THE JOURNAL OF
DIGITAL FORENSICS,
SECURITY AND LAW

Journal of Digital Forensics,
Security and Law

Volume 7 | Number 1 Article 4

2012

Comparing Android Applications to Find Copying

Larry Melling
Virtual System Platform Cadence Design Systems

Bob Zeidman
Zeidman Consulting

Comparing Android Applications to Find Copying

Larry Melling
Sr. Product Marketing Manager
Virtual System Platform Cadence Design Systems
USA
phone: 408-944-7432
fax: 408-910-2745
lmelling@cadence.com

Bob Zeidman
President
Zeidman Consulting
15565 Swiss Creek Lane
Cupertino, CA 95014 USA
phone: 408-517-1194
fax: 408-741-5231
Bob@ZeidmanConsulting.com

ABSTRACT

The Android smartphone operating system includes a Java virtual machine that enables rapid development and deployment of a wide variety of applications. The open nature of the platform means that reverse engineering of applications is relatively easy, and many developers are concerned as applications similar to their own show up in the Android marketplace and want to know if these applications are pirated. Fortunately, the same characteristics that make an Android application easy to reverse engineer and copy also provide opportunities for Android developers to compare downloaded applications to their own. This paper describes the process for comparing a developer's application with a downloaded application and defines an identifiability metric to quantify the degree to which an application can be identified by its bytecode.

General Terms: Android, Bytecode, Decompiled Code, Identifiability Metric, Java, Software Copying, Software Forensics, Software Plagiarism, Source Code.

Keywords: Android, BitMatch, Bytecode, CodeMatch, CodeSuite, Copying, Decompiling, Forensics, Identifiability, Intellectual Property, Java, Metrics, Plagiarism, Software, Source Code.

1. INTRODUCTION

In this paper we describe how to compare an Android application's source code with any downloaded Android application to find signs of copying. Many Android developers, and Android game developers in particular, are finding their applications being pirated from the online Android Marketplace (Ciancarini & Favini, 2009; Hornshaw, 2011).

We had a goal to define a comparison methodology and develop an "identifiability" metric to quantify how well a downloaded application could be identified from its bytecode. One purpose of the comparison is to determine whether a downloaded application was copied from another application, possibly leading to a copyright infringement charge. One purpose of the metric is to determine how much of an application's identifying information can still be obtained after its source code has been compiled into bytecode. Identifiability can be a positive or negative characteristic. A program that is easily identifiable after compilation may be easier to detect when it has been pirated, even if it is subsequently modified. A program that is difficult to identify after compilation may hide more of its trade secrets from reverse engineering and theft.

In this paper we present a case study that compares seven different Android Sudoku games applications and defines a measure called "identifiability" that represents how well the source code of an application can be identified by its compiled bytecode.

2. THE COMPONENTS OF AN ANDROID APPLICATION

Some programming languages, like the Java programming language, use a combination of compilers and interpreters. The Java compiler first turns the human-readable source code into intermediate code called "bytecode" that is a combination of computer-readable binary and human-readable text. A "Virtual Machine" ("VM") is a kind of interpreter that reads the bytecode and instructs the computer to perform the appropriate instructions. Android applications consist of bytecode that is delivered in an Android Package file (APK), a compressed archive file. Once unpacked the contents of the APK include:

- assets directory: This directory contains an unstructured hierarchy of files, defined by the app developer, for files that are retrievable as raw byte streams.
- META-INF directory: This directory stores signature data that allows the application to verify that the APK download and expansion completed successfully.
- res directory: This directory is used to store resource files for the application and includes information for the layout, names, and other elements used by the application.
- AndroidManifest.xml file: This is a required file that contains the application name, version, access rights referenced library files, etc.
- resources.arsc file: This is the binary resources file after compilation.

- classes.dex file: This is the Java bytecode file that will run on the Dalvik virtual machine used by Android and is not compatible with the typical Java virtual machine.

To find signs of copying, the APK has two categories of files to examine: the non-software source files (i.e. the AndroidManifest.xml file, the resource files, and the asset files) and the software bytecode (i.e. the classes.dex file).

3. EXAMINING THE NON-SOFTWARE SOURCE FILES

3.1 AndroidManifest.xml Files

To extract the manifest file in a readable form, we used the apktool (Google Code, 2011b) program. The extracted manifest file content is described in the Android developer documentation (Android Developers, 2012a):

Among other things, the manifest does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application—the activities, services, broadcast receivers, and content providers that the application is composed of.
- It names the Java classes that implement each of the components and their capabilities. These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the Android API and interact with other applications.
- It declares the permissions that other applications are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide application code profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries to which the application must be linked.

We visually inspected the manifest files of different applications to look for similarities. A utility like WinMerge or Diff can be used to find matches between two manifest files. We compared manifest files for different applications OpenSudoku (Google Code, 2011d)0 and Andoku (Google Code, 2011a) and no similarities were found. It is important to ignore similarities that are due to requirements of Android or are similar for reasons other than copying, such as Android APIs (Software Analysis & Forensic Engineering Corp., 2012).

3.2 Resource Files

In addition to the manifest, a res directory of folders and files was also examined. These files are resource files in XML format. The Android Developer documentation describes the importance of using resources (Android Developers, 2012b):

> You should always externalize resources such as images and strings from your application code, so that you can maintain them independently. Externalizing your resources also allows you to provide alternative resources that support specific device configurations such as different languages or screen sizes, which becomes increasingly important as more Android-powered devices become available with different configurations. In order to provide compatibility with different configurations, you must organize resources in your project's res/ directory, using various sub-directories that group resources by type and configuration.

Altova's DiffDog (Altova, 2012) utility made it easy to compare two res directories side by side. The tool automatically aligns directories with the same name and compares files with the same name. We compared resource files for the applications OpenSudoku and Andoku and no similarities were found. Again it is important to ignore similarities that are due to requirements of Android or are similar for reasons other than copying.

3.3 Asset Files

The assets directory contains a hierarchical directory of files used by the program. Asset files may be bitmapped images, HTML files, or any other type of file needed by the application. Not all applications have asset files. For the example applications, OpenSudoku had no asset files while Andoku did have asset files.

4. COMPARISON METHODOLOGY AND MEASUREMENTS

The bytecode for the application is found in the classes.dex file of the application APK. There are two approaches we considered for comparing the source code of one app to the bytecode of the downloaded app: 1) compare the bytecode form of the downloaded app or 2) decompile the bytecode into source code and compare the decompiled source code form of the downloaded app (Kalinovsky, 2004; Paller, 2009; Schulman, 2005a, 2005b, 2005c). We decided to try both approaches.

Selecting a tool to perform the comparisons was the next step.

4.1 Forensic Tool Selection

Working from bytecode means some information from the original source code will be lost, so a method to measure how much of the source code information is retained in the bytecode is important. A tool capable of examining bytecode is required. This requirement eliminated all but one of the forensic software analysis tools that are commercially available[1]. CodeSuite® by SAFE Corporation is the only tool that breaks down software into component elements and provides metrics on each of the component elements and thereby measures a baseline for source code coverage (Zeidman, 2006, 2008). It can also compare bytecode to source code (Software Analysis & Forensic Engineering Corp., 2011).

4.2 Identifiability Metric

We wanted a measure that signifies how easily application code can be identified after it has been compiled, because a goal of ours was to find out if a downloaded application was copied from the original application's source code. Some source code elements such as identifiers and strings remain in bytecode after source code is compiled into bytecode, while other source code elements such as statements and comments are usually removed during compilation[2]. As a basis for an identifiability measure we wanted to determine the percentage of source code elements that remain in an application's bytecode. We also wanted to decompile bytecode back into source code and again determine the percentage of the source code elements from the original source code that can be found in the resulting decompiled source code.

4.2.1 Source code element metrics

Bob Zeidman previously defined a process for examining source code to find copying that can be boiled down to: divide source code into basic elements, find all matches between elements of different programs, and then filter out matches that are not caused by copying (Zeidman, 2011). Based on this information, two measures for the source code elements were taken, the first is how many total elements exist and the second is how many of those elements are uncommon. Uncommon elements are more helpful at determining the identifiability of the application. Finding these uncommon elements in two different programs is a strong indicator that one may have been copied from the other (Zeidman, 2006, 2008, 2011).

Obtaining these metrics for an application involves running two CodeSuite tools. A CodeMatch® comparison of the application's source code to itself gives a list of all source code elements in the application. There are three types of elements that we consider: comments and strings (str), identifiers (id), and statements (stmt). The total number of source code elements of each type in a particular

[11] Note that we required a software forensics tool not a digital forensics tool. Our analysis is not about recovering data or determining the kind of data, but rather understanding the content of the data. CodeSuite is one of the few tools that analyze software on this level. See *The Software IP Detective's Handbook* (Zeidman, 2011), Chapter 9, for further clarification.

[2] A limitation of CodeMatch is that it lumps strings and comments together. For determining identifiability it would be better to consider these two source code elements separately.

application is represented as *SE*(str), *SE*(id), and *SE*(stmt). Running SourceDetective® then determines the number of times each source code element could be found on the Internet ("hits"). The Internet search hit count h is used to qualify the counts. In Table 1, these totals returned from CodeMatch and SourceDetective for the Android game OpenSudoku are shown. The numbers are taken from spreadsheets generated by CodeSuite. In this case, elements with less than 25 hits were considered uncommon and good potential indicators of copying, and elements with 0 hits were considered unique. Future researchers may want to test a different threshold than 25 hits for labeling a source code element as uncommon, but this number worked well in these tests and in our experience. Obviously an element that cannot be found elsewhere through an Internet search (i.e., has 0 hits) is unique to that application.

CodeMatch Metrics	**Total *SE*(str) + *SE*(id) +*SE*(stmt)**	**Comment/ String *SE*(str)**	**Identifier *SE*(id)**	**Statement *SE*(stmt)**
Total (*SE*)	**5,913**	**1,015**	**1,647**	**3,251**
Uncommon (h < 25 hits) (SE_{25})	**3,599**	**684**	**431**	**2,484**
Unique (h = 0 hits) (SE_0)	**3,171**	**621**	**324**	**2,226**

Table 1: CodeMatch analysis results of OpenSudoku source

4.2.2 Baseline for comparing bytecode

Next another CodeSuite tool, BitMatch®, was run to compare the application's source code with its own bytecode file (classes.dex). Then SourceDetective was run to generate the report of hits. This information is needed to create a baseline to quantify our likelihood of identifying copied code because we cannot expect better results comparing one application's bytecode to another application's source code (or bytecode) than when comparing one application's bytecode to its own source code.

Because CodeSuite provides these metrics by element type, it is valuable to define the identifiability metric by type as well as defining the total identifiability. There are three types of elements that we consider: comments and strings (str), identifiers (id), and statements (stmt)[3]. The number of source code elements that are also found in the application's bytecode are represented as *BE*(str), *BE*(id), and *BE*(stmt). The bytecode identifiability *IB* is the number of elements that can

[3] CodeMatch lumps comments and string together. Comments cannot be found in bytecode, so we refer to them simply as strings. Also, BitMatch extracts some text that it cannot determine to be strings or identifiers and so considers them to be both.

be found in the application's bytecode as a percentage of the total number of those elements in the application's source code. The Internet search hit count h is then used to qualify both the element (BE_h) and total (SE_h) counts so that identifiability can be determined for elements with h or fewer hits. Source code elements with high hit counts do not help uniquely identify an application while those with low hit counts do.

The formulas for calculating the identifiability of an application's bytecode are:

$$IB_h(\mathrm{str}) = BE_h(\mathrm{str})/SE_h(\mathrm{str})$$

$$IB_h(\mathrm{id}) = BE_h(\mathrm{id})/SE_h(\mathrm{id})$$

$$IB_h(\mathrm{stmt}) = BE_h(\mathrm{stmt})/SE_h(\mathrm{stmt})$$

$$IB_h= (BE_h(\mathrm{str})+BE_h(\mathrm{id})+BE_h(\mathrm{stmt}))/(SE_h(\mathrm{str})+SE_h(\mathrm{id})+SE_h(\mathrm{stmt}))$$

Table 2 shows the results for the analysis of the bytecode for the Android game OpenSudoku. As expected, because the bytecode does not include statements or comments from the source, the identifiability for statements was 0 and the comment/string identifiability comes only from strings. However, the identifiability for identifiers was high, which means that if code was copied, the comparison of bytecode with source code is very likely to find the copying (unless all of the identifiers were subsequently renamed). In addition, the coverage of unique identifiers (~90%) means that the compiling and packaging process did not eliminate many unique identifiers.

BitMatch Metrics	Total	String(str)	Identifier(id)
Elements (*BE*)	**1,513**	**227**	**1,286**
Identifiability (*IB*)	**25.6%**	**22.4%**	**78.1%**
Uncommon matches (BE_{25})	**443**	**44**	**399**
Uncommon identifiability (IB_{25})	**12.3%**	**6.4%**	**92.6%**
Unique matches (BE_0)	**326**	**34**	**292**
Unique identifiability (IB_0)	**10.3%**	**5.5%**	**90.1%**

Table 2: BitMatch analysis results of comparison of OpenSudoku's classes.dex with its source code

4.2.3 Baseline for comparing decompiled source with original source

We can measure the identifiability of the decompiled bytecode using the same methodology by comparing the application's source code to the source code from its decompiled bytecode.

4.2.3.1 Converting classes.dex to a JAR file

To get source code from the bytecode dex file requires decompiling. The JD-GUI decompiler (Java Decompiler, 2012) was selected (see section 0 for more information on the decompiler selection process). The decompiler works with either a Java archive (JAR) file or bytecode class files. The free dex2jar utility (Google Code, 2011c) was used to generate a JAR file from the classes.dex file.

4.2.3.2 Decompiling a JAR file

CodeMatch was used to compare the application's original source code with its decompiled source code. Then SourceDetective was run and decompiled code identifiability metrics were calculated. The number of source code elements that are also found in the application's decompiled bytecode are represented as *DE*(str), *DE*(id), and *DE*(stmt). The identifiability *ID* is the number of elements that can be found in the application's decompiled bytecode as a percentage of the total number of those elements in the application's original source code. The Internet search hit count h is then used to qualify both the element (DE_h) and total (SE_h) counts so that identifiability can be determined for elements with h or fewer hits. Source code elements with high hit counts do not help uniquely identify an application while those with low hit counts do.

The formulas for calculating the identifiability of an application's decompiled bytecode are:

$$ID_h(\mathrm{str}) = DE_h(\mathrm{str})/SE_h(\mathrm{str})$$

$$ID_h(\mathrm{id}) = DE_h(\mathrm{id})/SE_h(\mathrm{id})$$

$$ID_h(\mathrm{stmt}) = DE_h(\mathrm{stmt})/SE_h(\mathrm{stmt})$$

$$ID_h = (DE_h(\mathrm{str})+DE_h(\mathrm{id})+DE_h(\mathrm{stmt}))/(SE_h(\mathrm{str})+SE_h(\mathrm{id})+SE_h(\mathrm{stmt}))$$

CodeMatch Metrics	**Total**	**String (str)**	**Identifier (id)**	**Statement (stmt)**
Elements (*SE*)	**1,831**	**134**	**1,267**	**430**
Total Identifiability (*ID*)	**31.0%**	**13.2%**	**76.9%**	**13.2%**
Uncommon matches (SE_{25})	**697**	**52**	**393**	**252**
Uncommon Identifiability (ID_{25})	**19.4%**	**7.6%**	**91.2%**	**10.1%**
Unique matches (SE_0)	**572**	**43**	**304**	**225**
Unique Identifiability (ID_0)	**18.0%**	**6.9%**	**93.8%**	**10.1%**

Table 3: CodeMatch analysis results comparing OpenSudoku decompiled source code with its original source code

5. DECOMPILING ANDROID APPLICATIONS

Based on the results above it is clear the decompiler did not fully recreate the source code, so we wondered how good is the decompiled code? Table 4 shows the results of compiling the source code generated by the JD-GUI decompiler for three different applications. These results illustrate that the decompile process often does not produce source code that can be compiled or executed.

App	Compiles?	Executes?
Hello World	Yes	Yes
Notepadv1	No	No
OpenSudoku	No	No

Table 4: Validate JD-GUI decompiled code by attempting to compile and run

Because the decompiled code from JD-GUI does not compile, it made sense to look at other Java decompilers. The JAD decompiler (Varaneckas, 2001), another popular open source Java decompiler, was also tested using the same three applications. The results are shown in Table 5.

App	Compiles?	Executes?
Hello World	Yes	Yes
Notepadv1	No (warning class file version 50 not supported, but generated Java files)	No
OpenSudoku	No (errors and crashed decompiling CommandStack.class)	No

Table 5: Validate JAD decompiled code by attempting to compile and run

Based on this testing, while the JD-GUI decompiler didn't produce compilable code, it was selected because it was able to decompile all of the test cases while JAD failed to generate code in 2 out of the 3 cases tested.

Is decompiling a useful technique for identifying copying when source code is unavailable? The surprising result seen in Table 3 is that decompiling did improve the total identifiability. Comparing an application's source code with the source code that is decompiled from a suspect application's bytecode appears to be a slightly better way to detect copying than to directly compare an application's source code to a suspect application's bytecode. This is because bytecode decompilation produces source code statements that can then be compared, increasing the identifiability. And in general, being able to view source code will give you a better understanding of the context of any matching source code elements. The case study below can better illustrate how decompiling helps.

6. CASE STUDY: COMPARING DIFFERENT ANDROID SUDOKU APPLICATIONS

To illustrate the comparison methodology, we selected a variety of Android Sudoku applications for a case study. The Android Sudoku applications chosen were:

- Andoku
- Sudoku_bomb
- Enjoy Sudoku
- Mobile Sudoku
- Standard Sudoku
- Sudoku UI

Each of these was compared with OpenSudoku, the application used in the source code element coverage measures (see Section 0).

6.1 Bytecode to Source Code Comparison

The table below details the number of source code element matches found when comparing an application's bytecode with OpenSudoku's application source code.

Table 6 identifies application Andoku as having uncommon string and identifier matches with application OpenSudoku. The matched elements are listed in the CodeSuite report and shown in Table 7.

Because these elements are not commonly used—based upon Internet searches—the next step is to identify where they occur in the OpenSudoku application source code and how they are used. Searching the OpenSudoku source files for all occurrences of the elements shows that the matches occur in important files or code segments.

Application	Total element matches[4]	String matches total/uncommon/ unique	Identifier matches total/uncommon/ unique
Andoku	329	22 / 1 / 0	325 / 4 / 1
Sudoku_bomb	227	6 / 0 / 0	226 / 0 / 0
EnjoySudoku	387	14 / 0 / 0	384 / 0 / 0
Mobile37Sudoku	182	5 / 0 / 0	180 / 0 / 0
StandardSudoku	266	14 / 0 / 0	264 / 0 / 0
Sudoku.ui	204	12 / 0 / 0	202 / 0 / 0

Table 6: BitMatch results for app bytecode to OpenSudoku source code

Matching program elements	Search hits
Strings	
bad menuInfo	21
Identifiers	
DIALOG_RESET_PUZZLE	0
DIALOG_DELETE_FOLDER	1
EXTRA_FOLDER_ID	1
insertFolder	21

Table 7: OpenSudoku to Andoku uncommon matches

The string “bad menuInfo” is used in the OpenSudoku source code for error messaging.

The identifier “DIALOG_RESET_PUZZLE” is used in the OpenSudoku source code in a switch statement that controls the appearance of different dialogs based on a user’s action.

The identifier “EXTRA_FOLDER_ID” is found fifteen times in five different OpenSudoku source code files, all within the GUI.

This collection of information provides compelling evidence of possible copying between Andoku and OpenSudoku because it identifies matches in a number of different code files. Next the decompiled code is used to provide more context to this possible copying.

[4] Note that the total elements is less than the sum of string elements and identifier elements due to some elements being in both categories because BitMatch cannot be certain whether a single word is an identifier or string.

6.2 Decompiled Bytecode to Source Code Comparison

Table 8 shows the decompiled code results for all the applications. The analysis identifies Andoku with uncommon element matches.

App tested	Total Elements	String matches total/uncom/ unique	Identifier matches total/uncom/ unique	Statement matches total/uncom/ unique
Andoku	**468**	**10 / 1 / 0**	**376 / 5 / 1**	**82 / 1 / 0**
Sudoku_bomb	**300**	**2 / 0 / 0**	**247 / 0 / 0**	**51 / 1 /0**
EnjoySudoku	**417**	**3 / 0 / 0**	**337 / 0 / 0**	**77 / 1 / 0**
Mobile37Sudoku	**287**	**4 / 0 / 0**	**224 / 0 / 0**	**59 / 0 / 0**
StandardSudoku	**294**	**2 / 0 / 0**	**240 / 0 / 0**	**52 / 0 / 0**
Sudoku.ui	**363**	**7 / 0 / 0**	**296 / 0 / 0**	**60 / 1 / 0**

Table 8: CodeMatch results for comparing OpenSudoku source code to decompiled source code

In Table 9, the uncommon matches found between the Andoku decompiled byte code and OpenSudoku source code are listed.

Matching program elements	Search hits
Strings	
bad menuInfo	**21**
Identifiers	
DIALOG_RESET_PUZZLE	**0**
DIALOG_DELETE_FOLDER	**1**
EXTRA_FOLDER_ID	**1**
FolderListActivity	**1**
insertFolder	**21**
Statements	
Import android.widget.SimpleCursorAdapter.ViewBinder	**22**

Table 9: Uncommon matches between OpenSudoku and Andoku

A comparison of OpenSudoku source code with Andoku decompiled source code was performed. Even though close examination of the decompiled code revealed a functional error that would prevent the code from executing, the string “bad menuInfo” matched, the file name matched, the class name matched, and the method name matched, indicating that these two code segments have significant similarity.

We downloaded the actual Andoku source code (Google Code, 2011a) and found it to be nearly identical to the OpenSudoku source, thereby validating what our code analysis had flagged.

The other matched items from Table 9 also identified segments of code with copying. The additional information that the decompiled code provided gave context to the matched elements, offering more compelling evidence of copying.

7. CONCLUSIONS

In this paper we defined an identifiability metric for a software application that can give a developer an idea of how easy or difficult it is to identify bytecode as being derived from an application’s source code. A high identifiability means it will be easier to detect that another application was derived from or copied from the application. A low identifiability means that an application's trade secrets are better hidden.

In this paper, the viability of analyzing Android applications to discover possible copyright infringement without access to source code is demonstrated. The code comparison techniques identified uncommon element matches, offering developers an effective solution to identify code copying. The surprising result was that it was slightly more effective to use decompiled bytecode rather than bytecode in the comparison. It seems that decompiling puts information back into the code that is in the bytecode but difficult to identify.

While any evidence uncovered without access to source code may be compelling enough to convince a judge that there is reason for litigation, gaining access to source is ultimately needed to prove the extent of the copying. Because the techniques demonstrated apply to code that has been compiled and information has thus been removed, they do not cover 100% of the source code elements, and thus not finding any uncommon element matches does not disprove copying.

8. ACKNOWLEDGEMENTS

The authors would like to thank Jim Zamiska and Nik Baer for their detailed reviews and their in-depth discussions about the metrics and methodologies developed in this paper.

REFERENCES

Altova. (2012). DiffDog - XML-aware diff merge tool for file, folder, directory, and database differencing. Retrieved April 17, 2012, from http://www.altova.com/diffdog/diff-merge-tool.html

Android Developers. (2012a). The AndroidManifest.xml file. Retrieved April 17, 2012, from http://developer.android.com/guide/topics/manifest/manifest-intro.html

Android Developers. (2012b). Application Resources. Retrieved April 17, 2012, from http://developer.android.com/guide/topics/resources/index.html

Ciancarini, P. and Favini, G.P. (2009). Plagiarism detection in game-playing software. In *Proceedings of the 4th International Conference on Fondations of Digital Games*, Port Canaveral, FL, April 26-30, 2009.

Google Code. (2011a). ardorleo-p-andoku: p-andoku - soduko-puzzles clone. Retrieved April 18, 2012, from http://code.google.com/r/ardorleo-p-andoku

Google Code. (2011b). Android-Apktool: A tool for reverse engineering Android apk files. Retrieved April 17, 2012, from http://code.google.com/p/android-apktool

Google Code. (2011c). dex2jar: Tools to work with Android .dex and Java .class files. Retrieved April 17, 2012, from http://code.google.com/p/dex2jar

Google Code. (2011d). OpenSudoku-Android: Sudoku for Android. Retrieved April 17, 2012, from http://code.google.com/p/opensudoku-android

Hornshaw, P. (2011, March 18). Game developers struggle with piracy, malware in Android Market. *Appolicious Advisor*. Retrieved April 17, 2012, from http://www.androidapps.com/tech/articles/7177-game-developers-struggle-with-piracy-malware-in-android-market

Java Decompiler. (2012). Introduction. Retrieved April 17, 2012, from http://java.decompiler.free.fr

Kalinovsky, A. (2004). *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*. Indianapolis: Sams Publishing.

Paller, G. (2009). Understanding the Dalvik bytecode with the Dedexer tool. Retrieved April 17, 2012, from http://pallergabor.uw.hu/common/understandingdalvikbytecode.pdf

Schulman, A. (2005a, July 1). Finding Binary Clones with Opstrings & Function Digests: Part I. *Dr. Dobbs Journal*. Retrieved April 17, 2012, from http://drdobbs.com/184406152?queryText=Finding+Binary +Clones+with+Opstrings+%26amp%3B+Function

Schulman, A. (2005b, August 1). Finding Binary Clones with Opstrings & Function Digests: Part II. *Dr. Dobbs Journal*. Retrieved April 17, 2012, from http://drdobbs.com/184406203?queryText=Finding+Binary +Clones+with+Opstrings+%26amp%3B+Function

Schulman, A. (2005c, September 1). Finding Binary Clones with Opstrings & Function Digests: Part III. *Dr. Dobbs Journal*. Retrieved April 17, 2012, from http://drdobbs.com/tools/184406247?queryText=Finding +Binary+Clones+with+Opstrings+%26amp%3B+Function

Software Analysis & Forensic Engineering Corp. (2011). *CodeSuite User's Guide*, v4.3. Retrieved April 17, 2012, from http://www.safe-corp.biz/documents/CodeSuite%20Users%20Guide.pdf

Software Analysis & Forensic Engineering Corp. (2012) Our Process. Retrieved April 17, 2012, from http://safe-corp.biz/company_process.htm

Varaneckas, T. (2001) JAD Java Decompiler Download Mirror. Retrieved April 17, 2012, from http://www.varaneckas.com/jad

Zeidman, R. (2006). Software Source Code Correlation. In *Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06),* August 10-12/2006, Honolulu, HI.

Zeidman, R. (2008). Multidimensional Correlation of Software Source Code. In *Proceedings of the Third International Workshop on Systematic Approaches to Digital Forensic Engineering*, May 22, 2008, Oakland, CA.

Zeidman, B. (2011). *The Software IP Detective's Handbook*. Westford, MA: Prentice Hall.

AUTHOR BIOGRAPHIES

Larry Melling is a research engineer at Zeidman Consulting. He has over 30 years of executive management and engineering experience in developing new hardware and software technologies and bringing them to market. He has been engaged in applications engineering and marketing of electronic design automation (EDA) tools at major companies and small startups. He has also been involved in the development of sophisticated tools for source code and object code analysis for finding intellectual property infringement.

Bob Zeidman is a Senior Member of the IEEE, the president of Zeidman Consulting and the president of Software Analysis and Forensic Engineering. Among his publications are technical papers on hardware and software design methods as well as four textbooks – The Software IP Detective's Handbook, Designing with FPGAs and CPLDs, Verilog Designer's Library, and Introduction to Verilog. He has taught courses at engineering conferences throughout the world. Bob holds several patents. He earned a master's degree in electrical engineering at Stanford University and bachelor's degrees in physics and electrical engineering at Cornell University.