Stigma: A Tool for Modifying Closed-Source Android Applications*

Ed Novak[1], Shaamyl Anwar[2], Saad Mahboob[3]
Shokhinabonu Tojieva[4], and Chelsea Rao[5]

Computer Science Department
Franklin and Marshall College
Lancaster, PA 17604

[1]enovak@fandm.edu, [2]mshaamylanwar@gmail.com, [3]saadmahboob3@gmail.com
[4]shokhinatojieva@gmail.com, [5]crao@fandm.edu

Abstract

A difficult but potentially powerful advanced software engineering concept is to modify existing, compiled, closed-source applications to identify and potentially remedy security and privacy issues. This technically challenging concept is very applicable to the Android ecosystem, but existing approaches are bespoke, use-case specific implementations. In this paper we present Stigma, an open-source software tool which can make modifications to commodity Android applications. Our tool allows researchers and skilled users to define their own desired modifications for a range of purposes such as security and privacy analysis, improving app functionality, removing unwanted features, debugging, profiling, and others. We evaluate Stigma in terms of compatibility, efficacy, and efficiency on approximately 100 commodity Android applications.

1 Introduction

Android smartphone applications are most commonly pre-compiled and closed-source. This makes their functionality rigid and somewhat opaque, leading to

security concerns and contributing negatively to the general trend of concerns about user privacy. We seek to provide tools that modify and analyze the behavior of these closed-source applications. Such tools could one day be used to perform a variety of tasks such as identifying app uses of sensitive information, searching for security vulnerabilities, fixing bugs, and even improving app functionality independently of the original developer.

Unfortunately modifying commodity Android apps is generally only done in very limited ways via custom fit, temporary, and largely manual processes. Although reverse engineering and "cracking" / "modding" exists in a limited sense in the Android community, these efforts are largely disconnected and lack big picture strategies. Many of the existing research projects in this area [16, 11, 17] do not seem to be available, and therefore cannot be feasibly replicated, embraced, or extended. Furthermore, these legacy projects have usability and compatibility limitations [4].

Modifying apps is difficult due to the fact that the source code is not available. Instead users and tools must operate directly on the byte-code (machine code for a virtual machine). Writing byte-code directly is difficult due to numerous esoteric constraints, non-obvious syntax rules, and few conveniences most programmers take for granted (e.g., for loops). Additionally, the Android framework, runtime environment, and build system are complex leading to bespoke techniques and tools.

In this paper we present Stigma [13]; an open source (GPLv3), command line, python program which can help users make modifications to Android applications for security and privacy analysis. Users first obtain the Android Package (APK) of an app, which is used as input. Plugins, written in python for Stigma, determine what alterations will be made to the app. A new, modified APK is output, which can be run on any target device that the original APK was compatible with. No modifications are necessary to the device or OS itself making for easier reproducibility and long term compatibility. The contributions of this paper are as follows:

- We present the design and prototype implementation of Stigma, our open source and extensible software tool for modifying Android apps.
- As an exemplary use of Stigma, we implement (and also distribute in open source) two Stigma plugins. The first is a dynamic information flow tracking (DIFT) plugin and the second is a "SharedPreferences" extraction plugin.
- We highlight the esoteric details of the "reference pools" in the DEX file format, which to the best of our knowledge, have not been documented extensively elsewhere.

- We evaluate Stigma and our two plugins on approximately 100 popular Android applications. We seek to measure the compatibility of Stigma with arbitrary Android applications as well as the overhead incurred on the application.

2 Related Work

The most closely related works are "Dr. Android and Mr. Hide" [10], and "SIF" [9] in which the smali assembly code of Android apps is modified directly and automatically. "Dr. Android" makes limited modifications in the narrow scope of implementing a more fine grained permission system. SIF asks the user to specify their desired modification in a language called "SIFScript." The SIFScript includes the functionality itself as well as the general places and times in which the functionality should be inserted (called the "workload"). These works were published in 2012 and 2013 respectively. Due to their age, it is very likely that they are no longer compatible with modern Android. They seem to be orphaned and don't appear to be readily available online.

2.1 DIFT Systems

One relevant subfield is that of dynamic information flow tracking (DIFT), in which code is added into the target app to track and alert the user about the use of their own sensitive and/or personal, identifiable information (PII). This is the idea implemented by our Stigma plugin described in Section 4. Some of the most relevant works in this area include ViaLin [1], TaintMan [16], TaintArt [15], TaintDroid [8], AppCaulk [11], ConDySTA [18], and Capper [17].

2.2 Community Projects

Some less formal, community based efforts include the Cydia Substrate for Android [5], the (apparently defunct) Xposed Framework, and Android DDI [7] which allows the user to write their modifications using the Java Native Interface (JNI). None of these projects seem to have accompanying publications in peer reviewed conferences or journals.

2.3 Limitations of the Related Work

The existing works in this area all suffer from at least one of two critical flaws. Either the software described was never released to the public, or the system design has significant compatibility and usability concerns. Some projects exhibit both problems. Compatibility and usability concerns include requiring substantial changes to the Android OS, the Android Framework, the dex2oat

compiler, rooting, or changing other aspects of the platform / device itself instead of the app. These approaches are difficult for others to setup, brittle, and become outdated quickly as the Android OS is continually updated. In contrast, our system is carefully designed to follow the semantics of smali / dex byte-code itself, which is a standard that hasn't changed substantially since the introduction of Android.

3 Stigma System Design

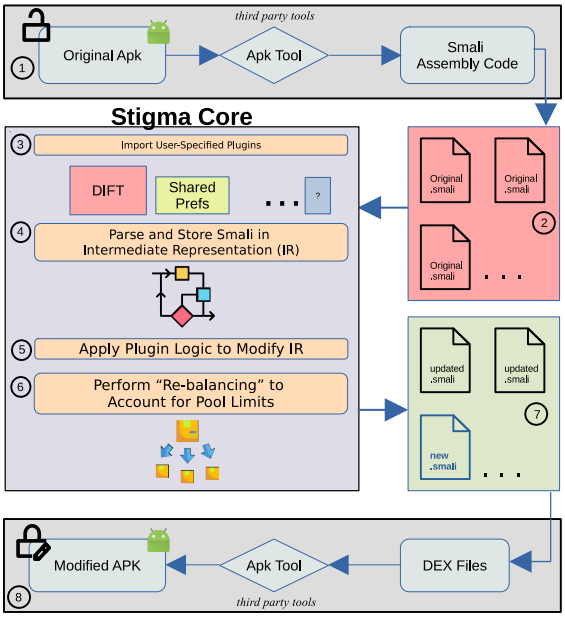


Figure 1: Stigma system architecture.

Stigma [13] is a fully open-source python program that accepts an APK file as input, and outputs a modified version of that APK file. The modifications made are specified by plugins, which are written by the user. As a proof of concept, Stigma comes with two pre-written plugins. One performs DIFT, which seeks to track the use of location (GPS coordinate) data. The other inserts code such that the app prints the keys and values of the default SharedPreferences database when the app is launched. Further information about these two plugins is given in Sec. 4 and 5. Other plugins can be imagined and implemented that allow researchers and other power users to specify precisely what modifications should be made to the app.

An overview of the architecture of the system can be seen in Fig. 1. First ① a third party tool `apktool`[3], is used to extract the Dalvik byte-code (DEX[1]) from the application and convert it to the assembly-like `smali` [12,

[1] DEX is designed to be run on a Java Virtual Machine (JVM), but in the modern Android ecosystem, it probably never will be. Instead, the DEX2OAT compiler is invoked at install

2] language ②. Stigma then parses these smali files into an intermediate representation (IR) ④. Stigma maintains in-memory representations (objects) for smali classes, smali methods, registers, and basic data-types (32-bit, 64-bit, and object references). Plugin logic is applied to the IR. Then in step ⑥, the code must be "re-balanced" to account for the constant pool limits as described in Section 3.1. Finally, the modified IR is written back to smali files on disk. The modified smali files ⑦, along with any new smali classes added by the plugin(s), are re-packed using the same third party tool `apktool` ⑧. The end result is an APK, digitally signed by Stigma, which can be installed on any device for which the original, input APK was compatible.

Stigma parses and allows the user to modify the smali assembly code of the target app. The original Java or Kotlin source code is not available, due to most apps being distributed close-sourced. And the immediately available DEX byte-code is not human readable, making it near impossible for users to define plugins for DEX directly.

As mentioned previously, the smali classes, methods, instructions, registers, and types of the original app are all represented in-memory by python objects. Stigma also builds a control flow graph for every method in the app, and does type analysis such that it can determine the known type of every data value stored in every register at any point in the execution. Of course, there are many points where the type information is unknown or undefined. For example, at the very beginning of a method most of the temporary, general purpose registers are empty and therefore have no type.

3.1 Reference Pools

As mentioned very briefly in the official Dalvik documentation "There are separately enumerated and indexed constant pools for references to strings, types, fields, and methods." [6]. This means that in a single DEX file (comprised of many smali files) all *references* to (1) strings, (2) types, (3) class fields, and (4) methods are collected into respective sets. Each set may contain at most 65,535 entries since the pools are enumerated using an unsigned short.

As applications have grown larger and larger, it has become increasingly common for a single Android application to exceed the 65k limit on one or more of the pools. To alleviate this the code is distributed into multiple DEX files (`classes.dex`, `classes2.dex`, `classes3.dex`, etc.) such that none of the pools are overloaded. Normally this is done by the `dx` converter (from the Android SDK) which converts Java `.class` files to Android `.dex` files. But, Stigma modifies the code *after it has been converted to DEX*, and so any

time to convert the DEX code to machine code matching the architecture of the device. The app is then run on that device via the Android RunTime (ART).

```
1  .class public Lcom/example/stigmatestapp/MainActivity;
2  .super Landroidx/appcompat/app/AppCompatActivity;
3  .source "MainActivity.java"
4
5  # static fields
6  .field static final GREEN_TRANSPARENT:I = 0x6600ff00
7
8  # instance fields
9  .field sputgetText:Landroid/widget/TextView;
10
11 # direct methods
12 .method public constructor <init>()V
13     .locals 0
14
15     .line 19
16     invoke-direct {p0}, Landroidx/appcompat/app/
           AppCompatActivity;-><init>()V
17
18     return-void
19 .end method
```

Listing 1: Sample code used to demonstrate how the constant reference pools are tabulated. This listing contains 9 string references, 5 type references, 2 field references, and 2 method references.

changes it makes that alter the amount of items in the pools may cause those pools to overflow and for the app to fail to compile / run.

Properly organizing a collection of smali files into an appropriate number of DEX files is not straightforward, since it is not clear how various smali instructions impact the four pools. To our knowledge, this relationship is not explained in any pre-existing documentation. The `smali` command line tool [12] can be used to create a DEX file from a single smali file. The resulting DEX file will likely be unable to run, since it does not even contain some of the necessary foundation code such as the Android support libraries. But, such a DEX file can be analyzed by the `dexdump` tool, from the Android SDK, to precisely determine how the code therein contributes to each pool. Using this tool-chain we are able to reverse engineer the relationship between smali code and the four pools.

Consider the code sample in Listing 1. According to `dexdump`, the DEX file containing *only this class* contains 9 string references, 5 class/type references, 2 field references, and 2 method references.

- **Strings** (9 total) - The class name and parent class name on lines 1 and 2 account for two strings. And, line 3 contains a literal string. These strings are used, presumably, for debugging purposes such as stack traces,

and compiler warnings as well as for Java reflection. Each of the fields declared on lines 6 and 9 contribute two string references each (one for the identifier, and another for the type). Finally, the method declaration on line 12 contributes two string references, (one for the method name, and the other for the method return type).

- **Types** (5 total) - The class and parent class on lines 1 and 2 are types. The `I` in the field declared on line 6 is a type (integer). The `TextView` referenced on line 9 is a type. And, the `V` indicating that the `<init>` method returns `void`, is a type. Note that the reference to `AppCompatActivity` on line 16 is not counted, because it is redundant with the reference on line 2.
- **Fields** (2 total) - This class references only two fields (as declarations) on lines 6 and 9.
- **Methods** (2 total) - This class references only two methods. The declaration of `MainActivity.<init>` on line 12 and the call to `AppCompatActivity.<init>` on line 16.

Stigma uses this logic to distribute smali files into a number of `classesX.dex` files appropriately. It is important to note that smali instructions need no modifications or special access rights in order to reference the classes, fields, and methods of smali files in other DEX files. Access rights are restricted only by the traditional Java access modifiers `public`, `private`, and `protected`.

3.2 Extensible Plugin Framework

Without any active plugins, Stigma will not make any changes to the target app. How should intended changes be specified? In our system, plugins specify callback handler functions. The callback handler function itself is written by the plugin author / user, specifying which key points it should be invoked on. These functions return new smali code that is inserted into the app at key point(s).

The handlers are invoked as the original application code is linearly iterated over. They can be called or triggered at various key points in the target app. First they can be called at the point in the app at which the app is launching. This is similar to "the start of the main()" in a traditional program. Second they can be called at the start of each original smali method. Finally, the most intricate trigger is applied individually to each of the 200+ types of smali instructions.

Stigma provides an "Instrumenter" class, which has a method `sign_up()`. Plugins can call the `sign_up()` method in order to register callbacks as shown in Fig. 2.

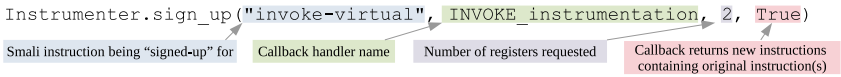


Figure 2: Example of a call to `sign_up()` made by a plugin registering a callback handler for the `invoke-virtual` smali instruction.

3.3 Other Implementation Challenges

Although smali assembly is the most user friendly form of the code to work with, there are still several technical details that must be accounted for when writing or modifying smali code. Stigma accounts for many of them automatically. Specifically, extracting the code from an APK file and converting it to smali, allocating and identifying machine registers that are free to use by the plugin code, accounting for code offset value limits, avoiding unintentional changes to control flow, correctly allocating "reference pools" among DEX files (as discussed in Sec. 3.1), and re-packing the modified smali code back into a usable APK with a valid cryptographic signatures.

Many of these esoteric and complex details of the smali and DEX languages are not well documented. Interested readers can see our technical report [14].

4 DIFT Plugin

As a proof of concept, we design a plugin for Stigma that implements dynamic information flow tracking (DIFT) of sensitive user information. Our plugin registers a variety of handlers for many smali instructions to (1) originate, (2) propagate, and (3) terminate tags that mark sensitive data. It also registers a handler for the start of each method to propagate tag values from the function parameters / inputs. The plugin essentially specifies new smali instructions, which Stigma inserts amongst the original app instructions, to implement the logic of sensitive information marking and tracking. Our plugin has several limitations (it is only a prototype), which are given in our technical report [].

4.1 Tag Origination

For tag origination, Stigma identifies several key functions from the Android API that can be used to obtain sensitive data. For example, `LocationManager.getLastKnownLocation(String provider)`, which is one method used to obtain the device's GPS coordinates. When this instruction/method call is identified in the smali assembly code, our Stigma plugin interleaves instructions to store a tag value on the register used to store the return value.

4.2 Tag Propagation

When a tag is applied to a register, that tag value should flow as the data in that register flows. For example, if the data is copied to another register or passed to a function, the tag should also flow. Our DIFT plugin interleaves new smali instructions to move the tag values, triggered by roughly 85 of the almost 250 smali instructions that move data.

4.2.1 Propagating Tags Across Function Calls

Well written Java code makes heavy use of methods. In order to track sensitive information in and out of method calls, we split all methods into two categories: "internal" and "external". Internal methods are all those defined in the smali code contained in the target APK file. External methods are those for which their smali source code is unattainable. For example, `java/lang/StringBuilder` is provided by the runtime so the code is not included in the APK.

For internal method calls, the tags for the arguments (at the call site) need to be propagated to the parameters (at the definition / callee site). At the call site, new instructions are added just before the function call. The tags for the method arguments are read (e.g., `public static CallingClass_foo_v1_TAG:F`) and then written into the tag locations for the method parameters at the callee site (e.g., `public static CalleeClass_bar_p0_TAG:F`).

When a method returns (keeping in mind there may be more than one return point) the returned value may contain sensitive information. Therefore, for every `return` instruction, the tag value of the returned register is copied into the special global tag field `public static return_field_TAG:F`. At the call site, the tag value is extracted from that field and propagated to the specified destination register.

4.3 Tag Termination

For (3) tag termination, Stigma identifies certain functions which indicate data transmission. In the current implementation this is limited to the various `write()` methods of `java/io/OutputStreamWriter;` and `java/io/OutputStream;`. These are used in network socket I/O and file I/O. When such a method is identified, our plugin writes new instructions into the application which retrieves the tag value associated with each of the input parameters. If the tag value of any parameter is not zero, an entry is written to the Android system log (logcat) alerting the user that sensitive information is being leaked.

5 SharedPreferences Extraction Plugin

"SharedPreferences" is an often used API in the Android framework. It allows app developers to store key-value pair information. Traditionally, it is intended to store the user's innocuous preferences that are specific to an app (e.g., repeat or shuffle in a music player app). Occasionally, developers store things that they should not, introducing security risks and vulnerabilities. Examples include plaintext passwords, private information, booleans to control paid only features, and encryption keys.

We wrote a plugin for Stigma that forces the app to print the entire contents of the default SharedPreferences database when the app is launched. This allows the user to search for suspicious or obviously improper use. Although there are other methods of obtaining the SharedPreferences contents, ours does not require rooting or modifying the OS / device. Additionally, our approach operates during runtime. Which is important, since applications that have not been run in a realistic way likely will not have added any actual values to the SharedPreferences database.

5.1 Implementation Details

Our plugin adds roughly 50 smali assembly instructions to application that invokes the Android SharedPreferences API. Since our code is inserted into the app itself, it runs with same privileges that app has. We simply iterate over the returned HashMap and print the values using the built-in Android logging system (logcat).

Finding the starting point of the application is not straightforward since Android applications follow an event driven architecture and there is no traditional `main()` function. To find the starting point of the application, Stigma parses the associated `AndroidManfiest.xml` file for `activity` and `activity alias` instances that specify the "`LAUNCHER`" attribute. The new code is then inserted into those activities at the start of the `onCreate()` method.

6 Evaluation & Case Studies

To evaluate the compatibility across many commodity Android applications, we acquired 100 random popular applications from `https://APKMirror.com` and ran Stigma on them.

First, we selected approximately 31 applications and, for each, we processed it with Stigma using our prototype DIFT plugin. If successful we installed and ran that app on an Android device. We found that approximately 45% of the apps we tested (14/31) had some sort of compatibility problem with the 3rd party depedency `apktool`, making it impossible to fully evaluate Stigma

on that app. Of the remaining 17 apps, only 11.76% of them (2/17) had compatibility problems with Stigma itself. Of those 15/31 apps that appeared to be fully compatible, Stigma was able to identify and track the use of GPS location information in 6 apps.

Similarly, for the remaining 67 random applications we processed each of them using Stigma with our prototype SharedPreferences plugin. We found that approximately 26% of the apps (18/67) had some sort of compatibility problem with `apktool`. From the remaining 49 apps, we were able to extract SharedPreferences data from 59% of them (29/49).

Stigma is not able to obtain GPS data or SharedPreferences data from every app. In the vast majority of cases where it was not, the reason is simply because the app in question doesn't appear to utilize location data or the SharedPreferences API at all.

6.1 LOC Overhead

To measure the overhead of the new instructions added by Stigma, we compare the number of lines of code in an application before and after Stigma is run with the DIFT plugin (which adds many more lines of code than our other prototype plugin). We compiled a short list of 5 "case study" applications. Three applications were selected from a list of popular Android applications. One (Open Chaos Chess) was selected from the open-source FDroid application market. And the final application is a simple, self made application used for testing and development of Stigma. All five are listed below. The LOC analysis is shown in Fig 3a.

1. "**Weather**" weather forecasts - `com.macropinch.swan`
   version: 5.1.7

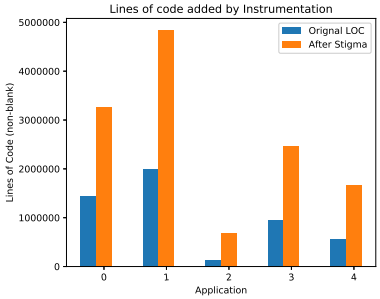
2. "**GroupMe**" messaging application - `com.groupme.android`
   version: 5.54.4

3. "**Open Chaos Chess**" chess game - `dev.corruptedark.openchaoschess`
   version: 1.7.0

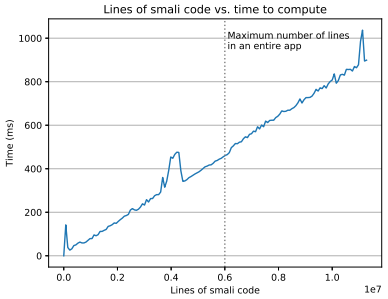
4. "**Office Documents Viewer**" office suite
   `de.joergjahnke.documentviewer.android.free`
   version: 1.29.13

5. "**Stigma Test App**" internal test application -
   version 0.1

The lines of code, both before and after modification, are measured as any non-blank lines of *smali assembly code*. This includes comments, function



(a) Lines of code added by Stigma.



(b) Time incurred by smali instructions.

signatures, field declarations, etc. in addition to actual opcodes / instructions. Therefore, some lines of code added don't incur much, if any, computational overhead. As a byproduct of the design of Stigma, many class fields are added to the application, which forms the bulk of the new lines of code shown in this analysis. As is shown in Fig. 3a our implementation increases the lines of code by about a factor of 2.5x.

6.2 Memory Overhead

We also measure the memory usage of applications on launch. For each application, we launched and performed basic functionality (logging in, starting a game, etc.) The `top` command on the Android command line (`adb shell`) is used to examine the memory usage. This was done before any instrumentation and again after. Results are shown in Table 1. `VIRT` represents the total size of the virtual address space for that process. `RES` represents the actual physical memory in use ("**RES**erved") for that process. Although there is some memory overhead, it is relatively small. This is likely due to the fact that in-memory data structures and code are insignificant compared with common assets such as audio, and images that already exist in most apps and which Stigma add none.

6.3 CPU Overhead

A significant concern of Stigma is overhead it might incur in responsiveness and performance of the application. Empirically, this impact is imperceptible. The computational complexity of the original app code is generally not changed, since our prototype plugins do not introduce any loops, recursion, or new function calls.

Process	**VIRT** before/after	**RES** before/after
Weather	4.3 / 4.4G	130M / 173M
Weather "remote"	4.1G / 4.1G	32M / 38M
Groupme	4.2G / 4.3G	111M / 143M
Groupme "sync"	4.1G / 4.2G	56M / 69M
Open Chaos Chess	4.2G / 4.2G	135M / 135M
Document Viewer	4.4 / 4.4G	139 / 151M
Stigma Test App	4.1G / 4.1G	62M / 67M

Table 1: Memory overhead before and after instrumentation.

To estimate the CPU overhead more precisely we wrote a simple Android application, which executes arbitrary assembly code similar to the code that might be inserted by Stigma to perform DIFT. We executed these instructions repeatedly in larger and larger batches, each time measuring the amount of time it took to complete the batch. The experiment was carried out on a Nexus 5x, which is a modest device released in 2015 running a Hexa-core CPU: (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57). The results can be seen in Fig.3b.

Generally, smali instructions incur a negligable amount of overhead. For an app of reasonable size of 2 million lines of (smali) code, *the entire codebase* could be executed in only $\sim$2/10 of a second. Usually, Android app performance is not CPU-bound, but rather I/O bound.

7 Conclusion

Modifying the code of commodity Android applications is a promising, foundational technique. Unfortunately, most of the tools and systems created in this area have a singular purpose and many are never released. Recent published works from the literature usually don't reveal the critical details necessary to safely modify smali code. In this work we uncover critical concepts for modifying smali code, highlight pitfalls that researchers will experience, and offer an open-source prototype implementation called Stigma. Our work aims to support and inspire future research efforts in this area.

References

[1] Khaled Ahmed et al. "ViaLin: Path-Aware Dynamic Taint Analysis for Android". In: *Proceedings of the 31st ACM Joint European Software En-*

gineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2023. <conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 1598–1610. ISBN: 9798400703270. DOI: 10.1145/3611643.3616330. URL: https://doi.org/10.1145/3611643.3616330.

[2] *Android Instrumentation with Smali: A survival guide*. Available At: http://paulsec.github.io/posts/android-smali-primer/. 2020.

[3] *APKtool Project Source Code*. Available At: https://ibotpeaches.github.io/Apktool/. 2022.

[4] Fabian Berner. and Johannes Sametinger. "Dynamic Taint-tracking: Directions for Future Research". In: *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT,* INSTICC. SciTePress, 2019, pp. 294–305. ISBN: 978-989-758-378-0. DOI: 10.5220/0008118502940305.

[5] *Cydia Substrate for Android*. Available At: http://www.cydiasubstrate.com/. 2014.

[6] *Dalvik Byte-Code Documentation*. Available At: https://source.android.com/devices/tech/dalvik/dalvik-bytecode. 2022.

[7] *DDI, Dynamic Dalvik Instrumentation Toolkit*. Available At: https://github.com/crmulliner/ddi. 2022.

[8] William Enck et al. "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 393–407. URL: http://dl.acm.org/citation.cfm?id=1924943.1924971.

[9] Shuai Hao et al. "SIF: A Selective Instrumentation Framework for Mobile Applications". In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '13. Taipei, Taiwan: Association for Computing Machinery, 2013, pp. 167–180. ISBN: 9781450316729. DOI: 10.1145/2462456.2465430. URL: https://doi.org/10.1145/2462456.2465430.

[10] Jinseong Jeon et al. "Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications". In: *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 3–14. ISBN: 9781450316668. DOI: 10.1145/2381934.2381938. URL: https://doi.org/10.1145/2381934.2381938.

[11] J. Schutte, D. Titze, and J. M. de Fuentes. "AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking into Android Apps". In: *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. 2014, pp. 370–379. DOI: 10.1109/TrustCom.2014.48.

[12] *Smali Project Source Code*. Available At: https://github.com/JesusFreke/smali. 2022.

[13] *Stigma Source Code*. Available At: https://github.com/fmresearchnovak/stigma. 2022.

[14] *Stigma Technical Report*. Available At: http://ednovak.net/documents/stigma_tr.pdf. 2024.

[15] Mingshen Sun, Tao Wei, and John C.S. Lui. "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 331–342. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978343. URL: http://doi.acm.org/10.1145/2976749.2978343.

[16] W. You et al. "TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices". In: *IEEE Transactions on Dependable and Secure Computing* 17.1 (Jan. 2020), pp. 209–222. ISSN: 1941-0018. DOI: 10.1109/TDSC.2017.2740169.

[17] Mu Zhang and Heng Yin. "Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 259–270. ISBN: 9781450328005. DOI: 10.1145/2590296.2590312. URL: https://doi.org/10.1145/2590296.2590312.

[18] Xueling Zhang et al. "ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 796–812. DOI: 10.1109/SP40001.2021.00040.