NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES
"Computing Systems: Software and Hardware"

MSc THESIS

Integration of Doop's static analysis to ReDex optimizer

Lydia H.Zoghbi

Επιβλέποντες: **Yiannis Smaragdakis,** Professor
George Fourtounis, Research Associate

ATHENS

September 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
"Υπολογιστικά Συστήματα: Λογισμικό και Υλικό"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ενσωμάτωση της στατικής ανάλυσης του Doop στο βελτιστοποιητή ReDex

Λυδία Χ.Ζογμπή

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής
Γιώργος Φουρτούνης, Μεταδιδακτορικός Ερευνητής

ΑΘΗΝΑ

Σεπτέμβριος 2019

MSc THESIS

Integration od Doop's static analysis to ReDex optimizer

Lydia H.Zoghbi
R.N.: M1530

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Yiannis Smaragdakis,** Professor
George Fourtounis, Research Associate

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ενσωμάτωση της στατικής ανάλυσης του Doop στο βελτιστοποιητή ReDex

Λυδία Χ.Ζογμπή
Α.Μ.: Μ1530

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής
Γιώργος Φουρτούνης, Μεταδιδακτορικός Ερευνητής

ABSTRACT

Mobile devices have limited processing power, memory, and battery life; thus, optimizing mobile applications for better performance is critical for their successful deployment. However, since optimization requires high-level technical expertise, it is a daunting task for an application developer. Therefore, automatic code optimization techniques and tools are widely used to achieve high performance in applications for mobile devices. An Android application is written in the Java language and a well-known optimizer for Android applications is ReDex, operating on the bytecode level. It statically analyzes the application and, based on that information, it applies optimizations wherever possible. However, ReDex's static analysis is strict and conservative, and thus it may miss various optimization opportunities. For that reason, in this thesis we attempt to enhance ReDex's static analysis and the optimization passes it runs on an application. We use Doop, which is a framework for pointer, or points-to, analysis of Java programs, and runs deeper and more detailed analysis, providing us with more information that we can leverage. Our main focus is the method inlining optimization and how we can further expand it so that it can also apply to virtual methods.

SUBJECT AREA: Static program analysis and program optimization

KEYWORDS: static program analysis, doop framework, program optimization, ReDex, method inlining

ΠΕΡΙΛΗΨΗ

Οι κινητές συσκευές έχουν περιορισμένη υπολογιστική ισχύ, μνήμη, και διάρκεια μπαταρίας. Επομένως, η βελτιστοποίηση των εφαρμογών για καλύτερη απόδοση είναι κρίσιμης σημασίας για την επιτυχημένη χρήση τους. Όμως, η βελτιστοποίηση σε υψηλό επίπεδο απαιτεί μεγάλη εμπειρογνωσία, γεγονός που την καθιστά μια δύσκολη πρόκληση για τους προγραμματιστές. Συνεπώς, τεχνικές και εργαλεία για αυτόματη βελτιστοποίηση κώδικα χρησιμοποιούνται ευρέως για την επίτευξη καλύτερης απόδοσης για εφαρμογές κινητών συσκευών. Μια εφαρμογή Android γράφεται στη γλώσσα προγραμματισμού Java και ένας πολύ γνωστός βελτιστοποιητής για τέτοιες εφαρμογές είναι το ReDex, το οποίο λειτουργεί σε επίπεδο ενδιάμεσης αναπαράστασης κώδικα (bytecode). Αναλύει στατικά την εφαρμογή, και με βάσει αυτή την πληροφορία, εφαρμόζει βελτιστοποιήσεις όπου κρίνει ότι είναι απαραίτητο και δυνατό. Παρόλα αυτά, η στατική ανάλυση του ReDex είναι αυστηρή και συντηρητική, επομένως, χάνει αρκετές ευκαιρίες για βελτιστοποίηση. Για το λόγο αυτό, στα πλαίσια αυτής της διπλωματικής εργασίας, επιχειρούμε να ενισχύσουμε τη στατική ανάλυση του ReDex και τις βελτιστοποιήσεις που εφαρμόζει στην εφαρμογή. Χρησιμοποιούμε το Doop, ένα εργαλείο που αναλύει προγράμματα γραμμένα σε Java και μας δίνει διάφορες χρήσιμες πληροφορίες, όπως για παράδειγμα, σε τι αντικείμενα στη μνήμη μπορεί να δείχνει μια μεταβλητή (pointer analysis). Το Doop εκτελεί βαθύτερη και πιο λεπτομερή στατική ανάλυση και μας παρέχει περισσότερη πληροφορία για το πρόγραμμα που μπορούμε να την εκμεταλλευτούμε για καλύτερα αποτελέσματα. Κύριος στόχος μας είναι να εστιάσουμε συγκεκριμένα στο method inlining και πώς μπορούμε να επεκτείνουμε την υλοποίηση του ReDex, ώστε να μπορεί να εφαρμοστεί και για virtual μεθόδους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική ανάλυση και βελτιστοποίηση προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση προγραμμάτων, doop framework, βελτιστοποίηση προγραμμάτων, ReDex, method inlining

To my closest person and best friend..

ACKNOWLEDGEMENTS

I am deeply thankful to professor Yiannis Smaragdakis for offering me the opportunity to work on such an interesting idea, and also for his guidance, and for keeping me motivated. I would also like to thank Vassilios Samoladas, George Fourtounis, and George Kastrinis for sharing their experience with me, and for their support and assistance, whenever any obstacle occured during the development of my thesis.

September 2019

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

PREFACE

This thesis attempts to enhance ReDex's method inlining optimization pass, the Android bytecode optimizer developed at Facebook, based on the static analysis of the Doop framework. It was developed as a MSc thesis in the Department of Informatics and Telecommunications, at the University of Athens, between July 2018 and February 2019.

1. INTRODUCTION

There are many methods to automate program optimization, ranging from rewriting high-level language code up to completely rewriting the program's machine code. Many optimizations are nowadays performed by compilers, after the source code has been compiled to an intermediate representation. This is because lower-level languages have simpler expressions that are easier to optimize. After a sequence of transformations, they produce a semantically equivalent output program that uses fewer resources and/or executes faster. However, it is rare that the compiler alone will provide the optimal outcome, so, for even better results, there are several optimizers, which focus specifically on various kinds of optimizations. Commonly, an optimizer analyzes the code and generates useful information to conclude if an optimization is possible without altering the semantics of the program. Most of these optimizers, including ReDex, an already established bytecode optimizer, developed at Facebook, run procedural routines to analyze the code, aiming in finding patterns that satisfy particular criteria of each optimization; such an analysis is difficult to expand and maintain. On the other hand, Doop is a Datalog-based framework and the analyses logic is declaratively written, which is its greatest advantage, compared to alternative pointer analysis implementations. Also, Doop is much faster, and scales better. Doop offers a rich set of analyses and is more precise in handling various Java features, even reflection, than alternatives. In this thesis we aim to integrate Doop's sophisticated analysis into ReDex to enhance its method inlining and take advantage of more optimization opportunities, and further enrich it by also dealing with polymorphic callsites and virtual method inlining. We evaluate our work using well-known Android applications, and also compare how other ReDex passes behave after our enhanced optimization pass.

The rest of this thesis is organized as follows:

- In Chapter 2 we give some background information about Android, the ReDex framework, static program analysis, and Doop.
- In Chapter 3 we provide technical details about our approach.
- In Chapter 4 we present our experimental evaluation and the results of our work.
- In Chapter 5 we summarize this thesis, and provide conclusions.

2. BACKGROUND

This section provides useful information about the Android operating system and the format of the applications it hosts, the ReDex optimization tool, and the static analysis framework that we used.

2.1 Android and Dalvik/ART Virtual Machines

Android [1] is a Linux-based, open source operating system developed by Google and it was initially used mainly for smartphones and tablets. Android is one of the most popular computing platforms, and it's also the mobile platform with the greatest diversity of devices. However, its use is more extensive nowadays, especially with the IOT growth, and it also finds application in cars, electronic devices in homes and offices, and other smart gadgets and wearables, such as watches.

Android applications are developed using Android Studio, the official IDE for Android software developers. When the application is ready, developers can build it into an APK file and sign it for release. An APK (Android Package) file is an archive that contains all the necessary information for an application to run on an Android device. Such information includes the source code which is compiled into .dex files, manifest files, platform-dependent compiled code for various platforms, a directory containing resources, as well as a file containing precompiled resources, and a directory containing all assets used.

Android applications are commonly written in Java and compiled to bytecode for the Java virtual machine (JVM), which is then translated to Dalvik bytecode and stored in .dex (Dalvik Executable) and .odex (Optimized Dalvik Executable) files, as shown in Illustration 1.

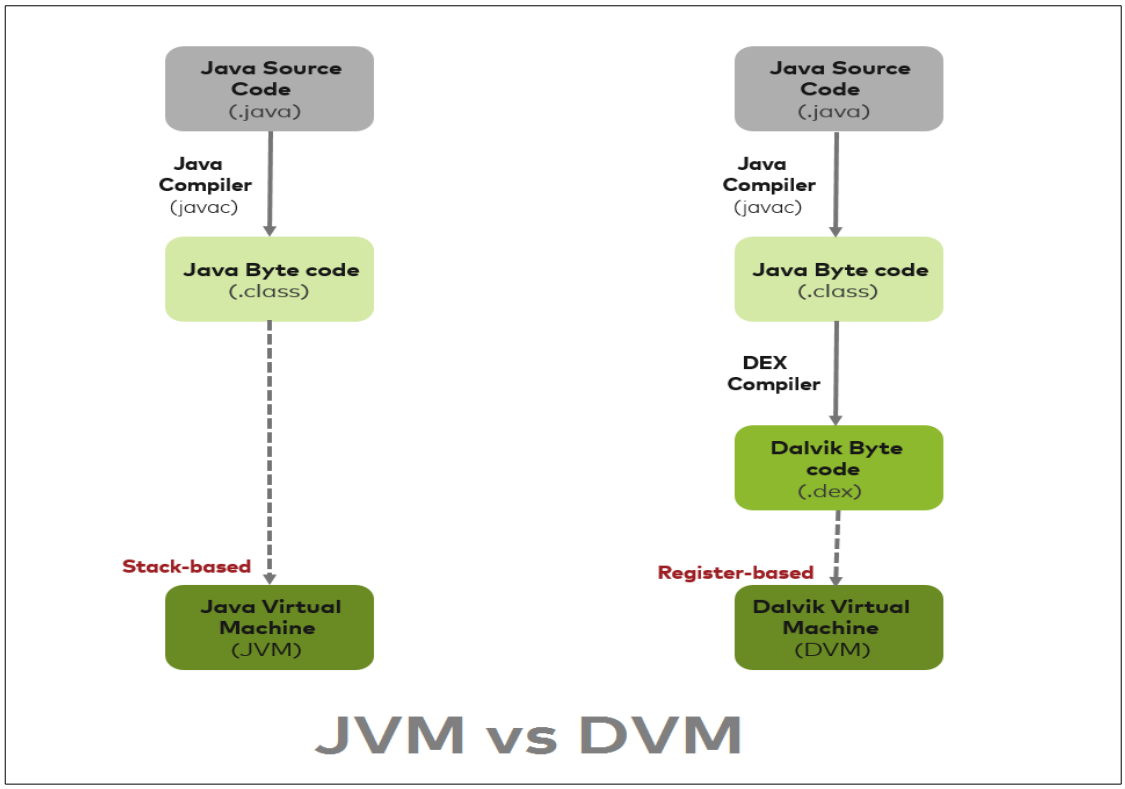

Illustration 1: DEX file creation

The Dalvik Executable format is more compact and benefits systems that are constrained in terms of memory and processor speed. In particular, JVM stores each class in an individual .class file. Each class file has a constant pool for strings, method names, class names, etc. If multiple classes reference the same string, then each .class file will have a copy of that string in its constant pool. This is not the case with .dex files, as the Dalvik Virtual Machine (DVM) stores multiple classes in a single .dex file, with a single constant pool. So, if multiple classes reference the same string, there will only be one copy of that string in the "global" constant pool for that .dex file.

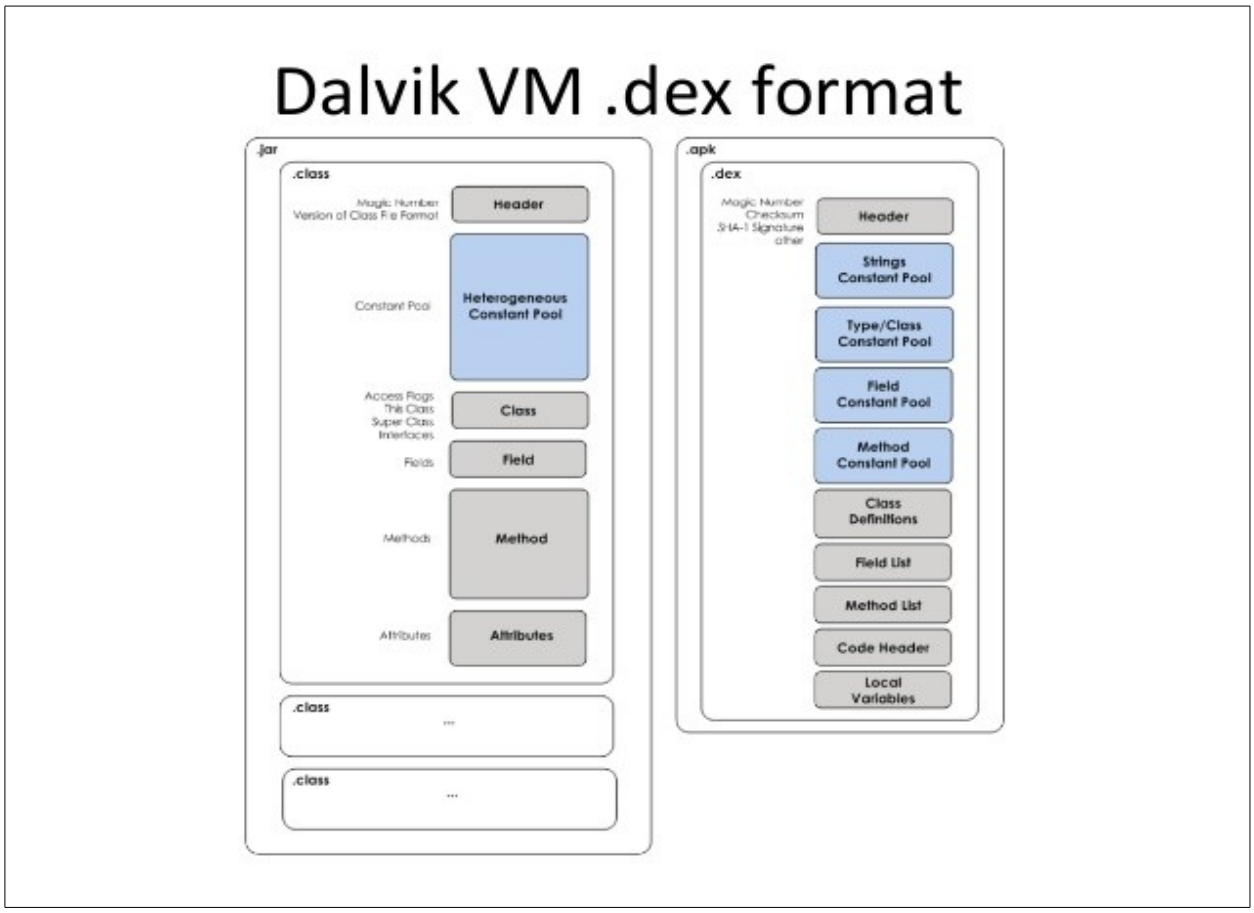

Illustration 2: DEX file layout

The DEX bytecode is translated to native machine code via either ART or the Dalvik runtimes, hence it is independent of device architecture. The Dalvik VM is an Android virtual machine optimized for mobile devices. The main difference between Dalvik and a typical Java VM is that the former is register-based while the latter is stack-based. Register-based VMs require fewer instructions and they are generally considered to exhibit faster startups and have better performance than their stack-based counterparts. Dalvik is a JIT (just-in-time) compilation based engine, meaning that compilation is done on demand at runtime; all the .dex files are converted into their respective native representations only when it is needed. However, that process is repeated every time the application runs, affecting the performance and the battery life of the device. In order to tackle that, beginning with Android 4.4, ART (Android Runtime)[2] was introduced as a runtime and since Android 5.0 it has completely replaced Dalvik. ART is the managed runtime used by applications and some system services on Android. ART and Dalvik are compatible runtimes running Dex bytecode, so apps developed for Dalvik should work when running with ART. ART introduces ahead-of-time (AOT)

compilation to entirely compile the application bytecode to machine code upon the application installation. ART largely increases the battery performance compared to DVM since the bytecode is not interpreted every time and also improves application performance, as it reduces the startup time and it makes the execution smoother and faster. However, it requires more storage space on the device as well as much more time to install the application. In order to tackle the aforementioned problems, Android 7.0 reintroduced JIT Compilation with code profiling along with AOT, and an interpreter in the ART, thus making it hybrid. The JIT compiler complements ART's current ahead-of-time (AOT) compiler and improves runtime performance, saves storage space, and speeds application and system updates[3]. It also improves upon the AOT compiler by avoiding system slowdown during automatic application updates or recompilation of applications during over-the-air updates.

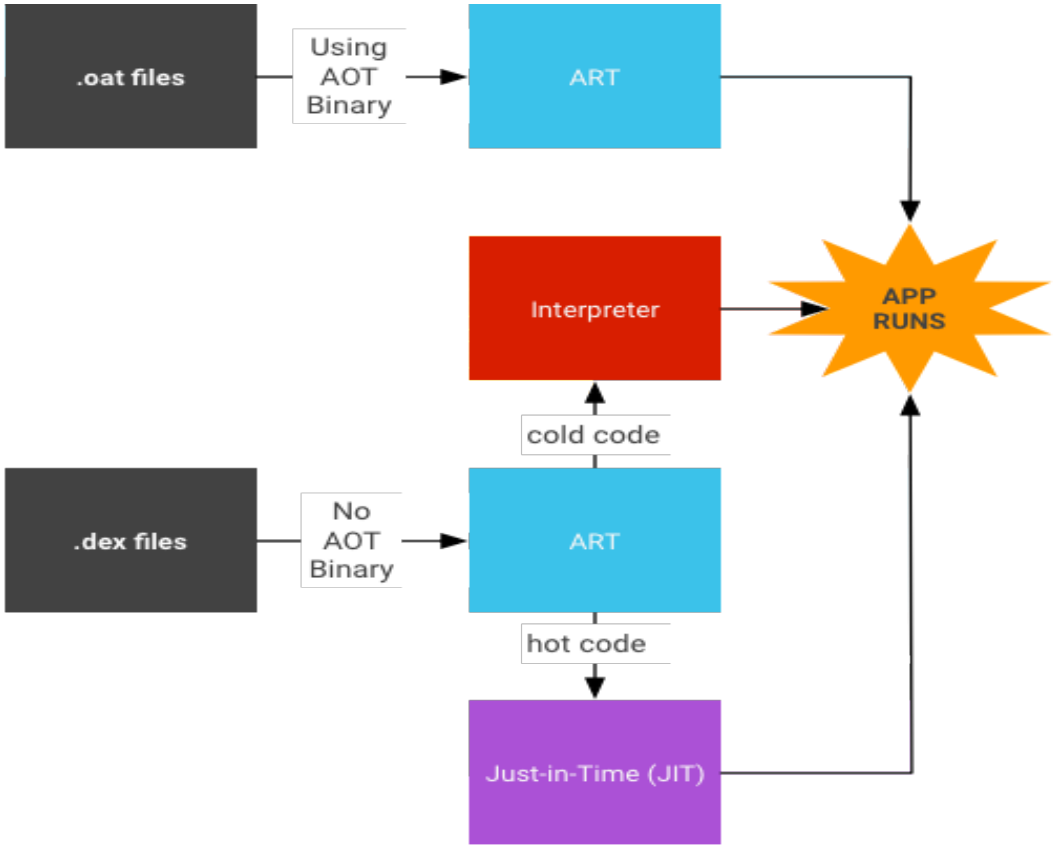

Illustration 3: JIT Architecture

2.2 ReDex

ReDex is an Android bytecode optimizer originally developed at Facebook [4]. It provides a framework for reading, writing, and analyzing .dex files, and a set of optimization passes that use this framework to improve the bytecode. At the time of the development of this thesis, ReDex only supported files with version 035 of the format, which is equivalent to Android 6 and prior versions. ReDex is conceptually similar to ProGuard [5], the default Android optimizer, in that both optimize bytecode, but ReDex operates on .dex files, whereas ProGuard on .class. Operating on .dex is more convenient and gives the opportunity to conduct more advanced and precise optimizations. For example, we can have knowledge of the number of virtual registers used by a method that is an inlining candidate, we are able to control the layout of classes within a .dex file, and it gives the opportunity for global, interclass optimizations

across the entire binary, rather than just doing local class-level optimizations. However, ReDex works alongside with ProGuard, as many ReDex passes require ProGuard configurations in order to be applied. For example, one of the optimizations that ReDex performs is to remove interfaces that have only one implementation. However, when there is a use of that interface through reflection or constructs like instanceof, then this is an unsafe removal, which should be prohibited by a ProGuard *keep* rule. Currently, ReDex provides support for simple *keep* annotations only for classes and interfaces.

An optimization pass is defined as a subclass of the abstract class Pass. The execution of passes happens in two steps. First, method Pass::eval_pass() is executed for each optimization, and then method Pass::run_pass() is executed. This allows each pass to evaluate its rules in terms of the original input, without other passes changing the identity of classes, since during the second phase the order of passes affects what each pass encounters. The optimization pipeline is organized as a series of stages, entering with the original .dex at the first stage and exiting with the optimized .dex at the final one. Each stage in the pipeline can be thought of as an independent “optimization plugin” that operates on the transformed bytecode from the previous stage, which gives us the opportunity to conduct multiple unrelated transformations in a row to produce the final, fully optimized outcome. ReDex offers a large variety of optimizations, but the subset and the order of the passes that are actually performed on an APK are specified by the configuration of each execution. Nevertheless, the order in which some optimizations are run matters significantly in some cases, thus we may observe that some passes may be defined more than one time in the configuration. Additionally, some optimizations completely rely on running along with others, in order to avoid further conflicts, especially during the type checking phase. For instance, if a pass, such as method inlining, that produces new bytecode, and thus introduces new registers, is run, then it is necessary to run RegAllocPass afterwards, which handles register allocation and eliminates redundant move instructions. Moreover, many passes can be configured in the configuration file, either for correctness or because some have various features that can be enabled or not, according to what the user wants to achieve and the depth she wants to reach. For instance, ReDex is quite aggressive about deleting things it considers unreachable, but often it does not know about reflection or other complex ways an entity could be reached. To ensure ReDex will not delete or rename a program element that it should not, it can be defined in the configuration file explicitly and ReDex will ignore any change to it. This is especially useful for methods and fields, because as mentioned above, classes and interfaces can also be protected through the ProGuard configuration file.

When optimizing apps, ReDex mainly aims to either reduce the size of the apk, or provide faster execution. Regarding the former, one of the first optimizations that were implemented targets replacing long human-readable identifiers with any shorter placeholder strings. This reduces how many bytes are dedicated to strings without affecting the application’s functionality. In general, fewer bytes also means faster download and install times, and less bytecode also typically translates into faster runtime performance. All other things being equal, less bytecode means fewer instructions to execute and fewer code pages to fault into memory, which is going to be a performance improvement for resource-intensive scenarios, like application cold start. One of the optimizations that fall into the latter category is method inlining, which improves execution time by omitting the method call overhead.

```
{
 "redex" : {
  "passes" : [
    "ReBindRefsPass",
    "BridgePass",
    "SynthPass",
    "FinalInlinePass",
    "DelSuperPass",
    "SimpleInlinePass",
    "PeepholePass",
    "ConstantPropagationPass",
    "LocalDcePass",
    "RemoveUnreachablePass",
    "RemoveGotosPass",
    "DedupBlocksPass",
    "SingleImplPass",
    "ReorderInterfacesPass",
    "RemoveEmptyClassesPass",
    "ShortenSrcStringsPass",
    "RegAllocPass",
    "CopyPropagationPass",
    "LocalDcePass"
  ]
 }
}
```

Illustration 4: Default ReDex configuration

```
"SimpleInlinePass": {
  "throws": true,
  "multiple_callers": true,
  "no_inline_annos" : [
    "Lcom/fasterxml/jackson/databind/annotation/JsonDeserialize;"
  ],
  "black_list": [],
  "caller_black_list": []
 }
```

Illustration 5: Configuration of SimpleInlinePass

2.3 Method inline benefits and ReDex's approach

Method inlining is an optimization technique used in compilers, that consists of inserting the complete body of a method wherever that method is used in the application. This optimization is one of the most common ones, as it comes with a handful of benefits. To begin with, it eliminates function call overhead and can provide significant speedup in programs in which small routines are called frequently. Another significant advantage of inlining is that afterwards various intraprocedural optimizations can also be applied. However, it must be used frugally, as too much inlining results in larger code size, which may be more inefficient, especially for mobiles with limited storage, than just calling the method in the first place. Also, if the final number of registers in the new caller context is big enough, then register pressure may occur. Most compilers won't inline a function, if its body is too large or if it has an excessive number of parameters. Another difficulty with method inlining is handling virtual methods. They can be inlined when the compiler or optimizer knows the exact type of the target object of the virtual function call, but that is not the general case, as virtual methods are used to achieve runtime polymorphism and are resolved during execution.

ReDex handles method inlining as one of the optimizations it applies on the bytecode, called SimpleInlinePass. It only concerns small and non-virtual methods, or virtuals that can be devirtualized beforehand. Initially, the pass gathers all the inline candidates, which are mainly selected based on their code size, or if they are explicitly annotated to force inline. Another criterion is if the method can be deleted or not, because afterwards the pass deletes the definition of all the inlined methods. Not all methods may be inlined, due to restrictions both for the caller and the callee. For this reason, all the candidates are passed to the inliner, which checks in more detail if they can be inlined without any conflicts, and if so, it eventually performs the optimization. This is based on the static analysis performed by ReDex, which, thus far, is primitive and rather strict, in order to avoid runtime errors. Some of the prerequisites that determine whether a method is inlinable are summarized below:

- If the callee is blacklisted in the configuration file the optimization does not proceed; typically, this is used to prevent deletion of a method that might be accessed through reflection.
- If the callee contains a catch block with an external catch type that is not public, inlining it is not feasible.
- The estimated final size of the caller should not be very large. However, we can configure the pass to not enforce a method size limit.
- If the callee contains an invoke-super to a different method in the hierarchy, and the callee and caller are in different classes, inlining an invoke-super off its class hierarchy would break the verifier.
- All the opcodes in the callee are analyzed to see if there is anything problematic, like accessing a virtual method or a field that is not known to the caller. In general, any type referenced in the callee body should be accessible in the caller context too; that means, that we might need to change the visibility of a method or a field. The problem occurs when a specific type is not known to ReDex, so to tackle this ReDex has predefined some known final types or well-known ones with no protected methods. In this case, the invocation is considered optimizable and the inlining proceeds.

Also, if a callee is inlined in multiple callsites, then its acceptable code size depends on the number of them; if it is being inlined in 2 callers, then it can have at most 7 instructions, whereas in the case of 3 callers the limit is 5 instructions.

One of the easiest cases where SimpleInlinePass comes in handy is that of wrapper methods. These are typically small methods added to provide a simpler class API to engineers or adapt methods to accept different lists of parameters. This also might include simple accessor methods (setter/getters) that are necessary to include in a class API but that might never even be called during runtime. During the initial compilation of class files, it isn't immediately obvious which of these functions are superfluous, but by the time we have the initial .dex file, it is trivial to determine which global optimizations are available.

2.4 Static program analysis

Static analysis is a way to gather useful information about a program without actually executing it and is the cornerstone of optimizing it. A very advantageous and fundamental type of static analysis is points-to analysis or pointer analysis [6], which determines information on the values of pointer variables or expressions, offering a static model of a program's heap. Basically, pointer/points-to analysis tries to answer "what objects can a variable point to?". Pointer analysis is a whole-program analysis with a modest performance cost, and the capability to scale to full realistic programs. A well-known and straightforward approach for points-to analysis was proposed by Andersen [7]. Andersen-style pointer analysis can be easily expressed as subset constraints, inducing inferences of the form "points-to set A is a subset of points-to set B". Datalog is a very suitable language to implement such constraints, as it is a logic programming language that achieves polynomial complexity, meaning that every Datalog program runs in polynomial time, and every polynomial algorithm can be written in Datalog. The analysis' logic is declaratively written and represented as various, probably recursive, relations that express information about the input program, such as program instructions, or type system information. Being a logic programming language, Datalog is largely based on first-order logic, which is a mathematic formalism; this allows us to write or enhance precise analyses at a higher level and without having to worry about the implementation of these algorithms, in terms of efficiency.

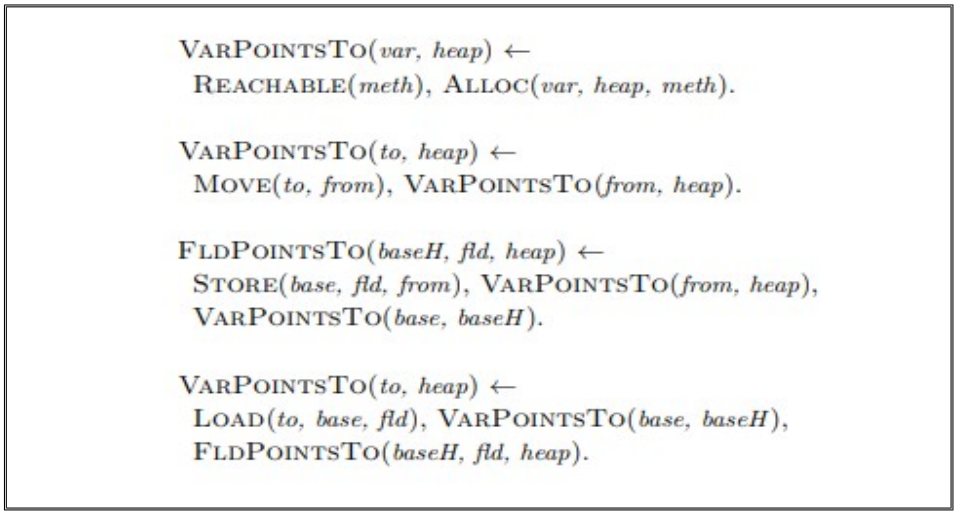

Illustration 6: Example of Andersen-style Datalog rules

Doop [8] is a versatile points-to analysis framework for Java programs, based on the use of Datalog for specifying the program analyses. It implements a range of different algorithms such as context insensitive, call-site sensitive, object-sensitive analyses and a lot of other variations of these algorithms. Also, it offers support for the complex semantics of Java, such as cast checking or reflection analysis. Doop is launched by specifying the type of analysis to run and the directory that contains the executable (JAR, APK, etc.) to analyze. At first, the facts are generated and imported to a database and then the specified analysis is run.

3. IMPLEMENTATION

3.1 ExplicitInlinePass

To apply our pass, named ExplicitInlinePass, we take advantage of ReDex's existing application programming interface (API) to manipulate the bytecode and reuse already implemented components, such as the inliner. In our case, we entirely omit the analysis performed by ReDex, as our main goal is to bring in external information from Doop, based on more extensive and precise analysis.

Initially, we parse the input file containing Doop's InvocationToInline relation, which has the following format. Each tuple indicates which method definition should be inlined, in which caller, and at which invocation site specifically. In the following example, '0' indicates the first call of the constructor in the class initializer of class okhttp3.CacheControl$Builder, and '1' the second call, respectively, as they appear in the bytecode in terms of original instruction order.

Table 1: Doop output format

<okhttp3.CacheControl: void <clinit>()>/okhttp3.CacheControl$Builder.<init>/0	<okhttp3.CacheControl$Builder: void <init>()>
<okhttp3.CacheControl: void <clinit>()>/okhttp3.CacheControl$Builder.<init>/1	<okhttp3.CacheControl$Builder: void <init>()>

First of all, we ensure that both the caller and callee are methods with definitions known to ReDex and we gather them using ReDex's API. Inlining is performed in a bottom-up manner, so that each method is fully resolved when it is being inlined to avoid extra work. Also, we discard tuples that indicate inlining a recursive method or methods that form a cyclical call chain, as such an action would lead to an infinite loop and ReDex would not terminate properly. Regarding the recursive methods, however, we can handle cases where there is tail recursion, meaning that the recurcive call is the last instruction of the method body. We perform tail recursion elimination, by transforming the recursion to a loop, and then we can proceed with inlining that method. In case of cyclical call chains, we just skip optimizing the callsite where the cycle occurs, and not the whole call chain. For example, if we have the following sequence of invocations, at first, C will be inlined in B, and then B will be inlined in A, accordingly. However, A will not be inlined in C, because at this point a cycle in the call chain is introduced.

$$A \rightarrow B \rightarrow C \rightarrow A$$

Some inlinable methods determined by Doop are also considered so by ReDex's analysis; we want to ensure that the file parsing mentioned above will take place before any other optimization pass makes changes to the bytecode, especially SimpleInlinePass, as it might mess up the invocation order of a method. To illustrate this with an example, imagine having the following code snippet:

```
class A{
   public void foo(int x){
      System.out.println("x in foo = " + x);
       bla(x+1);
   }

   public void bla(int x){
      System.out.println("x in bla = " + x);
   }
};

class B{
   public void bar(){
      foo(5);
      bla(10); //doop invocation identifier will be 0 here for method bla
   }
};
```

Illustration 7: Example for parsing Doop's output during EvalPass

Inside method bar, Doop will assume that the invocation offset of bla is 0. If SimpleInlinePass runs first and inlines foo inside bar, then another callsite for bla occurs first, making our point of interest appear second. If we parse the transformed bytecode in order to locate the invocation sites retrieved based on the initial bytecode the results will be inconsistent, inlining bla on the invocation that occurred from foo's body and not the proper one, causing potentially further conflicts, especially in case of method overloading where the wrong number and type of parameters are passed. This is why the whole parsing process described above is carried out during the eval_pass defined for our optimization. After the inlining is performed by our pass, the definitions of the inlined methods are deleted so that we do not bloat the bytecode.

3.2 Virtual method inlining

Polymorphism is one of the most powerful features of object-oriented programming and in Java it is enabled by class inheritance. Runtime polymorphism arises from method overriding and requires dynamic method dispatch. When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses)

of that method is to be executed based on the type of the object being referred to at the time the call occurs. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

In case an inline candidate is a virtual method, then the Doop analysis will provide us with the method definitions that are likely to be called at this callsite during runtime. We cannot always know statically the actual type of the object upon which that method is called; thus, we should proceed with the inlining with the assistance of guards. Essentially, guards are if-then-else blocks that check the actual type of the aforementioned object during execution, and, according to that type, the appropriate method body is run. To delve into more details, the possible types that we have to check are that of the base class or of any of its subclasses. However, we sort them in order to first inspect if the object type matches that of the subclass(es) and then the superclass, as any subclass object would also match the superclass types, otherwise. The first parameter passed to the method at the original invocation site holds the object, or alternatively it is the “this” pointer. We use that specific register to check it upon the first possible type with the “instanceof” Dalvik opcode and then we proceed with an “if..elseif..else” statement to apply the inlining accordingly. In case the result is true at runtime, the inlined method of that type will be run, otherwise, the execution flow will branch to the false-block and the alternative definition will be executed. Also, to tackle some unsoundness from Doop, if none of the above conditions checks true, then in the final “else” statement we rewrite the original invocation, so that the app will be fully functional, as it was in the first place. However, that still does not eliminate unsoundness entirely, because of reflection. For example, imagine loading at runtime a another subclass, that we have no information about statically; then, the invocation will call the definition of the base class, which is not the case, as it should be called the definition of the newly loaded class. Rewriting the original invocation instruction causes further increase to the size of the APK, because the method definitions are not deleted, as it happens with SimpleInlinePass. However, RegAllocPass balances out that increase, because it removes many move instructions that occur during inlining, and other passes further optimize the altered caller body, removing fields or unnecessary instructions.

```
.line 29
add-int v3, v2, v1

invoke-virtual {v0, v3}, Ltest/TestApp$A;->meth(I)I

move-result v3
```

Illustration 8: Example of virtual method call to be inlined

Illustration 8 shows the initial bytecode, with the virtual method call that will be replaced after our pass. Illustration 9 shows the outcome, after the guarded inlining. Specifically, it illustrates exactly how the type of the object is checked first, and according to that, the corresponding method is inlined.

```
.line 29
add-int v1, v4, v2

instance-of v0, v3, Ltest/TestApp$B;

if-eqz v0, :cond_1

.line 43
rem-int/lit8 v0, v1, 0x5

if-nez v0, :cond_0

.line 44
add-int/lit8 v1, v1, 0xa

.line 29
:cond_0
goto :goto_1

:cond_1
instance-of v0, v3, Ltest/TestApp$C;

if-eqz v0, :cond_3

.line 52
rem-int/lit8 v0, v1, 0x2

if-nez v0, :cond_2

.line 53
add-int/lit8 v1, v1, 0x2

.line 29
:cond_2
goto :goto_1

:cond_3
invoke-virtual {v3, v1}, Ltest/TestApp$A;->meth(I)I

move-result v1
```

Illustration 9: Example of guarded inlining

It is worth mentioning that all the execution paths that occur from the additional bytecode for guarding, should use the same register, in case we have returned values. This is important, because later that register will be used at specific instructions, meaning that it should have a specific type. ReDex conducts some verification checks before repacking the optimized APK and in case of inconsistency, it will prevent the transformation; the type of the register would errupt to bottom, meaning that it could be anything, which breaks the verifier. This is shown in the example of Illustration 9, where all branches store the result in register v1. Also, the red frame shows how the inilitial invocation is preserved for safety reasons at runtime. Finally, when we inline all the possible target methods to a virtual callsite, we do not delete their definitions afterwards, as mentioned above. We follow a more conservative approach here, as these methods may also be called by other program points, as well.

4. EXPERIMENTAL EVALUATION

In this section, we present the results of our work through experimental evaluation. Some of the key points we focused on are:

- to determine how our pass outperforms ReDex's SimpleInlinePass,
- how other passes benefit from ours,
- how is the final size of the APK affected,
- the runtime benefit of the transformed application

For our experimentation we used some well-known and widely used applications; Specifically, Instagram, Signal, Chrome, Viber, and WhatsApp. Also, to assess the performance tradeoff between virtual call or polymorphic inlining with guards, we wrote a micro-benchmark to measure the execution time in both cases.

For our analysis in Doop we focus mainly on the size of the methods to be inlined, in relation with the number of the callsites it will be inlined into. This is a similar approach to what ReDex does, however, the notion was to find the appropriate size so that we handle bigger methods than ReDex, but still small enough to not increase heavily the final size of the application. Bigger methods have more information and offer the opportunity to other optimization passes to perform more drastic transformations, contrary to ReDex's approach, where most of the inlined methods are accessors, constructors, or synthetic and wrapper methods introduced by the compiler. Shortly, the key points of our Doop analysis are:

- If a method will be inlined in one caller, its size is at most 20 instructions, for two callers it is limited to 10 instructions, and for three callers it drops under 6. Methods with at most 3 instructions are small enough to be considered inlinable in any case.
- Methods that will be inlined in virtual callsites that have 2 possible targets can be at most 10 instructions long, and in case we have three targets, the maximum size is restricted to 4 instructions.

4.1 ExplicitInlinePass performance

One of the metrics to evaluate our work, is to consider how many methods are inlined by ExplicitInlinePass compared to SimpleInlinePass. We already expect the numbers to be higher, but it is also important to ensure that the new opportunities that arised from Doop are handled properly and are reliable. Doop's analysis is very sophisticated and can detect more complicated cases than ReDex. With our pass, as already mentioned, we only delete method definitions inlined in monomorphic callsites. The most objective way to compare the two passes, is based on the total number of methods inlined, and the number of methods deleted as well. Table 2 and Illustration 10 show the results of ExplicitInlinePass when run alone, for the various applications, and presents some statistics about the invocation sites. The polymorphic callsites are strictly handled by our pass only, and is a clear advantage over ReDex's inlining policy.

Table 2: ExplicitInlinePass' invocation sites that will be inlined

Invocation types	WhatsApp	Viber	Instagram	Chrome	Signal
Static	6673	10677	1007	1224	5570
Direct	6150	12419	5267	3064	6274
Interface	942	2818	885	780	1129
Super	148	559	116	250	176
Virtual	2791	14064	1474	1843	8978
Total callsites	16704	40537	8749	7161	22127

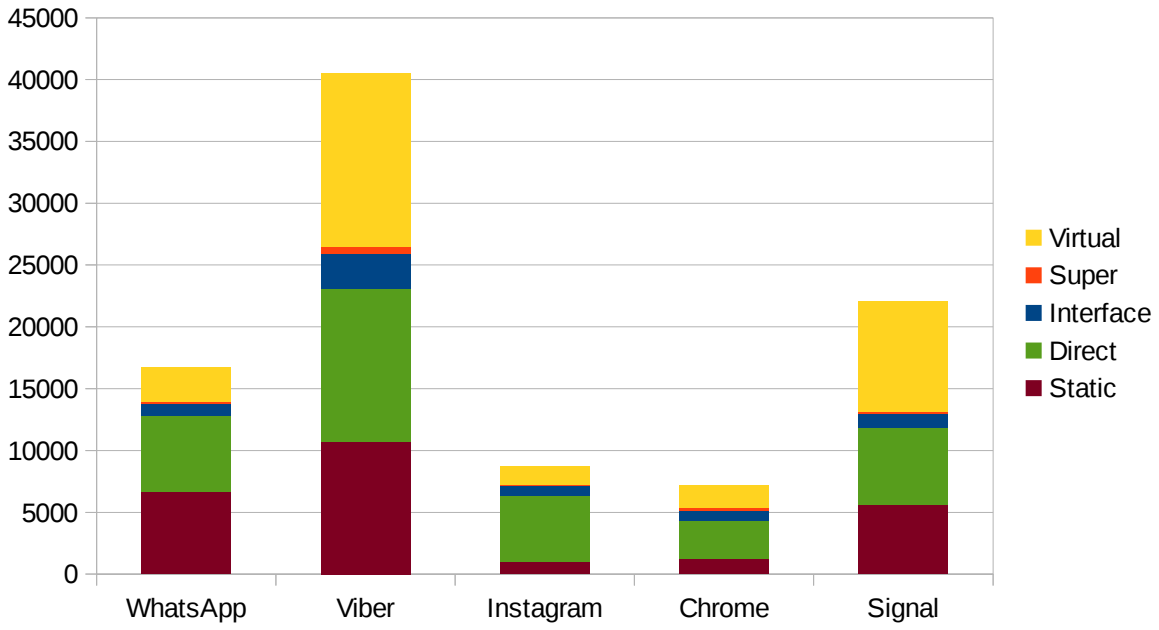

Illustration 10: ExplicitInlinePass' results

Table 3 and Illustration 11 present the total number of methods that are inlined by SimpleInlinePass and ExplicitInlinePass. Our pass shows significant increase, because even though Doop's analysis gathers small methods, their size is bigger than the size of those gathered from ReDex. ReDex's policy mainly inlines synthetic wrappers, bridge methods, small constructors, and setter/getter methods, which consist of, more likely, at most 7 instructions, as also mentioned above. Also, Doop and ExplicitInlinePass dominate at polymorphic invocation sites.

Table 3: Number of inlined methods

	SimpleInlinePass	ExplicitInlinePass
WhatsApp	5725	17593
Viber	17434	42199
Instagram	968	9547
Chrome	1096	7536
Signal	11338	24193

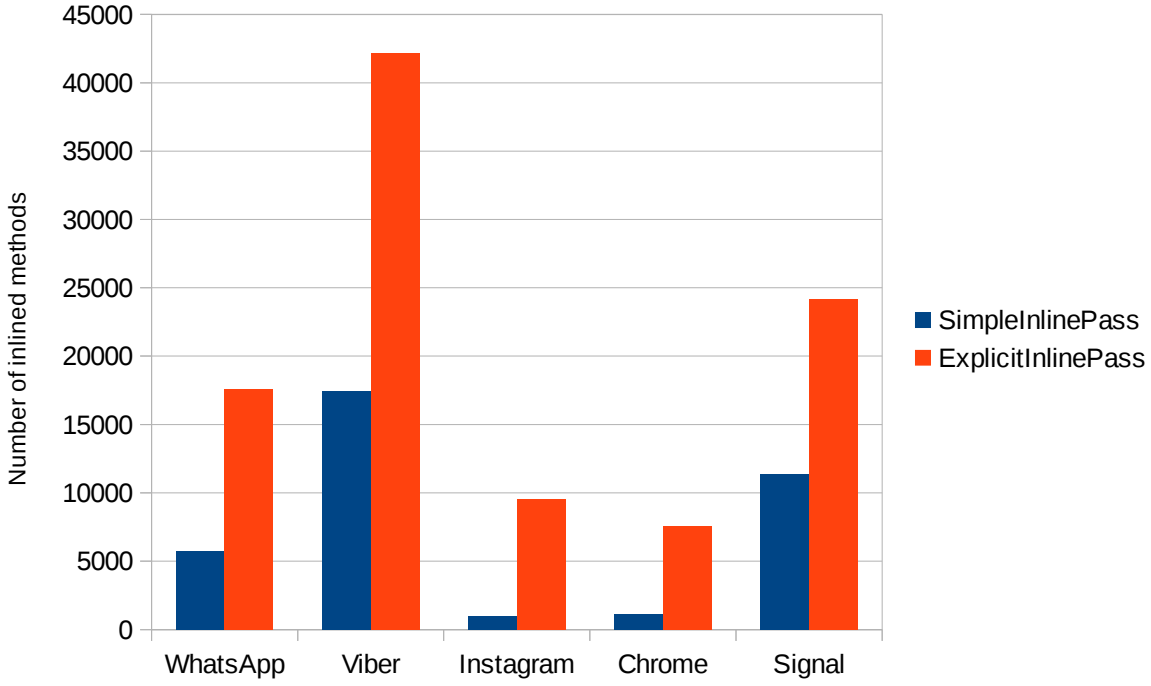

Illustration 11: Comparison of inlined methods between SimpleInlinePass and ExplicitInlinePass

Table 4 shows the number of methods removed from each pass respectively, and Illustration 12 also illustrates the aforementioned results. SimpleInlinePass aims to not only improve runtime performance by reducing method calls, but also to maintain the final size of the application almost the same, without increasing it through inlining. For this reason, ReDex inlines strictly very small methods, and also deletes their original definition, so the outcome is almost the same in terms of bytes. However, there are some methods that cannot be deleted, thus the number of methods that SimpleInlinePass removes is not identical with the number of methods it inlines, but the difference is relatively small. On the other hand, our pass, only deletes methods inlined in monomorphic callsites, and that explains the big discrepancy between the numbers of Table 3 and Table 4. We will present in section 4.3 how that affects the APK size.

Table 4: Number of deleted methods

	SimpleInlinePass	ExplicitInlinePass
WhatsApp	5664	11170
Viber	17182	27101
Instagram	953	5769
Chrome	936	4074
Signal	11244	12716

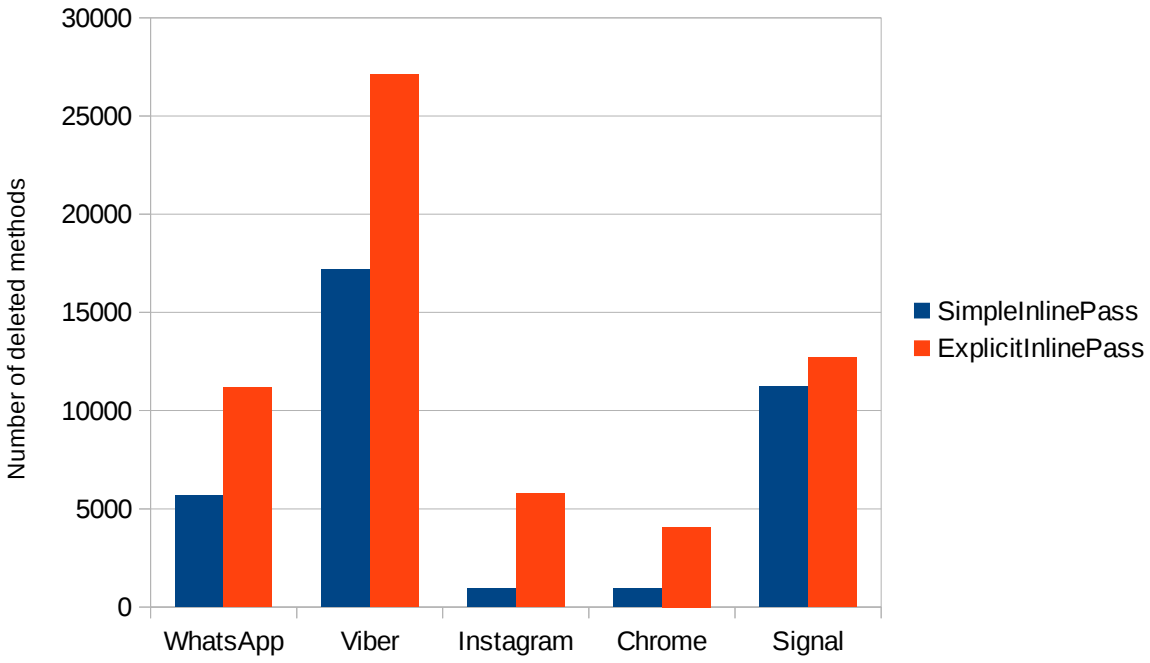

Illustration 12: Comparison of deleted methods between SimpleInlinePass and ExplicitInlinePass

4.2 Benefit of ExplicitInlinePass to other passes

Method inlining can create more opportunities to other optimizations as well. Some of the other ReDex's passes that benefit the most from our own are the following:

- **ConstantPropagationPass**, which propagates constants both intraprocedurally and interprocedurally, and also eliminates field writes if they all write the same constant value. Moreover, it removes dead branches from the control flow graph, based on whether the last instruction in a basic block is a dead if-instruction (i.e the branch is always taken or never taken). In our case, the extensive inlining we apply makes room for even more optimization opportunities.
- **CopyPropagationPass**, eliminates writes to register that already hold the written value. Also, it finds all alias registers and replaces them with one representative, which leads to more compact move instructions and also gives the chance for dead code elimination to optimize the bytecode more efficiently. It is also based on intraprocedural analysis, and thus after our pass this optimization operates better.
- **LocalDcePass**, which eliminates dead code using a standard backward dataflow analysis for liveness inside a method body.
- **RemoveUnreachablePass,** which removes unreachable classes, fields and methods. Likewise, inlined methods may no longer be reachable from other points in the application code, and their definitions are removed.

In the following subsections, we present some statistics to evaluate the aforementioned optimizations, comparing their performance when run alone and when run after our own. However, the fact that we do not delete all the method definitions, as described in

section 3.2, introduces some noise to the actual optimization results for the passes ConstantPropagationPass, CopyPropagationPass, and LocalDcePass. Thus, in order to evaluate the effect of our pass on those passes, we temporarily delete all the method definitions we inline, so that we do not encounter optimizations applied on the original definitions during our experiments.

4.2.1 ConstantPropagationPass

Table 5: Results for ConstantPropagationPass

	Before and After ExplicitInlinePass									
	WhatsApp		**Viber**		**Instagram**		**Chrome**		**Signal**	
Branches propagated	31	65	68	273	26	92	18	88	95	210
moves_replaced by_const_loads	17	17	12	13	2	2	0	0	119	148

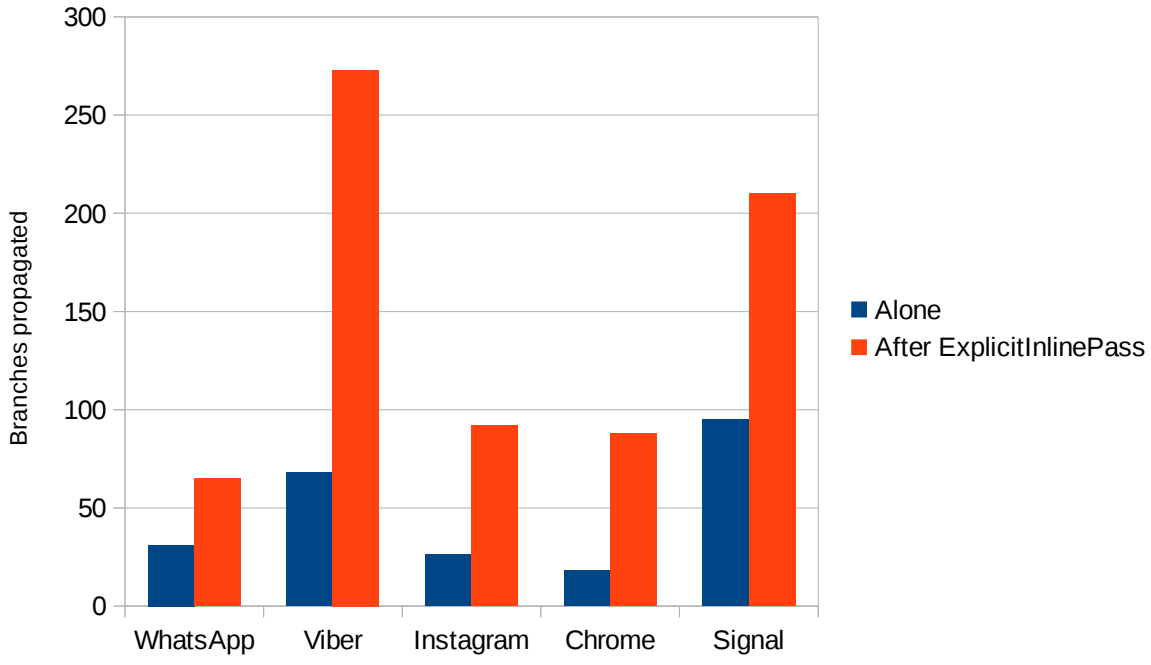

Illustration 13: Number of branches propagated before and after ExplicitInlinePass

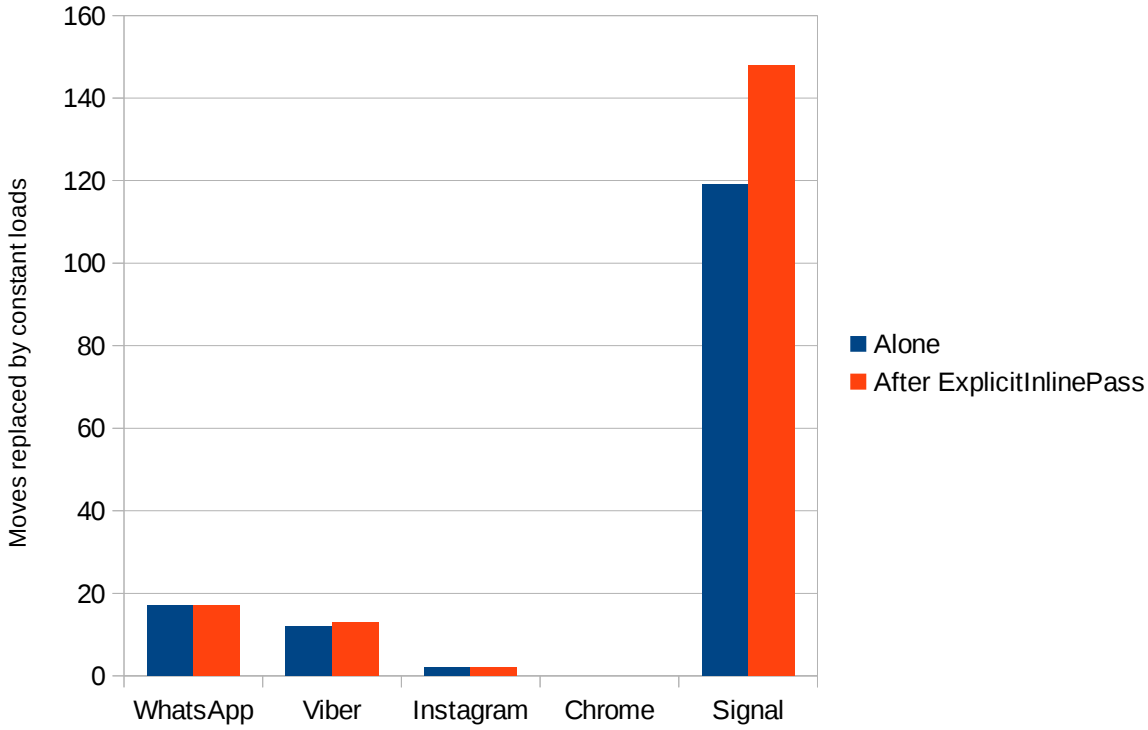

Illustration 14: Number of move instructions replaced by constant loads

It is clear that the significant benefit comes from the elimination of redundant branches. The results are more promising for Viber, with almost 120% increase, and Signal, with 75%, as they are the biggest applications with multiple dex files. This difference in the results can be explained considering that, after inlining it is easier to extract information about how values flow between variables and conclude whether the condition of an if-instruction, at specific points in the method, is always true or false. In such cases, the branches are removed. Without method inlining, such information is difficult to gather, as it needs flow sensitive program analysis, which is a demanding and hardly trivial task. Regarding the propagation of constants, it is presumable that the results are in most cases identical, because constant propagation is also performed interprocedurally, and thus, already handles most of the opportunities.

4.2.2 CopyPropagationPass

Table 6: Results for CopyPropagationPass

Before and After ExplicitInlinePass										
	WhatsApp		**Viber**		**Instagram**		**Chrome**		**Signal**	
redundant move instructions	5006	5223	6884	7188	7963	8048	1669	1727	10534	10777
source registers replaced with representative	6650	48044	9791	111430	6804	32753	2366	21963	16513	72008

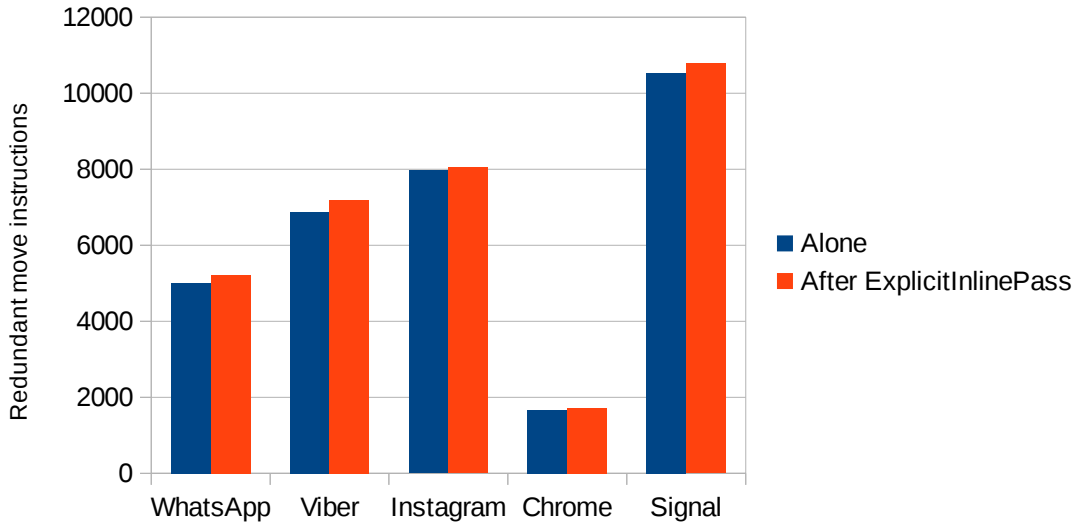

Illustration 15: Number of redundant move instructions

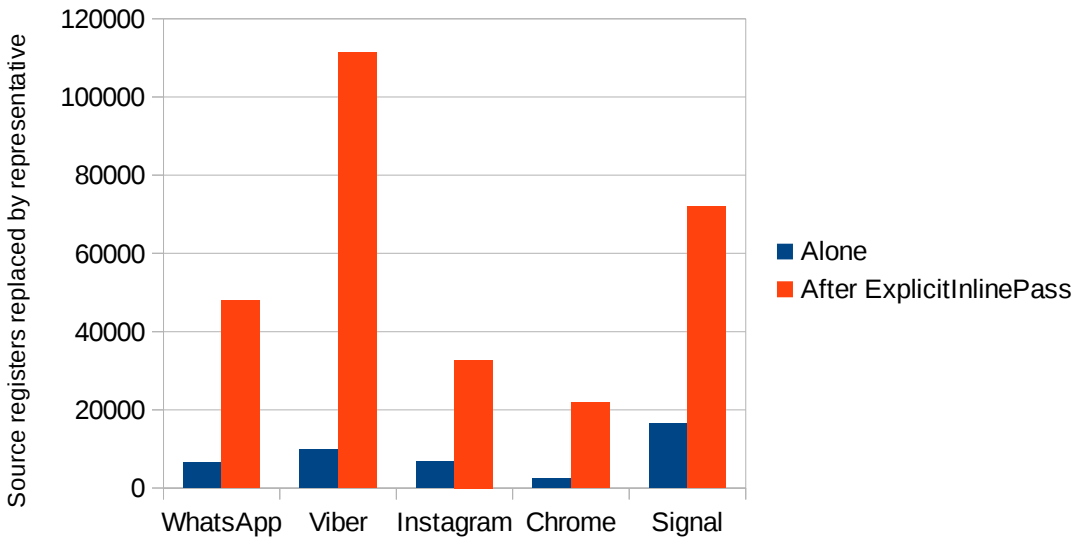

Illustration 16: Number of source registers replaced by a representaive

We observe that there is a significant improvement to the performance of CopyPropagationPass. This arises from the fact that intraprocedural analysis benefits incredibly from method inlining, as it has more information about how data flows and can conclude to more alias registers. When CopyPropagation is run after ExplicitInlinePass, some of the moves that are redundant are the ones introduced from the inliner, as it copies both the method arguments and the return value to new registers. Also, these new registers are aliases with the ones that held initially the method arguments passed during the invocation, and the return value after the call, thus the pass can create large sets of aliases and replace most of them with one representative register.

4.2.3 LocalDcePass

Table 7: Results for LocalDcePass

Before and After ExplicitInlinePass										
	WhatsApp		**Viber**		**Instagram**		**Chrome**		**Signal**	
Dead instructions	1567	4634	4099	13742	2542	4579	1748	3772	2859	9308
Unreachable instructions	0	18	6	152	0	58	0	20	0	341

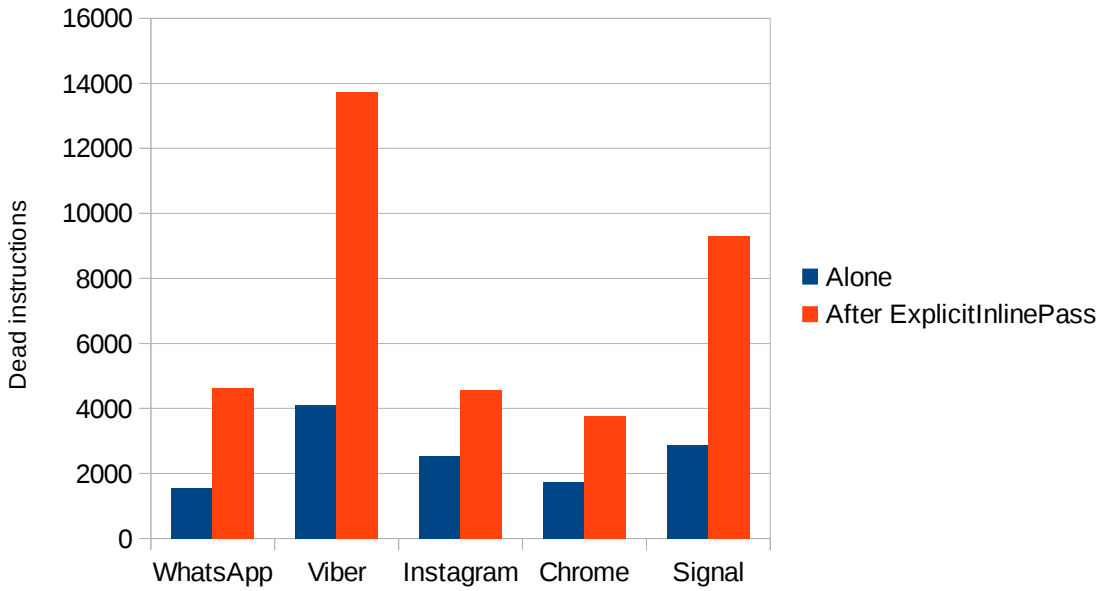

Illustration 17: Number of dead instructions

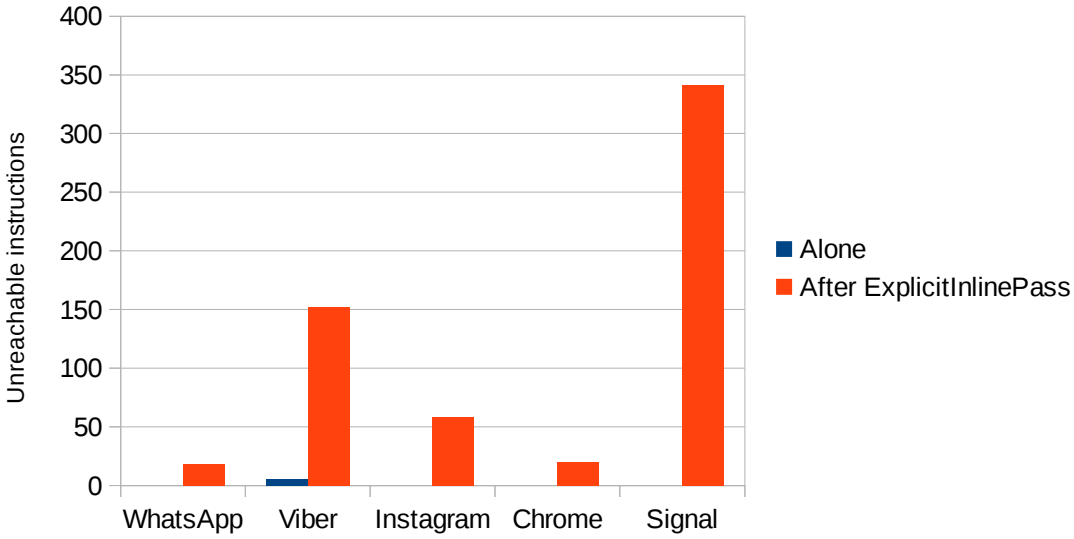

Illustration 18: Number of unreachable instructions

This pass removes dead and unreachable code. It has a similar benefit from our pass as constant and copy propagation, based on the fact that it performs intraprocedural analysis, like the aforementioned passes. So, extensive inlining contributes to encounter more optimization opportunities, as more code is analyzed within a method body and more information about it can be extracted.

4.2.4 RemoveUnreachablePass

Table 8: Number of items deleted

Before and After ExplicitInlinePass										
	WhatsApp		**Viber**		**Instagram**		**Chrome**		**Signal**	
classes	6719	3831	8989	8918	9979	10090	6446	6485	10234	10216
fields	23319	14954	42168	41329	31051	31132	17317	17351	58311	58141
methods	43361	25121	76982	62704	41282	36008	35355	32146	84312	75711

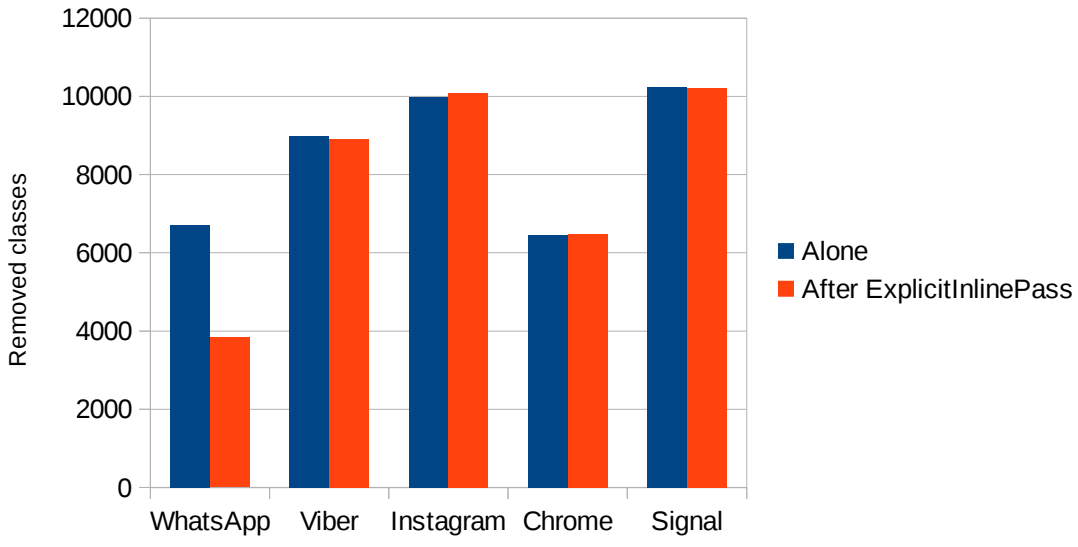

Illustration 19: Number of removed classes

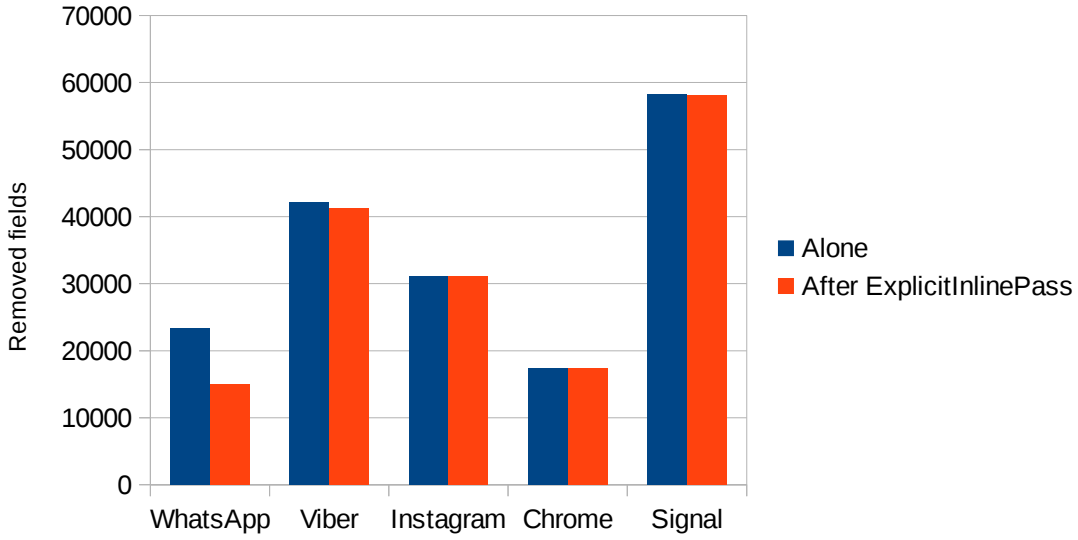

Illustration 20: Number of removed fields

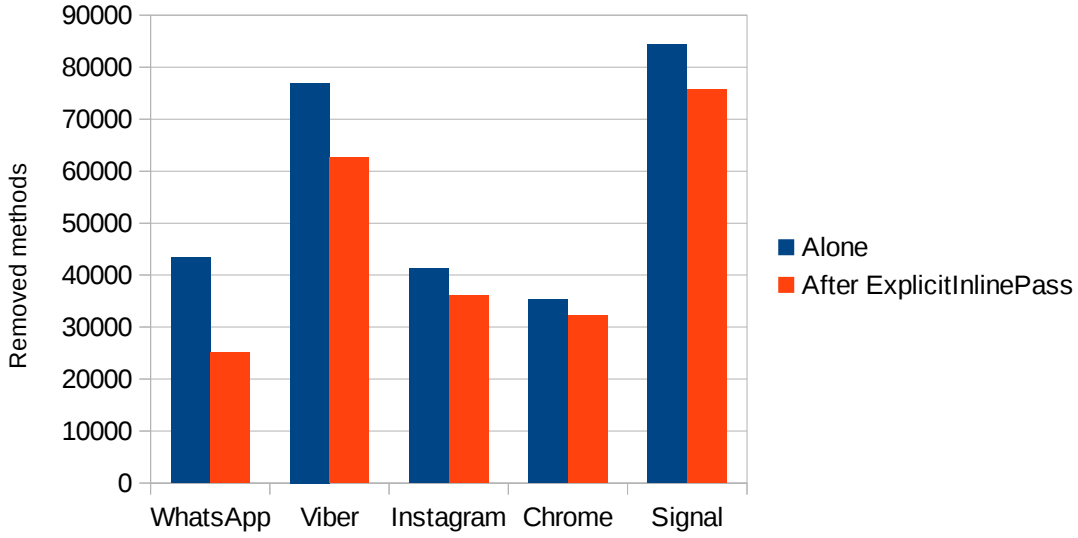

Illustration 21: Number of removed methods

This pass uses a mark-sweep algorithm to determine which classes, methods, and fields are reachable among the application code. Whatever remains unmarked is deemed as unreachable and eventually is removed. We are mostly interested in the changes of method removals. It might appear that after our pass, less methods are deleted, but in reality, we have to take into account that when RemoveUnreachablePass runs afterwards, it works on an already reduced total number of methods, because ExplicitInlinePass has removed the definitions of all methods inlined without guards. The main difference comes from removing the virtual methods inlined with guarding, so in that specific case, RemoveUnreachablePass shows an increase in the results. That can be easily verified, if we add to the amount of the methods removed from RemoveUnreachablePass, the total number of deleted methods from ExplicitInlinePass, as shown above. Specifically, for Signal it succeeds to delete 4115 more virtual methods, for Viber the difference is increased by 12823 more methods, for Instagram it achieves 495 additional deletions, and finally, for Chrome it removes 865 more methods. WhatsApp is the exception among our benchmarks and after our pass

RemoveUnreachablePass manages to remove 36291, which less than the amount it deletes alone by 7070 methods.

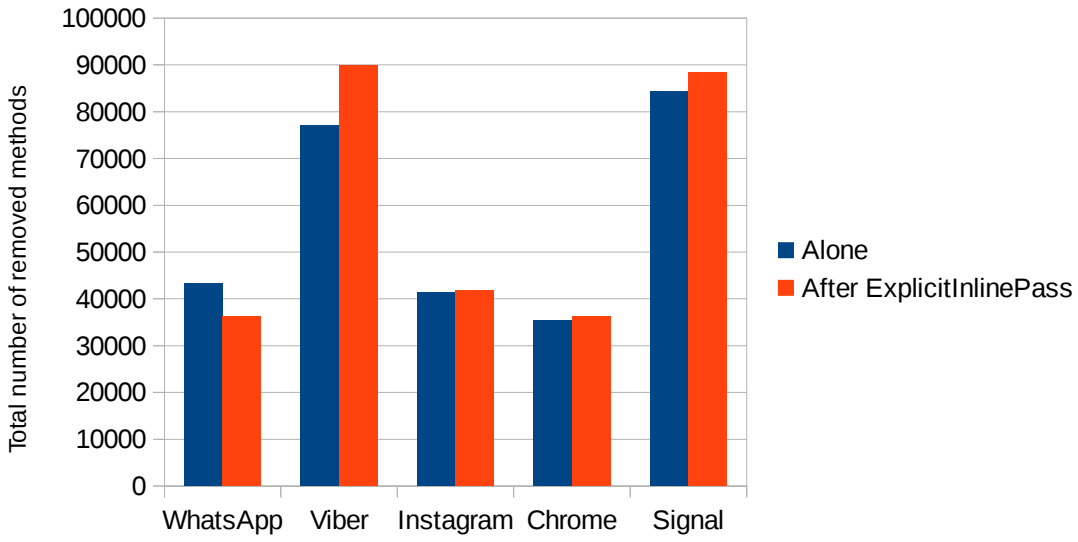

Illustration 22: Number of removed methods considering removals from ExplicitInlinePass

4.3 Final APK size

In this section we elaborate about why and how APK size is a crucial criterion for every application, and also provide some insight on how our pass affects it. As mobile applications have evolved over time, developers have added new features to serve and attract users, like higher resolution images and better graphics, causing further APK size increase. The size of the APK has an impact on how fast the application loads, how much memory it uses, and how much power it consumes, so it is a legitimate way to evaluate and compare applications. Therefore, an important aspect of our work is to maintain the final size of the APK, after all the inlining during our pass, in reasonable values, because if we exaggerate, we may cause bytecode bloating. Mobile storage is pretty limited and this is a serious constraint that we have to take into consideration, because we do not want to further burden an already critical matter. Our pass only affects the data in the dex files, so we calculate the difference in size of the dex(es) before and after our pass. In case of multidex APKs, then we add the size of all of them.

Table 9: DEX file size in bytes

	Before and After ExplicitInlinePass	
WhatsApp	8,674,944	8,107,028
Viber	18,211,476	17,050,640
Instagram	7,986,396	7,694,708
Chrome	6,522,252	6,297,268
Signal	17,949,448	17,120,876

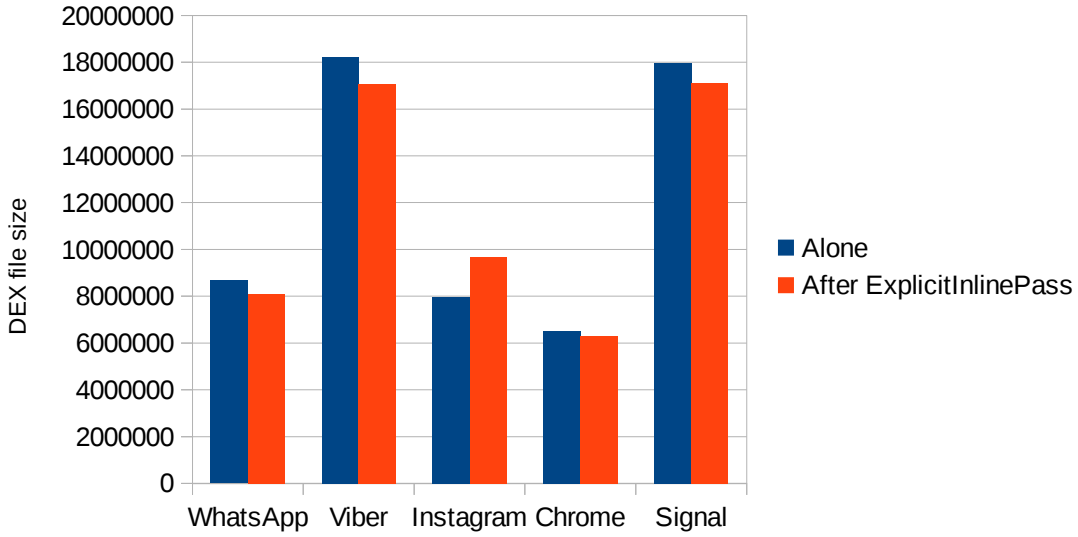

Illustration 23: Final application size in bytes

We observe that even though only our pass is run, despite all the inlining, the final outcome is smaller than the initial. That can be explained both because we delete all methods inlined without guards, and also, because after ExplicitInlinePass it is necessary to run RegAllocPass, as mentioned in section 3.2, which makes the instructions more compact after register allocation, and removes redundant move instructions.

4.4 Runtime benefits

In this section we showcase the execution times of our benchmarks, before and after our pass. Evaluating the applications that we used as our benchmarks so far is not straightforward, because they need user interaction. For this reason, we wrote our own simple microbenchmark to evaluate how it performs at runtime, when calling a method, and how the execution time changes if we inline it. Also, we evaluate the cost that the guards might introduce to the runtime, because they use the “instanceof” opcode, which carries out a series of type checks across a class hierarchy.

First, we aim to evaluate the runtime overhead of a method call compared to the improvement in the performance that method inlining provides. We create an object with of type A and, then, we proceed with calling a method on this object, calculating a mathematical operation, inside a loop that is executed for large number of iterations, for example 1.000.000. All the aforementioned procedure is repeated at least 10 times, and we finally, estimate the execution time of the whole process. The fact that the inner loop, which calls the virtual method, is performed so many times, ensures that the code will be JIT compiled. This is important, if we want to be as objective with our measurements as possible, because in reality all applications take advantage of JIT compilation at runtime.

Table 10: Runtime results of microbenchmark

Execution time in ms	
Method call	1745
Method inline	369
Guarded inline	654

Furthermore, we want to evaluate how virtual method inlining with the assistance of guards affects the execution time. We alternate our benchmark by having a simple class hierarchy, where A becomes the base class that has two subclasses, B and C, respectively. The object we create for this experiment is of static type A and the dynamic type might be either B or C. Then, we repeat the process as described above, before and after optimizing it with our pass. This time, to inline the method we need guards, because it is not trivial to conclude what the dynamic type of the object might be at runtime.

5. CONCLUSIONS

Automatic program optimizations are much more useful and effective, thus are the cornerstone of software engineering. Optimizers, such as ReDex, that are written procedurally are able to handle and manipulate the bytecode easily, however, writing analyses is not a trivial task. The fundamental idea of this thesis was to take advantage of the Doop framework, which consists of various very sophisticated analyses for Java programs that are written completely declaretively, in Datalog. The outcome of this work proves that more precise analyses can benefit ReDex and help it optimize applications more aggressively. Specifically, we observed that ExplicitInlinePass inlined more methods and also helped other passes perform better, which further resulted in decreasing the final APK size.

By the end of the development of this thesis, ReDex made an attempt to handle non-devirtualizable virtual methods as well, in order to explore more optimization opportunities. However, their approach was not based on whole-program analysis, like Doop. They concentrated mainly on finding abstract methods whose implementors have the same definition across the whole code base, and allow inlining, in such cases, based on that definition. Moreover, another optimization pass they implemented removes virtual methods that override other virtual methods, by merging them, under certain conditions. This gives the opportunity to the SimpleInlinePass to perform more inlining afterwards. Currently, they restrict this optimization, namely VirtualMergingPass, within each dex only (when overridden and overriding method are within the same dex). Also, the overriding method must be inlinable into the overridden method, using the standard inliner functionality that we presented in Section 2.3, and their implementation for this transformation pass is still vey conservative, for example they omit virtual scopes that are involved in invoke-supers. This concludes that their results might still be weaker than those from Doop, which gives us the ability to perform more detailed and sophisticated whole-program analyses, and we plan to further examine how our pass interacts with these new transformations and how many opportunities are discovered in each case.

ACRONYMS AND ABBREVIATIONS

APK	Application Package
JIT	Just-in-time
JVM	Java Virtual Machine
DVM	Dalvik Virtual Machine
ART	Android Runtime
DEX	Dalvik Executable
ODEX	Optimized Dalvik Executable
AOT	Ahead-of-time
JAR	Java Archive
API	Application programming interface

REFERENCES

[1] ”Android Operating System”, [Online]
Available: https://www.android.com/
[2] “Android Runtime (ART) and Dalvik”, [Online]
Available: https://source.android.com/devices/tech/dalvik
[3] “Implementing ART Just-In-Time (JIT) Compiler”, [Online]
Available: https://source.android.com/devices/tech/dalvik/jit-compiler
[4] “ReDex: An Android bytecode optimizer”, [Online]
Available: https://fbredex.com/
[5] “Proguard: The open source optimizer for Java bytecode”, [Online]
Available: https://www.guardsquare.com/en/products/proguard
[6] “Pointer Analysis”, [Online]
Available: https://yanniss.github.io/points-to-tutorial15.pdf
[7] “Program Analysis and Specialization of the C Programming Language”, Lars Ole Andersen PhD Thesis, DIKU, University of Copenhagen, May 1994.
[8] ”Doop: Framework for Java Pointer Analysis”, [Online]
Available: http://doop.program-analysis.org/