МАТЕМАТИЧЕСКИЕ ОСНОВЫ КОМПЬЮТЕРНОЙ БЕЗОПАСНОСТИ

УДК 004.492.3

АНАЛИЗ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ МОДЕЛЕЙ ПРИВИЛЕГИЙ И API-ВЫЗОВОВ ВРЕДОНОСНЫХ ПРИЛОЖЕНИЙ

А. А. Сковорода, Д. Ю. Гамаюнов

Московский государственный университет им. М. В. Ломоносова, г. Москва, Россия

Предложен метод автоматической классификации мобильных приложений на основе статического анализа и сопоставления моделей, полученных по его результатам, с моделями ранее известных вредоносных приложений. Модели основаны на привилегиях и API-вызовах, используемых в приложении. Все шаги анализа, а также построение моделей полностью автоматизированы. Таким образом, метод адаптирован для автоматизированного использования магазинами мобильных приложений или другими заинтересованными организациями.

Используя предложенный метод, мы проанализировали коллекции вредоносных приложений Drebin и ISCX, а также более 40000 приложений из Google Play, собранных в период с 2013 по 2016 г. Результаты анализа показывают, что комбинация достаточно простых признаков, таких, как запрашиваемые привилегии и цепочки используемых API-вызовов, достаточна для проведения бинарной классификации приложений на вредоносные и легитимные, а также для определения семейства вредоносных приложений, при этом количество ложных срабатываний приемлемо (около 3 %). Результаты исследования коллекции вредоносных приложений показывают, что нынешние вредоносные Android-приложения редко используют техники обфускации или шифрования для затруднения статического анализа, что пока не соответствует наблюдаемому в области семейства платформ «Wintel». Представлено экспериментальное сравнение предложенного метода с другим эффективным методом анализа Android-приложений, реализованном в программном средстве adagio.

Ключевые слова: *статический анализ, вредоносные Android-приложения.*

DOI 10.17223/20710410/36/7

AUTOMATED STATIC ANALYSIS AND CLASSIFICATION OF ANDROID MALWARE USING PERMISSION AND API CALLS MODELS

A. A. Skovoroda, D. Y. Gamayunov

Lomonosov Moscow State University, Moscow, Russia

E-mail: nastya_jane@seclab.cs.msu.su

In this paper, we propose a heuristic approach to static analysis of Android applications based on matching suspicious applications with the predefined malware models.

Static models are built from Android capabilities and Android Framework API call chains used by the application. All of the analysis steps and model construction are fully automated. Therefore, the method can be easily deployed as one of the automated checks provided by mobile application marketplaces or other interested organizations.

Using the proposed method, we analyzed the Drebin and ISCX malware collections in order to find possible relationships and dependencies between samples in collections, and a large fraction of Google Play apps collected between 2013 and 2016 representing benign data. Analysis results show that a combination of relatively simple static features represented by permissions and API call chains is enough to perform binary classification between malware and benign apps, and even find the corresponding malware family, with an appropriate false positive rate of about 3 %. Malware collections exploration results show that modern Android malware rarely uses obfuscation or encryption techniques to make static analysis more difficult, which is quite the opposite of what we see in the case of the "Wintel" endpoint platform family.

We also provide the experiment-based comparison with the previously proposed state-of-the-art Android malware detection method *adagio*. This method outperforms our proposed method in resulting detection coverage (98 vs 91 % of malicious samples are covered) while at the same time causing a significant number of false alarms corresponding to 9.3 % of benign applications on average.

Keywords: *static analysis, Android malware.*

Введение

С появлением мобильных устройств, обладающих широким спектром предоставляемых возможностей, стала существенной и проблема мобильной безопасности. Многие пользователи хранят на устройствах личную и конфиденциальную информацию, включая данные о банковских картах и данные, необходимые для авторизации в различных системах. Получение пользовательских данных является целью большинства вредоносных мобильных приложений.

Для борьбы с вредоносными приложениями (ВП) предложено множество методов обнаружения и предотвращения [1]. Из методов обнаружения выделяют обнаружение аномалий и обнаружение злоупотреблений. Методы обнаружения злоупотреблений можно условно разделить на две группы: методы, использующие некоторые предположения о свойствах (функциях) вредоносных приложений, и методы, использующие для обнаружения сигнатуры/модели известных вредоносных образцов. К первой группе можно отнести, например, taint-анализ, нацеленный на обнаружение утечек данных с устройства [2, 3]. Такие методы ограничены теми классами ВП, функции которых в них рассматриваются. Однако задание алгоритма обнаружения для произвольной вредоносной функциональности весьма затруднительно, и такие методы, как правило, ограничены узким классом обнаруживаемых приложений. Другая группа методов также имеет недостаток: такие методы существенно зависят от используемых сигнатур/моделей и неприменимы для обнаружения неизвестных ВП (zero-days). Методы обнаружения аномалий [4, 5] требуют задания моделей легитимной функциональности, что является ещё более затруднительным, так как требуется гораздо более широкое покрытие функциональности по сравнению с вредоносной.

По основополагающим техникам методы обнаружения ВП можно разделить на две крупных группы: статические и динамические, в зависимости от использования эмуляции/исполнения анализируемого приложения. Более детальная классификация

методов обнаружения включает методы машинного обучения, анализ привилегий, мониторинг уровня заряда аккумулятора, облачные вычисления [1].

В работе описывается метод статического анализа приложений, использующий модели ВП. Модели приложений отражают их поведение и строятся, исходя из используемых приложением системных привилегий и API-вызовов — вызовов программного интерфейса фреймворка Android или некоторых сторонних библиотек. Построение моделей ВП не требует ручной работы специалиста, в отличие от многих других методов статического анализа [6, 7]. Результаты анализа приложения представляют собой информацию о том, была ли найдена подобная модель среди вредоносных, и список подобных моделей. Данный метод может быть использован магазинами мобильных приложений для анализа новых приложений перед их публикацией, антивирусными приложениями как составная часть анализа на сервере, корпорациями с политикой BYOD (Bring Your Own Device), а также отдельными исследователями в области информационной безопасности.

Мы применили данный метод для бинарной классификации на множестве приложений из коллекций вредоносных приложений Drebin и ISCX, а также части собранной нами коллекции легитимных приложений. В этом эксперименте метод правильно распознал как вредоносные 91 % приложений, неверно классифицировав 2 % легитимных приложений. Проведение того же эксперимента с программным средством для анализа Android-приложений adagio выявило более широкую полноту покрытия adagio, но при этом и значительно большее число ложных срабатываний.

Мы также применили предложенный метод для исследования коллекций Drebin и ISCX. Вредоносные приложения, для которых анализатором обнаружено сходство, были сгруппированы в кластеры семейств ВП. Полученные кластеры не полностью соответствуют заранее заданному разделению коллекций по семействам, которое должно было быть получено исследователями в процессе ручного анализа каждого из образцов. Нужно отметить, что приложения в таких предопределённых семействах очень сильно разнятся, и для некоторых из них действительно сложно обнаружить сходство даже вручную. Исходя из этого, мы считаем предложенный метод подходящим для определения сходства между приложениями, относящимися к одному и тому же семейству ВП. В работе представлены некоторые особенности и характеристики ВП, полученные в процессе ручного и автоматического анализа исследованных коллекций.

Тестирование метода на коллекции из более 40000 легитимных приложений выявило долю ложных срабатываний 3,2 %, приемлемую для статических методов, но слишком высокую для полностью автоматизированного использования, например в качестве самостоятельного антивирусного средства. Тем не менее при использовании реализации предложенного метода исследователями в области безопасности информация о найденных общих цепочках API-вызовов позволяет без затруднений отсеять возникшие ложные срабатывания. Таким образом, метод может быть очень полезен для упрощения и автоматизации работы вирусных аналитиков. Для полностью автоматизированного использования метод может быть дополнен другими подходами к обнаружению для исключения/уменьшения числа ложных срабатываний.

1. Краткий обзор существующих исследований

С момента появления первых угроз, вызванных мобильными ВП, исследователями в области информационной безопасности предложено множество эффективных решений. В [8] представлен всесторонний обзор методов обнаружения ВП, наряду с эволюцией мобильных ВП и их способов противостоять анализу.

Сигнатурные методы обнаружения были использованы одними из первых [9], однако они неустойчивы к модификациям ВП. Для телефонов с небольшими вычислительными ресурсами показали свою эффективность методы мониторинга уровня заряда аккумулятора [5, 10]. Как показано в [11], такие методы больше неприменимы к современному состоянию мобильной среды.

Ряд методов обнаружения ориентируется на некоторые признаки вредоносной активности (такие, как передача по сети конфиденциальной информации без подтверждения пользователем) в статическом представлении приложения [2, 6] или в наблюдаемом поведении [3]. Класс обнаруживаемых ВП для таких методов строго ограничен соответствующими признаками. Так, например, методы, нацеленные на обнаружение утечек данных, не могут быть применены для обнаружения ВП, предназначенных для вымогательства. Кроме того, программные средства с динамическим анализом, предназначенные для выявления утечек данных, нуждаются в дополнении средствами для генерации цепочек событий, приводящих к утечкам [12]. Другая группа методов использует модели/сигнатуры некоторых известных ВП для обнаружения схожей вредоносной активности [13, 9]. Метод, предложенный в данной работе, относится именно к этой группе. В [13, 9] используется преимущественно сигнатурный подход, о недостатках которого упомянуто выше, в работе [13] он дополнен динамическим анализом.

Для обнаружения вредоносных мобильных приложений также широко применяются алгоритмы машинного обучения. Для задач классификации и кластеризации Android-приложений используются как признаки на уровне приложений [14 – 16], так и на уровне операционной системы [17] и генерируемой сетевой активности. В [18] описана кластеризация вредоносных мобильных приложений и генерация их сигнатур на основе наблюдаемого HTTP-трафика. Как и предлагаемый метод, подобные методы существенно зависят от данных, используемых для обучения. Однако, в отличие от нашего метода, в методах, использующих машинное обучение, необходима представительная выборка легитимных приложений для обучения, составление которой довольно затруднительно с учётом большого разнообразия таких приложений. Кроме того, методы, использующие машинное обучение, предоставляют в качестве результата, как правило, бинарную оценку либо вероятность отнесения анализируемого приложения к вредоносным без пояснения причин такой классификации.

Использование API-вызовов и привилегий Android в качестве основы анализа не является отличительной чертой только предлагаемого метода. Многие ранее предложенные в данной области методы машинного обучения используют множество признаков, преимущественно состоящее из такого рода характеристик приложения [19, 20, 16, 14]. Однако во всех перечисленных методах модели приложений (векторы признаков) формируются на основе значительно более поверхностного анализа приложений по сравнению с предложенным в данной работе, и эффективность использования данных методов обоснована прежде всего хорошей приспособленностью алгоритмов машинного обучения к работе на больших выборках приложений. Кроме того, хотя данные методы и предоставляют некоторое разъяснение результатов классификации, оно является гораздо менее детализированным и наглядным по сравнению с пояснением результатов анализа, выдаваемых предлагаемым методом. Экспериментальное сравнение с одним из таких методов проводится в п. 6.

Многие из методов нацелены не на бинарную классификацию приложений, а на создание некоторых структур для дальнейшего анализа либо информации о приложении, предназначенной для пользователей [21, 22]. Метод, предложенный в данной работе, выдаёт в качестве основного результата бинарную оценку вредоносности ана-

лизируемого приложения и список подобных ему вредоносных моделей. Помимо основного результата, предоставляются данные о найденных схожих цепочках API-вызовов, которые могут быть очень полезны для аналитиков мобильных приложений.

Поскольку ресурсы мобильных устройств весьма ограничены, многие схемы анализа приложений не могут быть запущены непосредственно на них. Предложены эффективные схемы организации анализа, уменьшающие нагрузку на мобильные устройства [23, 24]. В данной работе вопрос внедрения не рассматривается подробно, основным вариантом предполагается использование в качестве одной из автоматических проверок магазинами мобильных приложений и антивирусными приложениями.

2. Построение моделей

Анализ приложений заключается в построении моделей анализируемого приложения и их сравнении с предварительно построенными моделями ВП. Используется три вида моделей, каждый из которых отражает поведение приложения на определённой степени детализации. Самый общий вид — это модель привилегий. Модели API-вызовов Android Framework и цепочек API-вызовов являются последовательными детализациями первой модели.

2.1. Модель привилегий

Операционная система Android использует систему привилегий (permissions) для разграничения доступа к программным и аппаратным ресурсам устройства [25]. Все привилегии, необходимые приложению для корректной работы, должны быть объявлены в манифесте приложения (файле AndroidManifest.xml), который содержится в архиве apk-файла приложения. На рис. 1 приведён фрагмент манифеста приложения с описанием запрашиваемых привилегий.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="10023" android:versionName="1.0.2"
  package="com.wia.ucgepcdvlsl"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
</manifest>
```

Рис. 1. Запрашиваемые привилегии в файле AndroidManifest.xml

Запрашиваемые привилегии могут относиться как к системным ресурсам, так и к ресурсам сторонних приложений, установленных на устройство. Проанализировав привилегии, запрашиваемые вредоносными приложениями, мы составили список из 101 системной привилегии (т. е. привилегии, относящейся к ресурсам фреймворка). Поскольку целью анализа является определение сходства с какими-либо ВП, мы не расширяли список системными привилегиями, запрашиваемыми приложениями из коллекции легитимных.

Моделью привилегий для приложения является вектор $v \in \{0,1\}^{101}$, каждый из компонентов которого соответствует одной из привилегий в составленном списке (привилегии упорядочены в некотором заданном порядке):

$$v_p = \begin{cases} 1, \text{ если привилегия } p \text{ запрашивается,} \\ 0 \text{ в противном случае.} \end{cases}$$

2.2. Модель API-вызовов Android Framework

Более точное представление о поведении приложения можно получить, исходя из API-вызовов функций фреймворка. Для построения таких моделей необходима декомпиляция приложения, которая осуществляется с помощью средства Androguard [26].

В [27] проанализирована система привилегий Android и для каждой из системных привилегий составлен список API-вызовов, требующих её наличия у приложения. Будем называть такие API-вызовы защищёнными. Пользуясь этими результатами, мы проанализировали API-вызовы, используемые приложениями из вредоносной коллекции, и составили список тех из них, которые являются защищёнными. Поскольку некоторые ВП не использовали ни одного вызова из списка защищённых, мы расширили список вызовов некоторыми незащищёнными вызовами так, чтобы у каждого из ВП использовалось по крайней мере 20 вызовов из итогового списка. Всего в списке получилось 384 API-вызова.

Моделью API-вызовов Android Framework для приложения является вектор $v \in \in \{0,1\}^{384}$, каждый из компонентов которого соответствует одному из API-вызовов в списке (API-вызовы упорядочены в некотором заданном порядке):

$$v_f = \begin{cases} 1, \text{ если API-вызов } f \text{ используется,} \\ 0 \text{ в противном случае.} \end{cases}$$

2.3. Модель цепочек API-вызовов

Модель цепочек API-вызовов является наиболее подробной моделью, используемой в предлагаемом анализе. Она представляет собой список последовательностей (цепочек) API-вызовов, в которых вызовы упорядочены в порядке их возможного следования при запуске приложения. Каждая цепочка соответствует *точке входа* в приложение — функции, вызываемой фреймворком при некотором событии, сгенерированном пользователем или системой. Простейшим примером точки входа может служить метод onCreate в стартовой активности (Activity) приложения. Модели цепочек API-вызовов также строятся по декомпилированному представлению приложения.

Построение модели

Нахождение возможных точек входа в приложение

Для нахождения возможных точек входа в приложение используется алгоритм, описанный в [28]. Он заключается в следующем.

1) Строится первичный набор точек входа: методы основных компонентов приложения (Activity, Service, Broadcast Receiver, Content Provider), которые вызываются фреймворком.
2) По заданным точкам входа строится граф достижимых функциональных вызовов.
3) Все методы, перегружающие методы классов Android Framework и недостижимые из заданных точек входа, добавляются в точки входа, если в графе достижимых функциональных вызовов содержится конструктор их класса. Шаги 2 и 3 повторяются до тех пор, пока на очередной итерации множество точек входа не останется неизменным.

Построение графа функциональных вызовов для каждой точки входа

Выделяя вершины, соответствующие точкам входа в общем графе достижимых функциональных вызовов, а также все вершины, достижимые из этих вершин, и рёбра, инцидентные им, получаем графы функциональных вызовов для каждой из точек входа. В полученных графах для любой начальной вершины рёбра, инцидентные ей, упорядочены в порядке следования функциональных вызовов внутри метода, соответствующего данной начальной вершине.

Формирование цепочек API-вызовов

API-вызовы Android Framework являются листьями в построенных графах функциональных вызовов. Кроме них, интерес для анализа также представляют API-вызовы библиотек, вкомпилированных в приложение. Потомки таких вершин исключаются из обхода, поскольку могут привести к совпадениям в цепочках для приложений, совершенно не схожих по функциональности, а лишь использующих одинаковые библиотеки. Таким образом, для каждого функционального графа приложения в порядке обхода в глубину формируется цепочка из API-вызовов Android Framework и библиотечных API-вызовов.

На рис. 2 приведён пример графа функциональных вызовов для точки входа onCreate одного из сервисов приложения. Заданный порядок обхода смежных вершин на рисунке соответствует обходу слева направо и сверху вниз. Соответствующая графу цепочка API-вызовов:

Landroid/app/Service;->onCreate,
Landroid/os/PowerManager;->newWakeLock,
Landroid/os/PowerManager$WakeLock;->acquire,
Ljava/lang/Object;-><init>,
Landroid/database/sqlite/SQLiteOpenHelper;-><init>,
Landroid/os/Handler;->postDelayed

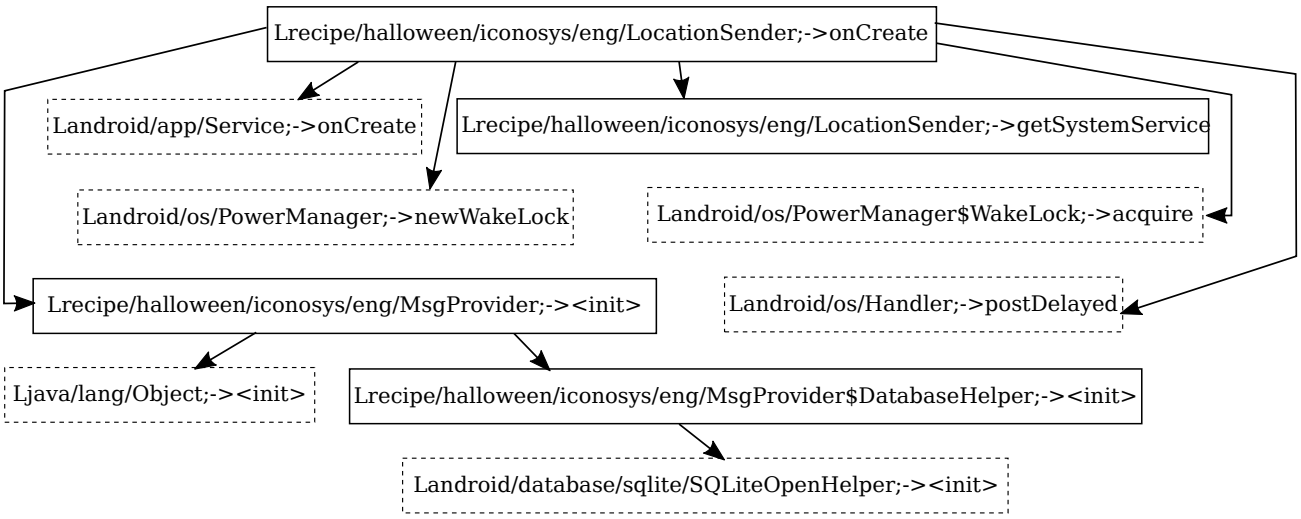

Рис. 2. Граф функциональных вызовов. Вершины, соответствующие API-вызовам, выделены пунктиром

В моделях цепочек API-вызовов для ВП содержатся и легитимные цепочки, которые могут быть встречены в невредоносных приложениях. Однако более длинные совпадения даже таких цепочек уже подозрительны и встречаются довольно редко. Больший интерес для анализа представляют цепочки моделей, содержащие ключевую

вредоносную функциональность модели. Приведём подцепочку такой цепочки в модели троянского приложения Mobiletx, которое собирает пользовательские данные (номер IMSI) и отправляет их в SMS-сообщении злоумышленникам:

Landroid/content/Context;->getSystemService,
Landroid/telephony/TelephonyManager;-> getSubscriberId,
Ljava/lang/StringBuilder;->append,
Ljava/lang/StringBuilder;->toString,
Landroid/telephony/SmsManager;->sendTextMessage

3. Анализ приложений

Анализ мобильных приложений состоит из построения моделей анализируемого приложения и сравнения их с предварительно построенными моделями ВП. Построение моделей приложения описано в п. 2. Сравнение полученных моделей с моделями ВП осуществляется в три этапа, на каждом из которых сравниваются между собой соответствующие модели. На первом этапе осуществляется сравнение с моделями привилегий всех ВП. Результатом является список ВП, модели привилегий которых оказались схожими с моделью привилегий анализируемого приложения, будем называть их *подобными по привилегиям приложениями*. На втором этапе сравнивается модель API-вызовов Android Framework с аналогичными моделями ВП, определённых как подобные по привилегиям на предыдущем шаге. В результате получается список ВП, подобных анализируемому по API-вызовам Android Framework. И наконец, на третьем этапе сравнивается модель цепочек API-вызовов с аналогичными моделями ВП, определённых как подобные на предыдущем этапе. Итоговым результатом анализа является список ВП, схожих с исследуемым по цепочкам API-вызовов. Соответственно анализируемое приложение считается вредоносным, если найдено хотя бы одно ВП, схожее с ним по цепочкам API-вызовов.

Вложенность результатов анализа обеспечивается за счёт вложенности используемых моделей. Действительно, приложения, имеющие схожие цепочки API-вызовов, имеют и схожие множества API-вызовов Android Framework. А так как возможность использования в приложении большинства API-вызовов Android Framework, рассматриваемых при построении этой модели, напрямую зависит от наличия соответствующей привилегии у приложения, то приложения, подобные по API-вызовам, в большинстве своем подобны и по привилегиям. Необходимо отметить, что результаты описываемого трёхэтапного анализа для большинства приложений будут совпадать с результатами анализа при использовании сравнения только модели цепочек API-вызовов с аналогичными моделями всех вредоносных приложений. Однако такой анализ был бы существенно более затратным по времени. К тому же результаты, полученные на первых двух этапах сравнения, могут представлять интерес для аналитиков мобильных приложений.

3.1. Сравнение моделей привилегий

Модели привилегий для приложений представляют собой двоичные векторы длины 101, значения которых соответствуют привилегиям из выбранного нами множества. Обозначим это множество привилегий как P. Введём веса для привилегий: обозначим вес привилегии p как w_p. Будем использовать большие веса для выделения наиболее опасных привилегий, таких, как android.permission.SEND_SMS; для большинства привилегий значение w_p равно 1. Более высокие веса опасных привилегий повышают значение функции сходства для анализируемых приложений, запрашивающих такие привилегии. Это не всегда соответствует большему сходству между сравниваемыми

приложениями, но, как правило, указывает на подозрительность приложения и необходимость его дальнейшего детализированного анализа.

Для сравнения модели привилегий v анализируемого приложения и модели привилегий u некоторого ВП вычисляется значение функции их сходства:

$$S_P(v,u) = \begin{cases} \dfrac{\sum\limits_{p\in P} w_p \cdot 1_{v_p=u_p \& u_p=1}}{\sum\limits_{p\in P} 1_{u_p=1}}, \text{ если } \sum\limits_{p\in P} 1_{u_p=1} \neq 0, \\ 1 \text{ в противном случае.} \end{cases} \quad (1)$$

Приложения считаются подобными по привилегиям, если $S_P(v,u)$ превышает наперед заданный порог $\text{Threshold}_{p_{\text{sim}}}$. В соответствии с (1) мы считаем все приложения подобными по привилегиям приложениям, не запрашивающим никаких системных привилегий. В качестве примера ВП, не запрашивающих никаких системных привилегий, можно привести ВП, эксплуатирующие уязвимости операционной системы для получения системных привилегий (root exploits).

3.2. Сравнение моделей API-вызовов Android Framework

Сравнение моделей API-вызовов Android Framework проводится аналогично сравнению моделей привилегий. Обозначим множество выбранных API-вызовов Android Framework как A. Для сравнения модели API-вызовов Android Framework v анализируемого приложения и модели API-вызовов Android Framework u некоторого ВП вычисляется значение функции их сходства:

$$S_{\text{API}}(v,u) = \frac{\sum\limits_{f\in A} 1_{v_f=u_f \& u_f=1}}{\sum\limits_{f\in A} 1_{u_f=1}}.$$

Приложения считаются подобными по API-вызовам Android Framework, если $S_{\text{API}}(v,u)$ превышает наперёд заданный порог $\text{Threshold}_{\text{API}_{\text{sim}}}$.

3.3. Сравнение моделей цепочек API-вызовов

Сравнение моделей цепочек API-вызовов основано на поиске наибольшей общей подпоследовательности между парами цепочек. Обозначим множество цепочек в модели приложения u за $\text{chains}(u)$. Для каждой пары цепочек c_v и c_u считается значение $\text{lcs}(c_v, c_u)$, равное количеству элементов в цепочке, являющейся их наибольшей общей подпоследовательностью. Саму наибольшую общую подпоследовательность для c_v и c_u обозначим $\text{cs}(c_v, c_u)$. Будем считать, что цепочки c_v и c_u имеют *общую подцепочку*, если их наибольшая общая подпоследовательность достаточно длинна, то есть относительная величина $\dfrac{\text{lcs}(c_v,c_u)}{|c_u|}$ превосходит порог $\text{Threshold}_{\text{common}}$ и абсолютное значение $\text{lcs}(c_v, c_u)$ не меньше $\text{Threshold}_{\text{common}_{\text{abs}}}$ API-вызовов.

Цепочки в модели анализируемого приложения рассматриваются в порядке уменьшения длины. Для каждой из цепочек c_v в модели анализируемого приложения, имеющих общие подцепочки с цепочками в модели ВП, выбираем цепочку $c_{u_{\text{similar}}}$, для которой общая подцепочка имеет наибольшую длину; в случае, если она не единственна, рассматриваются все такие цепочки. Общая подцепочка разбивается цепочкой c_v на *компоненты* — слитные участки. Если общих подцепочек наибольшей длины несколько, то из них выбирается та, которая разбивается на минимальное количество компонентов. На рис. 3 цепочки изображены в виде последовательностей символов, верхняя соответствует одной из цепочек анализируемого приложения, нижняя — одна из

цепочек ВП. Градиентной заливкой выделена их общая подцепочка с минимальным количеством компонентов, в данном случае их два.

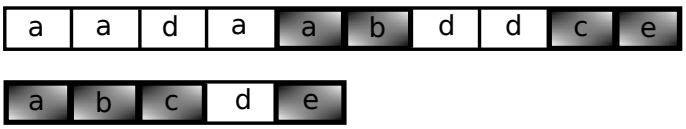

Рис. 3. Разбиение цепочки на компоненты

Обозначим количество компонентов, на которые разбивается общая подцепочка цепочек c_u и c_v как $\text{components}(c_v, c_u)$. Чем меньше компонентов в разбиении общей подцепочки, тем больше сходство цепочки c_v с цепочкой c_u, при этом слишком строгое ограничение на количество компонентов может привести к тому, что анализатором будут игнорироваться схожие цепочки даже с минимальными изменениями (например, добавление API-вызовов, не связанных с основной функциональностью цепочки). Поэтому в анализе мы учитывали только такие цепочки $c_{u_{\text{match}}}$ из $c_{u_{\text{similar}}}$, для которых выполнено следующее соотношение:

$$\text{components}(c_v, c_{u_{\text{match}}}) < \left\lceil \frac{1}{3} \operatorname{lcs}(c_v, c_{u_{\text{match}}}) \right\rceil. \tag{2}$$

Данное соотношение означает, что в общей подцепочке должен быть слитный участок из как минимум трёх звеньев (3 — величина, обратная коэффициенту в правой части соотношения). Если соотношение (2) выполнено для нескольких цепочек $c_{u_{\text{similar}}}$ из модели ВП, из них выбирается цепочка, содержащая защищённые API-вызовы (если таких нет, то любая цепочка). Таким образом, для каждой из цепочек c_v может быть выбрано не более одной цепочки в качестве наилучшего совпадения $c_{u_{\text{match}}}$ среди цепочек в модели ВП. Соответственно цепочки $\text{cs}(c_v, c_{u_{\text{match}}})$ формируют множество общих цепочек для сравниваемых моделей. Результатом сравнения являются четыре значения:

1) a — количество общих цепочек;
2) b — суммарная длина общих цепочек;
3) c — количество *длинных* общих цепочек (длинной считается цепочка с количеством элементов не меньше заданного порога $\text{Threshold}_{\text{long}_{\text{chain}}}$);
4) d — количество общих цепочек, содержащих защищённые API-вызовы.

Чем выше каждое из этих значений, тем больше сходство сравниваемых моделей. Мы считаем сравниваемые приложения подобными по цепочкам API-вызовов, если выполнено хотя бы одно из следующих условий:

— $a \geqslant \text{Threshold}_{\text{amount}}$ и $b \geqslant \text{Threshold}_{\text{length}}$ — наличие нескольких общих подцепочек с достаточно большой суммарной длиной;
— $c \geqslant 2$ — у приложений есть как минимум две длинные общие подцепочки;
— $c \geqslant 1$ и $d \geqslant 1$ — у приложений есть как минимум она длинная общая подцепочка и как минимум одна общая подцепочка, включающая защищённые API-вызовы;
— $d \geqslant 1$ и $b \geqslant \text{Threshold}_{\text{length}}$ — общие подцепочки приложений имеют достаточно большую суммарную длину и как минимум одна из них содержит защищённые API-вызовы;
— $\dfrac{a}{|\text{chains}(u)|} \geqslant 0{,}95$ и $\dfrac{b}{\sum\limits_{c \in \text{chains}(u)} \text{length}(c)} \geqslant 0{,}95$.

Последнее условие использовалось для обнаружения сходства с моделями, в которых количество цепочек и/или их суммарная длина невелики. Кроме этого, мы проверяем цепочки анализируемых приложений на полное совпадение независимо от их длины, считая приложения с одинаковыми цепочками подобными по цепочкам API-вызовов. Это сделано для обеспечения свойства рефлексивности отношения моделей, другими словами, каждое приложение подобно по цепочкам API-вызовов самому себе.

В приведённых условиях $|\text{chains}(u)|$ — количество цепочек API-вызовов в модели приложения u; $\text{length}(c)$ — длина (количество API-вызовов) цепочки c.

4. Подбор пороговых значений

Все пороговые значения, а также коэффициент в правой части соотношения (2) определены экспериментальным путём на части вредоносных и легитимных приложений из коллекций, подробно описанных в п. 5. В табл. 1 приведены пороговые значения, использованные в экспериментах. Эти значения могут быть перенастроены пользователями программного средства самостоятельно, хотя их изменение в ту или иную сторону может привести к изменению полноты обнаружения и количества ложных срабатываний (увеличение первого приводит к увеличению второго, и наоборот).

Таблица 1
Используемые пороговые значения

Наименование	Значение
$\text{Threshold}_{p_{\text{sim}}}$	0,9
$\text{Threshold}_{\text{API}_{\text{sim}}}$	0,7
$\text{Threshold}_{\text{common}}$	0,85
$\text{Threshold}_{\text{common}_{\text{abs}}}$	3
$\text{Threshold}_{\text{long}_{\text{chain}}}$	30
$\text{Threshold}_{\text{amount}}$	7
$\text{Threshold}_{\text{length}}$	50

Поскольку предложенный метод зависит от большого числа пороговых значений, практически невозможно исследовать зависимость его результатов от всех заданных пороговых значений одновременно. Поэтому мы изменяли каждое из пороговых значений по очереди при фиксированных остальных и выбирали значение, дающее наилучшие результаты.

Пороговые значения $\text{Threshold}_{p_{\text{sim}}}$ и $\text{Threshold}_{\text{API}_{\text{sim}}}$ должны быть выбраны так, чтобы итоговая доля ошибок 1-го рода была как можно ближе к нулю. При этом нужно минимизировать число подобных по привилегиям и API-вызовам Android Framework моделей, с которыми проводится сравнение на последнем (наиболее затратном по времени) шаге анализа. При подборе этих пороговых значений была использована половина коллекции *Drebin* для формирования моделей и ещё 300 приложений из этой коллекции для оценки результата работы предложенного метода (все эти приложения выбирались случайным образом). Подбирая $\text{Threshold}_{p_{\text{sim}}}$, мы пропустили второй шаг анализа, сразу сравнивая модели цепочек API-вызовов для приложений, подобных по привилегиям. Установлено, что доля ошибок 1-го рода одинакова для пороговых значений $\text{Threshold}_{p_{\text{sim}}} \in [0{,}1, 0{,}9]$ и равна 4,6 %; при значении $\text{Threshold}_{p_{\text{sim}}} = 0{,}95$ доля ошибок 1-го рода повышается до 5 %. В то же время среднее число подобных по привилегиям моделей падает с 1929 ($\text{Threshold}_{p_{\text{sim}}} = 0{,}1$) до 412 ($\text{Threshold}_{p_{\text{sim}}} = 0{,}9$). Поэтому для дальнейших экспериментов было выбрано пороговое значение $\text{Threshold}_{p_{\text{sim}}} = 0{,}9$.

В процессе подбора $\text{Threshold}_{\text{API}_{\text{sim}}}$ использован тот же критерий: максимально возможное сокращение числа моделей, подобных по API-вызовам Android Framework, при минимально возможной итоговой доле ошибок 1-го рода. На графике рис. 4 видно наилучшую точку с долей ошибок 1-го рода, равной 4,6 %, и средним числом подобных по API-вызовам Android Framework моделей, равным 142. Эта точка соответствует значению $\text{Threshold}_{\text{API}_{\text{sim}}} = 0{,}7$.

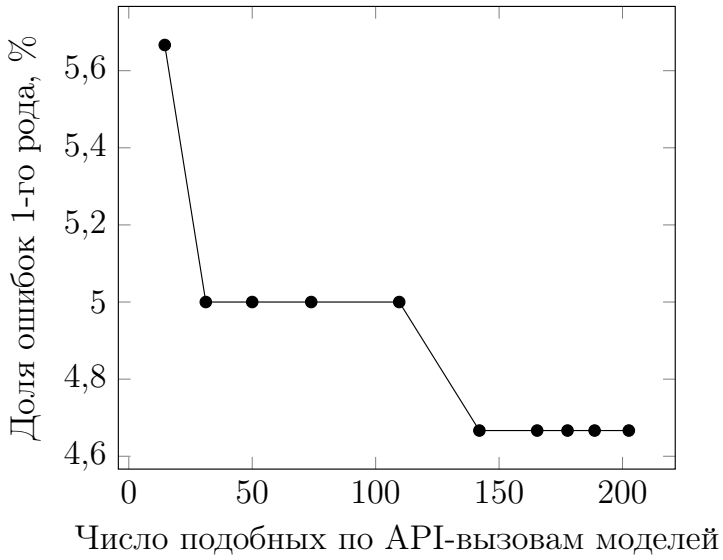

Рис. 4. Зависимость ошибки 1-го рода и среднего числа подобных по API-вызовам Android Framework моделей от $\text{Threshold}_{\text{API}_{\text{sim}}}$

Пороговые значения для сравнения цепочек API-вызовов были получены в процессе ручного анализа при сравнении некоторого числа приложений, относящихся к одному и тому же семейству ВП. Эти значения были проверены на части коллекции легитимных приложений, чтобы убедиться в их эффективности относительно ложных срабатываний.

5. Эксперименты

Далее описаны эксперименты, проведенные с использованием реализованного метода. Сначала метод был применен для решения задачи бинарной классификации, оценены доли ошибок 1-го и 2-го рода. Затем были исследованы коллекции ВП, показана способность метода обнаруживать похожие приложения внутри семейств ВП. Результаты этого же исследования были применены для выбора множества моделей ВП с минимальным количеством элементов таким образом, чтобы они покрывали все ВП в использованных коллекциях. Измерена доля ложных срабатываний метода на большой коллекции легитимных приложений.

5.1. Бинарная классификация

Для экспериментальной оценки метода были использованы две коллекции ВП: коллекция Drebin [16] и коллекция ISCX, предоставленная центром ISCX университа в Нью-Брансуик [29], в частности, её часть Android botnet. Общее число приложений в обеих коллекциях составляет 7489, однако с использованием предложенного метода не удалось построить модели 40 приложений, ещё для части приложений были построены нефункциональные модели, не содержащие ни одной цепочки API-вызовов. Более подробная информация об этом представлена далее. Таким образом, общее количество

проанализированных ВП составило 7449. В качестве выборки легитимных приложений использовано 3000 приложений, загруженных из Google Play. Предложенным методом не удалось построить модели семи из этих приложений. Сделана случайная выборка 2/3 ВП для формирования моделей ВП и протестирован метод на остальной части коллекции ВП, а также на всех легитимных приложениях. Этот эксперимент был проведён трижды, каждый раз со случайным разбиением общей коллекции ВП на модели и приложения для тестирования.

Предложенный метод показал в среднем долю ошибок 1-го рода менее 10 % и долю ложных срабатываний менее 2 % (табл. 2). Результаты промежуточных шагов анализа не предоставляются, поскольку они не существенны для оценки итоговой полноты покрытия метода, а также ложных срабатываний.

Таблица 2

Результаты анализа приложений (бинарная классификация)

Число моделей ВП	Срабатывания на вредоносной коллекции	Срабатывания на легитимной коллекции	Доля ошибок 1-го рода, %	Доля ошибок 2-го рода, %
4787	2254 / 2483	63 / 2993	9,3	2,1
4787	2240 / 2483	51 / 2993	9,7	1,7
4787	2262 / 2483	46 / 2993	9,1	1,5

5.2. Анализ коллекций вредоносных приложений

Предложенный метод был использован для исследования структуры двух вышеупомянутых коллекций ВП: Drebin и ISCX. Под исследованием здесь подразумевается обнаружение сходства и зависимостей между приложениями в коллекции. Основной целью было получение множества приложений из каждой коллекции, такого, что будучи взятым для создания моделей ВП, оно покроет всю оставшуюся часть коллекции, то есть каждое приложение из вредоносной коллекции будет подобным по цепочкам API-вызовов одной из моделей ВП.

Коллекция Drebin

Коллекция состоит из 5560 приложений, относящихся к 179 семействам. Все вредоносные образцы были собраны в период с августа 2010 по октябрь 2012 г. Из этих приложений 10 не было декомпилировано используемой библиотекой Androguard из-за программных ошибок в библиотеке или в приложениях (6 образцов не были распознаны как zip-архив утилитой file). Для 28 из оставшихся 5550 приложений не удалось построить ни одной цепочки API-вызовов. Эти приложения проанализированы вручную и выявлено, что 25 из них не запускаются эмулятором Android в среде динамического анализа Droidbox [30]. В ходе статического анализа в большинстве из них был обнаружен некорректный smali-код: он либо не содержал никаких инструкций методов, либо имена, объявленные в манифесте, не соответствовали действительным именам классов в приложении. Для трёх приложений не удалось построить цепочки, поскольку имена используемых в них пакетов совпадали с именами известных библиотечных пакетов, а такой код не учитывается при построении моделей. Про этот недостаток метода, а также способы борьбы с ним подробнее описано далее в п. 5.4 и 7. Таким образом, мы исключили 38 приложений из дальнейшего анализа.

На основе обнаруженного подобия по цепочкам API-вызовов среди приложений можно выделить 422 приложения так, что они покроют 5131 приложение в коллекции

(включая эти 422). Для 381 приложения не было найдено ни одного подобного по цепочкам приложения в коллекции. Далее были проанализированы кластеры подобных приложений, сформированные этими 422 приложениями. Каждому кластеру поставлена в соответствие метка семейства ВП (коллекции изначально были предоставлены размеченными) таким образом, чтобы метка кластера соответствовала меткам большинства приложений, образующих его. Затем было посчитано число приложений, у которых метка семейства не соответствовала метке кластера: 3,1 % (159 из 5131) приложений были неправильно кластеризованы. Таким образом, для формирования требуемого множества моделей ВП использовано $422+(5560-38-5131) = 813$ приложений из коллекции Drebin.

Кроме кластеризации, был проведён базовый анализ коллекции с целью получить представление об используемых приложениями практиках программирования и техниках обфускации. Результаты показывают, что 37 % приложений в коллекциях содержат встроенную рекламу; приложения преимущественно небольшого размера — 50 % приложений содержат меньше 50 классов. Анализ наиболее длинных цепочек API-вызовов выявил, что для 50 % приложений их длина меньше 70, в то время как для 92 % приложений длина таких цепочек меньше 200. Рефлексию используют 60,5 % приложений; 38 % используют API, связанные с выполнением нативного кода; 32 % используют криптографические API-вызовы (учитывался только пакет javax.crypto). Хотя предложенный метод не приспособлен для обнаружения использования нативного кода или рефлексии, с его помощью тем не менее можно обнаружить сходство между такими приложениями, основываясь на встреченных в цепочках API-вызовах рефлексии/исполнения нативного кода (Ljava/lang/Class;->forName, Ljava/lang/reflect/Method;->invoke, Ljava/lang/Runtime;->exec ...) и других частях приложений, не задействующих подобные техники. Мы также проанализировали 200 случайно выбранных приложений, чтобы получить представление об использовании обфускации имён в коллекции: 44 % не используют обфускации имён вовсе; в 51,5 % приложений обфусцирована часть названий классов, большинство из которых находятся внутри статически скомпилированных вместе с приложением библиотек; и только 4,5 % приложений используют обфускацию всех имён. В целом, ручной анализ приложений из коллекции выявил, что большинство приложений может быть достаточно легко подвергнуто обратной разработке и не использует сложной обфускации.

Коллекция ISCX

Коллекция ISCX была проанализирована по аналогии с коллекцией Drebin. Она состоит из 1929 приложений, принадлежащих к 14 семействам ВП, и содержит более новые образцы ВП, собранные с 2010 по 2014 г. Из них 30 приложений не были декомпилированы библиотекой Androguard, но в данном случае проблема была исключительно на стороне программного кода библиотеки. Предложенным методом не удалось построить цепочки API-вызовов для 66 образцов, из которых 19 содержали пакеты, имена которых совпадали с именами известных библиотек. Анализ оставшейся части приложений, для которых не удалось построить модели, выявил ещё один случай некорректных приложений: в некоторых приложениях все имена методов (включая такие методы, как onCreate, onStart) были модифицированы добавлением суффикса «abc123», что вызывало ошибки при попытке запуска компонентов. Суммарно из дальнейшего анализа были исключены 96 приложений коллекции.

На оставшейся части коллекции с помощью предлагаемого метода было сформировано 120 кластеров (состоящих из по меньшей мере двух приложений каждый), покрывающих 1723 приложения в коллекции. Другие приложения в коллекции не были кластеризованы и для 114 приложений не было обнаружено ни одного подобного по цепочкам API-вызовов приложения в коллекции. Из них 3,9 % (68 из 1723) были кластеризованы неверно, т. е. метка семейства ВП для них не соответствовала метке большинства приложений в кластере. Таким образом, мы сформировали требуемое множество моделей из $120 + (1929 - 96 - 1723) = 230$ приложений в коллекции.

Базовый анализ коллекции ICSX выявил, что приложения, представленные в ней, в целом крупнее и обладают более сложной структурой по сравнению с приложениями из коллекции Drebin: только 30 % приложений содержат менее 50 классов, почти 50 % содержат более 100 классов; 18 % приложений включают встроенную рекламу, 76 % используют рефлексию, 62 % содержат нативный код и более 53 % используют криптографические API-вызовы. Вручную проанализировано 100 приложений из коллекции, выбранных случайным образом, и выявлено, что 29 % не используют обфускации имён, 70 % обфусцируют имена части классов, относящихся к рекламным пакетам или другим библиотекам, и только в одном приложении все имена обфусцированы.

5.3. Анализ крупной коллекции легитимных приложений

Предложенный метод анализа мобильных приложений был протестирован на коллекции из 43819 мобильных приложений, загруженных из Google Play в период с 2013 по 2016 г., с использованием 850 приложений из коллекций Drebin и ISCX в качестве моделей ВП. Приложения для создания моделей получены объединением моделей для каждой коллекции с исключением дублирующихся и похожих приложений — коллекции частично пересекаются. В этом эксперименте пороговое значение $\text{Threshold}_{p_{\text{sim}}}$ было установлено в 0,7 вместо 0,9, чтобы выявить больше ложных срабатываний, вызванных именно подобием по цепочкам API-вызовов. Табл. 3 содержит результаты проведённого анализа.

Таблица 3

Результаты анализа коллекции легитимных приложений

Шаг анализа	Количество срабатываний	Доля ошибок 2-го рода, %
Привилегии (1)	43357 / 43819	98,9
API-вызовы Android Framework (2)	39451 / 43819	90,0
Цепочки API-вызовов (3)	1388 / 43819	3,2

Таким образом, доля ложных срабатываний (ошибок 2-го рода) составила 3,2 %. Эта оценка немного выше полученной в эксперименте с бинарной классификацией, поскольку использовано другое значение $\text{Threshold}_{p_{\text{sim}}}$. Тем не менее она лучше отражает устойчивость предложенного метода к ложным срабатываниям, поскольку основной частью анализа является сопоставление цепочек API-вызовов. Результаты промежуточных шагов (Привилегии (1) и API-вызовы Android Framework (2)) показывают количество анализированных приложений, для которых найдены подобные модели на соответствующем шаге. Как можно заметить из этих промежуточных результатов, только множества запрашиваемых привилегий и API-вызовов недостаточно для точного определения поведения приложения и характеристики его как вредоносного или легитимного. Первый и второй шаги анализа используются в основном для сокраще-

ния числа моделей ВП, используемых в сравнении на последнем и наиболее затратном по времени шаге анализа.

5.4. Анализ ошибок 1-го и 2-го рода

Ошибки 1-го рода в эксперименте с бинарной классификацией в основном связаны с отсутствием необходимых приложений в множестве моделей из-за их случайного выбора. Для обеспечения наилучшей полноты обнаружения предложенного метода необходим тщательный отбор моделей ВП.

Часть легитимных приложений из Google Play, для которых анализатором сгенерированы ложные срабатывания, была проанализирована вручную. Большинство ложных срабатываний возникло из-за наличия общего кода в пакетах, имена которых были изменены при обфускации приложений, а потому не были восприняты анализатором как библиотечный код. Как отмечено в п. 2, мы учитывали API-вызовы известных библиотек в построении цепочек, но исключали внутреннюю часть кода библиотек из дальнейшего анализа. Для реализации такого отсеивания был сформирован список известных библиотечных пакетов и отфильтровыван по именам. В программном средстве была реализована также фильтрация библиотечных пакетов по хеш-суммам их методов. Однако подходящий фильтр, содержащий все необходимые хеш-суммы, сформировать не удалось, это нетривиальная задача, требующая анализа большого числа легитимных приложений. Тем не менее у пользователей предложенного метода есть возможность задействовать свои фильтры Блума с хеш-суммами библиотечных методов и, таким образом, избавиться от этого недостатка.

Некоторые легитимные приложения имеют значительную часть общего кода с вредоносной моделью, в этих случаях, скорее всего, соответствующее ВП создавалось на основе ранней версии данного легитимного приложения (было переупаковано с добавлением функциональности). Ещё часть приложений содержала методы, схожие с некоторыми методами ВП, не включающими вредоносную активность (например, десериализация и сериализация XML-моделей).

6. Сравнение с adagio

Adagio — это программное средство, реализующее метод статического анализа Android-приложений, предложенный в [20]. Этот метод выбран для сравнения из-за сходства его основополагающих идей с предложенным методом, а также из-за доступности его исходного кода [31].

Adagio реализует алгоритм машинного обучения SVM, используя статические признаки приложений для формирования множества векторов признаков. Векторы признаков строятся по графам функциональных вызовов приложений по аналогии с моделями цепочек API-вызовов в предложенном методе. Однако графы функциональных вызовов интерпретируются по-другому: учитываются инструкции всех методов в графе вызовов, а также зависимости между ними для формирования множества хеш-сумм, впоследствии встраиваемого в пространство признаков. В предложенном методе учитываются API-вызовы Android Framework и известных библиотек, соответствующие параметрам только инструкций методов вида invoke*.

Мы провели трижды тот же эксперимент с бинарной классификацией на коллекции из 7449 вредоносных приложений из Drebin и ISCX и 3000 легитимных приложений, выделив 2/3 из них как обучающую выборку и 1/3 как проверочную. К сожалению, не удалось построить модели 509 вредоносных и 11 легитимных приложений. Табл. 4 содержит результаты эксперимента: число вредоносных и легитимных моделей немного разнится между экспериментами, поскольку 2/3 моделей для обучающей выборки

были выбраны случайным образом из объединения векторов признаков вредоносных и легитимных приложений.

Таблица 4

Результаты анализа приложений с использованием программного средства adagio

Вредоносные модели	Легитимные модели	Ошибки 1-го рода	Ошибки 2-го рода
4572	1983	45 / 2368 (1,9 %)	93 / 1006 (9,2 %)
4571	1984	51 / 2369 (2,2 %)	97 / 1005 (9,7 %)
4579	1976	57 / 2361 (2,4 %)	90 / 1013 (8,9 %)

Программное средство adagio с использованием конфигурационных значений по умолчанию показало среднюю долю ошибок 1-го рода 2,2 % и 2-го рода 9,3 %. Таким образом, предложенный метод имеет меньшую полноту покрытия со случайно выбранными вредоносными моделями, но в то же время является значительно более надёжным по ложным срабатываниям.

Касательно разъяснений классификации, для каждого предположительно вредоносного приложения adagio перечисляет методы в графе функциональных вызовов, которые внесли наибольший вклад в функцию потерь (desicion function). Таким образом, пояснение представляет собой список методов приложения, которые программное средство считает подозрительными (предположительно содержащими вредоносную активность), без объяснения, что в действительности является подозрительным в данных методах. Предложенный метод, напротив, даёт достаточно подробные разъяснения, представленные всеми совпавшими цепочками API-вызовов в анализируемом приложении и конкретной вредоносной модели.

Программное средство adagio способно анализировать приложения с гораздо более высокой скоростью по сравнению с реализацией предложенного метода. Время анализа составляет около 1 с для adagio, в то время как для предложенного метода оно составляет 2,3 с в случае ВП и около 23,8 с в случае легитимного приложения, поданного на вход. В то же время предложенный метод тратит меньше времени на генерацию моделей: 1,5 с по сравнению с 4,4 с для adagio в случае ВП и 11,2 с по сравнению с 20,5 с в случае легитимного приложения. Измерения производительности проводились на ПК с процессором Intel(R) Xeon(R) CPU E31225 @ 3.10GHz с использованием только одного ядра процессора.

7. Ограничения метода и возможные улучшения

Подводя итоги, нужно отметить, что предлагаемый метод проводит глубинный статический анализ приложений, давая при этом детальное объяснение результатов. Однако он требует тщательного выбора вредоносных моделей для поддержания высокого уровня обнаружения и приемлемой производительности, поскольку полнота покрытия метода существенно зависит от ВП, использованных для генерации моделей. На самом деле, это общая проблема для почти всех методов обнаружения ВП — они требуют постоянного переобучения/обновления множества моделей. В то же время множество моделей должно быть как можно более компактным, поскольку время, затрачиваемое на анализ, растёт линейно с ростом числа моделей.

Улучшение метода, которое можно предложить для снижения доли ложных срабатываний, — это тщательная фильтрация библиотечных пакетов, статически скомпилированных в приложении. Это может быть достигнуто с использованием фильтра по

хеш-суммам методов, однако такой фильтр должен быть получен на основе анализа большого числа легитимных приложений, чтобы включать в себя хеш-суммы всех известных библиотечных методов (кроме того, должны быть представлены все версии каждой библиотеки).

Возможно улучшение метода за счёт изменения алгоритмов построения цепочек API-вызовов или их сравнения. Например, в предлагаемом методе в графе функциональных вызовов приложения вершины, соответствующие методам классов, уникальны и в построенной цепочке API-вызовов соответствуют единственному участку, даже если вызываются из нескольких других методов. Повторение таких участков в цепочках для каждого из мест вызова существенно увеличит их длину (и время работы анализатора), но, возможно, увеличит и точность метода. Предложенный метод не учитывает взаимодействия между компонентами посредством передачи объектов класса Intent. Их учёт добавит в модели цепочек API-вызовов связи между цепочками, что также может увеличить точность метода. Для обнаружения обфусцированных образцов используемых вредоносных моделей метод может быть дополнен динамическим анализом, учитывающим фактические наблюдаемые при выполнении цепочки API-вызовов.

Заключение

В работе представлен метод анализа мобильных приложений для платформы Android, основанный на используемых в приложении системных привилегиях и API-вызовах. Результаты экспериментов на коллекциях вредоносных приложений показывают, что метод обнаруживает подобные приложения для 90–94 % приложений в них. Тестирование метода на коллекции приложений из Google Play выявило долю ложных срабатываний около 3,2 %. Метод может быть использован как часть многокомпонентного анализа на стороне магазинов мобильных приложений или аналитиками мобильных приложений как вспомогательное средство, предназначенное для сокращения времени, затрачиваемого на ручной анализ.

Анализ использованных коллекций вредоносных приложений показал, что приложения в них редко используют техники шифрования или обфускации для усложнения статического анализа. Это наблюдение противоположно наблюдаемому в семействе платформ «Wintel», и, вероятно, следует ожидать изменения ситуации в ближайшем будущем.

Представлены результаты сравнения предложенного метода с методом, показавшим свою эффективность в задаче обнаружения вредоносных Android-приложений и реализованным в программном средстве adagio. Предлагаемый метод имеет меньшую полноту покрытия при случайном выборе вредоносных приложений для генерации моделей, но показывает значительно лучшие результаты в отношении ошибок 2-го рода.

ЛИТЕРАТУРА

1. *Skovoroda A. and Gamayunov D.* Securing mobile devices: malware mitigation methods // J. Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications. 2015. No. 6(2). P. 78–97.
2. *Egele M., Kruegel C., Kirda E., and Vigna G.* PiOS: Detecting privacy leaks in iOS applications // Proc. 18th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2011. `http://www.eurecom.fr/publication/3282`

3. *Enck W., Gilbert P., Chun B.-G., et al.* TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones // Proc. 9th USENIX Conf. Design and Implementation OSDI'10, Vancouver, BC, Canada. USENIX Association, 2010. P. 1–6.
4. *Shabtai A., Tenenboim-Chekina L., Mimran D., et al.* Mobile malware detection through analysis of deviations in application network behavior // Computers & Security. 2014. V. 43. P. 1–18.
5. *Kim H., Smith J., and Shin K. G.* Detecting energy-greedy anomalies and mobile malware variants // Proc. 6th Intern. Conf. Applications, and Services MobiSys'08, Breckenridge, CO, USA. ACM, 2008. P. 239–252.
6. *Grace M., Zhou Y., Zhang Q., et al.* RiskRanker: scalable and accurate zero-day android malware detection // Proc. 10th Intern. Conf. Applications, and Services MobiSys'12, Low Wood Bay, Lake District, UK. ACM, 2012. P. 281–294.
7. *Feng Y., Anand S., Dillig I., and Aiken A.* Apposcopy: Semantics-based detection of android malware through static analysis // Proc. 22nd ACM SIGSOFT Intern. Symp. FSE 2014, New York, NY, USA. ACM, 2014. P. 576–587.
8. *Tam K., Feizollah A., Anuar N. B., et al.* The evolution of android malware and android analysis techniques // ACM Comput. Surv. 2017. V. 49. Iss. 4. Article No. 76.
9. *Yang L., Ganapathy V., and Iftode L.* Enhancing mobile malware detection with social collaboration // Proc. 2011 IEEE Third Intern. Conf. Social Computing (SocialCom), Boston, MA. IEEE, 2011. P. 572–576.
10. *Liu L., Yan G., Zhang X., and Chen S.* VirusMeter: preventing your cellphone from spies // LNCS. 2009. V. 5758. P. 244–264.
11. *Hoffmann J., Neumann S., and Holz T.* Mobile malware detection based on energy fingerprints — a dead end? // LNCS. 2013. V. 8145. P. 348–368.
12. *Wong M. Y. and Lie D.* Intellidroid: a targeted input generator for the dynamic analysis of android malware // NDSS. 2016. http://dx.doi.org/10.14722/ndss.2016.23118
13. *Zhou Y., Wang Z., Zhou W., and Jiang X.* Hey, you, get off of my market: detecting malicious apps in official and alternative android markets // Proc. 19th Annual NDDS Symp., San Diego, CA, USA. The Internet Society, 2012. https://www.internetsociety.org/sites/default/files/07_5.pdf
14. *Aafer Y., Du W., and Yin H.* DroidAPIMiner: Mining API-level features for robust malware detection in android // Security and Privacy in Communication Networks / eds. T. Zia, A. Zomaya, V. Varadharajan, and M. Mao. Springer International Publishing, 2013. P. 86–103.
15. *Zhang M., Duan Y., Yin H., and Zhao Z.* Semantics-aware android malware classification using weighted contextual API dependency graphs // Proc. ACM SIGSAC Conf. CCS'14, New York, NY, USA. ACM, 2014. P. 1105–1116.
16. *Arp D., Spreitzenbarth M., Huebner M., et al.* Drebin: efficient and explainable detection of android malware in your pocket // Proc. NDSS Symp., San Diego, CA, USA. The Internet Society, 2014. https://www.internetsociety.org/sites/default/files/11_3_1.pdf
17. *Damshenas M., Dehghantanha A., Choo K., and Mahmud R.* M0Droid: an android behavioral-based malware detection model // J. Inform. Privacy Security. 2015. V. 11. Iss. 3. P. 141–157.
18. *Aresu M., Ariu D., Ahmadi M., et al.* Clustering android malware families by http traffic // Proc. MALCON'2015, Fajardo, Puerto Rico, USA, 2015. https://pralab.diee.unica.it/sites/default/files/MALCON_0.pdf
19. *Mariconti E., Onwuzurike L., Andriotis P., et al.* Mamadroid: Detecting android malware by building markov chains of behavioral models. 2016. https://arxiv.org/abs/1612.04433

20. *Gascon H., Yamaguchi F., Arp D., and Rieck K.* Structural detection of android malware using embedded call graphs // Proc. AISec'13, New York, NY, USA. ACM, 2013. P. 45–54.
21. *Yang Z., Yang M., Zhang Y., et al.* AppIntent: analyzing sensitive data transmission in android for privacy leakage detection // Proc. CCS'13, Berlin, Germany. ACM, 2013. P. 1043–1054.
22. *Rosen S., Qian Z., and Mao Z. M.* AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users // Proc. Third ACM Conf. CODASPY '13, San Antonio, Texas, USA. ACM, 2013. P. 221–232.
23. *Portokalidis G., Homburg P., Anagnostakis K., and Bos H.* Paranoid android: versatile protection for smartphones // Proc. 26th Annual Conf. ACSAC'10, Austin, Texas. ACM, 2010. P. 347–356.
24. *Burguera I., Zurutuza U., and Nadjm-Tehrani S.* Crowdroid: behavior-based malware detection system for android // Proc. 1st ACM Workshop SPSM '11, Chicago, Illinois, USA. ACM, 2011. P. 15–26.
25. `http://developer.android.com/intl/ru/guide/topics/security/permissions.html` — Permissions, 2017.
26. `https://github.com/androguard/androguard` — Androguard, 2017.
27. *Au K. W. Y., Zhou Y. F., Huang Z., and Lie D.* PScout: analyzing the android permission specification // Proc. ACM Conf. CCS'12, New York, NY, USA. ACM, 2012. P. 217–228.
28. *Lu L., Li Z., Wu Z., et al.* CHEX: statically vetting android apps for component hijacking vulnerabilities // Proc. ACM Conf. CCS'12, New York, NY, USA. ACM, 2012. P. 229–240.
29. `http://www.unb.ca/research/iscx/dataset/iscx-android-botnet-dataset.html` — Коллекция вредоносных приложений UNB ISCX Android Botnet, 2017.
30. `http://code.google.com/p/droidbox/` — DroidBox: Сэндбоксинг Android-приложений, 2017.
31. `https://github.com/hgascon/adagio` — Adagio, 2017.

REFERENCES

1. *Skovoroda A. and Gamayunov D.* Securing mobile devices: malware mitigation methods. J. Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 2015, no. 6(2), pp. 78–97.
2. *Egele M., Kruegel C., Kirda E., and Vigna G.* PiOS: Detecting privacy leaks in iOS applications. Proc. 18th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2011. `http://www.eurecom.fr/publication/3282`
3. *Enck W., Gilbert P., Chun B.-G., et al.* TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. Proc. 9th USENIX Conf. Design and Implementation OSDI'10, Vancouver, BC, Canada. USENIX Association, 2010, pp. 1–6.
4. *Shabtai A., Tenenboim-Chekina L., Mimran D., et al.* Mobile malware detection through analysis of deviations in application network behavior. Computers & Security, 2014, vol. 43, pp. 1–18.
5. *Kim H., Smith J., and Shin K. G.* Detecting energy-greedy anomalies and mobile malware variants. Proc. 6th Intern. Conf. Applications, and Services MobiSys'08, Breckenridge, CO, USA. ACM, 2008, pp. 239–252.
6. *Grace M., Zhou Y., Zhang Q., et al.* RiskRanker: scalable and accurate zero-day android malware detection. Proc. 10th Intern. Conf. Applications, and Services MobiSys'12, Low Wood Bay, Lake District, UK. ACM, 2012, pp. 281–294.

7. *Feng Y., Anand S., Dillig I., and Aiken A.* Apposcopy: Semantics-based detection of android malware through static analysis. Proc. 22nd ACM SIGSOFT Intern. Symp. FSE 2014, New York, NY, USA. ACM, 2014, pp. 576–587.
8. *Tam K., Feizollah A., Anuar N. B., et al.* The evolution of android malware and android analysis techniques. ACM Comput. Surv., 2017, vol. 49, iss. 4, article no. 76.
9. *Yang L., Ganapathy V., and Iftode L.* Enhancing mobile malware detection with social collaboration. Proc. 2011 IEEE Third Intern. Conf. Social Computing (SocialCom), Boston, MA. IEEE, 2011, pp. 572–576.
10. *Liu L., Yan G., Zhang X., and Chen S.* VirusMeter: preventing your cellphone from spies. LNCS, 2009, vol. 5758, pp. 244–264.
11. *Hoffmann J., Neumann S., and Holz T.* Mobile malware detection based on energy fingerprints — a dead end? LNCS, 2013, vol. 8145, pp. 348–368.
12. *Wong M. Y. and Lie D.* Intellidroid: a targeted input generator for the dynamic analysis of android malware. NDSS, 2016. http://dx.doi.org/10.14722/ndss.2016.23118
13. *Zhou Y., Wang Z., Zhou W., and Jiang X.* Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. Proc. 19th Annual NDDS Symp., San Diego, CA, USA. The Internet Society, 2012. https://www.internetsociety.org/sites/default/files/07_5.pdf
14. *Aafer Y., Du W., and Yin H.* DroidAPIMiner: Mining API-level features for robust malware detection in android. Security and Privacy in Communication Networks / eds. T. Zia, A. Zomaya, V. Varadharajan, and M. Mao. Springer International Publishing, 2013, pp. 86–103.
15. *Zhang M., Duan Y., Yin H., and Zhao Z.* Semantics-aware android malware classification using weighted contextual API dependency graphs. Proc. ACM SIGSAC Conf. CCS'14, New York, NY, USA. ACM, 2014, pp. 1105–1116.
16. *Arp D., Spreitzenbarth M., Huebner M., et al.* Drebin: efficient and explainable detection of android malware in your pocket. Proc. NDSS Symp., San Diego, California, USA. The Internet Society, 2014. https://www.internetsociety.org/sites/default/files/11_3_1.pdf
17. *Damshenas M., Dehghantanha A., Choo K., and Mahmud R.* M0Droid: an android behavioral-based malware detection model. J. Inform. Privacy Security, 2015, vol. 11, iss. 3, pp. 141–157.
18. *Aresu M., Ariu D., Ahmadi M., et al.* Clustering android malware families by http traffic. Proc. MALCON'2015, Fajardo, Puerto Rico, USA, 2015. https://pralab.diee.unica.it/sites/default/files/MALCON_0.pdf
19. *Mariconti E., Onwuzurike L., Andriotis P., et al.* Mamadroid: Detecting android malware by building markov chains of behavioral models. 2016. https://arxiv.org/abs/1612.04433
20. *Gascon H., Yamaguchi F., Arp D., and Rieck K.* Structural detection of android malware using embedded call graphs. Proc. AISec'13, New York, NY, USA. ACM, 2013, pp. 45–54.
21. *Yang Z., Yang M., Zhang Y., et al.* AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. Proc. CCS'13, Berlin, Germany. ACM, 2013, pp. 1043–1054.
22. *Rosen S., Qian Z., and Mao Z. M.* AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users. Proc. Third ACM Conf. CODASPY '13, San Antonio, Texas, USA. ACM, 2013, pp. 221–232.
23. *Portokalidis G., Homburg P., Anagnostakis K., and Bos H.* Paranoid android: versatile protection for smartphones. Proc. 26th Annual Conf. ACSAC'10, Austin, Texas. ACM, 2010, pp. 347–356.

24. *Burguera I., Zurutuza U., and Nadjm-Tehrani S.* Crowdroid: behavior-based malware detection system for android. Proc. 1st ACM Workshop SPSM '11, Chicago, Illinois, USA. ACM, 2011, pp. 15–26.
25. `http://developer.android.com/intl/ru/guide/topics/security/permissions.html` — Permissions, 2017.
26. `https://github.com/androguard/androguard` — Androguard, 2017.
27. *Au K. W. Y., Zhou Y. F., Huang Z., and Lie D.* PScout: analyzing the android permission specification. Proc. ACM Conf. CCS'12, New York, NY, USA. ACM, 2012, pp. 217–228.
28. *Lu L., Li Z., Wu Z., et al.* CHEX: statically vetting android apps for component hijacking vulnerabilities. Proc. ACM Conf. CCS'12, New York, NY, USA. ACM, 2012, pp. 229–240.
29. `http://www.unb.ca/research/iscx/dataset/iscx-android-botnet-dataset.html` — UNB ISCX Android Botnet, 2017.
30. `http://code.google.com/p/droidbox/` — DroidBox, 2017.
31. `https://github.com/hgascon/adagio` — Adagio, 2017.