A Comparative Study on Optimization, Obfuscation, and Deobfuscation Tools in Android

Geunha You[1], Gyoosik Kim[2], Seong-je Cho[1]*, and Hyoil Han[3]
[1]Dept. of Computer Science & Engineering, Dankook University, Yongin, Republic of Korea
{geunhayou, sjcho}@dankook.ac.kr
[2]KT Infra Lab, Seoul, Republic of Korea
erewe4@dankook.ac.kr
[3]School of Information Technology, Illinois State University, Normal, IL, USA
hhan12@ilstu.edu

Received: December 10, 2020; Accepted: February 5, 2021; Published: February 28, 2021

Abstract

Code optimization is a program transformation process to make the program work more efficiently or consume fewer resources. Code obfuscation transforms a program and makes its code more difficult for a human to understand, which protects the code from reversing engineering. Deobfuscation is reverse-engineering the obfuscation. Optimization and obfuscation are widely used in Android apps. R8, the Android build process's default tool, does all of the code shrinking, obfuscation, and optimization. This paper compares and analyzes the functionalities of optimization, obfuscation, and deobfuscation tools in the Android platform. Besides R8, the other tools covered in this paper are ReDex, Obfuscapk, and DeGuard, which are optimization, obfuscation, and deobfuscation tools for Android apps, respectively. We investigate the characteristics of the four tools and compare their performance by performing experiments.

Keywords: Android app, Obfuscation, De-obfuscation, Optimization, R8 compiler, ReDex. Obfuscapk, DeGuard

1 Introduction

Code optimization or program optimization is a program transformation technique, which makes the program work more efficiently and uses fewer resources [23, 26]. Code optimization generally replaces the program constructs of the target software with efficient program codes. Therefore, it can improve the program's speed, and make the program operate with less CPU power and memory space. In the Android Dalvik just-in-time (JIT) compiler, optimization techniques include constant propagation, register allocation, elimination of redundant load/store and null-check, redundant branch elimination, induction variable optimization, loop optimization, method inlining, etc [19].

Code obfuscation is a practice in software development to obscure program code with semantics-preserving transformation [10, 8, 20, 21]. Obfuscation transforms the code into a more complicated form to protect the code from disassembling, decompiling, and debugging. Therefore, code obfuscation is frequently used by both software developers and malware writers. Normal software developers obfuscate their software to protect their intellectual property, such as the core logic and algorithms of software,

*Corresponding author: Department of Computer Science & Engineering, Dankook University, 152 Jukjeonro, Yongin-si, Gyeonggi-do, 16890, Korea, Tel: +82-31-8005-3239 however, malware writers obfuscated their malware to hide its malicious intention and logic and resist analysis.

On the other hand, *code deobfuscation* is reverse engineering obfuscated code and is useful for understanding obfuscated code [24, 7, 28, 27]. The objective of deobfuscation is to try to identify, simplify and remove obfuscation code. Given a program P_{org} and its obfuscated version P_{obf}, Yadegari et al. [28, 27] defined deobfuscation as the process of removing the effects of obfuscation from an obfuscated program, P_{obf}. That is, deobfuscation analyzes and transforms the code for P_{obf} to obtain a program P_{org} that is functionally equivalent to P_{obf} but is simpler and easier to understand. The deobfuscated code is easier to understand with less analysis time compared to the obfuscated one. Code obfuscation and deobfuscation are thus considered as a double-edged sword in the computer security community.

Code optimization and obfuscation tools are pervasively applied to Android apps. When an Android project is built using Android Gradle plugin 3.4.0 or higher, the plugin works with the *R8 compiler* as the default tool in the Android build process [18, 16]. The *R8 compiler* does all of the optimization, obfuscation, code shrinking, and resource shrinking. A legitimate software company needs to analyze which optimization tool (*optimizer*) or obfuscation tool (*obfuscator*) is good to use in order to protect the intellectual property of their apps. In addition, malware analysts want to know which optimizer or deobfuscation tool (*deobfuscator*) to use in order to efficiently analyze malware.

In this paper, we compare and analyze the functionalities of some optimization, obfuscation, and deobfuscation tools in Android platform. Besides *R8*, the other tools covered in this paper are *ReDex* [11, 12], *Obfuscapk* [5], and *DeGuard* [7], which are optimization, obfuscation, and deobfuscation tools for Android apps, respectively. We investigate the characteristics of the four tools and compare their performance through experiments.

The rest of this paper is organized as follows. Section 2 briefly describes the four tools: *R8*, *ReDex*, *Obfuscapk*, and *DeGuard*. Section 3 reviews related work, and Section 4 summarizes the method and analysis tools for our work. In Section 5, 6, and 7, we compare and evaluate the four tools. Finally, we conclude our findings in Section 8.

2 Background

Table 1 presents the four tools this paper investigates. The similarities and differences in their aims, functionalities, and input/output formats exist among these tools. These tools have many differences in processing input files.

Table 1: Summary of Four Tools.

Tools	Description	Functionalities
R8 compiler [18, 16]	Optimizer Obfuscator	Optimization (e.g., Removing unused `else{}` branches, Inlining), Identifier renaming obfuscation, Code shrinking, Resource shrinking
ReDex [11, 12]	Optimizer	Minification and compression, Inlining, Dead code elimination
Obfuscapk [5]	Obfuscator	Trivial techniques with four subcategories: Align, Re-sign, Rebuild, and Randomize manifest. Non-trivial techniques with four subcategories: Renaming, Encryption, Code, and Reflection.
DeGuard [7]	De-obfuscator	A tool to reverse the layout obfuscation of APKs performed by ProGuard or R8 compiler

2.1 The R8 compiler

The *R8 compiler* is a compiler suite to build Android apps and provide functionalities to optimize and obfuscate codes. *DX* was the built-in DEX compiler before Android Studio 3.1.0, *D8* from Android Studio 3.1.0, and *R8* from Android Studio 3.4.0 [15]. The *DX*, *D8*, and *R8* compliers convert a *.class* file to a *.dex* file.

Compared to *DX*, *D8* compiles faster and outputs smaller DEX files. Figure 1 shows a high-level overview of the compile process before *R8* was introduced [15]. *ProGuard*, a well-known obfuscator, transforms the names of packages, methods, and classes inside a target app, by substituting them with obscure names [7]. It can also remove unused classes, methods, and fields to minimize the size of the resulting APK. *ProGuard* works at the source-code level, mapping the original names to deformed ones based on the user's configuration [10]. Desugaring is a process that rewrites an Android app to provide Java 8 language features such as lambda expressions, default interface methods, etc. The desugaring engine has been extended to be able to desugar Java language APIs in Android Studio 4.0. *R8* integrates desugaring, shrinking, obfuscating, optimizing, and dexing (*D8*) all into one step, as shown in Figure 2 [18, 16, 15]. Note that *R8*, working with the *ProGuard* rules, is a new tool for shrinking, optimization, and obfuscation that replaces *ProGuard*.

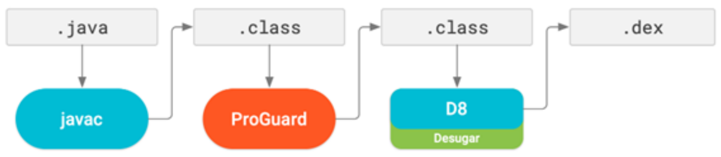


Figure 1: The Android app compile process before *D8* was introduced [15].

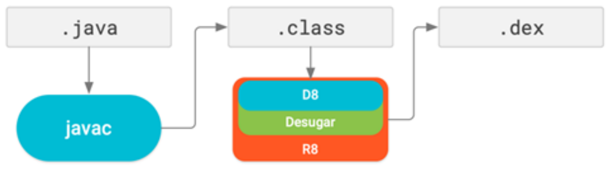


Figure 2: The Android app compile process after *R8* was introduced [15].

Given an Android app, the obfuscation part of *R8* reduces the app size by shortening the names of the app's classes, methods, and fields, but does not remove code from the app [18]. Code shrinking (or tree-shaking) removes unused classes, methods, fields, and attributes from the app and its library dependencies. Starting from each app's entry point, the code shrinking part examines the app's code to build a graph of all methods, member variables, and other classes that the app might access at runtime. It removes the code that is not connected to that graph. Moreover, if the app uses only a few APIs of a library dependency, shrinking can identify library code that the app is not using and remove that code.

After the code shrinking, resource shrinking removes unused resources from the app, including unused resources in the app's library dependencies. To decrease the size of the app's DEX files even further, the optimization part examines the code at a deeper level to remove more unused code. For example, it can remove the code for the `else {}` branch that is never taken, inline a method invoked in only one place, and combine a class with its subclass where the class has only the subclass.

To enable code shrinking, obfuscation, and optimization when building the final version of the app, you should include the '`minifyEnabled true`' statement in your project-level `build.gradle` file. You can also enable resource shrinking by including the '`shrinkResources true`' statement in the `build.gradle` file.

2.2 The ReDex optimizer

ReDex, developed by the Facebook engineering team, is an Android bytecode optimizer [11, 12, 31, 29, 30]. *ReDex* optimizes Android apps at the bytecode level via three pipeline stages: *Minification and compression*, *Inlining*, and *Dead code elimination*. Minification and compression store file paths in a compressed mode and recover them via a back-map in debugging. Inlining combines access methods such as setter/getter and decreases the method invocation overhead. Lastly, Dead code elimination traverses all branches and method invocations from the target app's entry points and eliminates the unused code. Each stage in the pipeline is independent of each other.

Wermke et.al. [25] considered *ReDex* a obfuscator similar to *ProGuard* [7, 25] because *ReDex* can obfuscate package/class/method/field names. But *ReDex*'s 'dead code elimination' has a de-obfuscation function and You et al. [31, 29, 30] considered *ReDex* a de-obfuscator.

ReDex can be installed by (1) downloading the source code from the Facebook/ReDex Github page and compiling it into an executable file (Executable Linking Format or Potable Executable Format) on the local system, or (2) using the docker image [13] deployed by the *ReDex* developers. Following the notation used in the previous paper [29], what is built from source code is called `Built-ReDex`, and what is distributed as a docker image is called `Docker-ReDex`. In this paper, we use the `Built-ReDex` version.

As of this paper writing, the current Docker version was distributed two years ago, and it is recommended to use the latest version of the source code that developers update periodically. *ReDex* outputs a *.dex* file after uncompressing a compressed APK file. Users can optimize a .dex file, repackage and sign the optimized *.dex* file and install it on an Android device.

2.3 Obfuscapk

Obfuscapk is a tool to obfuscate open-source black-box Android apps [5]. The inputs of *Obfuscapk* are APK files and obfuscation option, and its output is the obfuscated APK files. The obfuscation techniques provided by *Obfuscapk* is comprised of trivial techniques and non-trivial techniques. Trivial techniques and non-trivial techniques each consist of four subcategories. Among the techniques, we focus on non-trivial techniques in this paper.

The four subcategories of non-trivial obfuscation techniques are *Renaming*, *Encryption*, *Code*, and *Reflection*. *Renaming* converts identifiers' names (such as variable, methods, etc.) to meaningless names. *Encryption* encrypts strings, asset files (video, photo, text, etc.) of a native library, strings.xml, and constant strings in code and decrypts them in runtime.

Code subcategory contains several specific techniques: **DebugRemoval**, **CallIndirection**, **Goto**, **Reorder**, **ArithmeticBranch**, **Nop**, and **MedhodsOverload**. **DebugRemoval** removes debug metadata such as line numbers, types, and method names. **CallIndirection** changes the control-flow graph (CFG) by adding new methods and substituting existing methods with new wrapper methods. **Goto** modifies the CFG by inserting `goto` instructions. **Reorder** changes the order of basic blocks by inverting a branch instruction's condition and re-arranging the code, abusing `goto` instructions randomly. **ArithmeticBranch** complicates the CFG by inserting useless arithmetic computations and branch instructions. **Nop** inserts random nop instructions. **MethodsOverload** creates a new void method with the same name and arguments or adds new random arguments.

The *Reflection* subcategory looks for method invocations of an app, ignoring the calls to the Android framework. If an instruction with a suitable method invocation is found, such invocation is redirected to a custom method. **AdvancedReflection** is complementary to the Reflection technique but targeting the invocations of dangerous APIs.

2.4 DeGuard (de-obfuscator)

DeGuard is a free de-obfuscator to recover renamed symbols (identifiers) obfuscated by *ProGuard* [7, 29]. To evaluate the effectiveness of *DeGuard*, Bichsel et al. [7] arbitrarily selected 100 apps offered by *F-Droid* [2] and experimented with the dataset by applying *ProGuard*'s identifier renaming obfuscation. With the input of APK files, *DeGuard* perfectly recovered 79.1% of symbols obfuscated by machine learning approaches and predicted third-party libraries with an accuracy of 91.3%. *DeGuard* could not recover the final 20.9% of symbols, due to mis-deobfuscated symbols different from their original symbols or due to failed obfuscation where the obfuscated symbols remain. *DeGuard* did not open its source code but is open to an online web site so that people can use it freely.

Table 2 shows code obfuscation techniques that *DeGuard*, *R8*, and *ReDex* can resolve in the point of view of deobfuscation and optimization.

Table 2: The options that *DeGuard*, *R8*, and *ReDex* can deobfuscate.

Evaluation Tools	Obfuscaors and their options									
	R8 (as an obfuscator)	*Obfuscapk*								
	Renaming	Renaming	Encryption	Reflection	Code					
					Arithmetic Branch	Reorder	Call Indirection	Method Overload	Goto	Nop
DeGuard	O	X	X	X	X	X	X	X	Δ	Δ
R8 (as an optimizer)	X	X	X	X	O	X	X	X	O	O
ReDex	X	X	X	X	O	X	X	X	O	O

3 Related Work

Like *DeGuard*, *Anti-ProGuard* [6] is a tool to deobfuscate identifiers (or symbols) obfuscated by *ProGuard*. *Anti-ProGuard* uses similarity algorithms similar to SimHash and n-gram to conjecture symbols obfuscated by renaming. *Anti-ProGuard* experimented with four apps (offered by *F-Droid*) satisfying the following three criteria:

1. The app has at least a few libraries.
2. Making *ProGuard* active/inactive should be easy.
3. It is distributed with Gradle scripts so that it is easy to compile.

The experiment by the authors showed that *Anti-ProGuard* identifies more than 50% of symbols in packages.

Java-deobfuscator [3] is a deobfuscator that works for a Java program (*.jar*) on JVM. *Java-deobfuscator* can resolve (or deobfuscate) Java programs obfuscated by commercial Java obfuscators such as *Zelix Klassmaster*, *Stringer*, *Allatori*, *DashO*, *DexGuard*, *ClassGuard*, and *Smoke*. Most Android apps are written in Java, and *.class* files are created after compiling Java programs. Android apps' execution files named *.dex* is converted to *.jar* files. However, *Java-deobfuscator* is not a tool for Android apps. We extracted *.dex* files from APK files and converted *.dex* files to *.jar* files to use *Java-deobfuscator* for Android apps. Then we used *Java-deobfuscator* to convert the converted *.jar* files into *.dex* files and repackaged them. However, *Java-deobfuscator* outputs *.jar* files as well as many errors. That is, its outputs are incomplete *.jar* files. The main reason for *Java-deobfuscator*'s failure in optimization is the loss in converting *.dex* files to *.jar* files. Therefore, we could not execute the files on Android devices after repackaging *.dex* files obtained by converting incomplete *.jar* files due to such *Java-deobfuscator*'s failure.

Simplify [4] is a generic Android deobfuscator available in Github. *Simplify* operates Android apps in a virtual machine and optimizes the apps' activities. Optimization strategies of Simplify consist of constant propagation, dead code elimination, unreflection, and several peephole optimizations. *Simplify* does not deobfuscate renaming obfuscated but supports decrypting encrypted string, deobfuscating reflection obfuscation, and simplifying codes. In the future, we plan to experiment with *Simplify*.

In our previous work, we analyzed and compared the performance of *R8*, *ReDex*, and *DeGuard* [31, 29, 30]. In the current paper, we included our extended results to our previous work. In [31], we experimented with `Docker-ReDex` [27] to optimize Android apps obfuscated by *Obfuscapk*'s code category and analyzed the outcomes. In [29], we experimented with *DeGuard* to deobfuscate and with `Docker-ReDex` and `Built-ReDex` to optimize Android apps (APK$_F$) obfuscated by *Obfuscapk* or Android apps (APK$_P$) obfuscated by *R8* (or *ProGuard*). In [30], for Android apps (APK$_O$), which *R8* cannot apply to, we compared Android apps (APK$_P$) applied by *R8* with Android apps (APK$_{RO}$) optimized by `Built-ReDex`.

In Section 5 of this paper, for Android apps (APK$_O$), which *R8* cannot apply to, we compare Android apps (APK$_P$) applied by *R8* with Android apps (APK$_{F\_renaming}$) obfuscated by renaming the category of *Obfuscapk* and analyze the results. In Section 6, we compare the optimization performance between *R8* and *ReDex* for Android apps (APK$_P$) applied by *R8* and Android apps (APK$_{F\_code}$) obfuscated by *Obfuscapk*'s code category. Additionally, we analyze the optimization performance of *ReDex* for Android apps (APK$_P$) that *R8* applies to. Section 7 describes our experiment with *DeGuard* to deobfuscate Android apps (APK$_P$) that *R8* applied to. In this paper, we experimented with Android apps that were not used in the existing experiments.

4 Methods and Analysis tools

The methodology to compare the performance of tools depends on the input/out of each tool. Figure 3 shows the process of generating Android apps for experiments with *R8* and *Obfuscapk*. The dotted box in red in Figure 3 represents the Android apps used in our experiments. We chose three Android apps downloaded from *F-Droid* [2]. These apps are named "`tk.radioactivemineral.metronome_5.apk`," "`it.linuxday.torino_1.apk`," and "`eu.veldsoft.ithaka.board.game_5.apk`" and correspond to APK$_O$-1, APK$_O$-2, and APK$_O$-3, respectively. These apps are optimized or obfuscated for our experiment. We execute these Android apps in Android 8.1 Oreo on Android Virtual Device to experiment with the optimized apps, obfuscated apps, and deobfuscated app.

In Figure 3, the process that *R8* applies to APK$_F$ does not appear due to a lack of spaces. We cannot apply *R8* directly to APK$_F$ because APK$_F$ is already a compiled app and must use a process that extracts *.dex* and converts *.dex* to *.jar* using *dex2jar*. APK$_{PF}$ is generated by repackaging the *.dex* file obtained by applying *R8* to the converted *.jar* file. Table 3 explains the Android apps generated via the process in Figure 3.

The process of generating Android apps to compare the *R8 compiler* with *ReDex* appears in [30]. Table 4 shows the analysis tools used in our experiments to compare and analyze original apps, optimized apps, obfuscated apps, and deobfuscated apps.

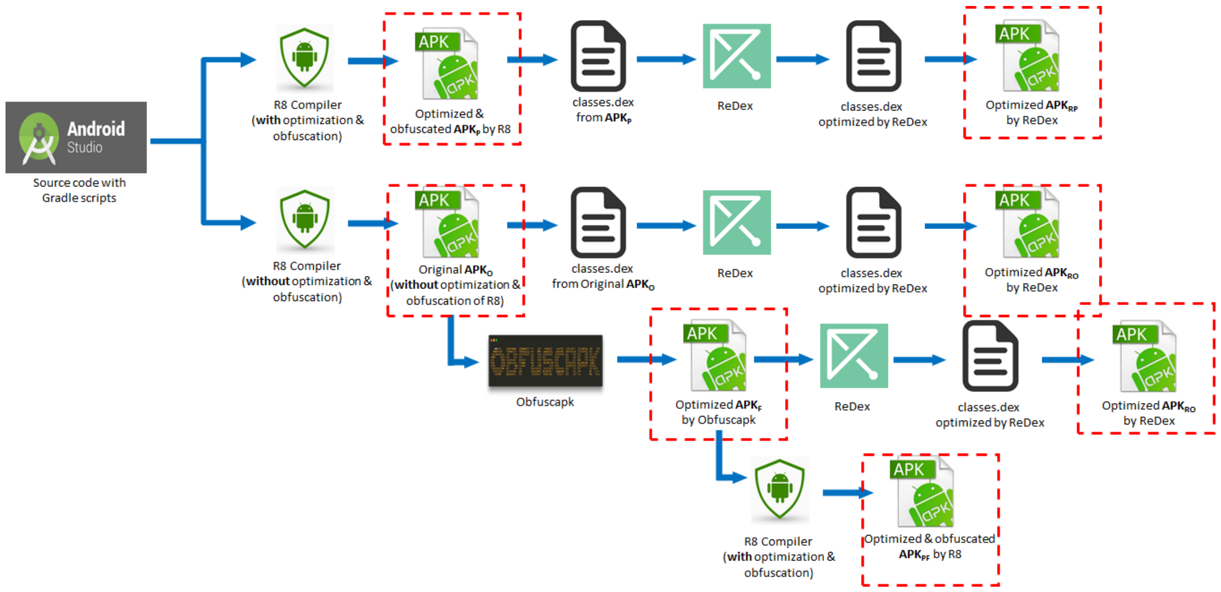

Android Studio
Source code with Gradle scripts
R8 Compiler (with optimization & obfuscation)
Optimized & obfuscated APK_P by R8
classes.dex from APK_P
ReDex
classes.dex optimized by ReDex
Optimized APK_RP by ReDex
R8 Compiler (without optimization & obfuscation)
Original APK_O (without optimization & obfuscation of R8)
classes.dex from Original APK_O
Optimized APK_RO by ReDex
Obfuscapk
Optimized APK_F by Obfuscapk
R8 Compiler (with optimization & obfuscation)
Optimized & obfuscated APK_PF by R8

Figure 3: The process of generating Android apps for our experiments.

5 Evaluation of R8 and Obfuscapk

This section describes the comparisons of *R8* with *Obfuscapk*. Figure 4 shows the classes of an original Android app and its obfuscated version by *R8* and *Obfuscapk*. Figure 4(a) shows the class with an original Android app (APK$_O$), Figure 4(b) shows a class of the Android app (APK$_P$), which is the original app obfuscated by *R8*'s identifier renaming, and Figure 4(c) shows a class of the Android app (APK$_F$), which is the original app obfuscated by *Obfuscapk*'s renaming category. We used a decompiler called JEB Pro 3.28 to compare Java source code. Figure 4 shows the source code comparisons. Figures 4(b) and 4(c) have a common property in the point of view of changing identifier names meaninglessly. Nonetheless, Figures 4(b) and 4(c) have differences. Figure 4(b) shows that the identifier names are shortened by *R8*, and the resultant *.dex* file size decreases. On the other hand, Figure 4(c) shows that *Obfuscapk* does not decrease the size of the identifier names, and the resultant *.dex* file does not decrease.

6 Evaluation of R8 and ReDex

We chose the original app (APK$_O$), and the obfuscated app (APK$_F$), which is the app obfuscated by *Obfuscapk*'s renaming category, to compare the optimization performance. *ReDex* developers built *ReDex* to futher optimize Android apps already optimized by the Android compiler, and *ReDex* decreased the size of Facebook apps by 25% [14]. Therefore, we also experimented with applying *ReDex* to the apps optimized by *R8*. Our experiment's Android apps appear in Table 3, and the results of our optimization experiments appear in Tables 5-10. Tables 5-6 show the experimental results based on APK$_O$-1 app (`tk.radioactivemineral.metronome_5.apk`). Tables 7-8 show results based on APK$_O$-2 app (`it.linuxday.torino_1.apk`), and Tables 910 show results based on APK$_O$-3 app (`eu.veldsoft.ithaka.board.game_5.apk`).

APK$_O$-1 is the original Android app that is not obfuscated and optimized, and APK$_P$-1 is the Android app obtained by compiling APK$_O$-1 with *R8*. Whereas APK$_{RO}$-1 is the Android app obtained by optimizing a *.dex* file by *ReDex* after the *.dex* is extracted from APK$_O$-1, and APK$_{RP}$-1 is the Android app optimized by *ReDex* after APK$_O$-1 is optimized and obfuscated by *R8*. In Table 5, row 4 represents the

Table 3: The explanation for Android apps used in our experiments.

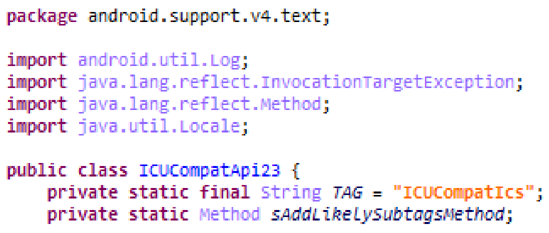
Notation	APK name	Description	Size (in bytes)
APK$_O$-1	tk.radioactivemineral.metronome_5.apk	Built from source code	1,710,352
APK$_O$-2	it.linuxday.torino_1.apk	Built from source code	1,603,967
APK$_O$-3	eu.veldsoft.ithaka.board.game_5.apk	Built from source code	2,384,201
APK$_P$-1	tk.radioactivemineral.metronome_5.apk	Obfuscated and optimized from source code using R8	640,069
APK$_P$-2	it.linuxday.torino_1.apk	Obfuscated and optimized from source code using R8	1,149,824
APK$_P$-3	eu.veldsoft.ithaka.board.game_5.apk	Obfuscated and optimized from source code using R8	690,489
APK$_F$-1	tk.radioactivemineral.metronome_5.apk	Obfuscated from APK$_O$-1 using Obfuscapk	3,831,684
APK$_F$-2	it.linuxday.torino_1.apk	Obfuscated from APK$_O$-2 using Obfuscapk	3,544,401
APK$_F$-3	eu.veldsoft.ithaka.board.game_5.apk	Obfuscated from APK$_O$-3 using Obfuscapk	3,830,524
APK$_{RO}$-1	tk.radioactivemineral.metronome_5.apk	Optimized from APK$_O$-1 using ReDex	1,665,665
APK$_{RO}$-2	it.linuxday.torino_1.apk	Optimized from APK$_O$-2 using ReDex	1,534,988
APK$_{RO}$-3	eu.veldsoft.ithaka.board.game_5.apk	Optimized from APK$_O$-3 using ReDex	2,342,769
APK$_{RP}$-1	tk.radioactivemineral.metronome_5.apk	Optimized from APK$_P$-1 using ReDex	632,719
APK$_{RP}$-2	it.linuxday.torino_1.apk	Optimized from APK$_P$-2 using ReDex	2,321,373
APK$_{RP}$-3	eu.veldsoft.ithaka.board.game_5.apk	Optimized from APK$_P$-3 using ReDex	690,113
APK$_{RF}$-1	tk.radioactivemineral.metronome_5.apk	Optimized from APK$_F$-1 using ReDex	2,274,768
APK$_{RF}$-2	it.linuxday.torino_1.apk	Optimized from APK$_F$-1 using ReDex	2,321,373
APK$_{RF}$-3	eu.veldsoft.ithaka.board.game_5.apk	Optimized from APK$_F$-1 using ReDex	2,975,136
APK$_{PF}$-1	tk.radioactivemineral.metronome_5.apk	Obfuscated and optimized from APK$_F$-1 using R8	2,015,794
APK$_{PF}$-2	it.linuxday.torino_1.apk	Obfuscated and optimized from APK$_F$-2 using R8	2,093,434
APK$_{PF}$-3	eu.veldsoft.ithaka.board.game_5.apk	Obfuscated and optimized from APK$_F$-3 using R8	2,726,662

O: denotes '*Original*', *P*: denotes '*R8 compiler*', *F*:denotes '*Obfuscapk*', *R*: denotes '*ReDex*'

size of all the resources in APK, rows 8-9 show the number of edges and nodes in a call graph (CG), rows 10-11 represent the number of basic blocks and edges of a control-flow graph (CFG), and row 11 shows the average size of methods in the Android app. The average size of methods was measured only for methods that appear in the original Android app, its obfuscated Android app, and its optimized Android

Table 4: Analysis tools used in our work.

Analysis Tools	Description
JEB Pro 3.28 [22]	A decompiler that decompiles .dex and generates Java source code
Androguard [9]	Androguard outputs an app's call graph (CG), used in analyzing nodes and edges.
Dexdump2 [17]	Dexdump2 outputs an app's control flow graph (CFG) used to analyze basic blocks and edges.
dex2jar [1]	dex2jar converts .dex into .jar.

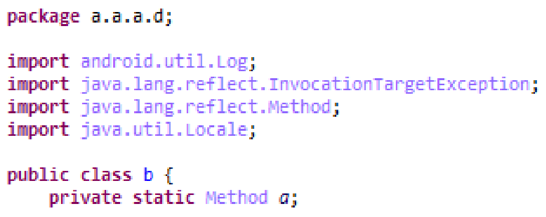


```
package android.support.v4.text;

import android.util.Log;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Locale;

public class ICUCompatApi23 {
    private static final String TAG = "ICUCompatIcs";
    private static Method sAddLikelySubtagsMethod;
```

(a) An original Android app (APK$_O$).

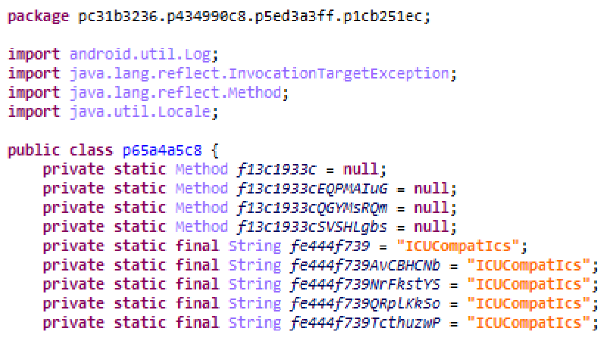


```
package a.a.a.d;

import android.util.Log;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Locale;

public class b {
    private static Method a;
```

(b) The Android app (APK$_P$).



```
package pc31b3236.p434990c8.p5ed3a3ff.p1cb251ec;

import android.util.Log;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Locale;

public class p65a4a5c8 {
    private static Method f13c1933c = null;
    private static Method f13c1933cEQPMAIuG = null;
    private static Method f13c1933cQGYMsRQm = null;
    private static Method f13c1933cSVSHLgbs = null;
    private static final String fe444f739 = "ICUCompatIcs";
    private static final String fe444f739AvCBHCNb = "ICUCompatIcs";
    private static final String fe444f739NrFkstYS = "ICUCompatIcs";
    private static final String fe444f739QRpLKkSo = "ICUCompatIcs";
    private static final String fe444f739TcthuzwP = "ICUCompatIcs";
```

(c) The Android app (APK$_F$).

Figure 4: (a) An original Android app (APK$_O$), (b) the Android app (APK$_P$), which is the original app obfuscated by *R8*'s identifier renaming, and (c) the Android app (APK$_F$), which is the original app obfuscated by *Obfuscapk*'s renaming category.

Table 5: The analysis results of the original app (APK$_O$-1) and its optimized apps by *R8* or *ReDex*. (Original app: `tk.radioactivemineral.metronome_5.apk`)

	APK size	.dex size	Res size	# of classes	# of methods	# of fields	# of nodes (CG)	# of edges (CG)	# of basic blocks (CFG)	# of edges (CFG)	Average size of methods	Executable
APK$_O$-1	1,710,352	3,205,844	234,152	3,289	31,264	14,430	28,814	48,190	57,000	52,027	77.21	O
APK$_P$-1	640,069	498,412	190,425	554	5,289	3,362	4,936	7,820	10,979	11,845	55.54	O
APK$_{RO}$-1	1,665,665	3,160,116	234,152	3,204	29,863	14,428	27,594	47,593	53,369	45,480	54.86	O
APK$_{RP}$-1	632,719	487,204	190,425	532	5,162	3,361	4,828	7,761	10,186	10,315	51.19	O

app. The last row in Table 5 explains the status of whether or not the Android app is executable.

R8's code shrinking significantly decreased the number of unused classes and methods, the number of nodes and edges in a call graph, the number of basic blocks and edges in a control-flow graph, and the size of *.dex*. Additionally, the resource shrinking significantly decreased the size of resources by removing unused resources. Compared with the results by *R8*, the improvement of APK$_{RO}$-1 optimized

Table 6: The analysis results of the original app (APK$_O$-1) and its optimized and/or obfuscated apps. (Original app: `tk.radioactivemineral.metronome_5.apk`)

	APK size	.dex size	Res size	# of classes	# of methods	# of fields	# of nodes (CG)	# of edges (CG)	# of basic blocks (CFG)	# of edges (CFG)	Average size of methods	Executable
APK$_O$-1	1,710,352	3,205,844	234,152	3,289	31,264	14,430	28,814	48,190	57,000	52,027	77.21	O
APK$_P$-1	3,831,684	12,003,476	275,083	3,289	64,019	14,430	45,195	67,753	832,968	757,798	175.10	O
APK$_{RO}$-1	2,015,794	4,429,520	275,083	3,289	62,,692	14,174	43,917	67,749	95,693	58,023	56,97	X
APK$_{RP}$-1	2,274,768	5,463,804	275,083	3,204	62,628	14,430	44,300	65,578	91,563	51,872	54.33	O

by *ReDex* is not dramatic. Finally, APK$_{RP}$-1 is the Android app obtained after *ReDex* optimized APK$_P$-1, which slightly decreased all related features in APK$_P$-1.

The Android apps in Table 6 are APK$_F$-1, APK$_{PF}$-1, and APK$_{RF}$-1. APK$_F$-1 is obtained by applying the code category of *Obfuscapk* with all options to APK$_O$-1. APK$_{PF}$-1 is obtained by optimizing APK$_F$-1 by *R8*. APK$_{RF}$-1 is obtained by optimizing APK$_F$-1 by *ReDex*. Among *Obfuscapk*'s code category options, `CallIndirection` and `MethodOverload` increased the number of nodes and edges in CG by increasing the number of methods. The `ArithmeticBranch` and `Goto` options increased the number of basic blocks and edges in CFG by inserting meaningless operators and branches. These options for obfuscation and the `Nop` option increased the size of *.dex*. For APK$_{PF}$-1, *R8* significantly decreased the number of nodes in CG and the size of *.dex*. For APK$_{RF}$-1, *ReDex* significantly decreased the number of basic blocks and edges in CFG. *Obfuscapk*'s code category options that impact the number of basic blocks and edges in CFG are `ArithmeticBranch`, `Goto`, `CallIndirection`, `MethodOverload`, and `Nop`. *ReDex* removes a lot of dummy code added by `ArithmeticBranch`, `Goto`, and `Nop`, and partially eliminates partially the methods added by `CallIndirection` and `MethodOverload`.

Tables 7-8 show the experimental results of Android apps optimized or obfuscated based on "`it.linuxday.torino_1.apk`", whereas, Tables 9-10 present the experimental results of Android apps optimized or obfuscated based on "`eu.veldsoft.ithaka.board.game_5.apk`." Our analysis shows that these results are similar to the results in Tables 56.

Table 7: The analysis results of the original app (APK$_O$-2) and its optimized apps by *R8* or *ReDex*. (Original app: `it.linuxday.torino_1.apk`)

	APK size	.dex size	Res size	# of classes	# of methods	# of fields	# of nodes (CG)	# of edges (CG)	# of basic blocks (CFG)	# of edges (CFG)	Average size of methods	Executable
APK$_O$-2	1,603,967	2.524.824	636	1,658	17,392	7,612	15,905	25,065	31,730	32,165	74.38	O
APK$_P$-2	1,149,824	872,764	605,379	993	9,843	4,122	9,346	15,892	20,689	22,836	52.72	O
APK$_{RO}$-2	1,534,988	1,752,364	635,574	1,584	16,425	7,609	15,126	24,681	29,715	27,918	51.02	O
APK$_{RP}$-2	1,132,953	837,296	605,379	944	9,299	4,121	8,893	15,780	19,248	19,975	49.19	O

Table 8: The analysis results of the original app (APK$_O$-2) and its optimized and/or obfuscated apps. (Original app: `it.linuxday.torino_1.apk`)

	APK size	.dex size	Res size	# of classes	# of methods	# of fields	# of nodes (CG)	# of edges (CG)	# of basic blocks (CFG)	# of edges (CFG)	Average size of methods	Executable
APK$_O$-2	1,603,967	2.524.824	636	1,658	17,392	7,612	15,905	25,065	31,730	32,165	74.38	O
APK$_F$-2	3,544,401	9,785,732	688,141	1,658	64,038	7,612	39,217	52,783	589,235	492,560	222.13	O
APK$_{PF}$-2	2,093,434	3,801,988	688,141	1,658	63,090	7,510	38,302	52,792	83,767	38,190	55.02	X
APK$_{RF}$-2	2,321,373	4,560,716	688,141	1,584	63,093	7,609	38,235	52,263	76,440	27,964	52.49	O

Table 9: The analysis results of the original app (APK$_O$-3) and its optimized apps by *R8* or *ReDex*. (Original app: `eu.veldsoft.ithaka.board.game_5.apk`)

	APK size	.dex size	Res size	# of classes	# of methods	# of fields	# of nodes (CG)	# of edges (CG)	# of basic blocks (CFG)	# of edges (CFG)	Average size of methods	Executable
APK$_O$-3	2,384,201	1,431,572	1,912,646	1,156	11,101	2,935	10,126	14,600	18,821	18,138	81.89	O
APK$_P$-3	690,489	25,656	673,606	33	237	71	236	356	468	596	47.26	O
APK$_{RO}$-3	2,342,769	1,062,464	1,912,646	1,098	10,502	2,932	9,669	14,254	17,652	15,711	46.97	O
APK$_{RP}$-3	690,113	24,408	673,606	33	237	71	236	354	404	504	44.45	O

Table 10: The analysis results of the original app (APK$_O$-3) and its optimized and/or obfuscated apps. (Original app: `eu.veldsoft.ithaka.board.game_5.apk`)

	APK size	.dex size	Res size	# of classes	# of methods	# of fields	# of nodes (CG)	# of edges (CG)	# of basic blocks (CFG)	# of edges (CFG)	Average size of methods	Executable
APK$_O$-3	2,384,201	1,431,572	1,912,646	1,156	11,101	2,935	10,126	14,600	18,821	18,138	81.89	O
APK$_F$-3	3,830,524	7,404,280	1,912,894	1,156	52,656	2,935	25,194	32,758	383,193	296,773	187.74	O
APK$_{PF}$-3	2,726,662	3,327,684	1,912,894	1,156	52,077	2,929	24,619	32,712	63,198	20,227	47.33	X
APK$_{RF}$-3	2,975,136	3,585,440	1,912,894	1,098	52,065	2,932	24,522	32,261	59,238	15,729	45.86	O

7 Evaluation of DeGuard

To evaluate *DeGuard*'s deobfuscation performance, we experimented with Android apps APK$_P$-1, APK$_P$-2, and APK$_P$-3, which were obfuscated by the *R8* compiler's renaming process. Table 11 shows the deobfuscation success rate and the ratio of correctly predicted libraries among the predicted libraries when *DeGuard* deobfuscated symbols of the Android apps. As shown in Table 11, *DeGuard* produced a good success rate in deobfuscating and predicting symbols. The renamed symbols (or renamed app element names) in libraries were correctly predicted over 94%. The Android support library, such as the '`android.support.v4`' package, is mainly predicted correctly, however, the symbols in the user-defined classes and methods within user-defined packages were not predicted correctly. The ratio of correctly predicted libraries to mispredicted libraries is proportional to the deobfuscation success rate shown in Table 11. Deobfuscation success rate represents the ratio of deobfuscating names of packages, classes, methods, constants, and variables obfuscated to their original names. The ratio of correctly predicted libraries is the ratio of correctly predicted libraries to the Android app's whole library.

Table 11: *DeGuard*'s deobfuscation success rate and the ratio of correctly predicted libraries.

APK	Deobfuscation success rate	Ratio of correctoly predicted libraries
APK$_P$-1	81.33%	97.23%
APK$_P$-2	74.67%	94.57%
APK$_P$-3	77.48%	95.67%

(Deobfuscation success rate: Percent of correctly recovered obfuscated element names)

8 Conclusion

This paper analyzed the performance of widely used tools for obfuscating, deobfuscating, and/or optimizing Android apps. We experimented with *R8* and *ReDex* for optimization tools, *DeGuard* for a deobfuscation tool, and *R8* and *Obfuscapk* for an obfuscation tool. *R8* is a tool for both optimization and obfuscation. Our experiment with applying *R8* to Android apps (APK$_P$ and APK$_F$) showed significant decreases in all aspects, including the number of classes, methods, and resources. *ReDex* also showed decreases in all aspects but showed lower decreases than *R8*. APK$_P$ was installed on AVD successfully because it was optimized and obfuscated by *R8*, however, APK$_{PF}$ was not installed on AVD successfully because an error occurred in converting *.dex* when applying *R8* to APK$_F$. On the other hand, APK$_{RO}$ and APK$_{RF}$, obtained after *ReDex* optimized APK$_O$ and APK$_F$, were executed successfully on AVD. Lastly, *DeGuard* deobfuscated 77.83% of the symbols renamed of APK$_P$ on average. The accuracy is competitive to the performance (79.1%) of the existing work for *DeGuard*.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2018R1A2B2004830) and the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2020-2015-0-00363) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation).

References

[1] dex2jar. `https://github.com/pxb1988/dex2jar` [Online; accessed on Januay 20, 2021].

[2] F-Droid. `https://f-droid.org/` [Online; accessed on January 20, 2021].

[3] Java-deobfuscator. `https://github.com/java-deobfuscator/deobfuscator` [Online; accessed on Januay 20, 2021].

[4] Simplify – Generic Android Deobfuscator. `https://github.com/CalebFenton/simplify` [Online; accessed on January 20, 2021].

[5] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, January-June 2020.

[6] R. Baumann, M. Protsenko, and T. Müller. Anti-proguard: Towards automated deobfuscation of android apps. In *Proc. of the 4th Workshop on Security in Highly Connected IT Systems (SHCIS'17), Neuchâtel, Switzerland*, pages 7–12. ACM, June 2017.

[7] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev. Statistical deobfuscation of android applications. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16), Vienna, Austria*, pages 343–355. ACM, October 2016.

[8] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.

[9] A. Desnos, G. Gueguen, and S. Bachmann. Androguard. `https://androguard.readthedocs.io/en/latest/` [Online; accessed on Januay 20, 2021].

[10] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *Proc. of the 14th International Conference on Security and Privacy in Communication Systems (SecureComm'18), Singapore, Singapore*, pages 172–192. Springer, August 2018.

[11] F. engineering. Optimizing Android bytecode with ReDex. `https://engineering.fb.com/2015/10/01/android/optimizing-android-bytecode-with-redex/` [Online; accessed on January 20, 2021].

[12] F. engineering. Redex - An Android Bytecode Optimizer. `https://fbredex.com/` [Online; accessed on January 20, 2021].

[13] F. engineering. ReDex – Docker Container Deployments. `https://fbredex.com/docs/docker` [Online; accessed on January 20, 2021].

[14] F. engineering. Open-sourcing ReDex: Making Android apps smaller and faster, 2016. `https://engineering.fb.com/2016/04/12/android/open-sourcing-redex-making-android-apps-smaller-and-faster/` [Online; accessed on Januay 20, 2021].

[15] Google. Android Studio – Android Gradle plugin release notes. `https://developer.android.com/studio/releases/gradle-plugin#3-4-0` [Online; accessed on January 20, 2021].

[16] Google. D8 dexer and R8 shrinker. `https://r8.googlesource.com/r8` [Online; accessed on January 20, 2021].

[17] Google. Git repositories on android. `https://android.googlesource.com/platform/dalvik/+/09239e3/dexdump` [Online; accessed on Januay 20, 2021].

[18] Google. Shrink, obfuscate, and optimize your app. `https://developer.android.com/studio/build/shrink-code`[Online; accessed on January 20, 2021].

[19] J. Kim, I. Kim, C. Min, H. K. Jun, S. H. Lee, W.-T. Kim, and Y. I. Eom. Static dalvik bytecode optimization for android applications. *ETRI Journal*, 37(5):1001–1011, 2015.

[20] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun. Stochastic optimization of program obfuscation. In *Proc. of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17), Buenos Aires, Argentina*, pages 221–231. IEEE, May 2017.

[21] J. Park, H. Kim, Y. Jeong, S.-j. Cho, S. Han, and M. Park. Effects of code obfuscation on android app similarity analysis. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 6(4):86–98, December 2015.

[22] P. Software. JEB. `https://www.pnfsoftware.com/` [Online; accessed on Januay 20, 2021].

[23] TutorialsPoint. Compiler Design - Code Optimization. `https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm` [Online; accessed on January 20, 2021].

[24] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proc. of the 12th Working Conference on Reverse Engineering (WCRE'05), Pittsburgh, Pennsylvania, USA*, pages 45–54. IEEE, November 2005.

[25] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl. A large scale investigation of obfuscation use in google play. In *Proc. of the 34th Annual Computer Security Applications Conference (ACSAC'18), San Juan, Puerto Rico, USA*, pages 222–235. ACM, December 2018.

[26] Wikipedia. Program optimization. `https://en.wikipedia.org/wiki/Program_optimization` [Online; accessed on January 20, 2021].

[27] B. Yadegari. *Automatic deobfuscation and reverse engineering of obfuscated code*. PhD thesis, The University of Arizona, 2016.

[28] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *Proc. of the 2015 IEEE Symposium on Security and Privacy (S&P'15), San Jose, California, USA*, pages 674–691. IEEE, May 2015.

[29] G. You, , S.-j. Cho, H. Han, and K. Suh. De-obfuscating android apps obfuscated by proguard and obfuscapk. In *Proc. of the 6th International Conference on Next Generation Computing (ICNGC'20), Busan, Republic of Korea*. Korean Institute of Next Generation Computing, December 2020.

[30] G. You, , S.-j. Cho, H. Han, and K. Suh. Performance comparison between r8 compiler and redex in code optimization of android apps. In *The 6th International Conference on Next Generation Computing (ICNGC'20), Busan, Republic of Korea*. Korean Institute of Next Generation Computing, December 2020.

[31] G. You, G. Kim, J. Park, S.-j. Cho, and M. Park. Reversing obfuscated control flow structures in android apps using redex optimizer. In *Proc. of the 9th International Conference on Smart Media and Applications (SMA'20), Jeju Island, Republic of Korea*. ACM, September 2020.

Author Biography

Geunha You received a B.E. degree in Dept. of Software Science from Dankook University, South Korea, in 2020. He is currently a master's student in Dept. of Computer Science and Engineering from Dankook University, South Korea. His current research interests include computer security, mobile security, reverse engineering and embedded system.

Gyoosik Kim received a BE in Applied Computer Engineering, and an ME in Computer Science and Engineering from Dankook University, South Korea in 2016, and 2018, respectively. He is currently a research engineer in infra R&D Lab at Korea Telecommunication. His current research interests include computer security and software intellectual property protection.

Seong-je Cho the B.E., M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 1989, 1991 and 1996, respectively. In 1997, he joined the faculty of Dankook University, Korea, where he is currently a Professor in Department of Computer Science & Engineering (Graduate school) and Department of Software Science (Undergraduate school). He was a visiting research professor at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. His current research interests include computer security, mobile app security, operating systems and software intellectual property protection.

Hyoil Han is an Associate Professor in the School of Information Technology at Illinois State University, USA. She obtained her BS and MS degrees in Electrical Engineering from Korea University and Korea Advanced Institute of Science and Technology. She worked for Samsung Electronics and Korea Telecom before obtaining a Ph.D. in Computer Science and Engineering from the University of Texas at Arlington in 2002. Her research interests include machine learning, natural language processing, big data management, and applying AI to security.