Back to the Binary: Revisiting Similarities of Android Apps

AUTHOR: TU Wien
Jakob Bleier & Martina Lindorfer|
{jakob | martina } [at] seclab.wien

Table of Contents

0 - A Bit of Context

The sky above the port was the color of a mobile phone, stuck in a boot loop.

It's been 13 years since "Similarities for Fun and Profit [0]," aka Elsim, was published in April 2012. Python 2.7 was still supported, Android 4.1 wasn't released yet, and it would take another 2 years until a libstagefright bug would get its very own shiny logo. Heck, even Google Code was still around!

Things changed, and herein lies our challenge: Code-based Android app similarity is not working (in 2025). The original Elsim code presented by Pouik and G0rfi3ld has seen major changes and improvements over the years, but unfortunately the updated version vanished (worry not, we saved a copy [1], tho we're still tracking down some dependencies). Alternative approaches were developed, but even if they have public descriptions and published code, they lack the generality of Elsim.

Similarities between apps are still interesting: Detecting malware is one use case. Finding suspicious copies of apps is another. A fast similarity tool can help you find cloned apps and get you started on finding license violations, impostors, or trojans. With similarities there's also the possibility of clustering apps to find "families". Allegedly the biggest app store for Android uses machine learning to detect outliers [2]: Apps that should be similar are compared, and outliers investigated. For example, if most apps providing control over the Flashlight don't request access to your contact list, the few that do are worth taking a closer look at.

But what is similarity anyway? Android apps can be easily downloaded [3], unpacked, changed, and repackaged. The presentation of an app, its icon and styling, are easy to change. So instead of relying on these properties, we want to use the code to reason whether apps are similar, ideally based on what they do, but this leads us very close to unsolvable problems. Imagine we want to be exact and see how many properties are shared between two apps: We'd need to define those properties and check if an app has them. For declared permissions, this is trivial. An app should not be able to access contact information without declaring the permission. If it actually has a particular property, it is, generally speaking, undecidable: Imagine we replace the call to get contact information with an infinite loop.

Voila, we are trying to solve the Halting problem.

So, instead of requiring similarity to represent what exactly apps do, we want to have an approximation of whether their code is similar. And while Android changed in ways that threw some convenient assumptions overboard, it also provided us with new opportunities.

To understand why Android works the way it does in 2025, we'll travel a bit back in time and see how it used to work, what problems it had, and how they were solved. With this, we'll revisit the state of the art for similarities and then present a new approach to working with apps, treating them as binaries instead of bytecode. A good PoC can't be missing, so we show how to do what Elsim did by using our binary trickery: scoring the similarity of an app from zero to one based on code.

TL;DR: We'll show how to turn APKs into binary ELF files and use those to calculate similarities between apps, such as Signal and TM SGNL, using BinDiff. Since this requires BinExport files and disassembly, it can take hours to analyze modern apps. We also provide a PoC to bring this down to minutes, which, for now, has a side effect of making BinDiff slower than it should be.

1 - A Bit of History

And you may find yourself inside an Android Runtime. And you may ask yourself - Well ... How did I get here?

It's 2008. The HTC Dream has just been released with a whopping 528 MHz Processor and 192Mb of RAM. Java was chosen to be the language used to program Android apps. People know Java. People already programmed apps for mobile devices in Java ME, so it was brought to Android. Instead of using the stack-based Java Virtual Machine (JVM), the register-based Dalvik Virtual Machine (DVM) found its way into Android. The dalvikvm executable aimed at having a smaller memory footprint than the jvm, and Dalvik bytecode is easy to transpile from Java bytecode. It differs in some specifications that surely will not come back to bite us `:)`.

For app developers this meant they could write Java code using the Android Software Development Kit (SDK) that would take the Java source code, compile it with a Java compiler like javac to Java bytecode, run Java-based plugins for optimizations and obfuscations (such as ProGuard), and then invoke the Dalvik compiler dx to create Dalvik Executable (.dex) files. The popular plugin ProGuard renamed method and class identifiers, creating a mapping.txt file that the developer can use to make sense of error information, such as stack traces. The .dex files were bundled in an Android Package (APK) zip, which in turn was distributed to devices, where the dalvikvm interprets the apps' code. Android provided a re-implementation of standard Java libraries, which also kept some lawyers busy.

```
+-------------+   +-------------+
| source.java |   | library.jar |
++------------+   ++------------+
 |                 |
+v--------------+  |    +-------------+
| tools to      +--+----> mapping.txt |
| obfuscate and |  |    +-------------+
| optimize      |  |
++--------------+  |
 |                 |
+v--------------+  |
| javac         |  |
++--------------+  |
 |                 |
+v-----------------v-+
| dx                 |
+---------+----------+
          |
          |                   above is accessible to developers
----------+----------------------------
          |                   below is running on devices
          |
+---------+--------------------------+
| APK     |                          |
|         v                          |
|    classes.dex                     |
|         |                          |
+---------+--------------------------+
          |
+---------+--------------------------+
|         |                          |
|    +----v-----+                    |
|    | dalvikvm |                    |
|    +----+-----+                    |
+------------------------------------+
```

It's 2010. The Nexus One puts a 1Ghz processor, 512MB of RAM, and 512MB Storage into people's pockets. Apps need to go fast, and the dalvikvm implemented Just-In-Time (JIT) compilation of Dalvik to machine code. Another executable, dexopt, prepares an app's DEX code for this by saving it as an ODEX file, containing optimized DEX code.

For some code, this is not fast enough. The Native Development Kit (NDK) allows developers to include libraries written in C and C++ in their apps. The Java Native Interface (JNI) allows this code to be called, allowing bundling of optimized libraries.

```
+-------------+   +-------------+
| source.java |   | library.jar |
++------------+   ++------------+
 |                 |
+v--------------+  |    +-------------+
| tools to      +--+----> mapping.txt |
| obfuscate and |  |    +-------------+
| optimize      |  |
++--------------+  |
 |                 |
+v--------------+  |    +------------+
| javac         |  |    | source.cpp |
++--------------+  |    ++-----------+
 |                 |     |
+v-----------------v-+  +v-----------+
| dx                 |  | NDK        |
+---------+----------+  +--+---------+
          |                |
          |                |  above is accessible to developers
----------+----------------+----------
          |                |  below is running on devices
          |                |
+---------+----------------+---------+
| APK     |                |         |
|         v                v         |
|    classes.dex     nativelib.so    |
|         |                |         |
+---------+----------------+---------+
          |                |
+---------+----------------+---------+
|         |                |         |
|    +----v-----+         +--v--+    |
|    | dexopt   |         | JNI |    |
|    +----+-----+         +-----+    |
|         |                          |
|    +----v-----+     +----------+   |
|    | app.odex +-----> dalvikvm |   |
|    +----------+     +----------+   |
+------------------------------------+
```

It's 2014. The Nexus 6 has a processor with not just one but four 2.7GHz cores and 3 GB of RAM. Its 32GB of storage allows for much more code to be stored, and so the JIT compilation is replaced by a complete Ahead-Of-Time (AOT) compilation. ODEX files no longer contain Dalvik code. They contain binary code and become OAT (allegedly "of-ahead-time") files. This compilation adds another layer of optimizations, a fact that will come back in Section 3.4. We will call the binary ODEX files OAT instead of ODEX, but they still use the file ending .odex. They are technically ELF shared objects with some additional metadata, but they are meant to only run with the Android Runtime (ART). It replaces the dalvikvm and can handle oat files.

On the developers' side, the pipeline of javac-to-plugins-to-dx also sees changes: Jack and Jill replace the whole stack, unifying compilation and optimizations. Android apps are fast(er).

```
+-------------+   +-------------+
| source.java |   | library.jar |
++------------+   ++------------+
 |                 |
 |                 |    +------------+
 |                 |    | source.cpp |
 |                 |    ++-----------+
 |                 |     |
+v-----------------v-+  +v-----------+
| jack               |  | NDK        |
| and                |  |            |
| jill               |  |            |
+---------+----+-----+  +--+---------+
          |                |
          |                |  above is accessible to developers
----------+----------------+----------
          |                |  below is running on devices
          |                |
+---------+----------------+---------+
| APK     |                |         |
|         v                v         |
|    classes.dex     nativelib.so    |
|         |                |         |
+---------+----------------+---------+
          |                |
+---------+----------------+---------+
| ART     |                |         |
|    +----v-----+       +--v--+      |
|    | dex2oat  |       | JNI |      |
|    |          |       +-----+      |
|    |   full   |     +-----------+  |
|    |   AOT----+-----> app.odex  |  |
|    |          |     +-----------+  |
|    +----------+                    |
+------------------------------------+
```

It's 2017. The Xperia XZ1 has eight 2.3GHz cores and 4GB RAM. Lessons were learned. First, getting rid of the Java compiler is rolled back and instead ProGuard's functionality is included in the R8/D8 compiler suite that consumes Java bytecode. Kotlin is added as the preferred language. It also turned out that compiling all of an app's code during installation can take a while, even with all those cores available. And, of course, all apps have to be recompiled after system updates, which eats performance like Android codename snacks. JIT is re-introduced, and profile-based compilation is added. The compilation profiles track per app which methods should be compiled ahead-of time, for example, after a system update. This saves resources.

```
+-------------+   +-------------+
| source.java |   | library.jar |
++------------+   ++------------+
 |                 |
 | +-----------+   |    +------------+
 | | source.kt |   |    | source.cpp |
 | ++----------+   |    ++-----------+
 |  |              |     |
+v--v------------+ |     |
| javac/kotlinc  | |     |
+---------+------+ |     |
          |        |     |
+---------v--------v-+  +v-----------+
| R8/D8              |  | NDK        |
|  includes ProGuard |  |            |
|  shrinking and     |  |            |
|  renaming          |  |            |
+---------+----+-----+  +--+---------+
          |    |           |    +-------------+
          |    +-----------+----> mapping.txt |
          |                |    +-------------+
          |                |
          |                |  above is accessible to developers
----------+----------------+----------
          |                |  below is running on devices
          |                |
+---------+----------------+---------+
| APK     |                |         |
|         v                v         |
|    classes.dex     nativelib.so    |
|         |                |         |
+---------+----------------+---------+
          |                |
+---------+----------------+---------+
| ART     |                |         |
|    +----v-----+       +--v--+      |
|    | dex2oat  |       | JNI |      |
|    |   JIT    |       +-----+      |
|    |    +     |     +-----------+  |
|    |   AOT----+-----> app.odex  |  |
|    |          |     +-----------+  |
|    +----------+                    |
+------------------------------------+
```

2 - A Bit of the Current State

What's in a name? An APK by any other name would smell of Java still.

2.1 - Building Android Apps and "Obfuscation"

While ProGuard is not new, its obfuscation capacities in the Android SDK are limited to renaming identifiers of classes, methods, and fields. If an app crashes and the stack trace is collected, this renaming would make it inconvenient to debug, so a mapping.txt is created for the developer to translate the "obfuscated" names back to the original ones. Sadly, we can't find this in an APK, and if renaming took place, we can't trust the class and method identifiers.

In some cases, this renaming should not be applied, for example, when reflection is used to dynamically refer to a class by its name. This can happen when Java/Kotlin code is called from native libraries. Since the NDK and D8/R8 are mostly unaware of each other's internals, these names must stay the same. Developers can exclude identifiers from renaming with the appropriate entries in the ProGuard configuration.

In addition to ProGuard, the full capabilities of the R8 compiler further aggressively optimize the code, which means two things: First, unused (or "dead") code is removed from the app. This mainly affects libraries as they can provide much more functionality in their precompiled .jar file than an app uses. Second, methods can disappear due to inlining. If the compiler can verify that a method is only ever called at specific places, it can decide not to call the method from these places, but instead include the target methods code where it would be called. This reduces the number of unique methods (which is limited per DEX file!) and allows for additional optimizations.

While renaming and compiler optimizations are not actual obfuscation, the official docs called it such until very recently [4].

Shoutout to whoever finally removed the example config of two lines containing two unrelated bugs that would stop compilation, it's been bugging us for a couple years.

2.2 - Similarity-Calculating Tools

Some tools for app similarity calculation are unusable when class and method identifiers are changed. SimiDroid [5] uses names to match methods and then compares their instructions, classifying them as full, partial, or non-matches. The ratio of the resulting sets is then used to calculate an overall similarity.

Diffuse [6] also uses method names for matches and could use mapping.txt files to account for obfuscation. Unfortunately, only developers have those, but we'd like to compare any APKs we can get our hands on, sources be damned.

Dexofuzzy [7] implements a fuzzy hash based on code that is robust against small changes but not against structural ones -- eliminating large parts of dead library code or inlining many methods results in entirely different hashes.

Finally, Elsim [0] uses Normalized Compression Distance (NCD) to provide a "rip-off indicator." It was still available when we started looking at this topic but had trouble analyzing bigger apps. In some cases, we had to stop the comparison after multiple days without a result.

We pushed some optimizations, but it still was too slow to analyze modern apps with hundreds of thousands of methods.

Various approaches are described in academia, but since they do not provide a PoC, they can Give Time For Others. In practice, the standard Android SDK optimizations are indistinguishable from obfuscations for available similarity tools.

2.3 - Dalvik is the Limit

The trouble with analyzing Android apps is that we must deal with Dalvik bytecode. Since it effectively exists only in the Android ecosystem, any tools and techniques will need to be adjusted specifically for it.

(Shoutout to Androguard development picking up after a dry spell!)

To avoid maintaining special Dalvik tooling, some tools opt to transpile Dalvik bytecode into a form usable by tools with a larger audience, such as dex2jar used by SootUP [8]. Its predecessor, Soot, is used by SimiDroid.

Unfortunately, Dalvik is not Java. It has a hard limitation on the number of identifiers in a single DEX file, so tools need to be able to merge those. It's no big deal nowadays. However, Dalvik does not impose an upper limit on the number of instructions in a method, so if it is lifted to Java bytecode, certain functions must be split because they contain more than 65536 bytes, which Java bytecode forbids. This was an open issue that crashes Soot (and thus SimiDroid) and, for a while, caused SootUP to skip analyzing a method. While the upstream dex2jar has fixed the issue [17], SootUP removed this dependency and integration is left as an exercise to the reader. Also, there might still be a way to break analysis with large methods `:)`.

2.4 - Summary and Back to Binary

To summarize, tools to compute similarity don't work well with modern Android apps. Optimizations are having an overly obfuscating effect, and Dalvik-based tools have trouble keeping up to date. So what are our options? What do other fields do to meet their similarity needs? Turns out: BinDiff is forever [9] (and now open source!). Let's leave Dalvik be and compile apps to their binary OAT version using the ARTs ahead-of-time compilation. Since these binaries are valid ELF files, we can feed them to a disassembler, create a BinExport [13] file, and then feed it further to halvarflake's friendly neighborhood diffing tool based on graph-isomorphism [10].

3 - A Bit of Examples

What is a function? A miserable little pile of bytes.

Let's go over some questions you might have by now. We've tried all this on a Pixel 8 hardware phone running Android 15 (BP1A.250305.019) just to show off this works on real devices, but an emulator works just as well. Since the system owns the system-generated OAT files, we need root to access them. Our test app will be Organic Maps [11] (version 2025.05.20-5-FDroid).

3.1 - How Does the ART Compile an App's Code Ahead-of-Time?

The ART is supposed to run AOT compilation during installation, and dex2oat is part of the art, so let's start logcat with `adb logcat artd:I *:S` and install an app.

```
$ adb install app.organicmaps.apk
Performing Streamed Install
Success
```

In the logcat output, we see the complete invocation of dex2oat:

```
$ adb logcat artd:I *:S # formatted for convenience
[...]
02-29 13:37:42.123  6074  6074 I artd    : Running dex2oat:
/apex/com.android.art/bin/art_exec --drop-capabilities
--set-task-profile=Dex2OatBootComplete --set-priority=background
--keep-fds=6:7:8:9:10:11 --
  /apex/com.android.art/bin/dex2oat64
    --zip-fd=6 --zip-location=/data/app/[...]/base.apk
    --oat-fd=7 --oat-location=/data/app/[...]/oat/arm64/base.odex
    --output-vdex-fd=8 --swap-fd=9 --class-loader-context-fds=10:11
    --class-loader-context=PCL[]{PCL[...]}
    --classpath-dir=/data/app/[...] --instruction-set=arm64
    --instruction-set-features=default
    --instruction-set-variant=cortex-a55
      --compiler-filter=verify
    --compilation-reason=install --compact-dex-level=none
    --max-image-block-size=524288 --resolve-startup-const-strings=true
    --generate-mini-debug-info --runtime-arg -Xtarget-sdk-version:35
    --runtime-arg -Xhidden-api-policy:enabled --cpu-set=0,1,2,3,4,5,6,7,8
    -j8 --runtime-arg -Xms64m --runtime-arg -Xmx512m
    --comments=app-name:app.organicmaps,[...]
```

There's a lot in there, and `dex2oat --help` makes for a lovely afternoon read. But we'd like to point out some interesting things. The obvious one is that it's just a binary that we can call ourselves. Sure, there is some trickery with file descriptors, but the flag `--dex-file` accepts a DEX, JAR, or APK. Second, the flag `--compiler-filter` is used to verify the code, creating the .vdex file. It also accepts 'everything', which is nothing more than the full-AOT mode of Android 7 and 8.

Because the system didn't use a profile for this compilation, the resulting odex is quite small (65K).

```
$ adb shell pm path app.organicmaps
package:/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC
_k_T1gA==/base.apk
$ adb shell ls -lah
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
-rw-r--r-- 1 system all_a306 65K 2025-02-29 13:37
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
```

3.2 - How Can We Compile an App's Code Fully Ahead-of-Time?

We could run dex2oat ourselves, recreating the invocation as closely as possible. Fortunately we don't have to deal with this, and can instead ask Android nicely to do it for us. The package manager `pm` will do it:

```
$ adb shell pm compile -m everything app.organicmaps
Success
```

Looking at the logs, we see a very similar invocation, but now the compiler filter has been set to 'everything':

```
$ adb logcat artd:I *:S # formatted for convenience
[...]
02-29 15:51:15.515  6074  6583 I artd    : Running dex2oat:
/apex/com.android.art/bin/art_exec --drop-capabilities
--set-task-profile=Dex2OatBootComplete --set-priority=background
--keep-fds=6:7:8:9:10:11:12 --
  /apex/com.android.art/bin/dex2oat64
    --zip-fd=6 -zip-location=/data/app/[...]/base.apk
    --oat-fd=7 --oat-location=/data/app/[...]/oat/arm64/base.odex
    --output-vdex-fd=8 --swap-fd=9 --class-loader-context-fds=10:11
    --class-loader-context=PCL[]{PCL[...]}
    --classpath-dir=/data/app/[...] --input-vdex-fd=12
    --instruction-set=arm64 --instruction-set-features=default
    --instruction-set-variant=cortex-a55
      --compiler-filter=everything
    --compilation-reason=cmdline --compact-dex-level=none
    --max-image-block-size=524288 --resolve-startup-const-strings=true
    --generate-mini-debug-info --runtime-arg -Xtarget-sdk-version:35
    --runtime-arg -Xhidden-api-policy:enabled --cpu-set=0,1,2,3,4,5,6,7,8
    -j8 --runtime-arg -Xms64m --runtime-arg -Xmx512m
    --comments=app-name:app.organicmaps,[...]
```

And indeed, the odex now contains a lot more binary code:

```
$ adb shell ls -lah
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
-rw-r--r-- 1 system all_a306 14M 2025-02-29 15:51
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
```

Pull the file from your (virtual) device and you are ready to go! We'll save our example as `app.organicmaps.odex`.

3.3 - How Can We Disassemble OAT Files?

It's an ELF! Throw it in your favorite disassembler. They all do reasonably well, we checked by looking at the function boundaries [12]!

```
$ file app.organicmaps.odex
oats/PXL8_everything/app.organicmaps.odex: ELF 64-bit LSB shared object,
ARM aarch64, version 1 (GNU/Linux), dynamically linked, stripped
```

We use Ghidra 11.0.3 because it supports the BinExport [13] plugin and is freely available. We provide a script to do this headlessly, but using the GUI works just as well.

```
$ ./ghidra_headless_binexport.sh app.organicmaps.odex
app.organicmaps.odex.BinExport
[...]
      Total Time    75 secs
[...]
```

Not bad, let's try this with a more complex app like Signal [14]:

```
$ ./ghidra_headless_binexport.sh org.thoughtcrime.securesms.odex
org.thoughtcrime.securesms.odex.BinExport
[...]
      Total Time    2642 secs
[...]
```

Well, this is hardly ideal. Taking almost an hour to prepare an app will not scale, even though we need to disassemble each OAT file we want to compare only once. In the script we already included some speedups, like disabling the GCCExceptionAnalyzer, which takes time but in our case is pointless, since the OAT file has never seen GCC. But modern apps are ridiculously big.

The Signal app has around 500k Dalvik methods alone, not counting native libraries. Its Linux app has around 40k, including all dynamically loaded libraries such as glibc. No wonder it takes almost an hour to analyze the OAT.

Fortunately, Android provides another way to see the disassembly with the oatdump utility. Similar to objdump, it displays information from oat files in human-readable form. It is fast but produces a lot of output. We can get almost all information we could need like this:

```
$ adb shell oatdump --oat-file=/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/\
app.organicmaps-2xFay0QEb95pHRC_k_T1gA==/oat/arm64/base.odex \
> app.organicmaps.odex.oatdump
$ cat app.organicmaps.odex.oatdump
[...]
  3: double app.organicmaps.util.LocationUtils.correctAngle(double, double)
(dex_method_idx=27942)
    DEX CODE:
      0x0000: 1400 0400 0000              | const v0, #+4
      0x0003: 1401 0100 0000              | const v1, #+1
      0x0006: 9000 0001                   | add-int v0, v0, v1
      0x0008: 9400 0001                   | rem-int v0, v0, v1
      0x000a: 3c00 0500                   | if-gtz v0, +5
      0x000c: 2a00 1200 0000              | goto/32 +18
      0x000f: cb53                        | add-double/2addr v3, v5
      0x0010: 1805 182d 4454 fb21 1940    | const-wide v5, \
#+4618760256179416344
      0x0015: cf53                        | rem-double/2addr v3, v5
      0x0016: 1600 0000                   | const-wide/16 v0, #+0
      0x0018: 3002 0300                   | cmpg-double v2, v3, v0
      0x001a: 3b02 0300                   | if-gez v2, +3
      0x001c: cb53                        | add-double/2addr v3, v5
      0x001d: 1003                        | return-wide v3
      0x001e: 2a00 f1ff ffff              | goto/32 -15
[...]
    CODE: (code_offset=0x00816520 size=92)...
      0x00816520: d1400bf0  sub x16, sp, #0x2000 (8192)
      0x00816524: b940021f  ldr wzr, [x16]
        StackMap[0] (native_pc=0x812528, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00816528: f81e0fe0  str x0, [sp, #-32]!
      0x0081652c: f9000ffe  str lr, [sp, #24]
      0x00816530: fd000be8  str d8, [sp, #16]
      0x00816534: f94002b5  ldr x21, [x21]
        StackMap[1] (native_pc=0x812538, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00816538: 5c0001e8  ldr d8, pc+60 (addr 0x00816574) (6.28319)
      0x0081653c: 1e612800  fadd d0, d0, d1
      0x00816540: 1e604101  fmov d1, d8
      0x00816544: f942027e  ldr lr, [tr, #1024] ; pFmod
      0x00816548: d63f03c0  blr lr
      0x0081654c: 1e602008  fcmp d0, #0.0
      0x00816550: 1a9f37e0  cset w0, hs
      0x00816554: 1e682801  fadd d1, d0, d8
      0x00816558: 7100001f  cmp w0, #0x0 (0)
      0x0081655c: 1e611c00  fcsel d0, d0, d1, ne
      0x00816560: fd400be8  ldr d8, [sp, #16]
      0x00816564: f9400ffe  ldr lr, [sp, #24]
      0x00816568: 910083ff  add sp, sp, #0x20 (32)
      0x0081656c: d65f03c0  ret
      0x00816570: 5800007f  ldr xzr, pc+12 (addr 0x0081657c) \
(0xd1400bf0006c564f / -3368679395845974449)
      0x00816574: 54442d18  unallocated (Unallocated)
      0x00816578: 401921fb  unallocated (Unallocated)
```

We say it contains almost all information because the output refers to the code_offset relative to the oatdata section in the OAT file. To get this offset, we can use readelf:

```
$ adb shell readelf -s /data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/\
app.organicmaps-2xFay0QEb95pHRC_k_T1gA==/oat/arm64/base.odex \
| grep 'oatdata$'
     1: 0000000000000658 2865576 OBJECT  GLOBAL DEFAULT    5 oatdata
```

Now we know we need to add 0x658 to the addresses in the oatdump output. If we want to compare it directly to Ghidra's output, we also need to add 0x100000 unless Ghidra used a different base address.

3.4 - Wait, Where Did the Arithmetic Go?

Eagle-eyed readers might have noticed some dummy arithmetic operations in the Dalvik code, with a goto instruction at the end, that seemingly vanished during compilation. We're sorry we were not completely honest. We did not use the regular build of Organic Maps, but instead a version that was repackaged with ObfuscAPK [15], using the "ArtithmeticBranch" obfuscation. This adds a branch at the beginning of each method with an arithmetic expression that always resolves to the same code being executed.

The ART helpfully removed this obfuscation as part of its optimizations. This means we get some free normalization of code along the way, as a treat! Not enough to deal with advanced techniques, but enough to make a difference.

To compare the same method without obfuscations, here's another oatdump of the original Organic Maps app:

```
$ cat app.organicmaps_unobfuscated.odex.oatdump
[...]
  3: double app.organicmaps.util.LocationUtils.correctAngle(double, double)
(dex_method_idx=27942)
    DEX CODE:
      0x0000: cb53                        | add-double/2addr v3, v5
      0x0001: 1805 182d 4454 fb21 1940    | const-wide v5, \
#+4618760256179416344
      0x0006: cf53                        | rem-double/2addr v3, v5
      0x0007: 1600 0000                   | const-wide/fix16 v0, #+0
      0x0009: 3002 0300                   | cmpg-double v2, v3, v0
      0x000b: 3b02 0300                   | if-gez v2, +3
      0x000d: cb53                        | add-double/2addr v3, v5
      0x000e: 1003                        | return-wide v3
[...]
    CODE: (code_offset=0x00953118 size=92)...
      0x00953118: d1400bf0  sub x16, sp, #0x2000 (8192)
      0x0095311c: b940021f  ldr wzr, [x16]
        StackMap[0] (native_pc=0x953120, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00953120: f81e0fe0  str x0, [sp, #-32]!
      0x00953124: f9000ffe  str lr, [sp, #24]
      0x00953128: fd000be8  str d8, [sp, #16]
      0x0095312c: f94002b5  ldr x21, [x21]
        StackMap[1] (native_pc=0x953130, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00953130: 5c0001e8  ldr d8, pc+60 (addr 0x95316c) (6.28319)
      0x00953134: 1e612800  fadd d0, d0, d1
      0x00953138: 1e604101  fmov d1, d8
      0x0095313c: f9420a7e  ldr lr, [tr, #1040] ; pFmod
      0x00953140: d63f03c0  blr lr
      0x00953144: 1e602008  fcmp d0, #0.0
      0x00953148: 1a9f37e0  cset w0, hs
      0x0095314c: 1e682801  fadd d1, d0, d8
      0x00953150: 7100001f  cmp w0, #0x0 (0)
      0x00953154: 1e611c00  fcsel d0, d0, d1, ne
      0x00953158: fd400be8  ldr d8, [sp, #16]
      0x0095315c: f9400ffe  ldr lr, [sp, #24]
      0x00953160: 910083ff  add sp, sp, #0x20 (32)
      0x00953164: d65f03c0  ret
      0x00953168: 5800007f  ldr xzr, pc+12 (addr 0x953174) \
(0xd1400bf00077d5ba / -3368679395845220934)
      0x0095316c: 54442d18  unallocated (Unallocated)
      0x00953170: 401921fb  unallocated (Unallocated)
[...]
```

As you can see, the resulting binary code is the same, modulo offsets. Simple code-based obfuscations, such as dead code, are eliminated during the creation of the .odex file. The power of the optimizing compiler compels thee, binary!

3.5 - Cool, Anything Else?

sigh So why do we need root to access the .odex file? It's because it is typically compiled based on profiles, and those essentially register usage patterns.

Imagine an app containing a method called `buy_drugs_and_do_crime`.

- F. K.

If a method ends up in a profile, it means it's been called often enough to be considered for AOT compilation. And if it's in the profile, then it ends up in the odex as binary code. Reading the odex would leak the information which methods are called often, so Android tries to protect it. If you find a way to read this nevertheless, especially from another app, congratulations: this might earn you a bounty and/or spying^W information gathering capabilities.

4 - A Bit of Putting the Bits Together

YES ... HA HA HA ... YES!

Now that we can create BinExport files from oat files, which contain the executed machine code of an app's Dalvik code, we can calculate similarities!

We prepared a couple of applications to demonstrate results. We will use Organic Maps (app.organicmaps) as a smaller baseline and Signal (org.thoughtcrime.securesms) as a complex, real-world application. We prepared the latter in a couple versions: 2.42.2, 2.32.2, 2.22.2, 2.12.2, and 2.2.4. The oldest version is from Summer 2024, one year ago, and we chose it because it is the basis for the leaked TeleMessage (TM) SNGL app's source code. It's an unofficial clone of Signal with added functionality to save messages centrally and was used by White House staff in early 2025. We used the source [16] to build a release version and compare it with our approach to the OG Signal app.

```
1. org.tm.archive_7.2.4.2.apk
  (release build of leaked TM SGNL app)
2. org.thoughtcrime.securesms_7.2.4.apk
  (Signal version closest in source code to TM SGNAL app)
3. org.thoughtcrime.securesms_7.12.2.apk
4. org.thoughtcrime.securesms_7.22.2.apk
5. org.thoughtcrime.securesms_7.32.2.apk
6. org.thoughtcrime.securesms_7.42.2.apk
  (Different version of Signal)
7. app.organicmaps_25052005_orig.apk
  (Organic Maps)
8. app.organicmaps_25052005_obfuscated.apk
  (Repackaged using ObfuscAPK's ArithmeticBranch pass)
```

We downloaded the Organic Maps app from F-Droid and repackaged it ourselves. The TM SGNL app we built from source in a release configuration and downloaded the historic Signal versions from apkmirror.com because the official GitHub and website didn't offer such old versions of the app. We then created BinExport files for all OATs and present the results of the full cross-comparison below. All confidence scores were above 98%.

```
          ,-org.tm.archive_7.2.4.2
|    | 1./      ,-org.thoughtcrime.securesms_7.2.4
| 2. | 88% | 2./      ,-org.thoughtcrime.securesms_7.12.2
| 3. | 80% | 88% | 3./      ,-org.thoughtcrime.securesms_7.22.2
| 4. | 77% | 84% | 91% | 4./      ,-org.thoughtcrime.securesms_7.32.2
| 5. | 71% | 78% | 84% | 89% | 5./
| 6. | 69% | 76% | 81% | 86% | 94% | 6. -org.thoughtcrime.securesms_7.42.2
```

Two clear outcomes of this cross-comparison of app versions are visible: Versions that are closer also show, unsurprisingly, a higher similarity score. The similarity is also high between the app clone TM SIGNL and Signal, even though additional functionality was added that allows for "archiving" messages. This also holds for app versions close to the version used as basis for our TM SGNL build.

No table is needed for comparing Organic Maps to Signal versions: Any version of Signal scores only 6% similarity with Organic Maps. The confidence plummets as well to less than 30%.

However, BinDiff reports a similarity of 85% with a confidence of 98% for the original and obfuscated versions of Organic Maps.

5 - A Bit of an Outlook

Warning: API levels rising.

Creating BinExport files takes a long time for complex apps and if a use case like clustering is the goal, we're looking at days of preprocessing.

As mentioned, the oatdump output contains many details about the binary code, and we started working on a proof-of-concept binexport2oatdump tool, included in this submission.

The idea is simple: use the presented commands to compile an APK to an OAT file (takes seconds), run oatdump (takes seconds), parse the output (takes about a minute, they are large), and then just create a BinExport protobuf (protocol buffer) file, containing all disassembly information one would need Ghidra for.

We implemented this as a Python script that can utilize all the information oatdump provides to speed up the preprocessing step. We have exact function boundaries with mnemonics and resolved branch targets if they are known at compile time. We can parse the Dalvik code to create the call graph instead of trying to reverse it from the binary information.

Functions contain basic blocks in one sequence without weird branches (except thunks). Our PoC successfully creates BinExport files from oat files an order of magnitude faster than Ghidra.

But ...

But it currently makes BinDiff run an order of magnitude longer for calculating a similarity score. For clustering, this is the opposite of what we want. The preprocessing step only runs once per app, so it scales linearly with the number of apps: O(n). But the comparison runs between each app pair, O(n^2). (Well, half of that but big-O doesn't care about such details). Out of time, out of ideas, and motivated to test in production, we've still attached our work in progress and will continue improving it. For now, dear reader, you have the choice between slow preprocessing and slow comparisons. Hopefully, not for long, then comparisons will be fast, robust, and useful to you!

Just one more thing. We talked about how apps are using native libraries for performance. Integrating them into a code-analysis framework always required to bridge the gap between Dalvik and machine code. If we have the Dalvik code as binary, though, we don't need to bridge a gap that doesn't exist. We're working on using our trick of using dex2oat not just to process an app's Dalvik code, but also its native libraries, making holistic app analysis and comparison possible. Until then, you can use your favorite binary tooling to analyze them all without having to maintain two toolchains.

6 - A Bit of Summary and Thanks

ceterum censeo scientia vult esse libera

In a system as complex as Android, change is the only constant. The fact that Dalvik is unique to this ecosystem does not make it easy to maintain tools, and so a lot of things that once were useful become more error-prone with time. But this evolution also opens up new possibilities.

We looked at how Android changed and how it currently builds and executes apps. The problems this creates, but also the doors this opens. We walked through the process of creating binary odex files from APKs and how to prepare them for BinDiff. We show that it's practical to compare apps and that this approach even applies free code normalization, removing some obfuscations in the process of recompilation.

Finally, we also provide a PoC script showing that there is significant speedup to be gained: Sidestepping the long disassembly process and utilizing oatdump to create BinExport files directly. This hack will make code-based similarity scalable for a large amount of apps and allow hunting for suspicious APKs, be it to find clones, for malicious apps, just for fun, or for profit.

It's our pleasure to submit this work to Phrack, which inspired not only this line of work but also our enthusiasm for looking behind the curtain, to hack, and to share. And special shoutout to:

- Sebastian Bachmann, aka reox for telling us about the neat tool called Elsim
- Felix Kehrer for listening to our Android rants and also ranting about Android

7 - A Bit of References

[0] https://phrack.org/issues/68/15#article
[1] https://github.com/themoep/elsim
[2] https://web.archive.org/web/20170713094900/
https://www.theverge.com/2017/7/12/15958372/
google-machine-learning-ai-\app-store-malware-
security
[3] https://github.com/EFForg/apkeep
[4] https://web.archive.org/web/20250421110118/\
https://developer.android.com/build/shrink-code
[5] https://github.com/lilicoding/SimiDroid
[6] https://github.com/JakeWharton/diffuse
[7] https://github.com/ramazansancar/Dexofuzzy2
[8] https://github.com/soot-oss/SootUp
[9] https://youtu.be/ajGX7odA87k?t=1873
[10] https://github.com/google/bindiff
[11] https://organicmaps.app/
[12] https://dl.acm.org/doi/10.1145/3578357.3591219
[13] https://github.com/google/binexport
[14] https://signal.org/
[15] https://github.com/Mobile-IoT-Security-Lab/
Obfuscapk
[16] https://github.com/micahflee/TM-SGNL-Android/
[17] https://github.com/ThexXTURBOXx/dex2jar/
issues/16

8 - And That Makes a Byte (code)

Visit Phrack.org to see PoC for oatdump2binexport