Stigma: A Tool for Modifying Closed-Source Android Applications
Technical Report

Ed Novak[*] Shaamyl Anwar[†] Saad Mahboob[‡] Shokhinabonu Tojieva[§] Chelsea Rao[¶]
Department of Computer Science
Franklin and Marshall College
Lancaster, PA

August 5, 2024

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

Most Android smartphone apps are closed source software, which makes them rigid and opaque in operation. Generally, their precise functionality cannot be easily known or changed, and the data they operate on cannot easily be known or tracked. Culturally, this has bread numerous concerns of privacy, security, and convenience [24, 25, 16]. Application functionality is generally intricate and clandestine. Although power users are better able to effectively navigate the complex digital landscape, regular users are overwhelmed [17, 21, 20]. All users should remain the owners of their data and their software. They should be able to know what their installed applications are doing, make adjustments to the behavior, and monitor the use of their own private information.

Modifying or tracking the execution of Android apps can allow users to perform a variety of tasks from tracking use of sensitive information, stripping out advertisements, enhancing performance, searching for security vulnerabilities, improving existing functionality, adding features, or fixing bugs. But, modifying pre-compiled, closed source software is generally only done in very limited ways via custom-fit, temporary, and largely manual processes. Reverse engineering, "cracking", modding, re-packaging, instrumenting, and even software profiling are all examples of users working to modify existing software in various ways to achieve different goals. In the

[*]Email: `enovak@fandm.edu`
[†]Email: `mshaamylanwar@gmail.com`
[‡]Email: `saadmahboob3@gmail.com`
[§]Email: `shokhinatojieva@gmail.com`
[¶]Email: `crao@fandm.edu`

Android community these efforts are largely disconnected and lack transcending, big-picture strategies. A single tool that could modify closed source Android app code in a generic way could transform and unite these communities; empowering users to take back some control of the software they study, build, and use every day.

Modifying closed source Android applications is challenging. Android applications are commonly written in a higher-level language: Java or Kotlin. They are distributed as APK files, the code contained therein has already been compiled to the DEX machine code / byte-code format. DEX is rather obscure and although there is documentation about it online, few people are familiar with it and even fewer have any amount of practice writing or even editing it. It has several esoteric constraints, non-obvious syntax rules, and few conveniences most programmers take for granted (e.g., for loops). Furthermore, the code structure, variable names, comments and other important features from the original source code are removed during compile time. So, modifying DEX is extremely difficult and only barely feasible for researchers and other experts. Deep understanding of the original code, as well as large, sophisticated changes to the DEX byte-code are simply not practical, even for experts. Finally, the Android framework, build-system, and API in general are elaborate and complex. This leads to app modification systems that are one-off, custom, bespoke solutions. These are usually single-purpose, require onerous manual steps, and usually have significant usage and functionality limitations.

An application that contains even a few lines of code that inadvertently break the constraints and rules of the DEX language will not run. Such an app may crash at runtime due to invalid operations, get rejected by the Java verifier at runtime, or get rejected by the assembler. DEX has an unofficial, non-sponsored assembly language called Smali [5]. Documentation of smali is minimal and development tools for it are inadequate.

All research efforts in building such a system leave the implementation inadequately defined. Among the published literature, we were not able to find any systems that are easily obtainable in source code or binary format [30, 22, 31]. These works cannot be re-created by others and so their results cannot be easily replicated, embraced, or integrated into new systems. Based on the way these tools are described in the literature, it seems that they were not designed for extensibility, but rather always a single purpose. For example, there are several systems that seek to insert code into apps to track sensitive user information. All of these systems are independent implementations and have significant usability and/or compatibility limitations [10]. Furthermore, these legacy tools often make use of a wide variety of dependency projects, which are often orphaned or poorly maintained. New research efforts in this area are currently stalled, as any researchers seeking to improve upon these systems must first complete the arduous task of re-implementing a robust DEX byte-code modification tool.

We present Stigma [6]; an open source software tool which can make arbitrary modifications to commodity Android apps. Stigma is extensible and freely available under the GPLv3 open source license. It is designed with a plugin framework that allows users and researchers to define desired modifications to their target Android app(s). The contributions of this paper are as follows:

- We present the design and prototype implementation of Stigma, an open source and extensible software tool for modifying commodity 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 Shared Preferences extraction plugin.
- We highlight several esoteric constraints and pitfalls of the dex/smali language. These constraints are difficult to discover and, 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.

Stigma is a python program, which runs on any X86_64 computer. Users first obtain the APK for an app which is used as input to Stigma. The plugin(s) determine what modifications Stigma makes to the application. A modified version of the APK is output, which can be run on any target device that the original APK was compatible with. The dependencies Stigma relies on are minimal and, at the time of writing, are all actively maintained. It is our hope that this work invigorates new research in privacy, security, performance and other aspects of Android applications.

2 Related Work

The most closely related works from the research literature are "Dr. Android and Mr. Hide" [15], and "SIF" [14] in which smali assembly code is modified directly and automatically. Dr. Android makes limited modifications in the narrow scope of implementing a more precise 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. They seem to be orphaned and don't appear to be readily available online. Due to their age, it is very likely that they are no longer compatible with modern Android.

Some less formal community based efforts include the Cydia Substrate for Android [1], the (apparently defunct) Xposed Framework, and Android DDI [4] which allows the user to write their modifications using the Java Native Interface (JNI). These systems are also orphaned and likely obsolete due to changes implemented in Android. Namely, the introduction of the ART runtime in 2015. None of these projects seem to have accompanying publications in peer-reviewed conferences or journals.

There are many many projects from the literature that aim to modify or analyze closed source Android apps in various ways. Many are static analysis only. Of those that are dynamic and actually involve executing the target app(s), most require 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 will become outdated quickly as the Android OS is continually updated by its parent Google. In contrast, Stigma 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.

2.1 DIFT Systems

One relevant sub-field 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 [9], TaintMan [30], AppCaulk [22], ConDySTA [32] and Capper [31]. These systems each make numerous design choices often lacking any consistency with others. They often fail in describing the details of the numerous concepts needed to successfully modify smali code such as higher-numbered register allocation, impacts on control flow, and the constant reference pool limits. In contrast our description these challenges and our solutions is given in Section 6. Further, unless otherwise noted these systems do not seem to be available or practically usable by others.

ViaLin [9] modifies the byte-code of target applications directly. Although their implementation is not described in great detail it is available online [8]. The required setup is error-prone and somewhat cumbersome, requiring the user to make numerous manual changes to a specific version of the AOSP.

TaintMan [30] introduced the idea of system-wide DIFT through smali modification, and touches upon some important ideas like taint tag storage and persisting taint-tags through function calls. It also introduces novel techniques like system library reference hijacking and "strict control dependence" implicit information flow tracking.

AppCaulk [22] attempts to only track information flows on *relevant control flow paths* from a source of sensitive information to a sink. Such paths are first identified via static analysis, and then DIFT instructions are inserted into the smali code to perform dynamic analysis on those paths only. The system relies on method summaries to model behavior for system library code, which cannot be modified. For example, "get" methods taint their return tag with the bit-wise `OR` of the parameters.

Other DIFT systems, targeted at Android, which *have* released software include [19, 18, 12, 28, 26, 13, 23]. However, none of these works utilize the foundational technique of smali byte-code instrumentation. These older tools have been somewhat prolific, because they are available for others to use and analyze [10]. Their approaches are fundamentally different from modifying smali byte-code though. And because of this they exhibit very limited compatibility across the Android ecosystem and over time.

3 Stigma System Design

Stigma [6] 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 "Shared Preferences" 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`[2], is used to extract the Dalvik byte-code (DEX[1]) from the application and convert it to the assembly-like `smali`[5] 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 6.4. 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 all of the application smali code. 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

[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 it is immediately re-compiled, at install time, from DEX to native machine code matching that of the install device via the DEX2OAT compiler. The app is then run on the device via the Android RunTime (ART).

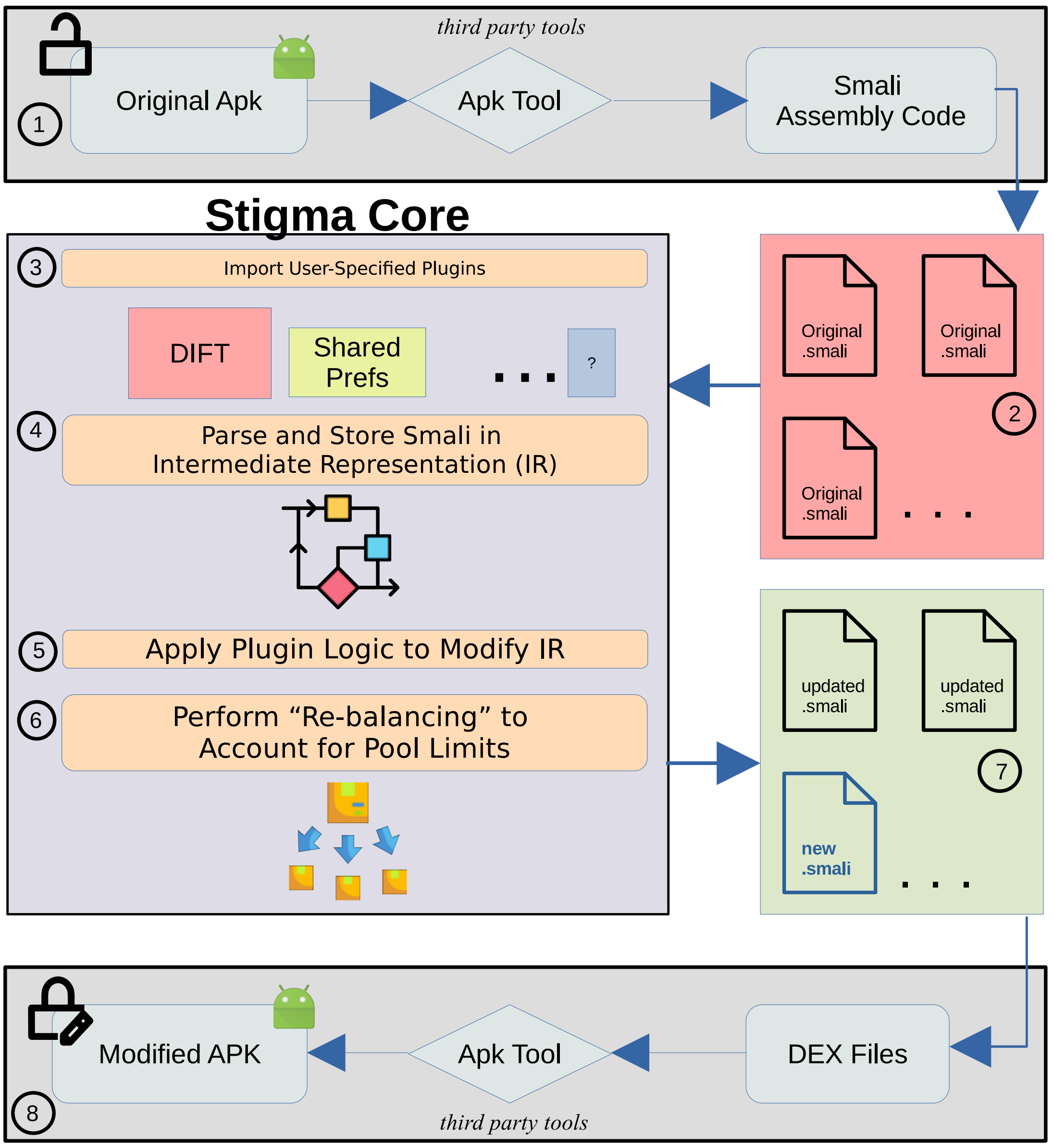

Figure 1: Stigma system architecture.

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 Allocating Registers

Any non-trivial plugin to Stigma will need to make use of registers to store data. Unfortunately, smali methods are defined (ultimately from the original Java code) such that they have a minimal number of registers. All of the registers originally allocated for the method are well utilized by the original functionality of that method. A key research challenge is how to allocate more registers for use by plugins.

Smali has only general purpose "v" registers in the range `[v0, v65535]`. But, an important aspect of smali is that most instructions can only operate using registers in the range `[v0, v15]`. For methods that require more than 16 registers, there are special instructions `move/16` and `move/from16` that can operate on the entire register range. Using these special `move` instructions it is possible to shuffle values in and out of lower- and higher- numbered registers in order to perform the desired instructions using lower-numbered registers only. Ordinarily, any necessary "shuffling" is handled entirely by the Java to dex compiler: `dx`. But, Stigma inserts new instructions / registers into methods *after* compilation. Adding registers to a method is not straightforward and has complex knock-on effects due to the register range limit of most instructions.

```
.method public foo(Ljava/lang/String;I)D
    .locals 5
```

Listing 1: Example of a smali method definition signature

The first line of any smali method (following the method signature) is always a "`.locals`" compiler directive indicating how many registers the method uses. Listing 1 shows a method `foo()` which takes a `String` and an `Integer` as input parameters and returns a `Double`. The `.locals` directive indicates that there are 5 local registers in the method, which act somewhat like local variables. This is used by the the dex2oat compiler (or the legacy Dalvik JVM) to help determine the stack frame size when this function is invoked. Each of the input parameters is also assigned a register (with a `pX` alias). `p0/v5` stores the implicit "this" parameter, since this method is not static. This makes for a total of 8 registers as shown below.

```
v0 (local register)    v1 (local register)
v2 (local register)    v3 (local register)
v4 (local register)    v5=p0 (1st parameter, 'this')
v6=p1 (2nd paramter, java/lang/String)
v7=p2 (3rd parameter, I=integer)
```

In order to allocate another register, we can re-write the `.locals` directive from 5 to 6. This creates a new, unused, local register `v5`, but also shifts `p0`, `p1`, and `p2` to occupy registers `v6`, `v7`, and `v8` respectively (a total of 9 registers). Multiple registers can be allocated this way, effectively expanding the size of the function call stack frame. Stigma increases the `.locals` value for every method in the project.

Adjusting the `.locals` directive in this way causes knock-on effects, since the original method code may refer to the registers in the original arrangement (e.g., expecting `v5` to contain the `'this'` reference). To resolve this, Stigma implements a procedure we refer to as ***growing*** the method. For methods that total more than 16 registers *after* increasing the `.locals` directive, the method may contain invalid instructions. This is because the original instructions of the method likely contain `pX` references which now, after increasing `.locals`, correspond to `vY` registers in which `Y` $>$ 15. Recall, most smali instructions do not support higher-numbered registers.

To solve this problem, Stigma adds additional `move/16` operations to move the parameter values back to their original locations. And, for all instructions throughout the method that reference any `pX` register, "`pX`" is replaced with the corresponding "`vY`." In the previous example `p0` would be moved from `v6` back to `v5` and all instances of `p0` in the method would be replaced with `v5`. After all the parameters are moved, `v8` is left completely unused. Following this procedure, it is straightforward to trivially grow a method by an arbitrary amount of registers. Of course, for some methods the newly allocated registers might be larger than `v15`.

3.1.1 JIT Register Acquisition

When interleaving instructions, registers must be selected for use therein. For many instructions, these should be lower-numbered registers. So, before passing control to a plugin, Stigma first attempts to acquire a certain amount of lower-numbered registers (`vX` where `X` < 16). Stigma makes a best-effort attempt, sourcing registers from three distinct categories. First, Stigma identifies any registers not yet used in the original program code (up until this point). Second the "top end" registers, that were allocated by ***growing*** can be used if they are lower-numbered. And, third, as a last resort, Stigma can acquire some registers by inserting necessary `move-*` instructions to free up lower-numbered registers at the precise point in the code at which they are needed. The values stored therein are moved to high valued registers, several of which are always available since the top X registers were created by Stigma earlier when ***growing***. They are moved back immediately following the new instructions inserted.

Should such `move` operations be necessary, it is important to use the correct `move` variant according to the data type being `move`-ed. Since the plugin / plugin author does not know the type(s) of all data stored in all registers, it is necessary for Stigma to insert any such `move` operations. Stigma performs rigorous static analysis to identify the type of every value in every register at every point in the method.

1. Word-sized (32-bit) values (int, float, boolean, etc.) require `move/16`.
2. Word-sized object references require `move-object/16`.
   This includes arrays and "special" objects such as Exceptions and "this."
3. Double-word-sized (64-bit) values (long and double) require `move-wide/16`.
   Such types actually use two consecutive registers (e.g., `v2` and `v3`) to store a single value since smali registers are all only 32-bits wide.

Stigma operates opportunistically and invokes the plugins whenever possible. But, in some extreme cases (for example when none of the register types are known) it may not be possible to acquire enough free, lower-numbered registers. In such circumstances plugin handlers cannot be invoked.

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 that are invoked as the original application code is linearly scanned. These handlers can be invoked at various key points in the target app. First they can be called when the app is launched. Second they can be called at the start of each method of original smali code. Finally, the most intricate method is for the plugin to specify a callback handler function individually for each of the 200+ types of smali instructions.

The callback handler function itself is written by the plugin author / user, and registers or "signs-up" with Stigma, specifying which key points it should be invoked on. These functions return new smali code that is inserted into the app at the key point.

```
1 invoke-virtual {v1, p1}, Ljava/lang/StringBuilder;->append(I)Ljava/lang/StringBuilder;
2
3 invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
4
5 move-result-object v1
```

Listing 2: Example of smali method calls to the `StringBuilder.append(int x)` and `StringBuilder.toString()` methods. The parameter passed (`v1`) in this example is the instance of the `StringBuider` on which the method is being called. After execution of line 5, `v1` contains a `String`.

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.

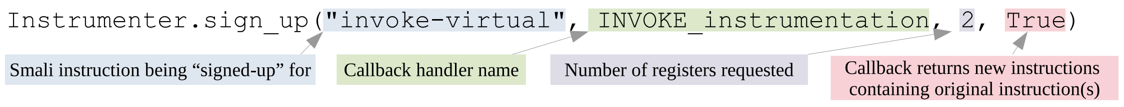
Instrumenter.sign_up("invoke-virtual", INVOKE_instrumentation, 2, True)
Smali instruction being "signed-up" for
Callback handler name
Number of registers requested
Callback returns new instructions containing original instruction(s)

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

Some common operations actually consist of two lines of smali code that must be used in sequence. Listing 2 demonstrates such an operation. A call to the `StringBuilder.append()` method consist of a single `invoke-virtual` instruction on line 3. Immediately following is a `move-result-object` instruction to capture the returned value. Any `move-result-*` instruction must immediately follow some `invoke-*` instruction, since the purpose of `move-result-*` instructions are to store the returned values of method calls. Should Stigma interleave other instructions between these two instructions the entire class will be made invalid.

To account for this case, both instructions are passed to the handler. When `sign_up()` is called, the final parameter passed is set to True, which allows the handler to include the original two instructions along with its proposed new lines. This allows the plugin to appropriately interleave instructions before the `invoke-*` instruction or after the `move-result-*` instruction (or both) depending on what is necessary / desirable.

4 DIFT Plugin

As a proof of concept, we design and implement 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 tag values. 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 amongst the original app instructions, to implement the logic of sensitive information marking and tracking.

4.1 Tag Origination

For (1) tag origination, Stigma identifies several key functions from the Android API that can be used to obtain sensitive data. For example, the method call `Landroid/location/LocationManager;->getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;`, 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 the tag value `2.0` for the register used to store the return value.

4.2 Tag Propagation

When a tag value is placed into 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. Our DIFT plugin needs to interleaves new smali instructions to move the tag value as well.

Some instructions do not necessitate tag propagation at all, such as `goto`, `if-eq`, `throw`, and `packed-switch`. But, most do. For example, roughly 85 of the almost 250 smali instructions are binary operations. Consider a binary smali instruction that might occur in an app (before any new instructions are added by any plugin(s)): `add-long v2, v7, v0`. This instruction adds the long values in registers (`v7,v8`) and (`v0,v1`). The result is stored in (`v2,v3`). Because `v7` and/or `v0` may contain sensitive information, the tag values associated with those registers must be propagated to the tag for `v2`. For a typical instruction, roughly 5 - 10 *new* instructions may be added to the program by the plugin to achieve this. An example of this case is shown in Listing 3.

```
1  # IFT INSTRUCTIONS ADDED BY STIGMA for ADD-LONG
2  const/16 v11, 0x0
3
4  sget v12, Lnet/sstorage/StorageClass2;->foo_v7:F
5
6  add-float v11, v11, v12
7
8  sget v12, Lnet/sstorage/StorageClass2;->foo_v0:F
9
10 add-float v11, v11, v12
11
12 sput v11, Lnet/sstorage/StorageClass2;->foo_v2:F
13
14 # IFT INSTRUCTIONS ADDED BY STIGMA for ADD-LONG
15
16
17 add-long v2, v7, v0 // original instruction
```

Listing 3: Smali code instrumented in order to propagate taint-tag values.

First, on line 2 the value 0 is placed into register `v11`. This is necessary or the application won't pass runtime verification done by the Java verifier. Specifically, instructions that operate on data (i.e., `add-float` on line 6) will not pass verification unless the operand register(s) contain value(s) of the correct type(s). Lines 4 - 10 obtain and merge together the tag values for registers `v7` and `v0` using simple addition. Lines 4 and 8 obtain the current tag for registers `v7` and `v0` respectively. Line 6 is technically unnecessary, but is present as an idiosyncrasy of our implementation. Note, registers `v11` and `v12` were introduced into the method exclusively to operate on tag values as described in Section 3.1. They are not used by the original method logic whatsoever ensuring that they can be used safely to store tag values.

4.2.1 Propagating Tags Across Function Calls

Well written Java code makes heavy use of methods. In smali code, methods are called using one of the `invoke-*` instructions and the result (if any) can be captured by an immediately subsequent `move-result` instruction. An example is shown in Listing 2.

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 (see Section 6.3. The examples shown in Listing 2 are calls to external methods, since `java/lang/StringBuilder` is provided by the run-time.

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 in the `move-result` instruction.

For external methods we do not have access to any of the method's smali code. Our solution is to combine the tags of the parameters passed using the `add-float` operation. The resulting tag value is then propagated directly to the register specified in the subsequent `move-result` instruction (when present).

4.2.2 Limitations

Depending on the instruction semantics, and the context of the instruction, it may be possible to achieve tag propagation with less overhead. Such optimizations are a key novelty of some related literature [22, 30], but are outside the scope of this work.

In our current implementation, array instructions (`new-array`, `array-length`, `aget`, `aput`, etc.) are implemented in the same primitive way that many other works handle them. A single tag is allocated for the entire array.

Some functions do not have a subsequent `move-result` instruction. For example, the call to `StringBuilder.append()` on line 1 of Listing 2. The `append()` changes the state of the String and returns nothing. We do not propagate any tags in such situations, leaving this to future work.

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 (a) retrieves the tag value associated with each of the input parameters and then (b) if the tag value of any parameter is not zero, writes an entry to the Android system log (logcat) alerting the user that sensitive information is being leaked. More sophisticated remediation, such as differential privacy analysis [27, 11, 29], is a potential area of future work.

4.4 Tag Storage

Where to store the tag data in an existing app/program is an open research question. To maintain broad compatibility, the tag information cannot be stored in the Dalvik virtual machine, or the ART runtime. We outline several possible locations below, which are compatible for a smali instrumentation based approach.

- Store tags in the fields of new classes added to the application (*Stigma implementation*).
- Store tags in a file on the filesystem that is world read/write-able.
- Store tags in the function call stack (expanding each stack frame to accommodate tags as necessary; a technique roughly described in other works [13, 30]).
- Store tags in a secondary, dedicated application that implements an Android `ContentProvider` and/or wraps a database.

It may seem attractive to store the tags in new fields added to the *existing* classes in the application. This likely breaks the logic of the application rendering it unable to compile and/or run. This is due to software practices such as object-relational databases, serialization, and reflection which require classes to have a certain set of specific fields. We found at least one app (Alibaba) that made assumptions about the instance fields of some classes in order to interact with SQLite schema. Additionally, some smali files / classes may not be allowed to carry any fields at all such as interface classes.

Internally, Stigma generates a list of `StorageClass` smali classes, each of which stores tags in public, static fields. Each `StorageClass` has a configurable limit of `m` fields. Plugins can call `add_taint_location()`, which automatically creates additional `StorageClass` instances when more then `m` fields are requested. An in-memory cache of mappings from method registers to their tag locations is maintained for fast retrieval while Stigma/the plugin is processing the app.

5 Shared Preferences Extraction Plugin

"Shared Preferences" 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 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, 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 Shared Preferences 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 Shared Preferences 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 will not have added any items to the Shared Preferences database.

6 Critical Concepts and Research Challenges

Some aspects of smali are esoteric and mysterious. We discovered many such details through careful analysis and reverse engineering. Below we summarize each of four different critical concepts discovered (Sections 6.1 through 6.6). This knowledge is critical for researchers building and working with systems that attempt non-trivial smali modification.

6.1 Obtaining Application Code

Android applications are packaged and distributed as APK files. These contain the application manifest (`AndroidManifest.xml`), and the assets of the application (images, sounds, data files, etc.). Most importantly, the application's DEX byte code is included in a file called `classes.dex`. For technical reasons, explained in Section 6.4, there may also be `classes2.dex`, `classes3.dex`, and so on. The format of DEX code is public knowledge [3], but it is not human readable or easy to parse or edit. When an application is installed on a modern Android device, the DEX code is immediately compiled by the `dex2oat` compiler to a binary executable compatible with the device's native architecture (e.g., ARM Cortex-A). When classes are loaded (just prior to run-time) they are validated by the Java verifier. Finally, classes are run with the Android ART runtime on the device.

There are popular tools such as Jadx [7], which can be used to de-compile Android app code. Such "DEX-to-Java' de-compilers can generate equivalent Java code, but they cannot recreate the actual original Java source code. An additional complexity is that many developers use obfuscation tools, and even the most popular de-compilation tools generally are not robust enough to fully de-compile arbitrary apps such that they can be re-compiled.

Stigma uses a pre-existing tool, `apktool` [2], which can be used to easily convert the binary DEX byte-code to a human readable assembly: smali [5]. The smali language consists of roughly 200 different instructions that are essentially a one-to-one mapping of the semantics of DEX opcodes. There is roughly one smali file for every pre-existing Java class in the project. After modifying the smali assembly code, the application can be re-packaged into an APK file using `apktool` again.

6.2 Cryptographic Signatures

To be run on an Android device, an APK file must be cryptographically signed. By unpacking, editing, and re-packing the application, the signature is destroyed. Fortunately, the resulting APK can still be signed by an arbitrary new key. Signing can therefore be achieved using the standard Java toolchain: `keytool` and `jarsigner`. It is important to note that after re-signing, the application is no longer signed by the original developer. As mentioned in [30] this may raise compatibility issues if the application checks its own signature, or if it coordinates with other applications running on the same device that expect to all be signed by the same developer. Instead of trusting the original developer, the user of the application must trust the new signer. In this context, that means trusting the authors of Stigma and this work. Because users are choosing to use Stigma, we feel this is a valid change to the chain-of-trust.

6.3 System Classes, Libraries, and Unattainable Code

Following the method described in Section 6.1 will not provide access to all of the code of the target app. Notable categories of unattainable code includes dynamically loaded code, code generated through Java reflection, and native (C or C++ via JNI) code.

A substantial amount of unattainable code is hidden in system classes such as `java/lang/String;`. The authors of TaintMan [30] propose a sophisticated reference hi-jacking technique in order to replace those system classes on the device with versions that already include DIFT code. In general, it is outside the scope of this work.

6.4 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." [3]. 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. At the same time, modifying smali code (e.g., to implement DIFT) will very likely add elements to the pools. To alleviate this the smali files are distributed into multiple dex files (`classes.dex`, `classes2.dex`, `classes3.dex`, etc.) such that none of the pools are overloaded.

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 [5] 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 toolchain we are able to reverse engineer the relationship between smali code and the four pools.

```
 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 4: 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.

Consider the code sample in Listing 4. 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 one string each. 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`.

6.5 Control-Flow

Adding new smali instructions may influence the control flow even when the new instructions do not interact with any of the existing program data and do not explicitly alter the flow (`if-*`, return, etc.). When any instruction is added inside a try/catch block, a new path may be introduced since that new instruction may cause an exception to be thrown. Consequently, new control flow paths may affect the type verification done during load-time by the Java verifier, *before* the newly added paths are even executed.

6.6 Code Offsets

Instructions such as `if-eqz` use a code offset as a target to jump to. Although the offset appears in smali code as a human-readable label (e.g., "`:catch_0`"), the compiler actually replaces all offset labels with hexadecimal values. Those values are stored using a signed short (16-bit), so it must not exceed the range (-32767, 32768) [3]. Of course a program may need to jump to an instruction that is more than +/- 32768 bytes away. In such cases the compiler will write smali code that will first jump to a relatively close location, and then immediately use a `goto/32` (32-bit) instruction to jump to the final destination. When stigma adds new smali instructions, the distance between jump instruction(s) and the jump destination(s) is inadvertently increased. So the 16-bit offset constraint may be un-intentionally violated resulting in invalid smali code.

7 Evaluation & Case Studies

To evaluate the compatibility across many commodity Android applications, we acquired 100 random popular applications and ran Stigma on them.

First, we downloaded 31 random, popular applications from `https://APKMirror.com` and, for each application, 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 some sort of 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, we downloaded an additional 67 random, popular applications from `https://APKMirror.com` and, for each application, we processed it with Stigma using 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 "Shared Preferences" data from 59% of them (29/49).

Overall, the ability of Stigma to correctly modify smali code is actually very high when considering the significant amount of changes it makes to an application. For example, when running Stigma with the DIFT plugin on our internal testing application "Stigma Test App" it modifies approximately 2300 files. Even though the application itself consists of only a handful of Java / Smali files, the support libraries are packaged into the app by the Android build system and Stigma modifies all of them. Stigma interleaves approximately 1.2 million new assembly instructions into these 2300 files without introducing any obvious bugs into the app whatsoever. For applications that we find to be not compatible, they typically use some advanced features of the Android framework.

7.1 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.

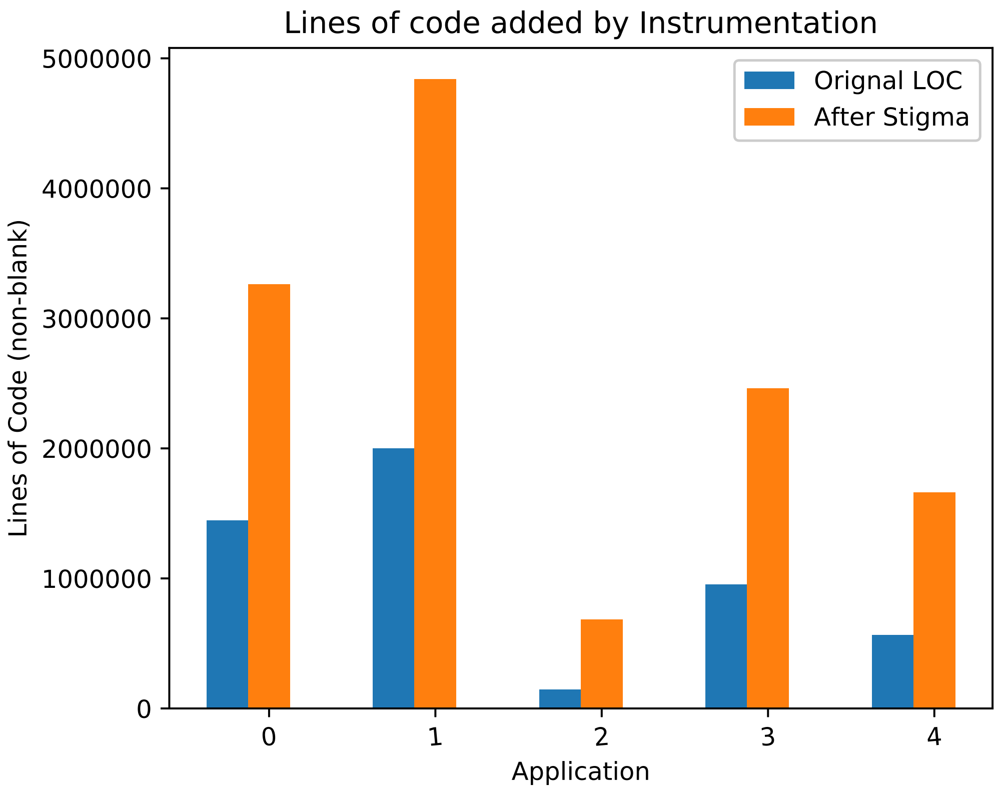

Figure 3: Lines of code added by instrumentation.

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 3.

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

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

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

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

4. "**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 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. 3 our implementation increases the lines of code by about a factor of 2.5x.

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.

7.2 Memory Usage

Lines of code added to the application don’t tell the full story of overhead. To investigate further we 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 does not add any.

7.3 CPU Overhead

A significant concern of Stigma is the hit to responsiveness and general performance of the application. Empirically, the user experience of the applications is not substantially impacted by the new code. 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.

Still, it is prudent to given an approximation of what type of overhead might be expected by inserting new instructions. We wrote a simple Android application, which executes arbitrary code similar to the code that might be inserted by Stigma to perform DIFT (i.e., mostly `sget`, `sput`, and `add-float` instructions). 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.4.

Generally, smali instructions incur a negligable amount of overhead. For an app of reasonable size of 2 million lines of (smali) code, *the entire code-base* could be executed in only ∼2/10 of a second. So, it is not surprising that Stigma does not appear to impact the responsiveness or performance of the apps we tested. Usually, Android app performance and responsiveness is not CPU-bound, but rather I/O bound. “Blocking” operations such as network download/upload, and database queries are a more prevelant concern for developers.

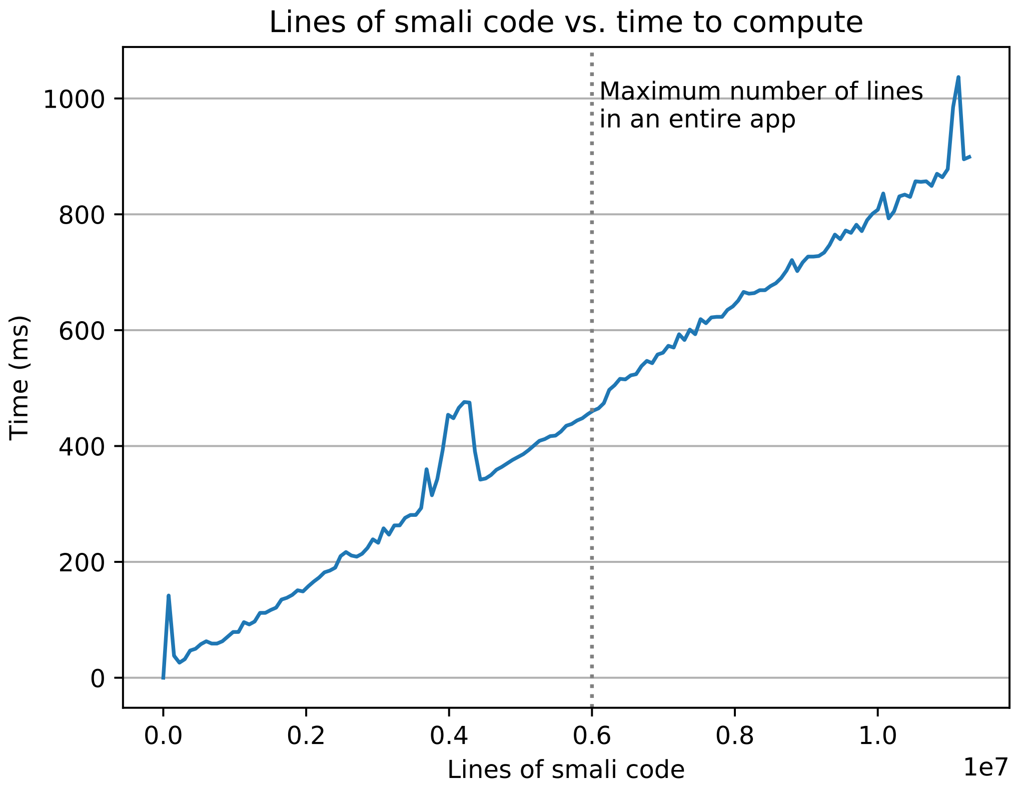

Figure 4: Time incurred by smali instructions.

8 Conclusion

Modifying the smali byte-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, higlight 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] Cydia substrate for android, 2014. Available At: `http://www.cydiasubstrate.com/`.

[2] Apktool project source code, 2022. Available At: `https://ibotpeaches.github.io/Apktool/`.

[3] Dalvik byte-code documentation, 2022. Available At: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`.

[4] Ddi, dynamic dalvik instrumentation toolkit, 2022. Available At: `https://github.com/crmulliner/ddi`.

[5] Smali project source code, 2022. Avialable At: `https://github.com/JesusFreke/smali`.

[6] Stigma source code, 2022. Available At: `https://github.com/fmresearchnovak/stigma`.

[7] Jadx project, 2023. Available At: `https://github.com/skylot/jadx`.

[8] Vialin project, 2023. Available At: `https://resess.github.io/`.

[9] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. Vialin: Path-aware dynamic taint analysis for android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1598–1610, New York, NY, USA, 2023. Association for Computing Machinery.

[10] 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,*, pages 294–305. INSTICC, SciTePress, 2019.

[11] Yang Cao, Yonghui Xiao, Li Xiong, Liquan Bai, and Masatoshi Yoshikawa. Priste: Protecting spatiotemporal event privacy in continuous location-based services. *Proc. VLDB Endow.*, 12(12):1866–1869, August 2019.

[12] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for android. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 481–494, San Diego, CA, 2014. USENIX Association.

[13] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 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, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[14] Shuai Hao, Ding Li, William G.J. Halfond, and Ramesh Govindan. Sif: A selective instrumentation framework for mobile applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, page 167–180, New York, NY, USA, 2013. Association for Computing Machinery.

[15] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 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, page 3–14, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Louise Matsakis. The wired guide to your personal data (and who is using it), Feb 2019.

[17] Moses Namara, Reza Ghaiumy Anaraky, Pamela Wisniewski, Xinru Page, and Bart P. Knijnenburg. *Examining Power Use and the Privacy Paradox between Intention vs. Actual Use of Mobile Applications*, page 223–235. Association for Computing Machinery, New York, NY, USA, 2021.

[18] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 451–466, Savannah, GA, 2016. USENIX Association.

[19] Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon P. Cox. Sandtrap: Tracking information flows on demand with parallel permissions. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 230–242, New York, NY, USA, 2018. ACM.

[20] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.

[21] Maxim Schessler, Eva Gerlitz, Maximilian Häring, and Matthew Smith. *Replication: Measuring User Perceptions in Smartphone Security and Privacy in Germany*, page 165–179. Association for Computing Machinery, New York, NY, USA, 2021.

[22] 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*, pages 370–379, 2014.

[23] 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, pages 331–342, New York, NY, USA, 2016. ACM.

[24] Jennifer Valentino-DeVries, Natasha Singer, Keller H. Michael, and Aaron Krolik. Your apps know where you were last night, and they're not keeping it secret. *The New York Times*, Dec 2018.

[25] Jennnifer Valentino-DeVries. Tracking phones, google is a dragnet for the police, April 2019.

[26] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914, 2015.

[27] Yonghui Xiao and Li Xiong. Protecting locations with differential privacy under temporal correlations. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1298–1309, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, aug 2012. USENIX Association.

[29] L. Yao, X. Wang, X. Wang, H. Hu, and G. Wu. Publishing sensitive trajectory data under enhanced l-diversity model. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pages 160–169, June 2019.

[30] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang. Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing*, 17(1):209–222, Jan 2020.

[31] 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, page 259–270, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. Condysta: Context-aware dynamic supplement to static taint analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 796–812, 2021.