

Android GPS-Spoofing



Projektbericht

im Rahmen der Veranstaltung
Sicherheit von mobilen Applikationen
an der Hochschule Flensburg

vorgelegt von:

Andrej Ott, Tim Eickelmann, Friedemann Dohse, Yannik Goldgräbe

ausgegeben und geprüft von:

Prof. Dr. rer. nat. Tim Aschmoneit und Aalbrecht Irawan, M. Sc.

Flensburg, 10. Dezember 2018

Inhaltsverzeichnis

1	Einleitung	3
1.1	Motivation	3
1.2	Ziele des Projekts	4
2	Entwicklung	6
2.1	GPS-Manipulation unter Android	6
2.2	GPS-Spoofing	9
2.3	Implementation	10
2.3.1	MainActivity	10
2.3.2	MyLocationListener	11
2.3.3	MockLocationService	11
2.4	Roads API	16
2.5	Arbeiten mit Versionsmanagement	18
3	Diskussion	19

Abbildungsverzeichnis

Abbildung 1: Typisches Angriffsszenario nach Tippenhauer et al. 2014 und Warner & Johnston, 2002	3
Abbildung 2: Strecke, die durch das Abändern von GPS-Signalen resultiert	5
Abbildung 3: Die blaue Linie verdeutlicht die gewünschte Strecke, welche dem Opfer angezeigt werden soll; die graue Linie ist die Datenbasis, auf der die blaue Linie erzeugt wird	6
Abbildung 4: Code-Snippet: Obsolete Methode zum Aktualisieren der Mock-Location	7
Abbildung 5: Code-Snippet: Methode zum Erkennen von MockLocations (Versionssicher)	8
Abbildung 6: Terminal-Befehle zum Senden einer Mock-Location an den Emulator	8
Abbildung 7: Screenshot der Extended Controls vom Pixel 2 Emulator am Beispiel einer Route in Flensburg	9
Abbildung 8: Screenshot der Anwendung	10
Abbildung 9: LocationListener onLocationChanged()	11
Abbildung 10: Einbinden des Service in der Manifest-Datei	11
Abbildung 11: onStartCommand()	12
Abbildung 12: Thread, in dem der Standort alle 500ms neu verfälscht wird	13
Abbildung 13: callRoadsAPI()	14
Abbildung 14: Permission zum Internetaufruf in der Manifest-Datei	14
Abbildung 15: Screenshot der Anwendung und gleichzeitigen Anzeige des verfälschten Standortes in Google Maps. Bredstedt (rot umkreist) ist der echte Standort, während sich der verfälschte Standort auf einer Straße außerhalb befindet	15
Abbildung 16: Update des MockLocation-Providers	16
Abbildung 17: An Straßen andockte Punkte abrufen	17
Abbildung 18: Ablaufdiagramm zum Versetzen der aktuellen Position mit Hilfe der snapToRoads-Funktion	18

1 Einleitung

In diesem Projekt geht es um GPS-Spoofing. Das GPS (Global Positioning System), welches offiziell “NAVSTAR GPS” genannt wird, dient zur Positionsbestimmung und nutzt hierfür ein globales Navigationssatellitensystem. Beim Spoofing (Verschleierung) eines GPS-Signals, werden Störsignale ausgesendet, welche das originale Signal imitieren. Im Gegensatz zu einem GPS-Jammer, welcher das Satellitensignal lediglich stört, werden beim GPS-Spoofing manipulierte GPS-Daten gesendet. GPS-Jammer, welche den Ausfall des GPS-Empfangs provozieren, können vom Opfer leicht identifiziert werden. GPS-Spoofing hingegen ist für das Opfer unauffälliger, da eine abgeänderte Positionserkennung weniger offensichtlich ist.

1.1 Motivation

Sich mit GPS-Spoofing auseinanderzusetzen ist sowohl aus Angreifer-, als auch aus Nutzersicht interessant. Warner & Johnston (2002) beschreiben ein Szenario aus Angreifer-Perspektive. Hierbei hat ein Lastwagen (das Opfer) eine GPS-Einheit, die in einem manipulationssicheren Gehäuse untergebracht ist und kryptografisch authentifizierte Statusaktualisierungen mit einer festen Rate an eine Überwachungszentrale sendet. Der Angreifer verfolgt das Ziel, den Lastwagen zu stehlen. Hierfür will er den Lastwagen an einen abgelegenen Ort steuern, um dann an die geladenen Güter zu kommen. Um nicht aufzufallen, sendet der Angreifer dem Überwachungszentrum gefälschte Signale, sodass diesem die Position des Lastwagens so angezeigt wird, als wäre er an der erwarteten Position. **Abbildung 1** verdeutlicht das beschriebene Szenario. (a) Situationsbeschreibung. (b) Abstrakte Repräsentation der Szene. (c) Der Angreifer sendet eigene verschleierte (spoofed) Signale. (d) Das Opfer synchronisiert sich mit dem Signal des Angreifers.

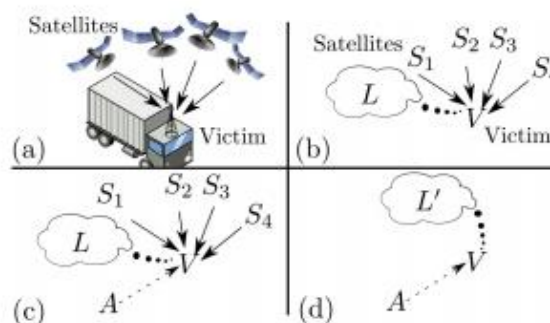


Abbildung 1: Typisches Angriffsszenario nach Tippenhauer et al. 2014 und Warner & Johnston, 2002

Digitaltrends¹ unterstreicht ebenfalls die potentielle Gefahr von GPS-Spoofing aus Angreifer-Sicht in einem Szenario. Forscher einer Universität aus Austin haben einen Cyber-Angriff vorgetäuscht, indem sie gefälschte GPS-Signale eingespeist haben, die Kontrolle über das Navigationssystem einer 80 Millionen Dollar Yacht übernahmen und diese auf einen potentiell gefährlichen Weg führten.

Ein Szenario aus Nutzersicht beschreiben Zhao & Sui (2017). Sie zeigen auf, dass in sozialen Medien vermehrt GPS-Spoofing zum Einsatz kommt. Beispielsweise kann im standortbasierten Spiel “Pokémon Go” der Standort so verschleiert werden, dass der Spieler Vorteile hat (Wehner, 2016). Ziel in diesem Spiel ist es, Fantasiewesen zu fangen und in virtuellen Kämpfen gegeneinander antreten zu lassen. Je mehr unterschiedliche (physische) Orte der Spieler hierbei besucht, desto höher ist die Wahrscheinlichkeit, dass er mehr und bessere Fantasiewesen fängt. Um Orte nicht in Wirklichkeit besuchen zu müssen, wird GPS-Spoofing angewandt, um der App vorzutäuschen, dass der Spieler sich gerade an einer anderen Position befindet.

Sowohl aus Angreifer-, als auch aus Nutzersicht ist GPS-Spoofing also von Bedeutung. Aus beiden Perspektiven ist es sinnvoll, nicht nur die GPS-Signale zu manipulieren, sondern diese so abzuändern, dass das Gerät realistische Wege zurücklegt. Bei dem von Warner & Johnston (2002) beschriebenen Szenario, wäre es beispielsweise für die Überwachungszentrale auffällig, wenn sich der LKW plötzlich nicht mehr auf Straßen bewegt, sondern auf Wasser. Genau an dieser Stelle setzt das in diesem Bericht behandelte Projekt an.

1.2 Ziele des Projekts

Ziel des Projekts ist es, GPS-Spoofing so umzusetzen, dass nicht nur das GPS-Signal um feste Werte abgeändert wird, sondern das GPS-Signal so manipuliert wird, dass sich das Zielgerät versetzt auf Straßen mitbewegt. Verdeutlicht wird dieses Verhalten in **Abbildung 2** und **Abbildung 3**.

¹ <https://www.digitaltrends.com/mobile/gps-spoofing/>

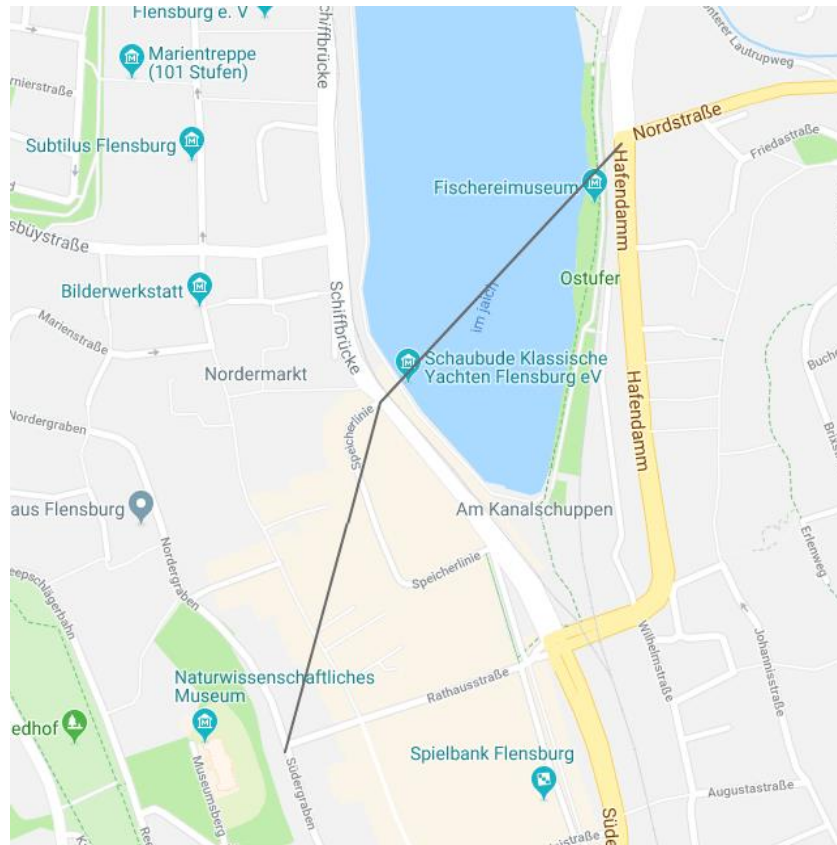


Abbildung 2: Strecke, die durch das Abändern von GPS-Signalen resultiert

In dieser Abbildung ist der Weg gekennzeichnet, welcher durch das Fälschen von GPS-Signalen entstehen kann. Dieser Weg ist für die in **Kapitel 1.2** beschriebenen Szenarien unrealistisch und würde den Opfern auffallen. Aus diesem Grund entwickeln wir in diesem Projekt ein GPS-Spoofing, welches nicht nur die Standorte versetzt, sondern diese zusätzlich auf Straßen positioniert. Der von uns gewünschte Weg bei der vorliegenden Datenbasis aus **Abbildung 2**, ist in **Abbildung 3** ersichtlich.

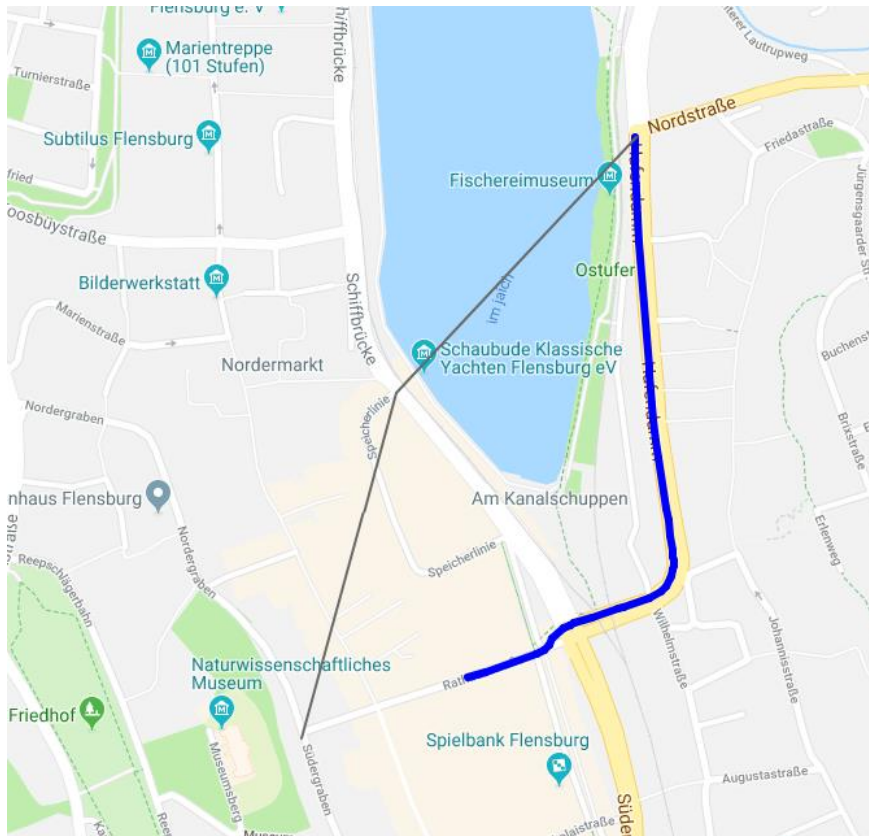


Abbildung 3: Die blaue Linie verdeutlicht die gewünschte Strecke, welche dem Opfer angezeigt werden soll; die graue Linie ist die Datenbasis, auf der die blaue Linie erzeugt wird

Es resultieren die folgenden zwei Ziele: (1) Verschleierung der GPS-Signale um festgesetzte Längen- und Breitengrade. (2) Positionierung der veränderten GPS-Standorte auf Straßen.

2 Entwicklung

Im Folgenden gehen wir auf das von uns entwickelte GPS-Spoofing ein, welches die Ziele aus **Kapitel 1.2** erfüllen soll.

2.1 GPS-Manipulation unter Android

Google hat mit dem Update der Android-Version auf 6.0 (Marshmallow) die Handhabung der Fake-Standorte geändert. Vor dieser Android-OS-Version konnte in den Systemeinstellungen ein Haken gesetzt werden, dass Mock-Standort erlaubt sind. Damit war es mit einem gerooteten System möglich der Applikation Root-Rechte zu gewähren und anschließend im Programmcode die Option “Allow Mock-Locations” einzuschalten, einen falschen Standort an das System zu

senden und anschließend die Option wieder auszuschalten. Da dies nur eine sehr kurze Zeitspanne umfasste, war es anderen Apps kaum möglich festzustellen, dass der Standort verändert wurde. Wie in **Abbildung 4** gezeigt, wird in der Methode `setMockLocationSettings()` die Einstellung `ALLOW MOCK LOCATION` ein- und in der Methode `restoreMockLocationSettings()` wieder ausgeschaltet. Der Teil unter dem Kommentar im unteren Bereich in **Abbildung 4** sollte nun an jeder Stelle des Codes verwendet werden, an dem der Standort manipuliert werden sollte.

```
private int setMockLocationSettings() {
    int value = 1;
    try {
        value = Settings.Secure.getInt(getContentResolver(),
Settings.Secure.ALLOW MOCK_LOCATION);
        Settings.Secure.putInt(getContentResolver(),
Settings.Secure.ALLOW MOCK_LOCATION, 1);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return value;
}

private void restoreMockLocationSettings(int restore_value) {
    try {
        Settings.Secure.putInt(getContentResolver(),
Settings.Secure.ALLOW MOCK_LOCATION, restore_value);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// every time you mock location, you should use these code
int value = setMockLocationSettings(); //toggle ALLOW MOCK_LOCATION on
try {
    mLocationManager.setTestProviderLocation(LocationManager.GPS_PROVIDER,
fake_location);
} catch (SecurityException e) {
    e.printStackTrace();
} finally {
    restoreMockLocationSettings(value); //toggle ALLOW MOCK_LOCATION off
}
```

Abbildung 4: Code-Snippet: Obsolete Methode zum Aktualisieren der Mock-Location²

Seit dem Android-API-Level 23 ist die Einstellung `ALLOW MOCK LOCATIONS` obsolet und wurde durch die Möglichkeit ersetzt, in den Entwickler-Einstellungen eine App als “Pseudo”- oder “Simulierter”-Standort-App zu kennzeichnen. Dieser dort eingetragenen App ist es dann

² <https://stackoverflow.com/questions/17412826/use-mock-location-without-setting-it-in-settings> [Stand: 03.12.2018]

möglich, dem System Pseudo-Standorte zu senden, die andere Apps (bspw. Google Maps) dann anstelle des realen Standorts interpretieren.

Dies ist jedoch sehr simpel zu überprüfen, da Google selbst eine Methode in der Location-Klasse³ bereitstellt, die überprüft, ob der Standort von einem Mock-Provider stammt.

```
public static boolean isLocationFromMockProvider(Context context, Location
location) {
    if (android.os.Build.VERSION.SDK_INT >= 18) {
        return location.isFromMockProvider();
    } else {
        if (Settings.Secure.getString(context.getContentResolver(),
            Settings.Secure.ALLOW MOCK_LOCATION).equals("0")) {
            return false;
        } else {
            return true;
        }
    }
}
```

Abbildung 5: Code-Snippet: Methode zum Erkennen von MockLocations (Versionssicher)⁴

Mit dieser Methode ist es, wie in **Abbildung 5** gezeigt, sehr einfach zu erkennen, ob es sich bei dem Standort um einen Mock-Standort handelt. Die Methode *isLocationFromMockProvider()* (siehe **Abbildung 5**) überprüft nicht nur anhand der von Google gestellten Methode *Location.isFromMockProvider()*, sondern überprüft außerdem die oben beschriebene Möglichkeit auf Geräten mit einer älteren OS-Version von Android.

Eine weitere Methode, unter Android Mock-Locations and das System zu senden, ist die Nutzung von Android Studio bzw. dem CLI-Tool *telnet* und einem Emulator mit Android. Wie Google selbst beschreibt muss der Emulator lediglich per *telnet* verbunden werden (siehe **Abbildung 6**) und anschließend kann per *geo fix* (siehe **Abbildung 6**) ein Standort an das Android-System geschickt werden.

```
telnet localhost <console-port>
geo fix -121.45356 46.51119 4392
```

Abbildung 6: Terminal-Befehle zum Senden einer Mock-Location an den Emulator

Alternativ kann, wie oben erwähnt, der Android Virtual Device Manager von Android Studio genutzt werden, um Standorte an den Emulator zu senden. Dafür gibt es in den Extended

³ [https://developer.android.com/reference/android/location/Location#isFromMockProvider\(\)](https://developer.android.com/reference/android/location/Location#isFromMockProvider()) [Stand: 03.12.2018]

⁴ <https://github.com/smarques84/MockLocationDetector/blob/master/library/src/main/java/com/inforoeste/mocklocationdetector/MockLocationDetector.java> [Stand: 03.12.2018]

Controls den Location Bereich. Darüber lassen sich definierte Standorte oder ganze Pfade in Form von einer .gpx-Datei an den Emulator übersenden.

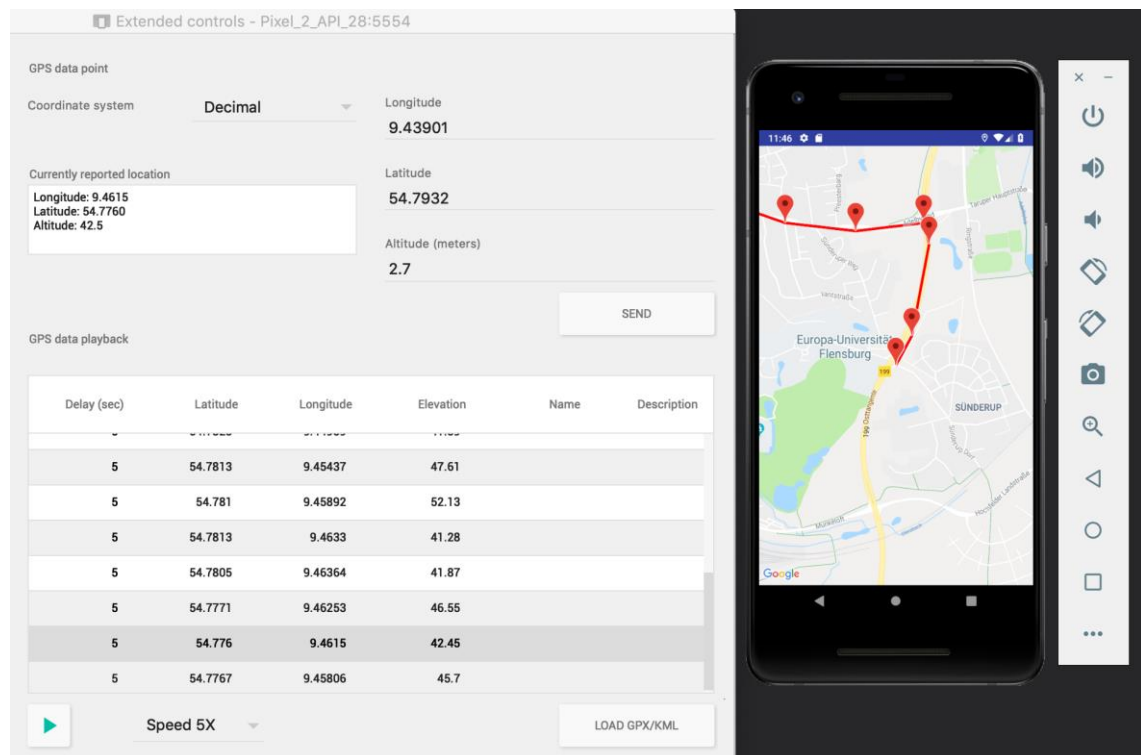


Abbildung 7: Screenshot der Extended Controls vom Pixel 2 Emulator am Beispiel einer Route in Flensburg

2.2 GPS-Spoofing

Beim vorliegenden Projekt wurde sich dafür entschieden, die oben beschriebene Variante des Mock-Location-Providers auf einem Gerät mit einem höheren API-Level als Marshmallow umzusetzen. Für die Anzeige des Standortes auf dem Gerät wird die bereits vorinstallierte App Google Maps verwendet, die den Standort aus dem System ausliest, welcher durch die erstellte App zum GPS-Spoofing verändert wurde. Die eigentliche App muss daher als ein Background-Service ebenfalls die Berechtigung erteilt bekommen, den realen Standort abzurufen (*ACCESS_COARSE_LOCATION* oder *ACCESS_FINE_LOCATION*), sowie den Standort verändern zu dürfen (*ACCESS_mock_location*).

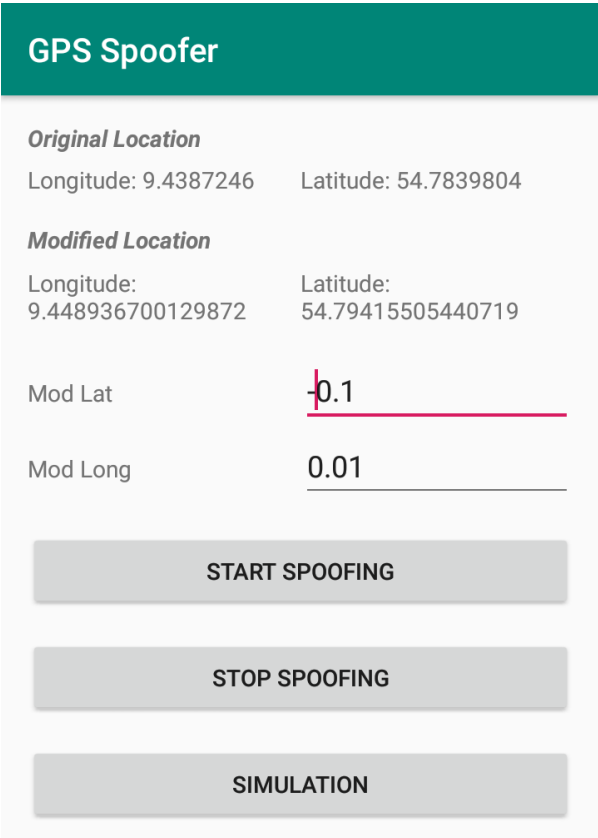
Damit die App ordnungsgemäß funktioniert, muss der App über die Einstellungen die erwähnte Berechtigung gewährt werden und zusätzlich muss sie in den Entwickler-Einstellungen als "Pseudo/Simulierter"-Standort-App eingetragen werden. Außerdem muss der GPS-Sensor des Gerätes aktiviert sein, wobei der Modus frei wählbar ist. Empfohlen ist jedoch eine Kombination aus GPS und Netzwerk, da die Ortung in diesem Fall robuster funktioniert.

2.3 Implementation

Die erstellte App besteht hauptsächlich aus drei Komponenten: *MainActivity* (View und Hintergrundklasse), *MyLocationListener* und *MockLocationService*. Diese werden im Folgenden erläutert, indem die wichtigsten Funktionen vorgestellt werden.

2.3.1 MainActivity

Die *MainActivity* stellt wie bei den meisten Apps den Startpunkt der vorliegenden App dar. Das User-Interface der App ist sehr schlicht gehalten und umfasst lediglich einen Button zum Starten und zum Beenden der Standort-Manipulation, zwei Textfeldern, die die manipulierten Längen- und Breitengrade anzeigen, einem Button zum Starten einer Simulation der App, ohne sich bewegen zu müssen und jeweils ein Textfeld zur Eingabe des gewünschten Offsets vom aktuellen zum manipulierten Standort (Längen- und Breitengrad) (siehe **Abbildung 8**). In der Code-Behind-Datei wird im jeweiligen onClick-Listener zum Starten bzw. Beenden der Manipulation die Klasse *MockLocationService* als Hintergrund-Service gestartet.



The screenshot shows the 'GPS Spoofer' application interface. It features a teal header with the title 'GPS Spoofer'. Below the header, there are two sections: 'Original Location' and 'Modified Location'. The 'Original Location' section displays 'Longitude: 9.4387246' and 'Latitude: 54.7839804'. The 'Modified Location' section displays 'Longitude: 9.448936700129872' and 'Latitude: 54.79415505440719'. Below these sections, there are two input fields: 'Mod Lat' with a value of '-0.1' and 'Mod Long' with a value of '0.01'. At the bottom of the interface, there are three buttons: 'START SPOOFING', 'STOP SPOOFING', and 'SIMULATION'.

Original Location	
Longitude: 9.4387246	Latitude: 54.7839804

Modified Location	
Longitude: 9.448936700129872	Latitude: 54.79415505440719

Mod Lat	-0.1
Mod Long	0.01

START SPOOFING

STOP SPOOFING

SIMULATION

Abbildung 8: Screenshot der Anwendung

2.3.2 MyLocationListener

Die Klasse *MyLocationListener* ist für das Abrufen des nicht manipulierten Standortes zuständig. Bei der Initialisierung der Klasse muss zwingend ein Standort-Provider und ein Kontext übergeben werden. Durch den Parameter des Providers kann die Klasse entweder für das Abrufen der Standort-Koordinaten über den GPS- oder über den Netzwerk-Provider verwendet werden. Des Weiteren implementiert die Klasse das Interface *LocationListener* von Google, um den Standort abrufen zu können. Jedes Mal, wenn das Gerät eine Änderung des Standortes erkennt, wird die Methode *onLocationChanged()* aufgerufen.

```
@Override
public void onLocationChanged(Location location) {
    [...]
    mLastLocation.set(location);

    Intent intent = new Intent("UpdateLocation");
    intent.putExtra("Latitude", mLastLocation.getLatitude());
    intent.putExtra("Longitude", mLastLocation.getLongitude());
    LocalBroadcastManager.getInstance(context).sendBroadcast(intent);
}
```

Abbildung 9: LocationListener onLocationChanged()

In der Methode *onLocationChanged()* wird der aktuelle Standort in der Klassenvariable *mLastLocation* für den späteren Zugriff und Verwendung gespeichert. Im Anschluss sendet die Klasse *MyLocationListener* einen Intent an die *MainActivity*, welcher den Längen- und Breitengrad beinhaltet, welche diese wiederum im User-Interface anzeigt.

2.3.3 MockLocationService

Damit die Standort-Manipulation auch funktioniert, wenn die App minimiert ist, wird die Klasse *MockLocationService*, welche für die Manipulation zuständig ist, als Hintergrund-Service eingetragen (siehe **Abbildung 10**) und implementiert das Interface *Service*.

```
<application>
    [...]
    <service android:name=".MockLocationService" android:exported="false"/>
</application>
```

Abbildung 10: Einbinden des Service in der Manifest-Datei

Durch die Implementierung des Interfaces *Service* muss die Klasse *MockLocationService* eine Methodendefinition *onStartCommand()* implementieren (siehe **Abbildung 11**). Für Services,

die einen expliziten Start sowie ein explizites Ende haben, sollte die Konstante *START_STICKY* übergeben werden⁵.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);

    diffLatitude = intent.getDoubleExtra("diffLatitude", 0.01d);
    diffLongitude = intent.getDoubleExtra("diffLongitude", 0.01d);

    String mode = intent.getStringExtra("mode");

    mLocationManager = (LocationManager)
    getApplicationContext().getSystemService(Context.LOCATION_SERVICE);

    switch (mode) {
        case "start":
            mLocationListeners =
                new MyLocationListener[] {
                    new MyLocationListener(LocationManager.GPS_PROVIDER,
                        getApplicationContext()),
                    new MyLocationListener(LocationManager.NETWORK_PROVIDER,
                        getApplicationContext())
                };

            [...] // Initialisierung der LocationListeners

            handler.post(mockRunnable);
            break;
        case "simulate":
            try {
                simulationList = loadGpxData(XmlPullParser(),
                    getResources().openRawResource(R.raw.gpx_data));
            } catch (XmlPullParserException | IOException e) {
                e.printStackTrace();
            }
            simulationListIndex = 0;
            handler.post(simulateRunnable);
            break;
    }

    return START_STICKY;
}
```

Abbildung 11: onStartCommand()

Durch den Aufruf als Service, wird die Methode *onCreate()*, sowie die Methode *onStartCommand()* ausgeführt, die alle nötigen Variablen initialisieren. Anschließend wird in der *onStartCommand()* ein Thread durch ein Handler gestartet, welcher diesen immer wieder

⁵ <https://developer.android.com/reference/android/app/Service#service-lifecycle> [Stand 04.12.2018]

neu mit einer Verzögerung von 500ms startet. Abhängig von der Art des Aufrufs wird für den Simulationsmodus bzw. den regulären Verfälschungsmodus ein anderer Thread gestartet. Während der reguläre Modus den realen Standort verfälscht, spielt der Simulationsmodus eine GPX-Datei mit etwas mehr als 100 Standorten ab.

```
private Runnable runnable = new Runnable() {
    @Override
    public void run() {
        Log.d(TAG, "Thread wird ausgeführt!");

        Location actLoc = null;

        for (MyLocationListener locList : mLocationListeners) {
            if (locList.mLastLocation != null) {
                actLoc = locList.mLastLocation;
            }
        }

        double newLat = actLoc.getLatitude() + diffLatitude;
        double newLng = actLoc.getLongitude() + diffLongitude;

        page[1] = new LatLng(newLat, newLng);

        if(firstRun) {
            firstRun = false;
            page[0] = new LatLng(newLat, newLng);
        }else{
            callRoadsAPI();
            loop = 0;
            if (newLoc != null) {

                LatLng newLL = newLoc.location;
                double SnappedLat = newLL.lat;
                double SnappedLng = newLL.lng;

                diffLatitude = SnappedLat - actLoc.getLatitude();
                diffLongitude = SnappedLng - actLoc.getLongitude();

                setMockLocation(SnappedLat, SnappedLng, 10);
                page[0] = new LatLng(SnappedLat, SnappedLng);
            }
        }

        handler.postDelayed(runnable, 500);
    }
};
```

Abbildung 12: Thread, in dem der Standort alle 500ms neu verfälscht wird

Für das Andocken der modifizierten Koordinaten an Straßen wird in diesem Projekt die RoadsAPI von Google genutzt.

```
private void callRoadsAPI() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                // snap
                try {
                    mContext = new
GeoApiContext().setApiKey(getString(R.string.google_maps_web_services_key))
;
                    SnappedPoint[] points =
RoadsApi.snapToRoads(mContext, false, page).await();

                    if (points != null){
                        newLoc = points[points.length-1];
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }).start();
}
```

Abbildung 13: callRoadsAPI()

Der Aufruf der RoadsAPI muss in einem eigenen Thread laufen. Die Funktion ruft einen Service bei Google auf. Die Internetberechtigung muss ebenfalls gegeben sein.

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Abbildung 14: Permission zum Internetaufruf in der Manifest-Datei

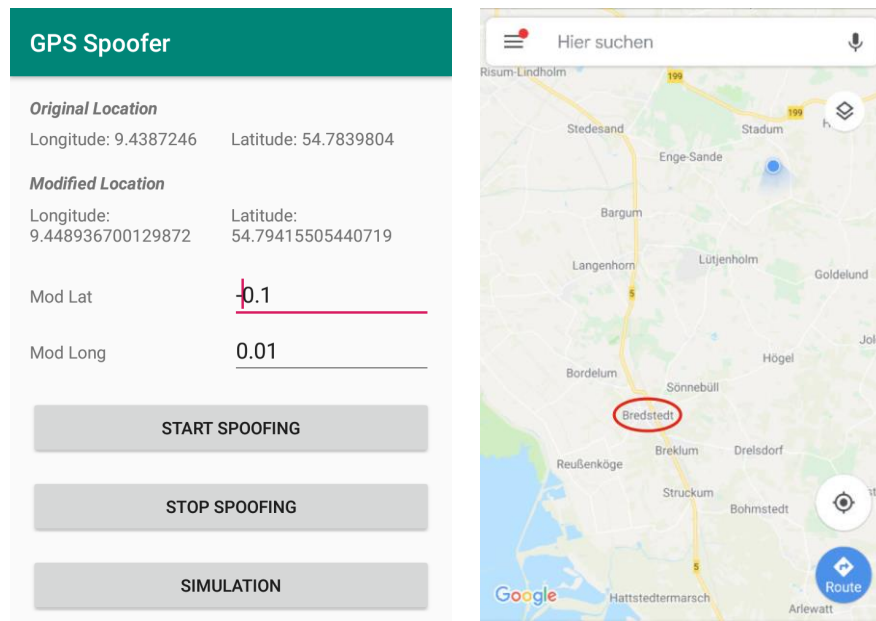


Abbildung 15: Screenshot der Anwendung und gleichzeitigen Anzeige des verfälschten Standortes in Google Maps. Bredstedt (rot umkreist) ist der echte Standort, während sich der verfälschte Standort auf einer Straße außerhalb befindet

Die eigentliche Anpassung des Standortes geschieht in der Methode *setMockLocation()* (siehe **Abbildung 16**), die als Parameter einen Gleitkomma-Zahlen-Wert für jeweils den Breiten- (Latitude) und Längengrad (Longitude), sowie die Genauigkeit der Koordinate entgegennimmt. Damit dem System ein modifizierter Standort übergeben werden kann, muss zunächst dem *LocationManager* ein neuer *TestProvider* hinzugefügt werden. Im Anschluss wird ein neues Objekt der Klasse *Location* erzeugt, welches die Parameter für die Initialisierung übergeben bekommt.


```

private void setMockLocation(double latitude, double longitude, float accuracy) {
    locationManager.addTestProvider(LocationManager.GPS_PROVIDER,
        "requiresNetwork" == "",
        "requiresSatellite" == "",
        "requiresCell" == "",
        "hasMonetaryCost" == "",
        "supportsAltitude" == "",
        "supportsSpeed" == "",
        "supportsBearing" == "",
        android.location.Criteria.POWER_LOW,
        android.location.Criteria.ACCURACY_FINE);

    Location newLocation = new Location(LocationManager.GPS_PROVIDER);

    newLocation.setLatitude(latitude);
    newLocation.setLongitude(longitude);
    newLocation.setAccuracy(accuracy);
    newLocation.setTime(Calendar.getInstance().getTime().getTime());
    newLocation.setElapsedRealtimeNanos(SystemClock.elapsedRealtimeNanos());

    locationManager.setTestProviderEnabled(LocationManager.GPS_PROVIDER, true);

    locationManager.setTestProviderStatus(LocationManager.GPS_PROVIDER,
        LocationProvider.AVAILABLE,
        null, System.currentTimeMillis());

    locationManager.setTestProviderLocation(LocationManager.GPS_PROVIDER,
        newLocation);
}

```

Abbildung 16: Update des MockLocation-Providers⁶

Neben dem Längen-, Breitengrad und der Genauigkeit, fordert die Google-API außerdem noch zwei Zeitstempel an, die einmal vom 01.01.1970 (setTime()) und vom Zeitpunkt des letzten Boot-Vorgangs aus gemessen werden.

Als letztes wird dem TestProvider im LocationManager der neue Standort übergeben, wodurch der manipulierte Wert vom System entgegengenommen wird und andere Apps darauf zugreifen können.

2.4 Roads API

Eine Anfrage an die Roads-API von Google akzeptiert bis zu 100 GPS-Punkte, die entlang einer Route erfasst werden. Die API gibt einen ähnlichen Datensatz zurück. Der Unterschied besteht darin, dass die GPS-Punkte an die wahrscheinlichsten bzw. naheliegendsten Straßen in Bezug auf die vorgegebenen GPS-Punkte angepasst werden. Wie bereits in **Kapitel 1.2** beschrieben,

⁶ <https://stackoverflow.com/questions/38251741/how-to-set-android-mock-gps-location>
04.12.2018]

[Stand

wollen wir uns diese Funktionalität zu Nutzen machen, um unseren verfälschten Standort so realistisch wie möglich wirken zu lassen. Ansonsten würden Routen entstehen, die auf dem Land in der realen Welt gar nicht bewältigt werden könnten. Aus diesem Grund übersenden wir jeden neuen versetzten Standort als Abfrage an die Google-Roads-API um einen Standort zurück zu erhalten, welcher auf der naheliegendste Straße positioniert ist. Die Roads-API bietet noch viele weitere Funktionen (wie beispielsweise die Abfrage der Geschwindigkeitsbegrenzungen der einzelnen Straßen), jedoch ist die für uns hauptsächlich relevante Funktionalität die sogenannte *Snap to Roads*⁷. Für die Implementierung musste zunächst ein API-Key in der Google Developer Konsole generiert werden. Dafür ist ein Google Account und das hinterlegen einer Zahlmethode notwendig, denn die Abfragen sind kostenpflichtig. Bei der ersten Hinterlegung wird einem jedoch etwas Guthaben kostenlos zur Verfügung gestellt, wodurch unsere Entwicklung für dieses Projekt nicht eingeschränkt wurde. Für einen leichteren Einstieg mit der API in Android, stellt Google Maps außerdem ein Beispielprojekt bei GitHub zur Verfügung⁸. Dies diente als hilfreiche Vorlage und hat uns bei der Gestaltung der Abfragen geholfen. Der hauptsächliche Aufruf sieht jedoch meistens folgendermaßen aus (siehe **Abbildung 17**).

```
SnappedPoint[] points = RoadsApi.snapToRoads(mContext, false,
page).await();
```

Abbildung 17: An Straßen angedockte Punkte abrufen

Die Funktion `.snapToRoads()` erwartet drei verschiedenen Parameter:

1. den aktuellen Kontext der App,
2. einen Boolean, welcher definiert, ob über den Pfad interpoliert werden soll (dadurch würden zusätzliche Punkte auf dem Pfad generiert werden, um ihn möglichst glatt aussehen zu lassen),
3. dem Pfad als Array bestehend aus mindestens einem Start- und Endpunkt.

In unserem Fall senden wir also bei einer Änderung im Standort den zuletzt bekannten und den aktuellen Standort als Pfad an die API. Die als Antwort erhaltenen gesnappten Punkte werden anschließend als unser neuer gespoofter Standort in das Android System übergeben.

⁷ <https://developers.google.com/maps/documentation/roads/snap> [Stand 06.12.2018]

⁸ <https://github.com/googlemaps/roads-api-samples> [Stand 06.12.2018]

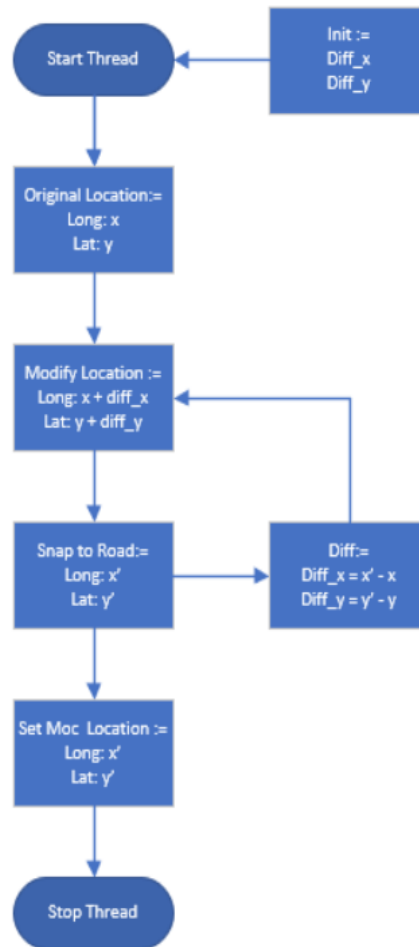


Abbildung 18: Ablaufdiagramm zum Versetzen der aktuellen Position mit Hilfe der `snapToRoads`-Funktion

Ausgehend von der originalen Position wird anfangs ein Differenzwert festgelegt, der die aktuelle Position ändert. Dann wird in einer Schleife die `snapToRoads` Funktion aufgerufen, um die Position auf einer nahegelegenen Straße zu bekommen. Diese Position wird mittels `mobile location` gesetzt und der Differenzwert wird neu ermittelt.

2.5 Arbeiten mit Versionsmanagement

Während unserer Entwicklungsarbeit sind mehrere Android-Projekte entstanden, die wir mit Git auf *Bitbucket* in einem Repository verwaltet haben. Dort befindet sich eine Anfangsversion, bei der die Standortsimulation noch über den Simulator entwickelt wurde (siehe **Abbildung 7**). Zusätzlich gibt es ein Beispielprojekt von der Roads-API. Durch das Zusammenbringen beider Projekte ist wiederum ein neues Android Projekt entstanden, welches dann noch um die obigen Funktionen erweitert und schlussendlich zu unserem finalen Projekt wurde. Da wir das Repository aufgrund von den einbezogenen API-Keys nicht veröffentlichen können, verschicken wir die Einladung gerne per E-Mail.

3 Diskussion

Die in **Kapitel 2** beschriebene Implementation erfüllt die in **Kapitel 1.2** beschriebenen Ziele, allerdings bringt die Verwendung der Roads API Nachteile mit sich. Da die Roads API ein Dienst von Google ist, besteht eine Abhängigkeit. Google kann die Algorithmen der Roads API jederzeit anpassen und somit bedeutenden Einfluss auf unsere Anwendung nehmen. Des Weiteren besteht die Möglichkeit, dass Google den Dienst für Anwendungen mit Absichten, wie die in diesem Projekt, verbietet/sperrt. Ein weiterer Nachteil ist, dass die Verwendung des *Snap to Roads* Dienstes kostenpflichtig ist. Aus diesen Gründen wäre eine Implementation über die Weltkarte "OpenStreetMap" sinnvoll. In diesem Projekt werden frei nutzbare Geodaten gesammelt, strukturiert und in einer Datenbank zur Verfügung gestellt. "OpenStreetMap" beinhaltet Informationen über Straßenverläufe, Bahnstrecken, Flüsse, Wälder, Häuser und vieles mehr. Mithilfe der Straßenverlauf-Daten könnte ein Algorithmus entwickelt werden, welcher als Eingabe den Längen- und Breitengrad des aktuellen Standorts übergeben bekommt. Zurückgeben müsste der Algorithmus dann Standortdaten, welche auf den naheliegendsten Straßen in Bezug auf den aktuellen Standort platziert werden.

Quellen

Literatur

- Tippenhauer, N. O., Pöpper, C., Rasmussen, K., B., Capkun, S. (2014). On the Requirements for Successful GPS Spoofing Attacks. ETH Zürich.
- Warner, J. S., Johnston, R. G. (2002). A simple demonstration that the global positioning system (GPS) is vulnerable to spoofing. Journal of Security Administration.
- Wehner, M. (2016). "How to Cheat at Pokémon Go and Catch Any Pokémon You Want without Leaving Your Couch." Accessed August 2 2016.
<http://www.dailydot.com/debug/how-to-cheat-pokemon-go-gps/>.
- Zhao, B., & Sui, D. Z. (2017). True lies in geospatial big data: detecting location spoofing in social media. Annals of GIS, 23(1), 1-14.