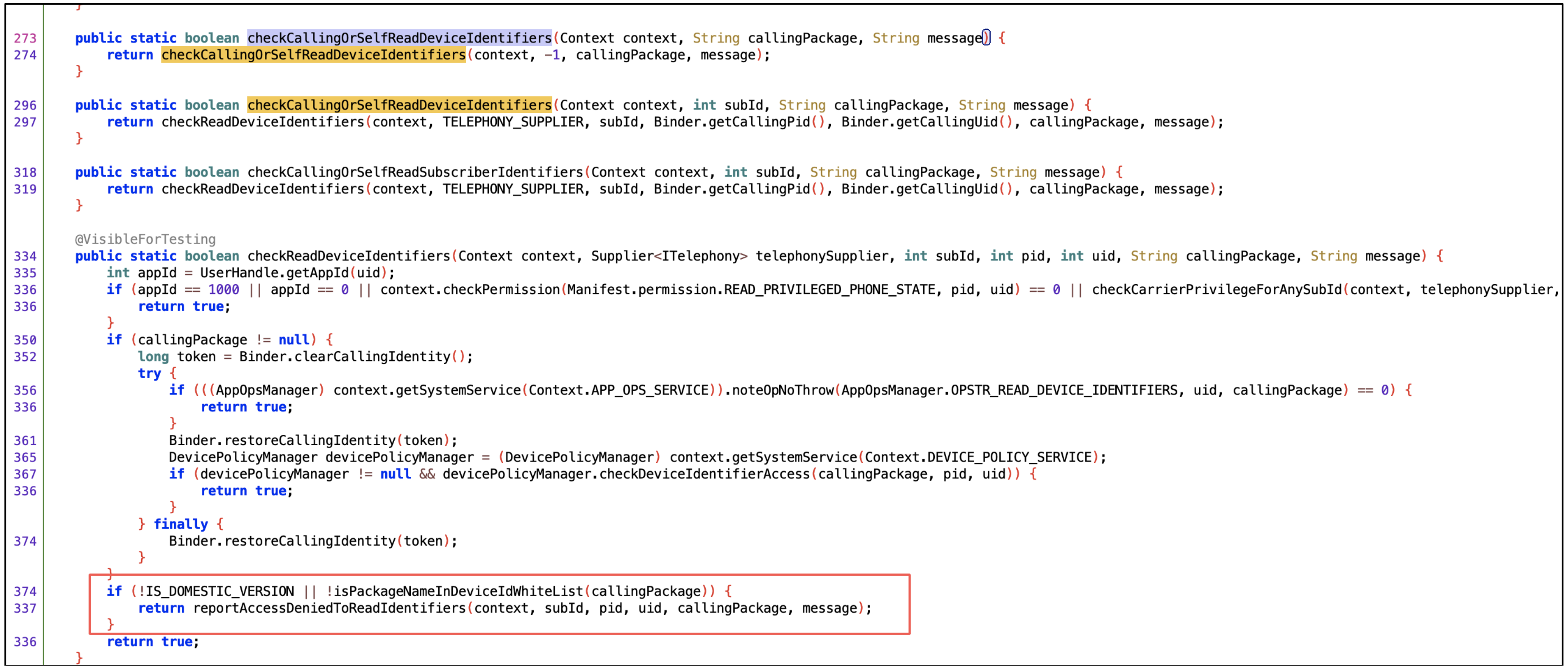
HUAWEI Whitelist Technical Details

Prepared by Mark, 28 July 2021

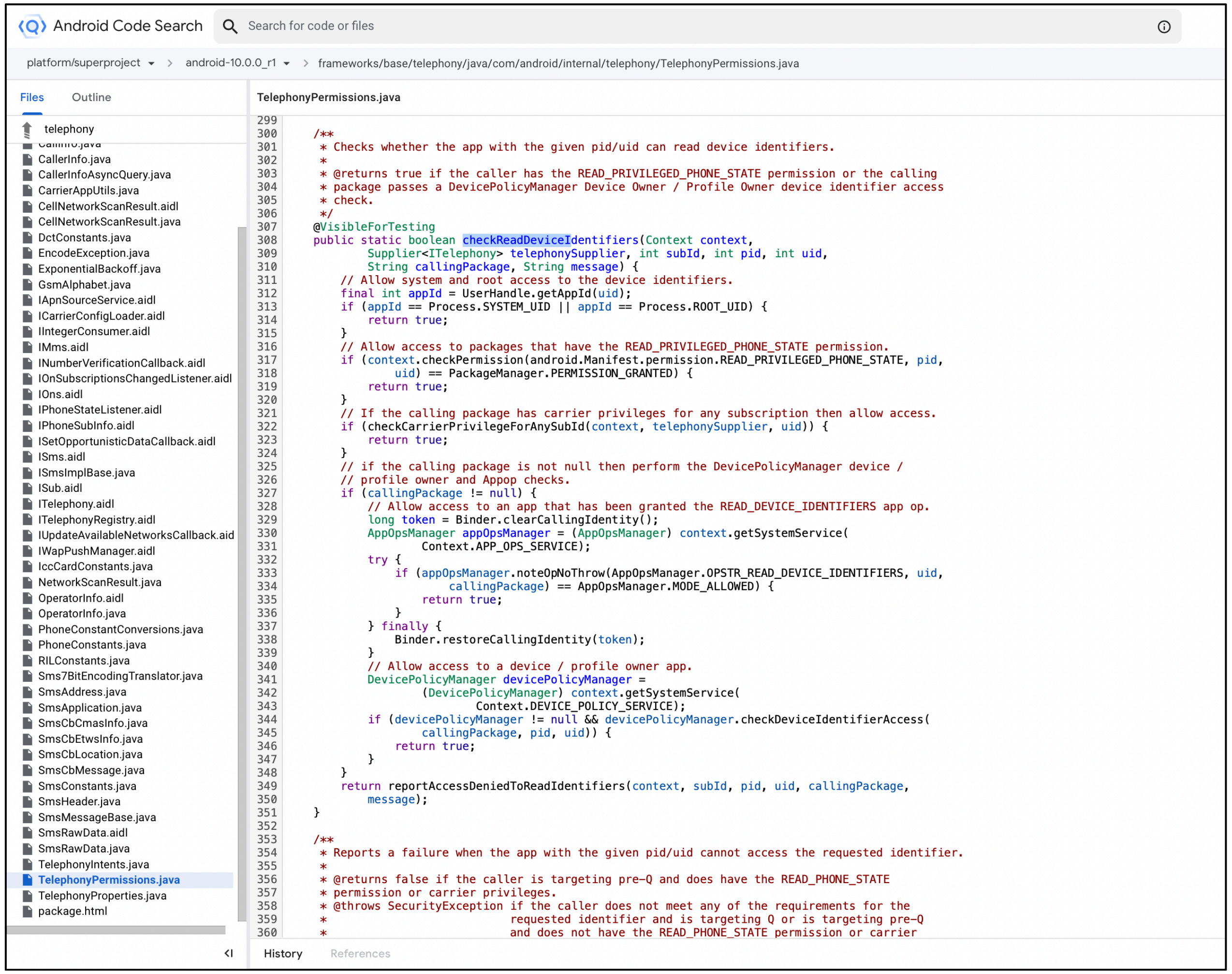
Source code of Huawei's OS (Version 10)



```java
public static boolean checkCallingOrSelfReadDeviceIdentifiers(Context context, String callingPackage, String message) {
    return checkCallingOrSelfReadDeviceIdentifiers(context, -1, callingPackage, message);
}

public static boolean checkCallingOrSelfReadDeviceIdentifiers(Context context, int subId, String callingPackage, String message) {
    return checkReadDeviceIdentifiers(context, TELEPHONY_SUPPLIER, subId, Binder.getCallingPid(), Binder.getCallingUid(), callingPackage, message);
}

public static boolean checkCallingOrSelfReadSubscriberIdentifiers(Context context, int subId, String callingPackage, String message) {
    return checkReadDeviceIdentifiers(context, TELEPHONY_SUPPLIER, subId, Binder.getCallingPid(), Binder.getCallingUid(), callingPackage, message);
}

@VisibleForTesting
public static boolean checkReadDeviceIdentifiers(Context context, Supplier<ITelephony> telephonySupplier, int subId, int pid, int uid, String callingPackage, String message) {
    int appId = UserHandle.getAppId(uid);
    if (appId == 1000 || appId == 0 || context.checkPermission(Manifest.permission.READ_PRIVILEGED_PHONE_STATE, pid, uid) == 0 || checkCarrierPrivilegeForAnySubId(context, telephonySupplier,
        return true;
    }
    if (callingPackage != null) {
        long token = Binder.clearCallingIdentity();
        try {
            if (((AppOpsManager) context.getSystemService(Context.APP_OPS_SERVICE)).noteOpNoThrow(AppOpsManager.OPSTR_READ_DEVICE_IDENTIFIERS, uid, callingPackage) == 0) {
                return true;
            }
            Binder.restoreCallingIdentity(token);
            DevicePolicyManager devicePolicyManager = (DevicePolicyManager) context.getSystemService(Context.DEVICE_POLICY_SERVICE);
            if (devicePolicyManager != null && devicePolicyManager.checkDeviceIdentifierAccess(callingPackage, pid, uid)) {
                return true;
            }
        } finally {
            Binder.restoreCallingIdentity(token);
        }
    }
    if (!IS_DOMESTIC_VERSION || !isPackageNameInDeviceIdWhiteList(callingPackage)) {
        return reportAccessDeniedToReadIdentifiers(context, subId, pid, uid, callingPackage, message);
    }
    return true;
}
```

HUAWEI Whitelist Technical Details

Source code of AOSP 10.0.0_r1
https://cs.android.com/android/platform/superproject/+/android-10.0.0_r1:frameworks/base/telephony/java/com/android/internal/telephony/TelephonyPermissions.java



```java

    /**
     * Checks whether the app with the given pid/uid can read device identifiers.
     *
     * @returns true if the caller has the READ_PRIVILEGED_PHONE_STATE permission or the calling
     * package passes a DevicePolicyManager Device Owner / Profile Owner device identifier access
     * check.
     */
    @VisibleForTesting
    public static boolean checkReadDeviceIdentifiers(Context context,
            Supplier<ITelephony> telephonySupplier, int subId, int pid, int uid,
            String callingPackage, String message) {
        // Allow system and root access to the device identifiers.
        final int appId = UserHandle.getAppId(uid);
        if (appId == Process.SYSTEM_UID || appId == Process.ROOT_UID) {
            return true;
        }
        // Allow access to packages that have the READ_PRIVILEGED_PHONE_STATE permission.
        if (context.checkPermission(android.Manifest.permission.READ_PRIVILEGED_PHONE_STATE, pid,
                uid) == PackageManager.PERMISSION_GRANTED) {
            return true;
        }
        // If the calling package has carrier privileges for any subscription then allow access.
        if (checkCarrierPrivilegeForAnySubId(context, telephonySupplier, uid)) {
            return true;
        }
        // if the calling package is not null then perform the DevicePolicyManager device /
        // profile owner and Appop checks.
        if (callingPackage != null) {
            // Allow access to an app that has been granted the READ_DEVICE_IDENTIFIERS app op.
            long token = Binder.clearCallingIdentity();
            AppOpsManager appOpsManager = (AppOpsManager) context.getSystemService(
                    Context.APP_OPS_SERVICE);
            try {
                if (appOpsManager.noteOpNoThrow(AppOpsManager.OPSTR_READ_DEVICE_IDENTIFIERS, uid,
                        callingPackage) == AppOpsManager.MODE_ALLOWED) {
                    return true;
                }
            } finally {
                Binder.restoreCallingIdentity(token);
            }
            // Allow access to a device / profile owner app.
            DevicePolicyManager devicePolicyManager =
                    (DevicePolicyManager) context.getSystemService(
                            Context.DEVICE_POLICY_SERVICE);
            if (devicePolicyManager != null && devicePolicyManager.checkDeviceIdentifierAccess(
                    callingPackage, pid, uid)) {
                return true;
            }
        }
        return reportAccessDeniedToReadIdentifiers(context, subId, pid, uid, callingPackage,
            message);
    }

    /**
     * Reports a failure when the app with the given pid/uid cannot access the requested identifier.
     *
     * @returns false if the caller is targeting pre-Q and does have the READ_PHONE_STATE
     * permission or carrier privileges.
     * @throws SecurityException if the caller does not meet any of the requirements for the
     *                           requested identifier and is targeting Q or is targeting pre-Q
     *                           and does not have the READ_PHONE_STATE permission or carrier
```