів API відповідно, як самовизначені та обфусковані. Зауважте, що позначаємо виклик API як обфускований, якщо не можемо визначити, що його клас реалізує, розширює або успадковує через спотворення ідентифікатора. Загалом є 11 (9+2) сімейств, 340 (243+95+2) пакетів і 5973 (4855+1116+2) можливих класів.

Для кожної програми MaMaDroid приймає як вхідні дані послідовність абстрактних викликів API цієї програми (класів, пакетів або сімейств, залежно від вибраного режиму роботи) і створює ланцюг Маркова, де кожен клас/пакет/сімейство є станом і переходи представляють ймовірність переходу з одного стану в інший. Для

кожного ланцюга Маркова стан S0 є точкою входу, з якої здійснюються інші виклики в послідовності. Насправді MaMaDroid розглядає всі можливі виклики – тобто всі гілки, що походять від вузла – у ланцюзі Маркова, тому додавання викликів суттєво не змінить ймовірність переходів між вузлами (зокрема, сімействами, пакетами чи класами залежно від операційної системи). режим) для кожної програми.

Далі використовуємо ймовірності переходу з одного стану (абстрактний виклик) в інший у ланцюзі Маркова як вектор ознак кожної програми. Стани, яких немає в ланцюжку, представлені як 0 у векторі ознак. Вектор, отриманий з ланцюга Маркова, залежить від режиму роботи MaMaDroid. У сімействах існує 11 можливих станів, тобто 121 можливий перехід у кожному ланцюжку, тоді як при абстрагуванні до пакетів існує 340 станів і 115 600 можливих переходів, а з класами є 5973 стани, отже, 35 676 729 можливих переходів. Також застосовуємо аналіз основних компонентів (PCA) [33], який виконує вибір ознак шляхом перетворення простору ознак у новий простір, що складається з компонентів, які є лінійними комбінаціями вихідних функцій. Перший компонент містить якомога більшу дисперсію (тобто кількість інформації). Дисперсія надається як відсоток від загальної кількості інформації вихідного простору ознак. Ми застосовуємо PCA до набору функцій, щоб вибрати основні компоненти, оскільки PCA перетворює простір функцій у менший, де дисперсія представлена якомога меншою кількістю компонентів, таким чином значно зменшуючи складність обчислень/пам’яті. Крім того, PCA може зменшити переобладнання, побудувавши модель лише з основних компонентів функцій у нашому наборі даних, що, у свою чергу, може підвищити точність класифікації.

Останнім кроком є виконання класифікації, тобто позначення додатків як безпечних або шкідливих. З цією метою тестуємо MaMaDroid, використовуючи різні алгоритми класифікації: випадкові ліси, 1-найближчий сусід (1-NN), 3-найближчий сусід (3-NN) і опорні векторні машини (SVM). Зауважте, що оскільки і точність, і швидкість гірші з SVM, опускаємо результати, отримані з ним. У середньому показник

F-Measure із SVM із використанням радіальних базових функцій (RBF) на 0,09 нижчий, ніж із випадковими лісами, і навчання в сімейному режимі (який має набагато менший простір функцій) у 5 разів повільніше, ніж у 3-найближчих сусідів (найповільніший серед інших методів класифікації). Кожна модель навчається за допомогою вектора ознак, отриманого з програм у навчальному зразку. Зауважте, що через різну кількість функцій, які використовуються в різних режимах, використовуємо дві різні конфігурації для алгоритму випадкових лісів. Зокрема, при абстрагуванні до сімейств використовуємо 51 дерево з максимальною глибиною 8, тоді як для класів і пакетів використовуємо 101 дерево з максимальною глибиною 64. Для налаштування випадкових лісів дотримуємося методології, застосованої в [18].

1.2.3 Статичний аналіз на основі маніфесту і коду

DroidEnsebmle працює в чотири етапи, як показано на рисунку 1.7. Спочатку збираємо кілька додатків із чотирьох ринків додатків і malapps у дикій природі.

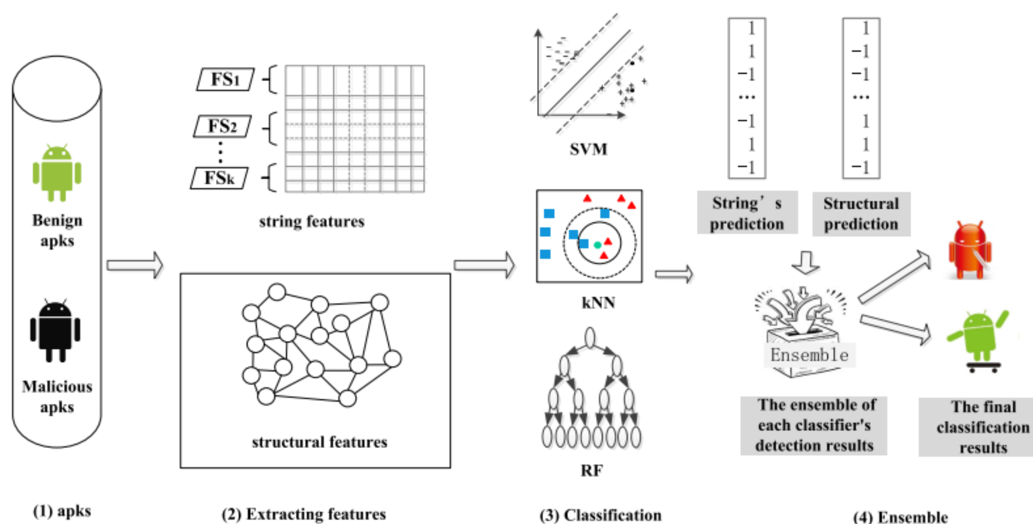


Рисунок 1.7 – Схема аналізу на основі маніфесту і коду

По-друге, витягуємо 6 типів рядкових функцій і графік викликів функцій як структурні функції з кожного apk. По-третє, створюємо три моделі навчання під наглядом, щоб оцінити ефективність наших методів з обома типами наборів функцій. По-четверте, підтверджуємо ефективність DroidEnsemble за допомогою ансамблю рядків і структурних особливостей. Нарешті, проводимо масштабні експерименти,

включаючи: (1) порівняння ефективності наших методів з кожним типом функцій; (2) порівняння класифікаторів і (3) оптимізація результатів виявлення за допомогою ансамблю ознак обох типів.

Витягуються 6 типів рядкових функцій із кожної програми. Усі функції описані нижче. FS1 (запитувані дозволи): якщо програмі потрібно виконати певні операції, вона має запитати відповідні дозволи у файлі маніфесту. Кожна програма містить файл маніфесту, що містить метаінформацію, що підтримує встановлення та подальше виконання програми. З цього файлу можна отримати цінну інформацію. Однак деякі програми запитують дозволи, які необов’язково потрібні для виконання їхніх функцій, що може вказувати на зловмисні наміри. В іншому випадку комбінація кількох дозволів може відображати певну шкідливу поведінку. Наприклад, якщо програма подає запит на дозвіл на підключення до мережі, а також дозвіл на доступ до SMS, програма може отримувати інформацію про SMS користувачів, а потім поширювати її через Інтернет. У цій роботі використовуємо всі дозволи, оголошені у файлі маніфесту, з елементами <uses-permission> як набір функцій. FS2 (характеристики апаратного забезпечення): у системі Android вимоги до апаратного та програмного забезпечення вказують на вимоги додатків щодо системних ресурсів. Наприклад, якщо програма отримує доступ до 4G і GPS, це може означати, що вона повідомляє зловмиснику місцезнаходження користувача, що розкриває зловмисну поведінку програми. Таким чином, витягуємо інформацію про обладнання та програмне забезпечення, визначену у файлі маніфесту, з елементами <uses-feature> як другим набором функцій. Існують відповідні роботи [19], [20], які використовували цей тип функцій.

FS3 (відфільтровані наміри): наміри обробляють зв’язок між компонентами, надсилаючи об’єкти намірів на Android. Фільтри намірів допомагають компонентам програми відхиляти небажані наміри, а також залишати бажані наміри. Витягується відфільтровані наміри як ще один набір функцій для виявлення malapp, і вони є сигналом з елементами <intent-filter>. Feizollah та ін. [21] оцінили ефективність

Android Intents (явних і неявних) як функції ідентифікації шкідливих програм, і рівень виявлення досяг 91%. Вилучається вище три набори функцій із файлу маніфесту за допомогою інструментів androguard і Android Asset Packaging Tool (aapt). Крім того, витягуємо ще три статичні функції з коду дизассемблування (FS4-FS6). FS4 (обмежені виклики API): система дозволів Android обмежує доступ до серії критичних викликів API, демонструючи, як програма взаємодіє з інфраструктурою Android. Обмежені виклики API захищені дозволами. Відповідно до зіставлення дозволів API, наданого PScout, легко визначити, які API захищені дозволами. Потім визначаємо словник, який відображає зв'язок між дозволами та відповідними обмеженими викликами API. Сканується розібраний код зразків програм і записуємо, чи викликають вони виклики API, захищені деякими дозволами, щоб отримати цей набір функцій.

Фактично використовується через відображення дозволів API, надані PScout [22] як інший набір функцій. FS6 (шаблони коду): система Android не забезпечує дійсний механізм автентифікації та захисту для зовнішніх завантажених ресурсів. Бібліотеки можуть бути шкідливими для модифікації та маскування. Таким чином зловмисники можуть спробувати приховати частини шкідливих функцій своїх програм у бібліотеках. Перевіряється, чи програма виконує команди оболонки, чи вона динамічно завантажує зовнішні файли в цьому наборі функцій. Крім того, перевіряємо, чи використовує програма методи відображення Java чи викликає криптографічні функції. Витягуємо наведені вище 6 типів статичних функцій як рядкові функції для виявлення malapp. Крім того, додаємо функції графіка викликів функцій як структурні особливості.

Перевірити, чи є два графіки ізоморфними, непросто за поліноміальний час. Для спрощення цього процесу, вимірюючи подібність між двома графіками, підраховуючи кількість однакових підграфів між графіками викликів функцій двох програм. У цій роботі генеруємо ізометричне кодування для вузлів виклику функцій програми на основі інструкцій Dalvik. Був використаний метод для ідентифікації доброякісних і

шкідливих програм шляхом обчислення подібності між двома програмами з кількістю однакових кодувань двох програм. Він включає наступні три кроки. Крок 1. Додаток розбирається за допомогою apktool, а його графіки викликів функцій витягуються за допомогою androguard. Потім вузли графа викликів функції позначаються 15-бітовою послідовністю. Формально граф формується як 4-кортеж G = (N, E, L, l), де N — набір вузлів, а кожен вузол n ∈ N пов'язаний з однією з функцій програми. E ⊆ N × N позначає набір спрямованих ребер, де ребро від вузла n1 до вузла n2 вказує на виклик функції, представленої n1, до функції, представленої n2. L — це мультимножина міток у графі, а l: N→L — функція позначення, яка призначає мітку кожному вузлу, враховуючи типи інструкцій функції, яку він містить.

Щоб описати поведінку додатків для подальшого аналізу, вбудовуємо всі 6 типів функцій у багатовимірний вектор функцій і вбудовуємо кожен графік викликів у простір функцій відповідно. На основі двох типів ознак будуємо три моделі навчання під наглядом, а саме опорну векторну машину (SVM), k-найближчий сусід (kNN) і випадковий ліс (RF), оскільки ці три методи широко використовувалися для бінарної класифікації. Наші набори функцій теоретично лінійно розділяються.

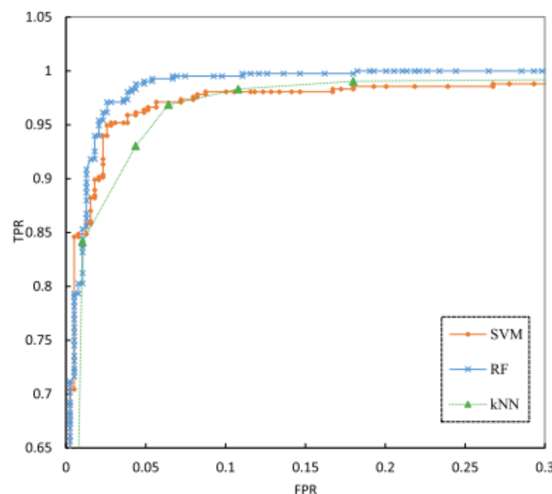


Рисунок 1.8 – Крива ROC основана на SVM, RF та kNN

На рисунку 1.8 можна побачити відмінність у ефективності цих класифікаційних моделей. На практиці SVM добре працює для задач лінійної роздільної класифікації

великої розмірності. Матриця, згенерована графіком виклику функції, використовується для вимірювання подібності між зразками. Щоб полегшити порівняння продуктивності, також класифікуємо kNN і RF. SVM більш ефективний для вирішення проблем роздільної класифікації великої розмірності. SVM досягає найкращих результатів із сукупністю двох типів функцій порівняно з kNN та RF.

1.2.4 Динамічний метод

Програми на основі Android корелюються з ОС пристрою за допомогою системних викликів, що дає змогу відстежувати, що відбувається між ними. Техніка динамічного виявлення відстежує зловмисне програмне забезпечення Android у контрольованому середовищі під час виконання, враховуючи покажчики зловмисного програмного забезпечення, сигнатури виявлення яких можна моделювати за їх допомогою. Він перевіряє взаємодію зловмисного програмного забезпечення з мобільними ресурсами та службами, такими як місцезнаходження, мережа, пакети, діяльність ОС. Для безпеки експериментального обладнання (фізичного пристрою) рекомендується, щоб виконання коду здійснювалося в кібернетичному середовищі. Дослідження застосувало цю техніку виявлення до 4034 і 10024 зловмисних програм і безпечних наборів даних відповідно. Використовуючи підхід ServiceMonitor, алгоритм класифікації випадкового лісу з точністю 96% виявив зловмисне програмне забезпечення в цих програмах. Використовуючи k-кратну перевірку та ланцюг Маркова, модуль класифікатора було навчено для вилучення ознак вибірки. Точність інформації, отриманої зловмисним програмним забезпеченням, як-от IMEI телефону, виявилася на 67%. Серед виявленого зловмисного програмного забезпечення 17% додатків, як було помічено, приєднали своє корисне навантаження до пристрою для оцінки преміум-сервісу. Помічено, що мобільні утиліти, такі як процесор і пам’ять, були інфіковані з продуктивністю накладного пристрою на рівні о,8% і 2% відповідно. Деякі зловмисні програми залишаються неактивними на пристрої після завантаження та встановлення, доки не буде запущено дію. У той час як деякі роблять це, інші

виконують своє корисне навантаження під час завантаження, встановлення та виконання. Це було продемонстровано в. Авторизація доступу, яка зазвичай надається користувачам Android під час завантаження та встановлення програми, створює великий простір у векторі атаки пристрою, навіть якщо під час сеансів завантаження та встановлення завжди зустрічаються дозволи за замовчуванням. Під час цих вправ шкідливий код приєднується до безпечних програм. Критичний моніторинг на цих етапах потрібен для кращої безпеки мобільних платформ.

Був розрібраний DroidCat, уніфікований підхід до виявлення зловмисного програмного забезпечення, що використовує систематичне динамічне профілювання та навчання під наглядом, щоб визначити, чи є дана програма безпечною чи належить до певної родини зловмисних програм. Як показано на малюнку 1, у нашому підході є два етапи: навчання та тестування. На етапі навчання DroidCat приймає як безпечні, так і шкідливі програми як вхідні дані. Для кожної програми він обчислює особливості поведінки, інструментуючи та виконуючи програму, а також збираючи різноманітні характеристики поведінки. Потім функції надаються керованому машинному навчанню для навчання багатокласового класифікатора, який використовуватиметься на другому етапі. Для тестування, враховуючи довільну програму, DroidCat обчислює її поведінкові характеристики так само, як згадано вище, а потім передає ці функції в класифікатор, щоб вирішити, чи є програма доброякісною чи належить до сімейства шкідливих програм.

Щоб обчислити динамічні функції програми Android, спочатку інструментували програму для збору трасування виконання. Зокрема, для файлу APK кожного додатка використовували Soot для декомпіляції виконуваного файлу в байт-код, а потім вставляли інструментарій байт-коду для відстеження кожного виклику методу та кожного ICC разом із вмістом Intent. У цьому процесі ми також позначили додаткову інформацію для інструментованих класів і методів, щоб полегшити вилучення функцій. Наприклад, був позначений тип компонента для кожного інструментованого

класу, категорію кожного інструментованого зворотного виклику та властивість джерела або приймача кожного відповідного Android API. Щоб визначити тип компонента класу, такого як Foo, застосували аналіз ієрархії класів (CHA), щоб визначити всі суперкласи. Якщо Foo розширює будь-який із чотирьох відомих типів компонентів, таких як Activity, його тип компонента позначається відповідно. Був використаний список зіставлення типу методу, щоб позначити категорію зворотних викликів і властивість джерела/приймача API. Далі запустили інструментальний APK кожної програми на емуляторі Android, щоб зібрати траси виконання, які включають всі виклики методів і ICC. Зауважте, що не інструментуємо системні виклики на рівні ОС, оскільки хочемо, щоб DroidCat був стійким до будь-яких системних викликів, спрямованих на атаку. Наш інструментарій також не обмежується чутливими API. Переконавшись, що конфіденційні API не є єдиною цільовою областю профілювання викликів методів, також робимо DroidCat стійким до будь-яких атак, націлених на конфіденційні API. Попередня робота показує, що навіть без виклику системних API або конфіденційних API деякі зловмисні програми можуть здійснювати атаки, маніпулюючи іншими програмами через ICC [23] . Тому також інструментуємо ICC, щоб виявити будь-які поведінкові відмінності між доброякісними та різними шкідливими програмами з іншої точки зору.

Щоб повністю охарактеризувати динамічну поведінку додатків, нам потрібно запускати кожну інструментальну програму протягом достатньо тривалого часу, використовуючи різні вхідні дані, щоб охопити якомога більше шляхів. Вводити дані в додатки вручну – справа нудна й неефективна. Щоб швидко запускати різноманітні виконання програми, використовували Monkey для випадкової генерації вхідних даних. Оскільки попереднє дослідження показує, що для середньої програми Monkey не покриває значно більше шляхів після перших 10 хвилин запуску, ми навмисно налаштували Monkey на виконання кожної програми протягом 10 хвилин. Це дозволило нам ефективно збирати величезну кількість даних трасування, не

жертвуючи значним динамічним покриттям. Нарешті, витягли 70 функцій із слідів виконання. Усі ці функції були визначені як відсоток певних викликів функцій або ICC з певною характеристикою. Наприклад, одна функція вимірює відсоток ICC, що передають лише дані URI, тоді як інша функція описує відсоток вихідних API, які мають шляхи, що досягають принаймні одного API приймача. Ці функції були визначені на основі нашого систематичного динамічного дослідження характеристик програм Android.

Щоб навчити класифікатор для уніфікованого виявлення зловмисного програмного забезпечення, нам потрібні дані про особливості поведінки не лише для безпечних додатків, але й для зловмисних додатків із різних сімейств зловмисного програмного забезпечення. Тому в наших навчальних даних кожна точка даних представляє одну програму та має такий формат: <вектор ознак, мітка>, де вектор ознак містить 70 значень ознак. Мітка – це або більший, або назва сімейства шкідливих програм (наприклад, DroidKungfu). Використовуємо Random Forest, алгоритм керованого машинного навчання (ML), щоб навчити класифікатор із позначеними даними. На етапі тестування, враховуючи невідому програму, DroidCat обчислює її функції, і передає вектор ознак добре навченому багатокласовому класифікатору. На відміну від даних навчання, кожна точка даних у даних тестування не позначена, оскільки не знаємо, чи є програма доброякісною. На основі вектора ознак наш класифікатор визначає, яку мітку категорії слід призначити додатку. Ми реалізували DroidCat на Python і використали Scikit-learn, безкоштовний інструментарій машинного навчання для Python, щоб навчити та перевірити класифікатор. DroidCat з відкритим кодом за адресою (посилання не надано для подвійного сліпого перегляду).

На основі зібраних трас виконання охарактеризували поведінку програми, визначивши 122 метрики в 3 ортогональних вимірах: структура, ICC і безпека. Інтуїтивно зрозуміло, що чим різноманітніше ці метрики фіксують одну трасу виконання, тим повніше вони характеризують поведінку програми. Ці показники

вимірюють не лише наявність певних викликів методів або ICC, але й частоту їх появи. Щоб спростити пояснення, обговоримо лише кілька показників у статті. Щоб отримати повне пояснення всіх метрик, зверніться до нашого веб-сайту (посилання не надано для подвійного сліпого перегляду). Структурний вимір містить 63 метрики для опису розподілу викликів методів, їхніх класів оголошення та зв’язків між абонентами, що викликаються. Серед цих показників 31 показник описує розподіл усіх викликів методів між трьома рівнями програмного забезпечення (тобто код користувача, бібліотеки сторонніх розробників і Android SDK) або між різними компонентами. Інші 32 метрики описують розподіл певного типу методів — зворотних викликів.

Розмір ICC містить 7 показників для опису розподілу ICC. Оскільки існує два способи класифікації ICC: внутрішні проти зовнішніх і неявні проти явних. Перерахування всіх можливих комбінацій призводить до чотирьох показників. Інші три метрики визначаються на основі типу даних, які містяться в Intents. Вимір безпеки містить 52 метрики для опису розподілу джерел, приймачів і доступності між ними. Зокрема, щоб перевірити досяжність між будь-якими двома API при отриманні трасування виконання, створюємо динамічний графік викликів, а потім перевіряємо будь-який шлях у графі викликів, який пов’язує API. Якщо джерело має принаймні один шлях, що досягає поглинача, воно вважається джерелом ризику.

Рисунок 1.9 – Топ-10 метрик для динамічного аналізу

На рисунку 1.9 продемонстровані 10 найбільш поширених метрик, які зазвичай викорстовують під час динамічного аналізу файлу APK. Визначивши 122 показники, оцінили показники для кожної програми в нашому наборі тестів. Якщо деякі показники завжди показували різні профілі між безпечними та шкідливими програмами, покладалися на них, щоб охарактеризувати розбіжності в поведінці, і вибрали їх як функції для навчання уніфікованого детектора зловмисного програмного забезпечення. Щоб ідентифікувати будь-який показник із різними профілями в двох групах додатків, виміряли значення кожного показника для кожної контрольної програми, а потім обчислили середнє значення для всіх безпечних додатків і для всіх шкідливих програм. Якщо метрика мала різницю в середньому значенні більше або дорівнювала 5%, вважали поведінковий профіль двох груп суттєво відмінним щодо метрики; якщо показник мав різницю, більшу або рівну 2%, говорили, що поведінковий профіль помітно відрізняється щодо метрики. Як і очікувалося, суттєво відмінні показники завжди є підмножиною помітно різних показників. У наших налаштуваннях 5% і 2% були обрані як евристики; наші пізніші експерименти показали, що обидва пороги працюють добре.

Як показано в таблиці 1, шляхом порівняння середніх значень показників у групах додатків виявили 36 суттєво відмінних показників і 70 помітно різних показників. Через обмеження простору показуємо лише 10 найпоширеніших показників на малюнку 2. На малюнку десять показників перераховано на осі Y, а вісь X відповідає середнім значенням показників, які змінюються від 0% до 100%. Кожна метрика, указана на осі Y, відповідає двом горизонтальним смугам: одна червона смуга, яка показує середнє значення всіх шкідливих програм, і одна зелена смуга, яка представляє середнє значення всіх доброякісних програм. Вусик на кожній смужці представляє стандартну помилку середнього3. Ці 10 показників найкраще демонструють поведінкові відмінності між шкідливими та безпечними програмами. Далі вивчаємо ці показники та групуємо їх за параметрами. У структурному вимірі

шкідливі програми викликають менше методів, визначених у SDK, більше методів, визначених у коді користувача, і включають більше зворотних викликів, пов'язаних з інтерфейсом користувача. Це вказує на те, що дії користувача можуть викликати надмірні або незаплановані обчислення. У вимірі ICC зловмисне програмне забезпечення включає більше зовнішніх явних ICC з більшою кількістю даних URI, які передаються Intents. Це означає, що зловмисне програмне забезпечення частіше використовує явні ICC для потенційної атаки на певні зовнішні компоненти або надсилає більше даних URI через ICC для поширення потенційно шкідливих URI. У аспекті безпеки зловмисне програмне забезпечення викликає більш ризиковані джерела API, але менше API приймача журналу. Виконуючи більш ризиковані джерела, зловмисне програмне забезпечення може спричинити витік конфіденційних даних.

Наше порівняння показників показує, що поведінка шкідливих програм помітно відрізняється від безпечних. Такі поведінкові відмінності відображені в показниках структурного, ICC та безпеки. За замовчуванням використовуємо 70 помітно різних показників як функції, отримані для навчання уніфікованих моделей виявлення зловмисного програмного забезпечення в DroidCat. Пам'ятайте, що ці функції показують не лише існування певних викликів методів або ICC, але й їхні динамічні швидкості виконання, які ніколи не можуть бути зафіксовані жодним статичним детектором зловмисного програмного забезпечення.

1.2.5 Гібридний метод

Цей метод поєднує в собі функції динамічних і статичних методів, щоб забезпечити більш надійний результат виявлення під час аналізу зловмисного програмного забезпечення. Підхід гібридного виявлення до виявлення зловмисного програмного забезпечення передбачає в основному фази навчання та виявлення, які можуть виконуватися відповідно динамічними та статичними методами. Здається, це дає кращу швидкість виявлення, ніж динамічні та статичні методи, оскільки сильні

сторони обох методів синергізовані. Використовуючи аспект глибокого навчання штучного інтелекту, було розроблена модель DroidDetector з деякими алгоритмами для виявлення зловмисного програмного забезпечення Android. Гібридна техніка зібрала загалом 192 зразки зловмисного програмного забезпечення та доброякісних програм для Android для навчання. Модель показала точність результату виявлення 96,60% з різницею 0,0021% серед використаних алгоритмів. У деяких складних випадках, коли зразок зловмисного програмного забезпечення невідомий, навчання та виявлення можуть не проводитися одночасно, щоб уникнути втручання функцій. Доведено, що приховані моделі Маркова мають високу продуктивність, коли мова йде про двостороннє вдосконалення виявлення зловмисного програмного забезпечення. Гібридна методика дозволяє провести порівняльний аналіз статичної та динамічної точності виявлення. Завдяки семантичному підходу цієї методики код операції та послідовність викликів зловмисного програмного забезпечення API були виділені за допомогою прихованих моделей Маркова. Нагадаємо, точність і специфічність визначили поріг кривої ROC. Щоб визначити та встановити зловмисність і нешкідливість програми, Android Buster Sandbox використовувався як аналізатор. Однак виявлення зловмисного програмного забезпечення Android шляхом застосування послідовності викликів API не змогло подолати проблему обфускації зловмисного програмного забезпечення. Крім того, послідовність спостереження функцій зловмисного програмного забезпечення не створює реляційної відповідності до окремих станів HMM, цей підхід не може створити початковий стан розповсюдження зловмисного програмного забезпечення в графі викликів і послідовності відповідно. Подібним до цього було дослідження, яке за допомогою графіків викликів API вилучало smali-файли зловмисного програмного забезпечення. З 1216 підозрілих Android-додатків було вилучено 1022. Отримана точність виявлення склала 96,12%. Незважаючи на те, що атаки ухилення від зловмисного програмного

забезпечення Android було подолано за допомогою цього підходу, атаку отруєння Android не вдалося впоратися за допомогою машинного навчання.

Зосереджуючись на блокових викликах API, дослідження розробило інструмент виявлення (Droiddelver) з алгоритмом Deep Belief Network, який видобув асемантні сліди як відомого, так і невідомого шкідливого програмного забезпечення. У такому сценарії зловмисне програмне забезпечення найчастіше надає індекси програми smali, яку зловмисне програмне забезпечення може мати намір врізати в мобільне ядро Android. Для розпакування та декомпіляції програм Android перед видобуванням рівня виклику API потрібен невеликий код, який стоїть між віртуальною машиною Dalvik та інтерфейсом програми. Деякі зловмисні програми Android в основному призначені для збору інформації, пов'язаної з системними викликами, файловими системами, мобільним місцезнаходженням і зображеннями, знятими камерою пристрою. Шкідлива програма з цією ціллю робить користувача пристрою фізично та інформаційно вразливим. Його можна легко вистежити та атакувати, або його системні файли можуть бути використані для отримання фінансової вигоди чи іншим чином. Це було продемонстровано в дослідженні з відносно невеликим набором даних шкідливого програмного забезпечення Android.

1.2.6 Виявлення на основі емуляції

Ця техніка вимагає створення змодельованої екосистеми за допомогою емулятора для запуску зразків зловмисного програмного забезпечення, щоб відокремити їх від фактичних фізичних ресурсів пристрою. Симуляцію можна виконувати на ОС Android або апаратному забезпеченні. Однак виявлення стає набагато складнішим, коли зловмисне програмне забезпечення запускається в мобільній реальній ОС. Ця техніка потребує створення пісочниці та налаштування віртуальних машин систематичним і безпечним способом, щоб уникнути зараження інших пристроїв у мережі. Ця техніка ефективна, особливо якщо належним чином відстежується файл Dalvik (.dex). Шкідливі програми можна виявити в системі

ізольованого програмного середовища, отримавши файл dex і перетворивши його у форму, зрозумілу людям. Зловмисне програмне забезпечення нульового дня та зловмисне програмне забезпечення, яке підвищує привілеї, ефективно виявляється за допомогою цієї техніки.

1.3 Формалізована постановка задачі

Метою роботи є створення інформаційної технології кіберзахисту мобільних платформ. Необхідно дослідити та оптимізувати модель ознакового опису мережі, її методи навчання та розпізнавання. На основі вхідної вибірки з описом набору реалізацій шкідливих та безпечних зразків програмного забезпечення необхідно здійснити їх аналіз та класифікацію. Отримана система повинна забезпечити оптимальні показники точності виявлення та класифікації шкідливих зразків.

Для досягнення поставленої задачі в роботі необхідно вирішити наступні завдання:

1. Визначення методу екстракції ознакового опису спостережень;
2. Вибір методу класифікаційного аналізу для прогнозування впливу шкідливого програмного забезпечення та побудова вирішальних правил;
3. Визначити критерії оцінювання ефективності моделі розпізнавання шкідливих програм;
4. Побудова моделі та методу навчання моделі розпізнавання шкідливого програмного забезпечення;
5. Програмна реалізація алгоритмів технології розпізнавання;
6. Формування даних для навчальної матриці з подальшим аналізом та класифікацією шкідливого програмного забезпечення;
7. Тестування та оцінювання ефективності розробленого програмного забезпечення, аналіз результатів;

2 ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗПІЗНАВАННЯ ШКІДЛИВИХ ПРОГРАМ НА МОБІЛЬНИХ ПЛАТФОРМАХ

2.1 Модель екстракції ознакового опису спостережень

APK (пакет Android) — це комплексний ZIP-файл, який інкапсулює код, ресурси, підпис, маніфест і кілька інших файлів, які потрібно виконати, щоб повна програма для Android була запущена. AndroidManifest.xml містить дозволи, запитувані програмою. Усвідомлюючи той факт, що безпека сильно пов’язана з дозволами Android, головним завданням було отримати цей файл. ОС Android працює на коді з «Проекту з відкритим вихідним кодом Android» або AOSP. Цей код є відкрити, і, отже, розробники можуть отримувати вихідний код і створювати з нього власні операційні системи (Hoffman і т.д)

На етапі навчання будемо використовувати три категорії функцій:

1) функції, пов’язані з API (фільтри намірів, підозрілі виклики);

2) функції маніфесту (дозвіл програми, приймачі трансляції);

3) стандартні команди фреймворку ОС і Android.

Функції, пов’язані з API, отримують шляхом видобутку файлу Classes.dex. Вони складаються з ключових слів, які дозволяють виявити використання вибраних стандартних викликів Android API (за допомогою яких програма взаємодіє з різними функціями пристрою), а також вибраних викликів Java API, які використовуються для розширення функціональності програм.

Функції дозволів і приймачів трансляції – це ключові слова, які зіставляються зі стандартним файлом Android, оголошеним у файлі маніфесту, таким чином дозволяючи отримувати дозволи, які запитує програма для своїх функцій. Наприклад, ключове слово SEND_SMS дозволяє виявити запит дозволу на надсилання SMS-повідомлень додатку, якщо це зазначено у файлі маніфесту.

Функції, пов’язані з командами, – це ключові слова, які виявляють наявність команд Linux, таких як «chown», «mount» тощо, або певних параметрів, які можуть використовуватися з цими командами. Ці команди зазвичай вбудовано в приховані файли в APK і викликаються шахрайськими програмами, щоб увімкнути незвичайні дії, як підвищення привілеїв, запуск прихованих сценаріїв або вбудованих шкідливих двійкових файлів або приховування шкідливих дій.

2.2 Модель і метод навчання класифікатора шкідливих програм

Методи навчання на основі ознак присвоюють значення кожній ознакі вхідного зразка залежно від того, наскільки вона релевантна для рішення щодо класифікації. Ці значення релевантності часто називають атрибуціями. У цій роботі для навчання класифікатора ми будемо застосовувата навчання на основі градієнта. Автоматична нелінійна класифікація є поширеним і потужним інструментом аналізу даних. Дослідження машинного навчання створили практичні методи, які можуть класифікувати невидимі дані навчання на обмеженому навчальному наборі позначених прикладів. Тим не менш, більшість алгоритмів не пояснюють своє рішення. Проте в практичних даних аналізу важливо отримати пояснення на основі екземплярів, тобто ми хотіли б отримати розуміння, які вхідні характеристики змусили нелінійну машину дати відповідь для кожної окремої точки даних.

Як правило, пояснення надаються спільно для всіх екземплярів навчального набору, наприклад методи вибору ознак (включно з автоматичним визначенням релевантності) потрібні, щоб з’ясувати, які вхідні дані є важливими для гарного узагальнення. Хоча це може дати приблизне враження про глобальну корисність кожного вхідного виміру, це все ще не повна картина, яка не дає відповіді на основі екземпляра. Також у літературі про нейронні мережі лише ансамблевий вигляд використовувався в таких алгоритмах, як скорочення вхідних даних. Єдина класифікація, яка надає окремі пояснення, це дерева рішень.

2.2.1 Визначення навчальних векторів

Для випадку з кількома класами припустимо, що нам задані точки даних x1,..., xn ∈ ℜd з мітками y1,..., yn ∈ {1,...,C}, і ми маємо намір навчити функцію, яка передбачає мітки непозначених точок даних. Припускаючи, що дані є IID-вибіркою з деякого невідомого спільного розподілу P(X,Y), ми визначаємо Класифікатор Байєса

$$g^{*}(x) = \arg\min_{c\epsilon\{1,\dots,C\}} P(Y \neq C | X = x) \tag{2.2.1}$$

що є оптимальним для функції втрат 0-1.

Для класифікатора Байєса ми визначаємо вектор пояснення точки даних x0 як похідну відносно x при x = x0 умовної ймовірності Y 6=g∗(x0) якщо X = x, або формально,

$$\zeta(\mathrm{x}_0) := \frac{\partial}{\partial x} P(Y \neq g^{*}(x) | X = x)|_{x=x_0} \tag{2.2.2}$$

Зауважте, що ζ(x0) є d-вимірним вектором, як і x0. Класифікатор g∗ розділяє простір даних ℜd на C частин, на яких g∗ є постійним. Припустимо, що умовний розподіл P(Y = c | X =x) є диференційовним першого порядку x для всіх класів c і по всьому простору введення. Наприклад, це припущення виконується, якщо P(X =x | Y =c) є для всіх першого порядку диференційовним по x і опори щільності класів перекриваються навколо кордону для всіх сусідніх пар у розділі класифікатора Байєса. Вектор ζ(x0) визначає на кожній із цих частин векторне поле, яке характеризує витік з відповідного класу. Таким чином, записи в ζ(x0) з великими абсолютними значеннями підсвічують особливості, які впливатимуть на рішення про мітку класу x0. Позитивний знак такого запису передбачає що збільшення цієї функції зменшить ймовірність того, що x0 буде присвоєно g∗(x0).

Ігнорування орієнтації векторів пояснення ζ утворює безперервно змінне (без орієнтації) векторне поле, уздовж якого змінюються мітки класу. Це векторне поле дозволяє нам локально зрозуміти класифікатор Байєса. Зауважимо, що ζ(x0) стає

нульовим вектором, наприклад, коли P(Y 6= g∗(x) | X = x)|x=x0 дорівнює одиниці в деякому околі x0. Наш вектор пояснення добре підходить до класифікаторів, де умовний розподіл P(Y = c | X = x) зазвичай не є абсолютно плоским у деяких регіонах. У випадку детермінованих класифікаторів, незважаючи на цю проблему, віконні оцінки Парзена з відповідними значеннями ширини можуть надати значущі вектори пояснення для багатьох зразків на практиці.

У випадку бінарної класифікації ми безпосередньо визначаємо локальні вектори пояснення як локальні градієнти функції ймовірності p(x) = P(Y = 1 | X = x) вивченої моделі для позитивного класу. Для функції ймовірності p : ℜd → [0,1] моделі класифікації, отриманої з прикладів {(x1, y1),...,(xn, yn)} ∈ ℜd × {−1,+1} мають вектор пояснення для класифікованої тестової точки x0, при якому локальний градієнт p при x0:

$$\eta_p(x_0) \coloneqq \nabla_p(x)|_{x=x_0} \quad (2.2.3)$$

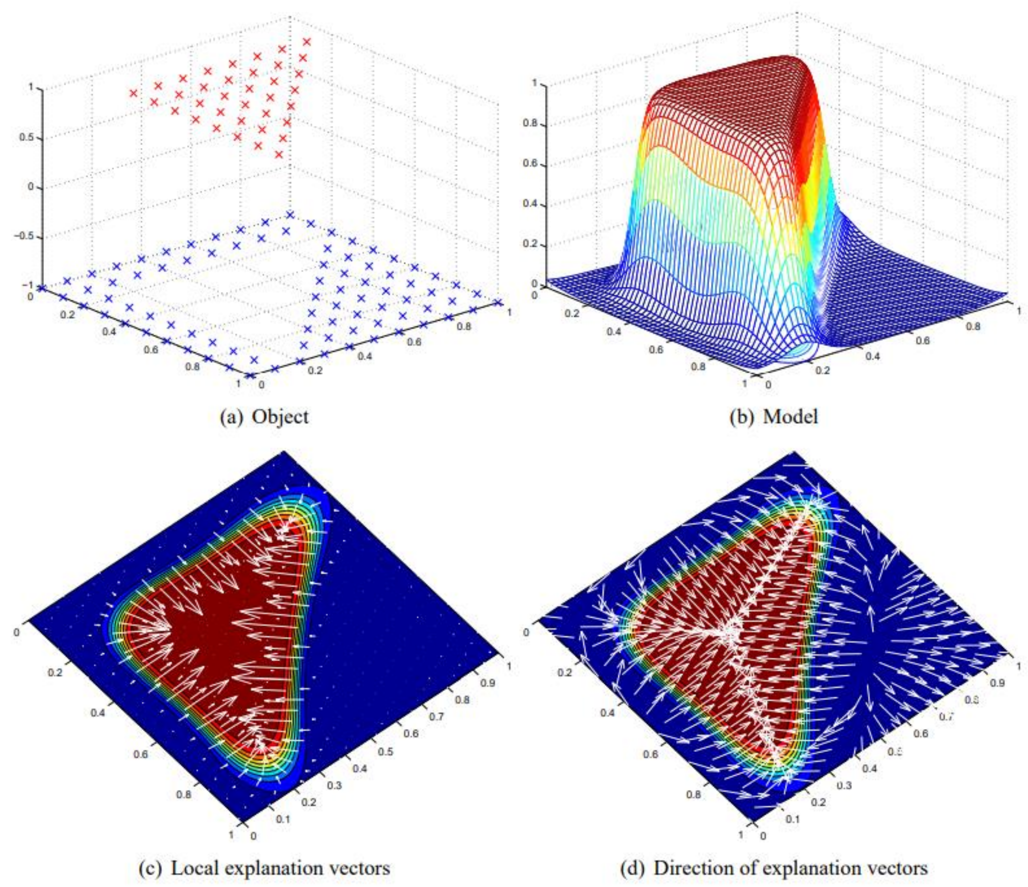
За цим визначенням пояснення η знову є d-вимірним вектором, як і контрольна точка x0. Знак кожного окремого запису вказує, збільшиться чи зменшиться прогноз коли відповідна характеристика x0 збільшується локально, а абсолютне значення кожного запису дає ступінь впливу на зміну прогнозу. Вектор η задає напрямок найкрутішого підйому від контрольної точки до вищих імовірностей для позитивного класу. Для бінарної класифікації негативна версія −ηp(x0) вказує на зміни в характеристиках, необхідні для збільшення ймовірності для негативного класу, який може бути особливо корисним для x0, передбаченого в позитивному класі.

Для прикладу ми застосовуємо визначення 2.2.3 до моделювання прогнозів, отриманих за допомогою класифікації процесів Гауса (GPC). GPC використовується тут з трьох причин:

1) У нашій реальній програмі ми зацікавлені в класифікації даних про злоякісні програми, це область, де гаусівські процеси показали

найсучаснішу продуктивність. Природно очікувати модель з високою точністю прогнозування на комплексі проблем захоплення відповідної структури даних, яку варто пояснити та яка може дати конкретні відомості на додаток до прогнозованих значень;

2) GPC дійсно моделює функцію ймовірності класу, яка використовується у визначенні 2.2.3 безпосередньо. Для інших методів класифікації, таких як опорні векторні машини, які не забезпечують функцію ймовірності, на виході ми надаємо приклад для методу оцінки, починаючи з визначення;
3) Локальні градієнти функції ймовірності можуть бути розраховані аналітично для диференційованих ядер.



(a) Object (b) Model

(c) Local explanation vectors (d) Direction of explanation vectors

Рисунок 2.1 – Класифікація об’єкта за допомогою процеса Гауса

Панель (a) рисунка 2.1 показує навчальні дані простого завдання класифікації об’єктів, а панель (b) показує модель, отриману за допомогою GPC. Дані позначені −1 для синіх точок і +1 для червоних балів. Як показано на панелі (b), модель є функцією ймовірності для позитивного класу, який дає кожній точці даних ймовірність бути в цьому класі. Панель (c) показує градієнт ймовірності моделі разом із локальними векторами пояснення градієнта. Уздовж гіпотенузи і в кути пояснень трикутника з обох об’єктів взаємодіють з класом трикутників, тоді як по краях домінує важливість одного з двох вимірів об’єктів. При переході від негативного до позитивного класу довжина векторів локального градієнта представляє збільшення важливості відповідних особливостей. На панелі (d) ми бачимо, що пояснення розташовані близько до країв ділянки (особливо в правому кутку) від позитивного класу. Однак панель (c) показує, що їх величина дуже мала.

2.2.2 Оцінка навчальних векторів

Кілька методів класифікації безпосередньо оцінюють правильне рішення, яке часто не має тлумачення як функція ймовірності. Наприклад, функція оцінки опорних векторних машин має форму

$$f(x) = \sum_{i=1}^{n} a_i k(x_i, x) + b \quad (2.2.4)$$

де αi,b ∈ ℜ. Припустимо, у нас є два класи (кожен з одним кластером) в одному вимірі і навчене SVM з ядром RBF. Для точок поза кластерами даних f(x) прагне до нуля. Таким чином, похідна f(x) (показана стрілками над кривими) для точок ліворуч або праворуч сторони осі буде вказувати на неправильну сторону.

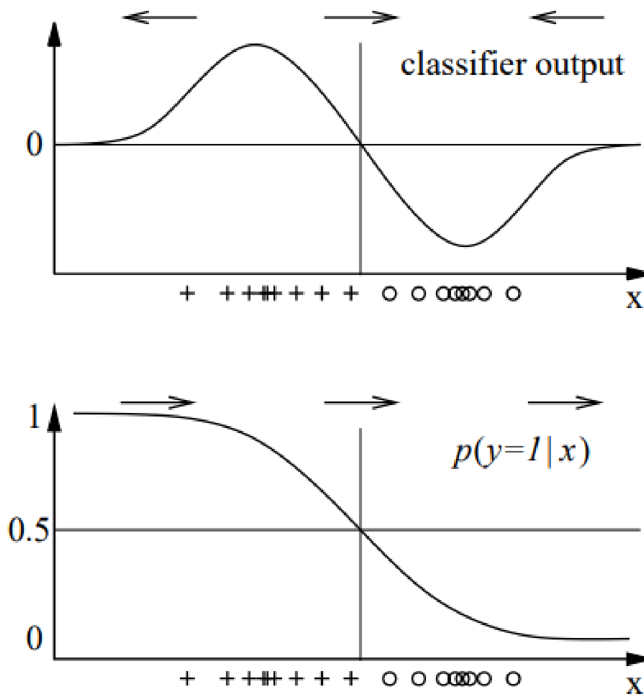


Рисунок 2.2 – Вивід класифікатора SVM порівняно з p(y=1|x)

На рисунку 2.2 можна побачити різницю у даних після порівняння SVM з представленою функцією. На практиці ми не маємо доступу до справжнього розподілу P(X,Y). Отже, ми не маємо доступу до класифікатора Байєса. Натомість ми можемо застосувати складні механізми навчання, такі як опорні векторні машини які оцінюють деякий класифікатор ¨ g, який намагається імітувати g∗. Для тестових точок даних z1,...,zm ∈ ℜd, які, як припускається, взяті з того самого невідомого розподілу, що й навчальні дані, g оцінює мітки g(z1),...,g(zm). Тепер, замість того, щоб намагатися пояснити g∗, до якого ми не маємо доступу, ми визначимо вектори пояснення, які допоможуть нам зрозуміти класифікатор g на тестових даних.

Оскільки ми не припускаємо, що маємо доступ до якогось проміжного реального результату класифікатора (з яких g може бути пороговою версією, а яка далі може не бути оцінкою P(Y =c | X =x)), пропонуємо апроксимувати g іншим класифікатором ˆg, фактичний вигляд якого нагадує класифікатор Байєса. Є кілька варіантів для ˆg, наприклад, GPC, логістична регресія та Вікна Парзен [24] .

У цій статті ми застосовуємо вікна Парзена до навчальних точок, щоб оцінити зважені щільності класів P(Y =c)·P(X | Y =c), для набору індексів Ic = {i | g(xi) = c}

$$\hat{\rho}_{\sigma}(x, y = c) = \frac{1}{n}\sum_{i \in 1_c} k_{\sigma}(x - x_i) \tag{2.2.5}$$

і з kσ(z) є ядром Гауса kσ(z) = exp(−0,5 z⊤z/σ2)/√2πσ2 (як завжди інші ядра також можливі).

Підводячи підсумок, ми імітуємо класифікатор g, який ми хотіли б пояснити локально Парзеном віконний класифікатор ˆg, який має ту саму форму, що й оцінка Байєса, і для якого ми можемо оцінити пояснювальні вектори, використовуючи визначення 2.2.1. Практично є деякі застереження: імітуючий класифікатор gˆ має бути оцінено з g навіть у великих розмірах; це потрібно робити обережно. Однак, в принципі ми маємо довільну кількість навчальних даних, доступних для побудови ˆg, оскільки ми можемо використовувати наш оцінений класифікатор g для генерації позначених даних.

2.2.3 Пояснення класифікації додатків за допомогою процесів Гауса

У цьому розділі ми описуємо застосування нашої методології пояснення локального градієнта на складний набір даних з реального світу. Наша мета полягає в тому, щоб знайти структуру, специфічну для проблемної області, яка не була введена в навчання явно, але неявно зафіксована моделлю GPC у багатовимірному просторі ознак, який використовується для визначення його прогнозу.

GPC застосовувався таким чином:

- Клас 0 складається з доброякісних застосунків;
- Клас 1 складається з злоякісних застосунків;
- Випадково розділяєм набір даних на половину навчальних і половину тестових прикладів;
- Кожен приклад представлений з ознак, які ми отримали під час декомпіляції APK-файлу;

- Нормалізуємо навчальний і тестовий набір, використовуючи середнє значення та дисперсію навчального набору;
- Застосуємо модель GPC з ядром RBF;
- Визначимо продукивність.

Разом із прогнозом ми обчислимо вектор пояснення (як введено у визначенні 2.2.3).

2.3 Критерії оцінювання ефективності моделі розпізнавання шкідливих програм

Основними критеріями ефективності ми зазначимо точність розпізнавання. Також надамо короткий опис інших можливих критеріїв:

Чутливість – це відсоток фактичних одиниць, які були правильно передбачені. Він показує, який відсоток одиниць охоплено моделлю.

Загальна кількість одиниць становить 71, з яких 70 було правильно передбачено. Отже, чутливість 70/71 = 98,59%

Чутливість має більше значення, коли правильна класифікація одиниць важливіша, ніж класифікація 0. Так само, як те, що нам потрібно тут у випадку раку молочної залози, коли ви не хочете пропустити будь-яку злоякісну пухлину, яка буде класифікована як «доброякісна».

Подібним чином Специфічність – це частка фактичних нулів, які були правильно передбачені. Отже, у цьому випадку це 122 / (122+11) = 91,73%. Конкретність має більше значення, коли правильна класифікація нулів важливіша, ніж класифікація одиниць. Максимальна конкретизація більш актуальна в таких випадках, як виявлення спаму, коли ви суворо не хочете, щоб справжні повідомлення (0) потрапляли в спам (1).

Швидкість виявлення – це частка всієї вибірки, де події були виявлені правильно. Отже, це 70 / 204 = 34,31%.

3 ПРОГРАМНА РЕАЛІЗАЦІЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ РОЗПІЗНАВАННЯ ШКІДЛИВИХ ПРОГРАМ НА МОБІЛЬНИХ ПЛАТФОРМАХ

3.1 Формування навчальних та тестових даних

3.1.1 Завантаження датасету

Ми починаємо із завантаження набору даних програм для Android під назвою DrebinRed, що складається з 12 000 доброякісних і 550 шкідливих зразків, витягнутих із набору даних Drebin [25] :

Таблиця 3.1 — Вміст набору даних

Назва	Кількість
Застосунки	12550
Доброякісні	12000
Злоякісні	550
Ознаки	1227080

3.1.2 Навчання класифікатора

Ми навчаємо класифікатор Support Vector Machine на половині набору даних (6000), а решту використовуємо для тестування. Ефективність завдання розпізнавання нешкідливих і зловмисних програм повідомлятиметься за допомогою коефіцієнта виявлення при 2% хибнопозитивних результатів, оцінки F1 і побудови кривої робочих характеристик приймача (ROC).

3.1.3 Пояснення рішень

У цьому розділі пояснимо додаткові пояснення для навченого детектора зловмисного програмного забезпечення Android, аналізуючи навчену модель, щоб зрозуміти, які компоненти програм є більш релевантними на етапі прийняття рішення (класифікації).

Ми використовуємо метод пояснення Gradient * Input на основі градієнта, який реалізовано класом CExplainerGradientInput. Наш алгоритм оптимізовано для

роботи з розрідженими даними, що робить його придатним для використання в цьому прикладі програми. Для кожного зразка, одного доброго й одного зловмисного, ми обчислюємо пояснення щодо позитивного (зловмисного) класу та перераховуємо 10 найбільш впливових функцій разом із відповідною релевантністю (%).

```
Пояснення для APK 137 (є зловмисним: 0)
 -8.30  req_permissions::android.permission.CALL_PHONE
 -6.59  suspicious_calls::android/net/Uri;->toString
  5.60  req_permissions::android.permission.INTERNET
 -5.39  used_permissions::ACCESS_NETWORK_STATE
 -5.08  api_calls::android/media/MediaPlayer;->start
 -4.24  used_permissions::VIBRATE
 -3.65  intent_filters::android.intent.category.LAUNCHER
 -3.56  req_permissions::android.permission.ACCESS_FINE_LOCATION
 -3.43  used_permissions::ACCESS_FINE_LOCATION
 -2.63  api_calls::android/location/LocationManager;->getLastKnownLocation
```

Рисунок 3.1 – Пояснення для доброякісного прикладу

Дивлячись на перший зразок (рисунок 3.1), доброякісну програму, ми можемо спостерігати, як більшість функцій мають негативне значення для рішення, що означає, що для цього класифікатора є показником доброякісної поведінки.

```
Пояснення для APK 138 (є зловмисним: 1)
 15.18  suspicious_calls::android/telephony/TelephonyManager;->getNetworkOperator
 12.55  req_permissions::android.permission.SEND_SMS
  8.69  req_permissions::android.permission.READ_SMS
  5.83  req_permissions::android.permission.INTERNET
  5.58  intent_filters::android.intent.action.BOOT_COMPLETED
 -4.41  used_permissions::VIBRATE
  3.96  intent_filters::android.intent.category.HOME
 -3.80  intent_filters::android.intent.category.LAUNCHER
  3.66  receivers::com.google.android.c2dm.C2DMBroadcastReceiver
  3.39  req_permissions::android.permission.READ_PHONE_STATE
```

Рисунок 3.2 – Пояснення для злоякісного прикладу

Що стосується другого зразка (рисунок 3.2), шкідливої програми, ми можемо спостерігати протилежне, оскільки більшість ознак мають позитивне значення релевантності, що означає, що для цього класифікатора є ознакою шкідливої поведінки.

Ми також спостерігаємо, що понад ~50% релевантності призначається лише 10 функціям в обох випадках. Це підкреслює відому поведінку цих класифікаторів, які, як правило, призначають більшу частину ваги невеликому набору функцій, що робить їх вразливими до атак противника.

3.1.4 Створення змагальних прикладів

Тепер ми встановили атаку максимальної впевненості ухилення на основі градієнта, щоб створити змагальні приклади проти класифікатора SVM, на якому базується детектор зловмисного програмного забезпечення Android.

По-перше, ми вибираємо параметри розв’язувача. Оскільки ми працюємо з логічними функціями (кожна може приймати значення 0 або 1), ми використовуємо крок сітки пошуку рядків eta 1. Потім ми вибираємо l1 як відстань, щоб виконати розріджену атаку рівня L1. Лише одна функція змінюється (з 0 на 1 або навпаки) на кожній ітерації. Нарешті, оскільки ми хочемо, щоб шкідливі зразки класифікувалися як доброякісні, ми встановлюємо y_target = 0 для здійснення цілеспрямованої атаки.

Обмеження нижньої та верхньої меж є критичними в цій програмі. Щоб створити зловмисне програмне забезпечення, здатне обдурити класифікатор, зловмисник теоретично може як додавати, так і видаляти функції з оригінальних програм. Однак на практиці видалення функції є нетривіальною операцією, яка може легко скомпрометувати шкідливі функції програми та, загалом кажучи, виконуватись лише для непроявлених компонентів. Додавання функцій є безпечнішою операцією, особливо коли введені функції належать до маніфесту; наприклад, додавання дозволів не впливає на наявні функції програми.

Отже, у цьому прикладі, щоб дозволити лише додавання функцій, ми встановили lb = 'x0' і ub = 1. Щоб також дозволити видалення функцій, можна встановити lb = 0

3.1.5 Оцінка безпеки

Щоб оцінити надійність детектора зловмисного програмного забезпечення Android щодо збільшення кількості модифікованих (доданих) функцій, SecML надає спосіб легко створити криву оцінки безпеки за допомогою класу CSecEval.

Екземпляр CSecEval запустить атаку ухилення від класифікатора, використовуючи зростаючі значення eps збурення порядку L1. Цей процес має тривати приблизно 60 секунд, незважаючи на роботу над понад 1 мільйоном функцій із використанням атаки CAttackEvasionPGDLS, оскільки він оптимізований для роботи з розрідженими даними.

3.2 Короткий опис програмного забезпечення

3.2.1 Apktool

Щоб отримати файли з APK архіва, було застосовано зворотне проектування за допомогою програми «apktool», яка є незалежним від платформи інструментом на основі Java, яка використовується для декодування ресурсів до майже оригінальної форми та їх реконструкції після внесення деяких змін. Цей інструмент використовувався для зворотного проектування файлів apk Android і перетворення їх у відповідні папки для доступу до файлу AndroidManifest.xml та інших файлів.

```
C:\Users\Admin\Desktop>apktool d "Draw Signature
_v2.5.0_apkpure.com.apk"
I: Using Apktool 2.4.0 on Draw Signature_v2.5.0_
apkpure.com.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources..
.
S: WARNING: Could not write to (C:\Users\Admin\A
ppData\Local\apktool\framework), using C:\Users\
Admin\AppData\Local\Temp\ instead...
S: Please be aware this is a volatile directory
and frameworks could go missing, please utilize
--frame-path if the default storage directory is
 unavailable
I: Loading resource table from file: C:\Users\Ad
min\AppData\Local\Temp\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Рисунок 3.3 – Консоль під час декодування APK файлу

На рисунку 3.3 показано журнали терміналу під час запуску apktool.

3.2.2 Python

Наука про дані витягує цінну інформацію з даних, а машинне навчання (ML) дозволяє комп'ютерам автоматично навчатися даних і робити точні прогнози. Фахівці з роботи з даними використовують Python для вирішення наступних завдань:

- Виправлення та видалення неверних даних (очистка даних)
- Залучення та вибір різних характеристик даних
- Розметка даних додає дані значимі імена
- Пошук статистичної інформації в даних
- Візуалізація даних за допомогою діаграм і графіків: лінійних діаграм, столбчатих діаграм, гістограмм і кругових діаграм

Фахівці з роботи з даними використовують бібліотеки Python ML для моделей машинного навчання та створення класифікаторів, які точно класифікують дані. Класифікатори на основі Python використовуються в різних областях і застосовуються для виконання таких завдань, як класифікація зображень, текстів і мережевого трафіку, розпізнавання речей і розпізнавання осіб. Фахівці з роботи з даними також використовують Python для глибокого навчання — передової техніки машинного навчання.

3.2.3 SecML

SecML — це бібліотека Python з відкритим кодом для оцінки безпеки алгоритмів машинного навчання (ML). Він оснащений набором потужних функцій:

- Широкий спектр підтримуваних алгоритмів ML. Доступні всі алгоритми керованого навчання, які підтримуються scikit-learn, а також нейронні мережі (NN) через платформу глибокого навчання PyTorch;
- Вбудовані алгоритми атаки. Атаки ухилення та отруєння на основі розробленого на замовлення швидкого вирішувача. Крім того,

надаються з’єднувачі для інших сторонніх бібліотек Adversarial Machine Learning;

- Підтримка щільних/розріджених даних. Забезпечує повну прозору підтримку як щільних (через бібліотеку numpy), так і розріджених даних (через бібліотеку scipy) в одній структурі даних;
- Візуалізація результатів. Використовується структура візуалізації та побудови графіків на основі широко відомої бібліотеки matplotlib;
- Пояснення результатів. Зрозумілі методи ML для інтерпретації модельних рішень за допомогою впливових функцій і прототипів;
- Широкий спект моделей;
- Багатопроцесорність. Ця бібліотека забезпечує повну сумісність із усіма функціями багатопроцесорної обробки scikit-learn і pytorch, а також вбудовану підтримку бібліотеки joblib;
- Розширюваний. Легко створювані нові компоненти, наприклад моделі ML або алгоритми атак, розширюючи надані абстрактні інтерфейси.

3.3 Результати машинного навчання

В результаті застосування тестової вибірки даних з 12550 зразків шкідливого програмного забезпечення як вхідних даних побудованої нейронної мережі з використанням градієнтного методу були отримані графіки точності за епохами навчання на навчальних даних і перевірки ефективності моделі на тестових даних.

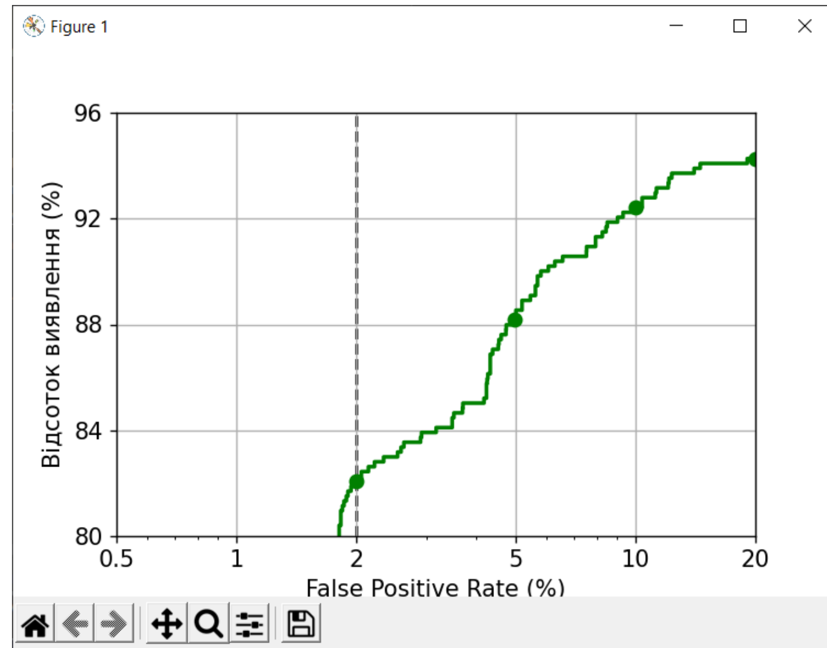


Рисунок 3.4 – Точність виявлення після навчання 300 семплів

Аналіз рисунка 3.4 показує, що на I епосі вдалося досягти точності 82.07% при 2% FPR після навчання моделі 300 прикладами.

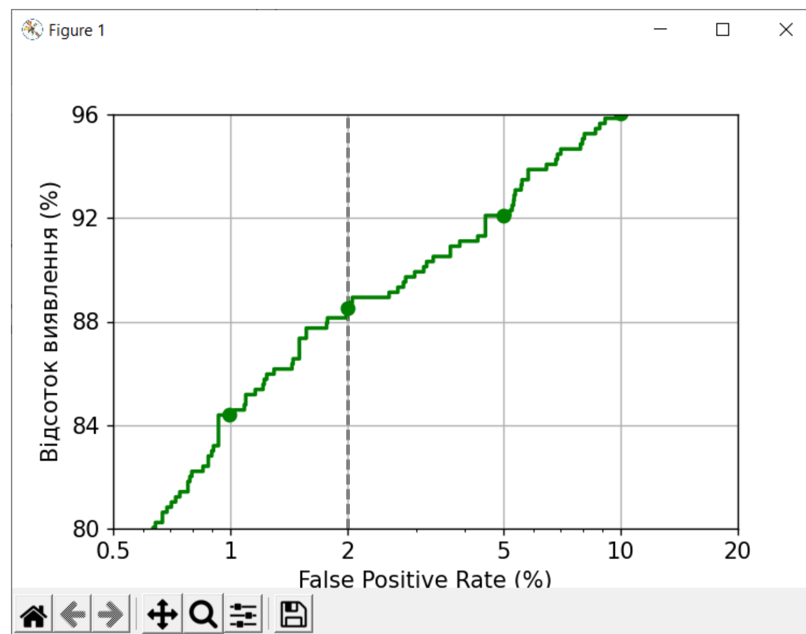


Рисунок 3.5 – Точність виявлення після навчання 1000 семплів

Аналіз рисунка 3.5 показує, що на II епосі вдалося досягти точність 88.54% при 2% FPR після навчання моделі 1000 прикладами. Це більше, порівняно з I епохою.

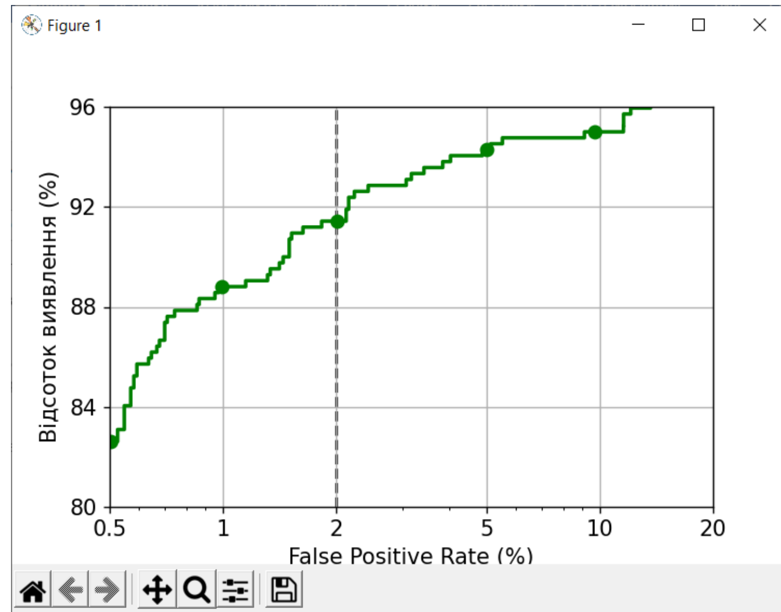


Рисунок 3.6 – Точність виявлення після навчання 3000 семплів

Аналіз рисунка 3.6 показує, що на III епосі вдалося досягти точність 91.43% при 2% FPR після навчання моделі 3000 прикладами. Це більше, порівняно з I і II епохами.

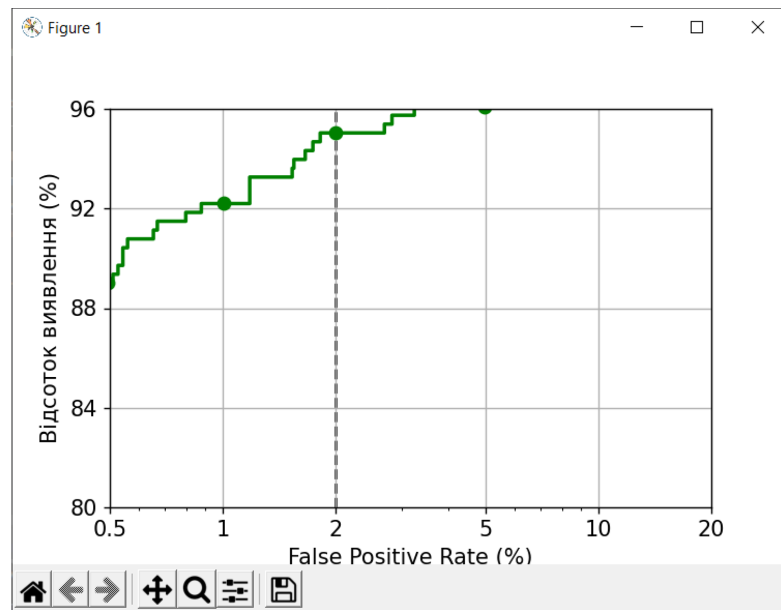


Рисунок 3.7 – Точність виявлення після навчання 6000 семплів

Аналіз рисунка 3.7 показує, що на IV епосі вдалося досягти точність 95.04% при 2% FPR після навчання моделі 6000 прикладами. Це більше, порівняно з усіма попередніми епохами.

Таким чином, точність вибірки даних змінювалась від значення 82.07% до максимального — 95.04% на IV епосі навчання. Отримані значення свідчать про високу точність моделі розпізнавання.

В результаті, на максимальній кількості навчань ми отримали точність виявлення 95.04% при 2% FPR. На останній (IV) епосі навчання моделі відбувалось на основі 6000 семплів.

Також ми розрахували F1 для останньої епохи, яка дорівнює 89.09%.

Графіки демонструють підвищення якості моделі зі збільшенням кількості епох для навчання моделі. Подальший ріст точності є можливим, але існує ризик перенавчання (англ. overfitting) мережі, при якому значення функцій втрат для тестових даних вже почнуть зростати зі збільшенням епох.

ВИСНОВКИ

Смартфони стали одним із основних джерел існування людини. У міру того, як користування смартфонами збільшується, у смартфонах людини з’являються повні відомості та джерела даних для щоденного використання. У цьому випадку зовнішнього зловмисника дуже легко дізнатися про користувача, лише викравши сам смартфон, тому безпека смартфонів стала однією з головних проблем сьогодні. Підхід об’єднання класифікаторів із алгоритмами машинного навчання, такими як Дерево рішень, SVM і Gaussian Gaussan NB для виявлення зловмисного програмного забезпечення Android, можна використовувати для ідентифікації пристрою зловмисного програмного забезпечення, яке використовують зловмисники. Запропонована система об’єднання зосереджена на алгоритмах голосування, які дозволяють об’єднання на вищому рівні з використанням обчислювального підходу, а не багатьох інших традиційних метакласифікаторів. Результати аналізу ефективності демонструють її ефективність у покращенні продуктивності за допомогою класифікаторів. Метод Гауса також допоміг нам підвищити точність на високому рівні до 95.04%. Ця модель може виявити зловмисне програмне забезпечення з навченого набору даних.

ЛІТЕРАТУРА

1. IDC статистики даних Android [EB/OL]. http://www.baijingapp.com/article/7842
2. X. Li, J. Liu, Y. Huo, R. Zhang and Y. Yao, «Метод виявлення зловмисного програмного забезпечення Android на основі файлу маніфесту Android», 4-та Міжнародна конференція з хмарних обчислень і інтелектуальних систем (CCIS), 2016 р., Пекін , 2016, С. 239-243.Р. Т. Ванг, «Назва гл
3. L. D. Coronado-De-Alba, A. Rodríguez-Mota and P. J. E. Ambrosio, "Вибір функцій і набір класифікаторів для виявлення шкідливих програм Android", 2016 8th IEEE Latin-American Conference on Communications (LATINCOM), Medellin, 2016, pp. 1-6.
4. Qin Z, Xu Y, Liang B та ін. Статичний метод виявлення шкідливих програм Android [J]. Журнал Південно-Східного університету, 2013, 43(6):1162-1167.
5. Felt A P, Chin E, Hanna S та ін. Дозволи Android демістифіковані[C]// Конференція ACM з безпеки комп’ютерів і зв’язку. ACM, 2011:627-638.
6. Янг З, Янг М. LeakMiner: Виявлення витоку інформації на Android за допомогою аналізу статичного забруднення[C]// Розробка програмного забезпечення. IEEE, 2012:101-104.
7. https://www.academia.edu/59115732/Malware_Detection_in_Android_Apps_Using_Static_Analysis
8. Qiao Y, Yang Y, He J та ін. CBM: безкоштовна система автоматичного аналізу зловмисного програмного забезпечення з використанням послідовностей викликів API [M]// Розробка та управління знаннями. Springer Berlin Heidelberg, 2014:225-236.
9. Там К., Хан С. Дж., Фатторі А. та ін. CopperDroid: автоматична реконструкція поведінки зловмисного програмного забезпечення Android[C]// Симпозіум із безпеки мережі та розподілених систем. 2015 рік.

10. Zheng C, Zhu S, Dai S та ін. SmartDroid: автоматична система для виявлення умов запуску на основі інтерфейсу користувача в програмах Android [J]. 2012 рік.

11. Qiao M, Sung A H, Liu Q. Об'єднання функцій дозволу та API для виявлення шкідливих програм Android [C]// Iiai International Congress on Advanced Applied Informatics. 2016: 566-571.

12. Муньос А, Мартін І, Гусман А та ін. Виявлення зловмисного програмного забезпечення Android із метаданих Google Play: вибір важливих функцій[C]// Комунікації та мережева безпека. IEEE, 2015:701-702.

13. Peiravian N, Zhu X. Машинне навчання для виявлення зловмисного програмного забезпечення Android за допомогою дозволів і викликів API [C]// IEEE, Міжнародна конференція про ІНСТРУМЕНТИ зі штучним інтелектом. IEEE, 2013:300-305.

14. Mas'Ud MZ, Sahib S, Abdollah MF та ін. Аналіз вибору функцій і класифікатор машинного навчання в виявленні зловмисного програмного забезпечення Android[C]// Міжнародна конференція з інформаційних наук і програм. IEEE, 2014:1-5.

15. Р. Валле-Раі, П. Ко, Е. Ганьон, Л. Хендрен, П. Лам і В. Сундаресан. Soot - фреймворк оптимізації байт-коду Java. На конференції Центру перспективних досліджень із спільних досліджень, 1999 р.

16. С. Растофер, С. Арцт, Е. Бодден. Підхід машинного навчання для класифікації та категоризації джерел і приймачів Android. На щорічному симпозіумі з безпеки мереж і розподілених систем (NDSS), 2014.

17. В. Растогі, Ю. Чень, Х. Цзян. DroidChameleon: Оцінка антишкідливого програмного забезпечення Android проти атак трансформації. В AsiaCCS, 2013

18. С. Бернард, С. Адам і Л. Хойтте. Використання випадкових лісів для розпізнавання рукописних цифр. На Дев'ятій міжнародній конференції з аналізу та розпізнавання документів (ICDAR), 2007 р.

19. X. Wang, W. Wang, Y. He, J. Liu, Z. Han і X. Zhang, «Характеристика поведінки програм Android для ефективного виявлення malapps у великих масштабах», Future Generat. обчис. Syst., том. 75, стор. 30–45, жовтень 2017 р., doi: https://doi.org/10.1016/j.future.2017.04.041

20. Д. Арп, М. Спрейтценбарт, М. Хюбнер, Х. Гаскон і К. Рік, «Дребін: Ефективне та зрозуміле виявлення зловмисного програмного забезпечення Android у вашій кишені», в Proc. Netw. Розпод. сист. Secur. Симп., 2014, С. 23–26.

21. A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "Андродіаліз: аналіз ефективності намірів Android у виявленні шкідливих програм ", Comput. Secur., вип. 65, стор. 121–134, березень 2016.

22. K. W. Y. Au, Y. F. Zhou, Z. Huang і D. Lie, «PScout: Аналіз специфікації дозволів Android», в Proc. ACM Conf. обчис. Комун. безпеки, 2012, с. 217–228.

23. K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based detection malware on Android", IEEE Trans. Інф. Криміналістика безпеки, том. 11, № 6, стор. 1252–1264, черв. 2016 р., doi: https://doi.org/10.1109/TIFS.2016.2523912

24. Фрад і Ф. Кльрот. Методологія пояснення класифікації нейронних мереж. Нейронні мережі, 15(2):237 – 246, 2002. doi: 10.1016/S0893-6080(01)00127-7.

25. Арп, Даніель та ін. «Drebin: Ефективне та зрозуміле виявлення шкідливих програм Android у вашій кишені». NDSS. том. 14. 2014 рік.]

ДОДАТОК А

Код програми

```
import re

import secml
from secml import settings
from secml.utils import fm
from secml.utils.download_utils import dl_file_gitlab
from secml.utils import pickle_utils
from secml.data.splitter import CTrainTestSplit
from secml.ml.classifiers import CClassifierSVM
from secml.ml.peval.metrics import CMetricTPRatFPR, CMetricF1, CRoc
from secml.figure import CFigure
from secml.explanation import CExplainerGradientInput
from secml.adv.attacks.evasion import CAttackEvasionPGDLS
from secml.array import CArray
from secml.adv.seceval import CSecEval
from secml.ml.peval.metrics import CMetricTHatFPR, CMetricTPRatTH

repo_url = 'https://gitlab.com/secml/secml-zoo'
file_name = 'drebin-reduced.tar.gz'
file_path = 'datasets/DrebinRed/' + file_name
output_dir = fm.join(settings.SECML_DS_DIR, 'drebin-red')
md5_digest = 'ecf87ddedf614dd53b89285c29cf1caf'

ds_path = fm.join(output_dir, file_name)

# ********** DOWNLOADING DATASET **********

if not fm.file_exist(ds_path):
    try:
        min_version = re.search(r'^\d+.\d+', secml.__version__).group(0)
        dl_file_gitlab(repo_url, file_path, output_dir,
```

```
                    branch='v' + min_version, md5_digest=md5_digest)

    except Exception as e:
        dl_file_gitlab(repo_url, file_path, output_dir, md5_digest=md5_digest)

dataset = pickle_utils.load(ds_path)

print("Кількість семплів: ", dataset.num_samples)

n_neg = sum(dataset.Y == 0)
n_pos = sum(dataset.Y == 1)

print("Кількість доброякісних семплів: ", n_neg)
print("Кількість злоякісних семплів: ", n_pos)
print("Кількість ознак: ", dataset.num_features)

# ********** TRAINING **********

train_apps_number = 6000

dataset_train, dataset_test = CTrainTestSplit(train_apps_number,
random_state=0).split(dataset)

classifier = CClassifierSVM(C=0.1)
print("Навчання почалось для семплів:", train_apps_number)
classifier.fit(dataset_train.X, dataset_train.Y)
print("Навчання завершено!")

# Classification of test set and computation of performance metrics
y_prediction, score_prediction = classifier.predict(dataset_test.X,
return_decision_function=True)

fpr_th = 0.02  # 2% False Positive Rate
```

```
detection_r = CMetricTPRatFPR(fpr=fpr_th).performance_score(y_true=dataset_test.Y,
score=score_prediction[:, 1].ravel())
print("Точність при @ 2% FPR: {:.2%}".format(detection_r))
f1 = CMetricF1().performance_score(y_true=dataset_test.Y, y_pred=y_prediction)
print("F1 оцінка: {:.2%}".format(f1))

fpr, tpr, _ = CRoc().compute(y_true=dataset_test.Y, score=score_prediction[:,
1].ravel())

# ********** DRAWING TRAINED VALUES **********

fig_dr_to_fpr = CFigure(height=4, width=6)
fig_dr_to_fpr.sp.plot_roc(fpr, tpr)
fig_dr_to_fpr.sp._sp.axvline(x=2, c='k', linestyle='--', zorder=-1)
fig_dr_to_fpr.sp.xlim(0.5, 20)
fig_dr_to_fpr.sp.ylim(80, 96)
fig_dr_to_fpr.sp.yticks([80, 84, 88, 92, 96])
fig_dr_to_fpr.sp.yticklabels([80, 84, 88, 92, 96])
fig_dr_to_fpr.sp.ylabel(r'Відсоток виявлення $(\%)$')
fig_dr_to_fpr.show()

explainer = CExplainerGradientInput(classifier)

print("Розрахунок пояснень '{:}'".format(explainer.__class__.__name__))


# ********** EXPLAIN METHOD **********


def explain_for_index(i):
    x, y = dataset_test[i, :].X, dataset_test[i, :].Y

    print("Пояснення для APK {:} (є зловмисним: {:})".format(i, y.item()))
```

```
    attr = explainer.explain(x, y=1)
    attr = attr / attr.norm(order=1)  # To display values in 0-100

    attr_argsort = abs(attr).argsort().ravel()[::-1]

    n_plot = 10

    for i in attr_argsort[:10]:
        print("{:6.2f}\t{:}".format(attr[i].item() * 100,
dataset.header.feat_desc[i]))


def output_features_for_index(i):
    x = dataset_test[i, :].X
    y = dataset_test[i, :].Y

    print("Ознаки для семплу {:} (є злоякісним: {:})".format(i, y.item()))

    attr = explainer.explain(x, y=1)
    attr = attr / attr.norm(order=1)  # To display values in 0-100

    attr_argsort = abs(attr).argsort().ravel()[::-1]

    for i in attr_argsort[:10]:
        print("{:}".format(dataset.header.feat_desc[i]))


explain_for_index(137)
explain_for_index(138)
output_features_for_index(138)

# ********** ATTACK  **********

params = {
```

```
    "classifier": classifier,
    "distance": 'l1',
    "double_init": False,
    "lb": 'x0',
    "ub": 1,
    "attack_classes": 'all',
    "y_target": 0,
    "solver_params": {'eta': 1, 'eta_min': 1, 'eta_max': None, 'eps': 1e-4}
}

evasion_attack = CAttackEvasionPGDLS(**params)

n_mal = 10

# Attack DS
mal_idx = dataset_test.Y.find(dataset_test.Y == 1)[:n_mal]
adv_ds = dataset_test[mal_idx, :]

# Security evaluation parameters
param_name = 'dmax'  # This is the `eps` parameter
dmax_start = 0
dmax = 28
dmax_step = 4

param_values = CArray.arange(
    start=dmax_start, step=dmax_step, stop=dmax + dmax_step)

sec_eval = CSecEval(
    attack=evasion_attack,
    param_name=param_name,
    param_values=param_values)

print("Running security evaluation...")
sec_eval.run_sec_eval(adv_ds)
```

```
print("Security evaluation completed!")

fig_dr_to_fpr = CFigure(height=5, width=5)

# Get the ROC threshold at which Detection Rate should be computed
th = CMetricTHatFPR(fpr=fpr_th).performance_score(y_true=dataset_test.Y,
score=score_prediction[:, 1].ravel())

# Convenience function for plotting the Security Evaluation Curve
fig_dr_to_fpr.sp.plot_sec_eval(sec_eval.sec_eval_data,
metric=CMetricTPRatTH(th=th),
                               percentage=True, label='SVM', color='green',
marker='o')
fig_dr_to_fpr.sp.ylabel(r'Відсоток виявлення $(\%)$')
fig_dr_to_fpr.sp.xlabel(r"$\varepsilon$")
fig_dr_to_fpr.show()
```