

Reverse Engineering Android Apps With CodeInspect

Siegfried Rasthofer¹, Steven Arzt¹, Marc Miltenberger¹, and Eric Bodden²

¹Fraunhofer SIT & TU Darmstadt, Darmstadt, Germany

²Paderborn University & Fraunhofer IEM, Paderborn, Germany

Abstract

While the Android operating system is popular among users, it has also attracted a broad variety of miscreants and malware. New samples are discovered every day. Purely automatic analysis is often not enough for understanding current state-of-the-art Android malware, though. Miscreants obfuscate and encrypt their code, or hide secrets in native code. Precisely identifying the malware’s behavior and finding information about its potential authors requires tools that assist human experts in a manual investigation. In this paper, we present `CodeInspect`, a novel reverse engineering tool for Android app that optimally supports investigators and analysts in that task.

1 Introduction

Mobile devices such as smartphones and tablets are increasingly used in everyday life and have long since become essential tools. This success is primarily due to the availability of apps for almost every need. While this abundance is helpful for users, it also attracts miscreants. Stealing sensitive user information or directly incurring charges on them is a profitable, albeit illegal business model. As Android has the largest market share among mobile operating systems [8], most malware is developed for Android as well. The rate with which new malware appears in the wild increases by the year [12].

Many approaches for automatically detecting Android malware have been proposed in the academic literature [1, 6, 5] and implemented into practical tools such as Drebin [1] or Chabada [6]. While automation

is crucial for mass analysis, these tools face challenges for highly obfuscated state-of-the-art malware and is usually completely ineffective for novel or targeted attacks. In these cases, to understand the behavior of a given sample the analyst must resort to manual labor. Furthermore, she usually needs to gather additional information such as potential hints on the miscreants behind the malware. Remote URLs, telephone numbers, e-mail addresses, or even coding patterns can give valuable insights to defenders and prosecutors alike. Though approaches exist to extract information from apps automatically [11, 10, 14, 7], gaining a complete understanding of a malware sample usually requires manual inspection.

With today’s numbers of new samples arriving every day, it has become of utmost importance to make manual investigations as efficient as possible. The analysis tool should thus support the human expert to reduce the mechanical parts of the investigation, allowing the human to focus on understanding the threat. In this paper, we present `CodeInspect`, a novel reverse-engineering tool for Android applications. `CodeInspect` features an expressive intermediate language with type information for local variables, an interactive debugger, and various Android-specific analyses such as data-flow tracking and permissions-usage scanning. We show how `CodeInspect` can be used to analyze a complex real-world malware sample [13] within less than one hour.

The remainder of this paper is structured as follows. In Section 2, we introduce the malware that will serve as a running example in this paper. Afterwards, we give an overview over `CodeInspect` in Section 3. In Section 4, we show how we used `CodeInspect` to reverse engineer the malware, before we conclude in Section 5.

2 Android/BadAccent Malware

The Android/BadAccents malware family was discovered in 2015 [13] and is a banking trojan that uses different obfuscation techniques. Its design is modular with several features such as SMS stealing, social engineering, and uninstalling AV apps. Furthermore,

Copyright © by the paper’s authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

In: D. Aspinall, L. Cavallaro, M. N. Seghir, M. Volkamer (eds.): Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS’16, London, UK, 06-April-2016, published at <http://ceur-ws.org>

the malware is designed to evade automatic detection approaches. To completely understand the behavior of the malware, a manual investigation is necessary. Out of the various components, we focus on the *SMS Interception* component, which intercepts incoming SMS messages and forwards them to the attacker. This is done in an attempt to obtain mobile transaction numbers (mTAN), which can then be used to conduct fraudulent transaction at the user's expense. For the investigator, it is important to understand where the stolen information is sent as any target address may give clues on the identity of the miscreant running the scam. From a previous investigation, we knew that the malware sends some information via e-mail. It was, however, unclear where the e-mails were sent to, and whether additional channels existed. Therefore, finding the target mail address and possible other channels was the focus of the manual investigation at hand.

3 CodeInspect Overview

As seen in the introduction, some situations require binary software to be inspected manually. Analysts can use existing command-line tools such as *APKtool* to decompile the binary APK file into readable text source. This tool, however, creates smali code. Smali is an untyped assembly language, leaving the analyst with the challenging task as making sense to registers operations and explicit reference management on the heap. Filling data structures, for instance, is a complex set of heap navigation instructions in smali. Furthermore, disassembly files only give a static look on the malware. They do not easily allow for runtime inspection.

A powerful IDE such as *CodeInspect*, on the other hand, is much more convenient to use. The tool is based on the Eclipse [4] IDE, so that developers usually have an intuition on how to use it. *CodeInspect* converts the APK file into a typed, higher-level intermediate representation that is much more convenient to read than smali. The code editor provides the analyst with syntax highlighting and navigation capabilities that allow the analyst to e.g., jump to the definition of a symbol of interest. The *CodeInspect* IDE allows the analyst to work with the decompiled code on a semantic, rather than just a textual level. If the analyst, e.g., searches for a specific method, she will only find occurrences of that method name, not arbitrary strings that happen to have the same name. *CodeInspect* can import APK file either directly if they are available on the analyst's machine, or it can load them from a real tablet or phone on which the respective apps are installed.

Besides reverse engineering for malware analysis, *CodeInspect* can also be used to analyze benign apps

in detail. Bundled third party libraries such as advertisement libraries are usually considered safe, although they might pose a security or privacy risk to the user of the application. The library code is often not available and, thus, cannot be checked by the app developers. *CodeInspect*, however, enables developers to validate the behavior of the compiled application including the actions performed by the libraries. Similar challenges arise when outsourcing app development to third parties that only deliver the binaries of the developed app, but not the source code. In that case, the purchasing company also need powerful analysis tools to look into the delivered black box. Otherwise, that black box development could contain serious security flaws or even malicious code that goes undetected.

3.1 Jimple Intermediate Representation

CodeInspect relies on the Soot framework for program analysis and transformation [9]. The Soot framework takes an Android application as input and converts it into a human readable type-based intermediate representation called Jimple [15]. From now on, all code analyses and transformations are performed on this intermediate representation rather than the original bytecode. Soot also offers the possibility to convert the (potentially modified) Jimple code back into an Android binary. *CodeInspect* inherits this feature and allows the analyst to modify the app, for instance to remove emulator checks or other challenges to the analysis. The human expert can also refactor the app to integrate conclusions she has already drawn about the app, e.g., by renaming methods to what their actual task is instead of some obfuscated name. The analyst can also merge additional Jimple or Java code into the app. With this feature, she can, for instance, implement a decryption method for some obfuscated strings in Java and use them during a dynamic analysis to better understand the original data processed inside the app. *CodeInspect* automatically merges the original app code and the new additions at compile time.

Although actual Java source code might be even easier to understand than Jimple, it is not always possible to decompile an app's bytecode back into valid Java code. The Dalvik bytecode language that Android uses allows for constructs that have no equivalent in Java, such as unconditional nested jumps (*goto* instructions). Existing obfuscators [3] allow to easily transform an app into such a non-reversible form. As long as the app still contains valid bytecode (i.e., runs on the device), it can, however, be represented in Jimple. This makes Jimple the ideal middle ground between bytecode and Java source code. It also makes sure that *CodeInspect* can re-compile every app (with potential changes from the analyst) and inspect its be-

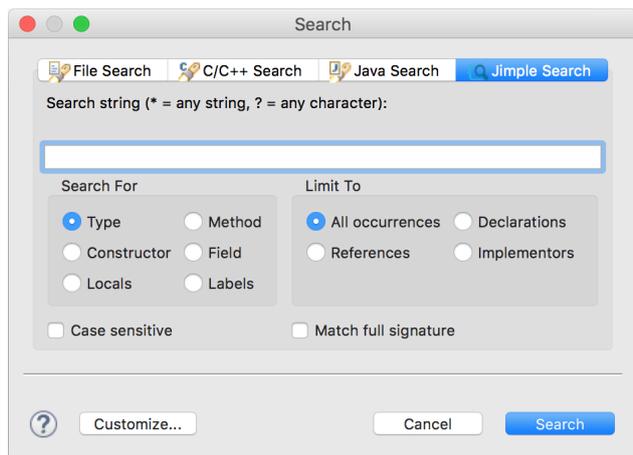


Figure 1: Jimple Search

```
public static boolean UsbAutoRunAttack(android.content.Context $param0)
{
    java.lang.String $SString;
    $SString = <smart.apps.droidcleaner.Tools: java.lang.String urlServer>;
    ...
    staticinvoke <smart.apps.droidcleaner.Tools: boolean
        DownloadFile(java.lang.String, java.lang.String, java.lang.String,
            java.lang.String, android.content.Context)>
        ($SString, "autorun.inf", "ftpuppper", "secret007", $param0);
    return true;
}
```

Figure 2: Jimple Code Snippet

havior dynamically. For those cases in which a Java decompilation is possible, CodeInspect integrates an existing state-of-the-art Java decompiler in a best-effort approach.

CodeInspect’s Jimple editor behaves similarly to a normal Eclipse code editor for Java. The user may modify the code as she wishes. When she saves her changes, the code is automatically recompiled. As a result, she receives a modified application as a new APK file, which behaves like the original application, but with the changed code.

In the same way, due to Soot’s class file feature, a main method can be written in Java, which calls the aforementioned “decryption” method from Java code, which runs on the JVM on the computer. In case the app loads a dex file at runtime, one can also inspect its code and debug it via the Dex file merge feature. It merges the code from an available dex file and adds it to an existing project.

The user also may search for specific fields, methods, classes as well as their usages using the Jimple search, as shown in figure 1.

Figure 2 shows the Jimple representation of a method taken from a real-world malware app. The body a Jimple method can be divided into two parts.

The first part contains the variable declarations. The second part contains the actual Jimple instructions. In this example, we see a read access to a field (`urlServer`) in the first line and a method call (`DownloadFile`) in the second line. In total, this example loads a file from a server on the web and specifies the user name and password required to access the file.

3.2 Project Explorer

In the normal Eclipse project explorer, CodeInspect lists all parts of the decompiled Android app. This includes not only the code, but also the manifest xml file (in human-readable form), the assets bundled with the app, the native libraries, and the layout XML files. The user is free to inspect and modify all of these files. For opening the manifest or the layout XML files, CodeInspect contains the Android ADT components for Eclipse. A layout file will thus be shown in graphical UI editor in addition to the plain XML representation. All the different editors are linked; if the user clicks on a class name in one editor (e.g., the manifest file), she can directly jump to the respective code in the Jimple code editor. These links are also automatically updated when the code is changed. If the analyst, for instance, renames an activity, the respective manifest entry will also be adapted automatically.

3.3 Debugging

To debug the application, it is not required to root the phone. Debugging may be performed on an emulator or a real device; the only requirement is that the Developer mode is activated on the device. The *Auto Stepper* view automates stepping through the application under analysis. If activated, it steps through the code in a predefined frequency. Of course, it is also possible to manually set breakpoints, jump into or over method calls, jump back from a method to its caller, or drop the execution pointer to the current stack frame. CodeInspect’s debugger is as powerful as the original Android debugger that app developers use on their source code.

3.4 Android-Specific Analyses

CodeInspect extends the normal Eclipse IDE with various Android-specific analyses that give a human expert more information on the current app. These analyses are implemented as additional views. The *Permission Usage View* (Figure 3) lists all the permissions requested by the app. For every permission, it also reports all locations in the code where the respective permission is required. More precisely, CodeInspect identifies all API calls that would fail if the respective permission were not present. This information

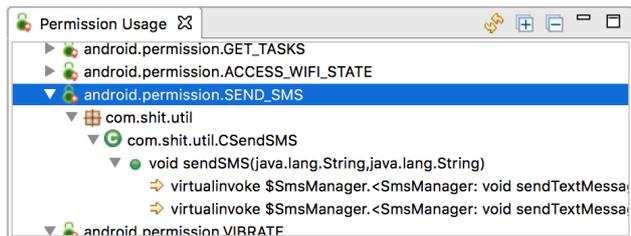


Figure 3: Permission Usage View

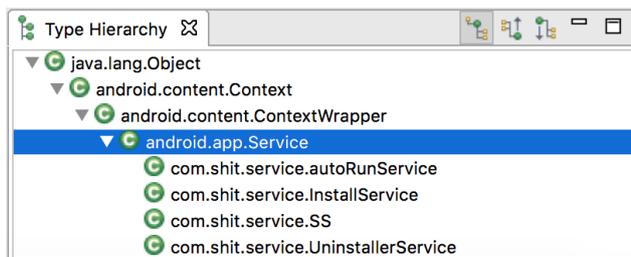


Figure 4: Type Hierarchy View

can give the analyst a first hint to potentially malicious behavior in the app, for instance in the case of SMS trojans.

Furthermore, the normal Java-based code analyses known from Eclipse are also available in CodeInspect. Similar to the Java Eclipse IDE Plug-In (JDT), CodeInspect contains a type hierarchy view, which can be used to examine inheritance and interface implementation relationships. Figure 4 shows the type hierarchy of the Service class, thus showing all Services of an application. Besides, we have implemented a call hierarchy view. Figure 5 shows all potential callers of method `getSDPath1`. This might give clues on how the respective method is used. In case of the `getSDPath1` method, the view shows that this method gets called from `deleteFoder1`, hinting that the app is trying to delete a folder.

3.5 Plugins

As CodeInspect has been developed at an academic institution, we have also compiled our research results into CodeInspect plugins. FlowDroid [2] is a popular static information flow tracker for Java and Android, which can be used to detect unwanted or potentially dangerous data flows. FlowDroid is tightly integrated into CodeInspect through CodeInspect’s extensible plug-in interface. This allows the analyst to configure and conduct data flow analyses easily and efficiently. The results are graphically displayed inside CodeInspect as shown in Figure 6. The user can select different data flows, represented by data sources and sinks. After selecting a particular data flow, the plug-in highlights the corresponding statements that are re-

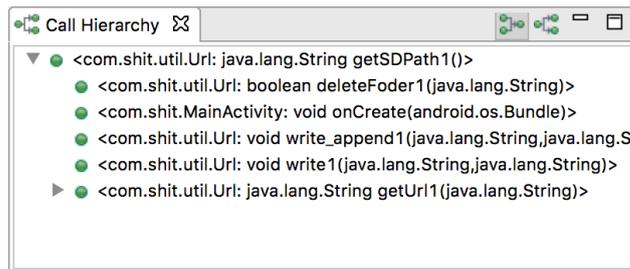


Figure 5: Call Hierarchy View

sponsible for the information flow and shows all relevant statements in the *Calltrace View*. With a click, the user can jump from the flow results directly into the source code.

To fully understand an Android app, it is often important to know certain runtime values. If the analyst has found out that a malware application leaks data via SMS, she must then find the target telephone number. For communication with a remote command&control server, she is interested in the URL of that server. Such values are, however, often not available in the app as plain text, but are obfuscated. Their final value only gets decrypted or computed at runtime. Manually undoing obfuscations is a cumbersome and inefficient task. We therefore provide Harvester [11], an approach that fully automatically extracts such runtime values from Android applications. It can also be used for e.g., deobfuscating reflective method calls by first finding the correct target method signature and then replacing the reflective call with a direct one. Harvester is well-integrated into CodeInspect. The user can select a code positions (e.g. method arguments) from which she wants to extract runtime values. Harvester will then fully automatically extract possible runtime values for this particular set of variables. Similarly, she can select the reflective method calls she wants to simplify. Harvester then looks for the receiver method and injects a direct call.

CodeInspect’s plugin interface is also open to other developers who want to extend the tool with additional functionality. As CodeInspect is based on Eclipse, normal Eclipse plugins can be used to provide new features such as support for more version control systems or specific file type editors.

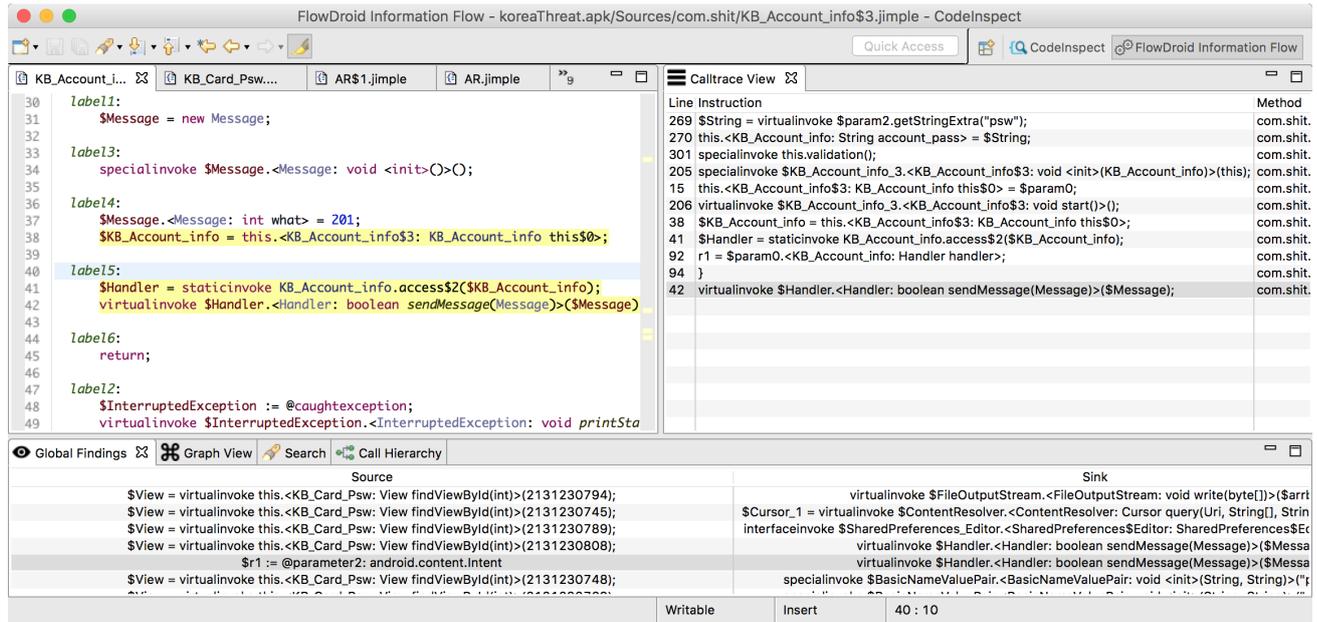


Figure 6: FlowDroid Plug-In

4 Reverse Engineering Bad/Accents Malware With CodeInspect

In this Section, we explain how CodeInspect can be used in a malware investigation. Such a manual reverse engineering task is usually important if automated approaches, such as a behavior analysis in a sandbox [10, 14], provide no or only little evidence on questions such as *which data is stolen?*, or *where is the data sent to?*. In such cases, a manual inspection is usually the only solution. We take the malware investigation [13] of the Android/BadAccents malware as an example to demonstrate the need for manual reverse engineering. An automated pre-analysis of this malware family hinted at a banking trojan, but it was not clear to the malware analyst where the stolen data from the SMS intercepting component of the malware was sent to. In the following we will explain in detail how one could use CodeInspect during investigations such as the one on Android/BadAccents. The goal is to answer detailed questions such as the receivers of stolen data on potentially obfuscated code.

4.1 APK Overview

The Android manifest provides a good starting point in a manual malware investigation since it contains information about the different components of the app along with additional meta-information. In the context of the Android/BadAccents malware, one can identify that `com.shit.MainActivity` is the class of the main activity, which gets called first when the user opens an application. Furthermore, the manifest also contains a broadcast receiver called `com.a.a.AR` which, in turn,

has an `onReceive()` method that gets executed whenever the device receives a new SMS message. Note that the receiver also reacts on other actions besides the `SMS_RECEIVED`, but these actions are less important. The main activity and the receiver class are two interesting parts, which needs to be analyzed in more detail.

Besides the manifest, one can also use CodeInspect’s search function. Since we are looking for user e-mail credentials, a simple lookup for “password” or “user” might return interesting code positions. Indeed, Android/BadAccents contains a few code statements that contain the two search words. The most interesting code location is the one shown in the lower part of Figure 7. It contains two api calls `stringUser()` and `stringPassword()` which are call native methods. These methods take no parameters and return strings, i.e., are simple native *getters*. The code section was executed in the `onCreate()` method of the main activity and the api calls were directly executed without any special triggering.

Now, one could either reverse engineer the native implementation of the two api calls or proceed with a dynamic analysis. We could have extracted the native code into a new app and called the two methods there to find the returned string values. Debugging the original malware app, however, required even less effort as we explain in the next section.

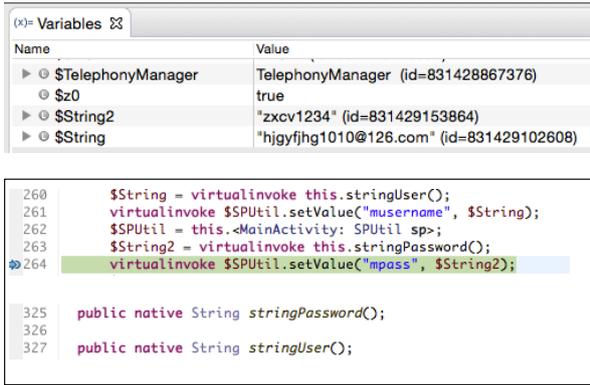


Figure 7: Access to Email Credentials in the Main Activity

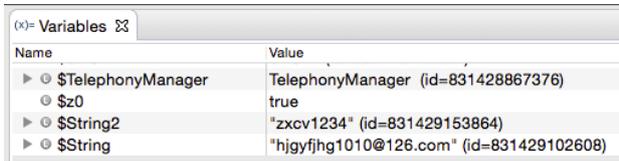


Figure 8: Runtime Values of Variables

4.2 Detailed Manual Analysis

One of the most powerful features of CodeInspect is the interactive debugger. It allows a human analyst to perform single-step debugging on the Jimple code. Whenever the execution is halted, she can examine the current contents of all variables in scope in the live variables view. This was very useful for extracting the concrete username and password which are returned from the two native API calls. A breakpoint in line 264 in Figure 7 stops the program at that point and gives the analyst the possibility to view the runtime values of the variable `$String` and `$String2`. These two values are temporarily stored on the file system and are later used for sending the stolen data to the attacker’s email account via the SMTP protocol. This answers the first question in our investigation. We have information about email credentials, which were hidden in native code. However, we do not have a proof that these credentials are actually used for authentication.

A quick search for “mail” in the Jimple code shows that the application contains a method called “MailSend” which sounds like the method which is responsible for sending emails to the attacker. We use the “Open Call Hierarchy” feature of CodeInspect (see Figure 9) and discover that the “MailSend” method gets triggered by the “onReceive” method which gets executed once the application receives an SMS (see Section 4.1). This shows that SMS data is indeed stolen and leaked via e-mail.

As a next step, we send an SMS message (number



Figure 9: Call Hierarchy View for the MailSend Method

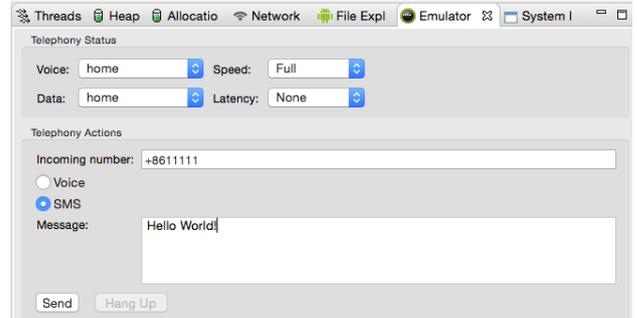


Figure 10: Sending an SMS Message to the Emulator via CodeInspect

+861111” and text “Hello World!”) to the emulator as shown in Figure 10. After single stepping through the Jimple code, we hit an interesting Code section (see Figure 11), which checks whether the incoming number starts with “+86” or “+82” which shows that the malware is expecting SMS messages from China or South Korea. This is an interesting result from the investigation which shows that the malware is especially targeting users from China or South Korea.

Further stepping through the code leads to another interesting code section as shown in Figure 12. Here we can see that the malware expects a certain SMS text “ak40_1”. The purpose of this special command is to activate and deactivate stealing the incoming SMS messages. After sending the “ak40_1” command to the emulator, all further incoming SMS messages are intercepted and sent to the attacker. Figure 13 shows the usage of the “MailSend” method which leaks the incoming SMS message to the attacker with the cre-

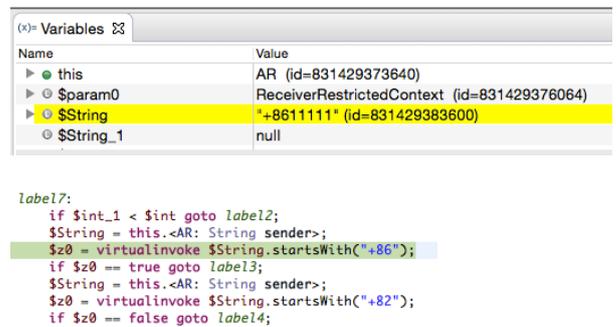


Figure 11: Incoming SMS Number Check

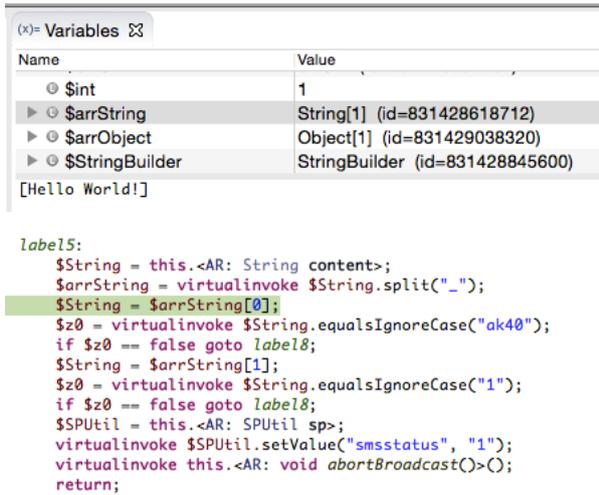


Figure 12: Activation Command for Stealing Incoming SMS Messages

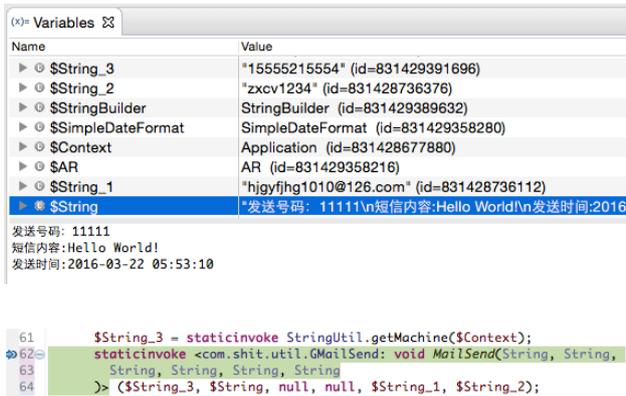


Figure 13: Variables View of the Debugger at the MailSend API Call

dentials that we have previously identified. This concludes the investigation on the malware’s e-mail interface. We have all information which were necessary to proof that the Android/BadAccents steals incoming SMS messages and leaks them to the attacker via email.

5 Conclusion

In this paper, we have presented CodeInspect, a novel tool for manually reverse engineering malicious Android apps. The tool supports the human expert with an expressive, typed intermediate representation, an interactive debugger, and various Android-specific analyses. It greatly reduces the effort of the investigation. As future work, we plan to integrate more analysis techniques and views into the tool.

References

- [1] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014.
- [2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, June 2014.
- [3] Michael Batchelder and Laurie J. Hendren. Obfuscating java: The most pain for the least gain. In Shriram Krishnamurthi and Martin Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2007.
- [4] W. Beaton and J. d. Rivieres. Eclipse platform technical overview. Technical report, The Eclipse Foundation, 2006.
- [5] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '13*, pages 13–24, New York, NY, USA, 2013. ACM.
- [6] Alessandra Gorla, Iliaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*, 2014.
- [7] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 1844–1851, New York, NY, USA, 2013. ACM.
- [8] Kantar. Android returns to growth in europe’s big five markets. whitepaper, 2015.
- [9] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oktober 2011.

- [10] Martina Lindorfer, Matthias Neugschwandner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, Wroclaw, Poland, September 2014.
- [11] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *2016 Network and Distributed System Security Symposium (NDSS)*, 2016.
- [12] Pulse Secure. Mobile threat report 2015. whitepaper, 2015.
- [13] Stephan Huber Siegfried Rasthofer, Irfan Asrar and Eric Bodden. How current android malware seeks to evade automated code analysis. In *9th International Conference on Information Security Theory and Practice (WISTP'2015)*, 2015.
- [14] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 1808–1815, New York, NY, USA, 2013. ACM.
- [15] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.