Blocking Advertisements on Android Devices using Monitoring Techniques

Khalil El-Harake, Yliès Falcone, Wassim Jerad, Mattieu Langet, Mariem Mamlouk

Blocking Advertisements on Android Devices using Monitoring Techniques ⋆ ⋆⋆

Khalil El-Harake, Yliès Falcone, Wassim Jerad, Mattieu Langet, and Mariem Mamlouk

Laboratoire d'Informatique de Grenoble, Vérimag, University of Grenoble-Alpes, France
First.Last@imag.fr

Abstract. This paper explores the effectiveness and challenges of using monitoring techniques, based on Aspect-Oriented Programming, to block adware at the library level, on mobile devices based on Android. Our method is systematic and general: it can be applied to block advertisements from existing and future advertisement networks. We also present miAdBlocker, an industrial proof-of-concept application, based on this technique, for disabling advertisements on a per-application basis. Our experimental results show a high success rate on most applications. Finally, we present the lessons learned from this experience, and we identify some challenges when applying runtime monitoring techniques to real-world case studies.

1 Introduction

Smartphone usage has dramatically increased over the past decade, presently accounting for 57.6% of mobile devices. On mobile devices, Android [1], is the leading platform holding 78.4% of the market [2]. The downside is that the popularity of Android made it a primary target of adware: studies show that 49% of the applications on the market are bundled with at least one ad library [3]. It has also become common practice for application developers to bundle multiple advertisement libraries into their software.

The prevalence of adware, reduces device performance, detracts from user experience, significantly contributes to battery drain [4], and raises privacy concerns through the collection of sensitive information (such as user location) [5].

In this paper we present how monitoring techniques can be used to disable advertisements in Android applications. More particularly, we are interested in enforcement monitoring where a so-called enforcement monitor receives the sensitive events from the application under scrutiny and uses an internal decision procedure to determine whether each event should be allowed or not. Using Aspect-Oriented Programming [6] (AOP), we insert, at the bytecode-level, enforcement monitors that give users the ability to disable advertisements in Android on a per-application basis. Our technique minimally modifies a targeted application in the sense that only the initial behavior related to

⋆ The work presented in this paper is partially funded by Institut Carnot LSI.

⋆⋆ This paper is an academic study of the effectiveness of using monitoring techniques on a large-scale and challenging case study. By no means it should be seen as an attempt to actually suppress advertisements in applications nor to jeopardize the source of income of the actors involved in the Android ecosystem.

the display of advertisements is impacted while the rest of the host application functions normally.

Unlike other methods that work by modifying the host operating system, our method works on an unrooted stock Android device. Our method also differs from similar solutions that perform bytecode transformation, and relies on custom security languages, or low-level transformation using intermediate representations of bytecode [7, 8]. We rely on Aspect-Oriented Programming, via the standard AspectJ compiler [9], which developers are more likely to be familiar with. A detailed comparison with related work, can be found in Sec. 7.

Contributions. The contributions of this paper are to:

- Introduce the use of AOP as a method for disabling ads on Android applications;
- Present miAdBlocker, an end-user application, implementing the technique;
- Discuss results evaluating the approach in practice;
- Explore the limitations of using AOP to modify closed-source applications and block advertisements.

Paper Organization. The rest of this paper is structured as follows. Section 2 presents background notions. The method for suppressing advertisements is presented in Sec. 3. Section 4 presents miAdBlocker, our industrial proof-of-concept that implements the method presented in Sec. 3. miAdBlocker comprises i) a completely re-developed version of Weave Droid [10], ii) an aspect that allows to suppress advertisements in many Android applications, and adds user-oriented features. Section 5 presents our evaluation of the method on a sample of 860 popular Android applications retrieved "off-the-shelf" from Google Play. In Sec. 6, we discuss some of the issues (and possible counter measures) encountered when applying miAdBlocker to Android applications. Section 7 discusses related work. Finally, Sec. 8 presents some concluding remarks and open perspectives.

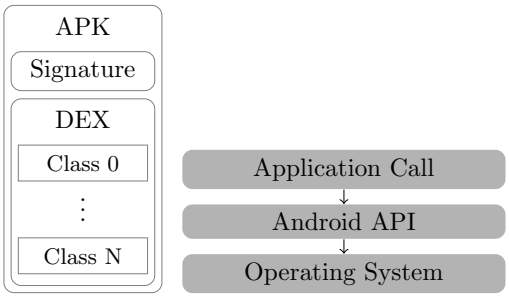
2 Background

This section presents some background notions on Android, advertisement libraries, aspect-oriented programming used in our approach.

2.1 Android and Advertisement Librairies

Android is an open-source operating system based on Linux. Android is primarily used on mobile devices such as smartphones and tablets. Android applications are primarily developed in Java, and while it is possible to use native code for development, only 4.52% of applications on the market use it [11]. Unlike typical Java applications which run on a Java Virtual Machine (JVM), Android applications use the Dalvik Virtual Machine (DVM). The DVM and JVM have significant differences, such as differences in bytecode encoding, their differences and resulting problems are discussed briefly in Sec. 6.

Android applications are distributed as APK (Android Package) files (see Fig. 1(a)). APK files consist of the application's manifest, resources, application bytecode encoded for the DVM as a single `classes.dex` file, and signatures over the APK file for verifying its authenticity. An Android application runs in its own process, with its own Dalvik Virtual Machine (DVM) instance. When a method call to a privileged resource



(a) Simplified structure of an APK.

(b) Android API call.

Fig. 1. General Information on the functioning of Android Application.

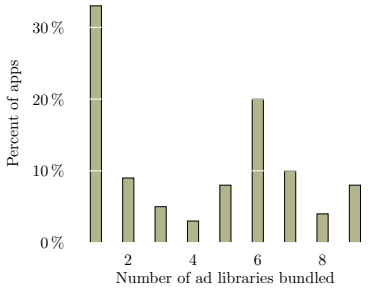


Fig. 2. Number of bundled advertisement libraries in applications.

is made, the call goes through the Android API, see Fig. 1(b), and the application framework checks if the originating application has the permission for proceeding with the request.

Advertisement libraries are often bundled with Android applications. An analysis of 100 applications containing advertisements on the market revealed that 40% of the applications contained 6 or more advertisement libraries, see Fig. 2. While another study, showed that 35% of applications contained 2 or more advertisement libraries [12].

Android does not have a built-in, in-process permission separation mechanism for libraries. As with all libraries bundled into an Android application, these advertisement libraries share the access permissions of the host application.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is an established paradigm developed in the 1990s at Xerox PARC [13]. AOP aims to facilitate modularity through the use of *aspects*, with each aspect being the embodiment of a so-called *cross-cutting concern*, i.e., parts of a program that rely on or must affect other parts of the system.

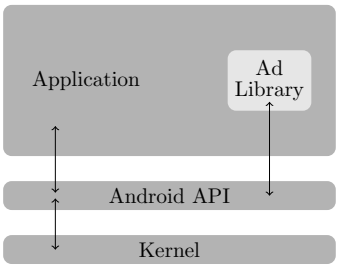


Fig. 3. View of an application at runtime with an ad library.

Aspects are implemented through the use of three main concepts: joint points, pointcuts, and advices.

Joint-Point: A join point is an identifiable point during the runtime of a program, such as on the execution or call of a method.

PointCut: A pointcut is an expression for matching on joint points, for example the below pointcut matches on join points where a call to a method in the package `com.google.ads` is made.

```
call(* com.google.ads..*(..))
```

Advice: An advice is a piece of code that can be attached to run either after, before, or around a pointcut. For example, by using an *around* advice on a method, one can decide to proceed or not with the actual method call. Context information for making decisions regarding granting permissions can also be obtained by matching on the call arguments, or on other information via AspectJ's *thisJoinPoint* special variable. See Listing 1 for an example of an advice definition.

```
Object around() :
  call(* com.google.ads..*(..)) {
  return null;
}
```

Listing 1: Example of an around advice that can be used to block certain ads by intercepting calls to the `com.google.ads` package.

AspectJ is an AOP implementation created at Xerox PARC for Java. The AspectJ compiler allows to perform weaving into JVM bytecode, through it, one can use aspects to modify compiled applications even without having access to the source code.

2.3 Weave Droid

Weave Droid [10] is a tool for weaving AspectJ aspects into an Android application. As input Weave Droid takes an APK, and a set of aspects that will be weaved into the APK. Weave Droid supports embedded weaving, where the entire weaving process is performed on the Android device, as well as cloud-based weaving, where the input

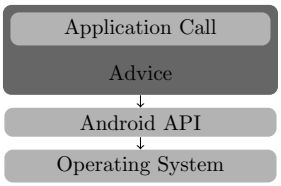


Fig. 4. Monitored Android API call.

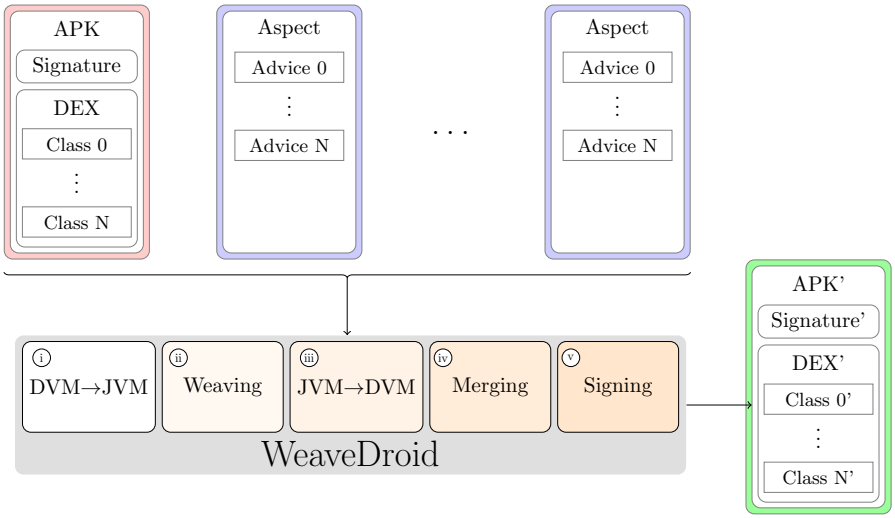


Fig. 5. Pipeline of weaving process

required for weaving is sent to be processed on a Weave Droid server after which the output is returned back to the device.

The Weave Droid process is split into 5 stages:

ⓘ Conversion of the input APK from DVM bytecode format to JVM format. This process uses the *dex2jar* library. This step is necessary as AspectJ is only capable of weaving JVM format bytecode. This conversion process has limitations that are discussed in Sec. 6.1.

ⓘⓘ Weaving of the input aspects with the JVM bytecode from stage ⓘ. The AspectJ compiler is used to handle the compilation and weaving of the aspects as well as injecting a library dependency required by the aspects.

ⓘⓘⓘ Conversion of the JVM bytecode to DVM bytecode format. This process uses the *dx* tool. This stage is necessary as Android expects the bytecode in DVM format.

ⓘⓥ Merging the modified bytecode into the input APK. From stage ⓘⓘⓘ we obtain a DVM bytecode file, we use this file to replace the `classes.dex` file present in the input APK.

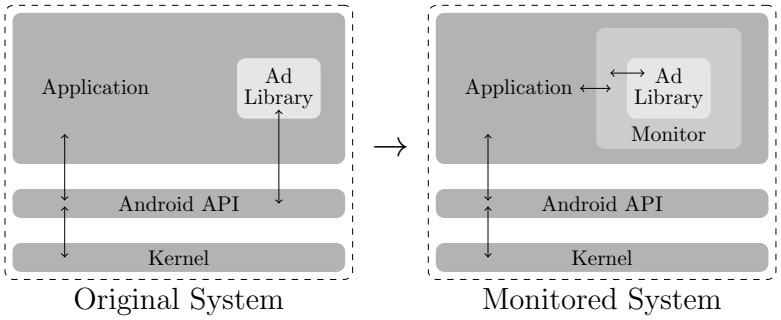


Fig. 6. Comparison of original application with the monitored application.

ⓥ Signing of the modified APK. For Android applications to run, a valid signature is required. Modification of the APK from stage ⓘⓥ results in the APK's signatures being broken, to resolve this, we erase the existing signatures and re-sign the APK.

The APK resulting from the Weave Droid process will exhibit the functionality introduced by the aspects, and will differ structurally from the original as follows:

- *Bytecode and size*, due to weaving of the aspects and inclusion of their library dependency.
- *Signature*, as a result of breaking the APK signature in stage ⓘⓥ and re-signing in stage ⓥ.

3 Ad Suppression Method

In our solution we use the Weave Droid engine to weave the ad-blocking aspects into the application.

```
Class.forName("com.google.ads.AdActivity")
  .getDeclaredMethod("startActivity")
  .invoke(null);
```

Listing 2: Example of a method invocation via the reflection API.

3.1 Aspect Creation

When writing aspects that modify the behavior of applications, we must take into account different mechanisms by which a method can be triggered. Adware applications are notorious for their use of dynamic method invocation as a means of defeating static analysis. For example through the reflection API a method call can be invoked. Listing 2 is an example call to a method that would not be intercepted by the pointcut example specified in Sec. 2.2.

Other factors that must be taken into account, include properly allocating and deallocating intercepted objects that require them. For example some objects such as *BroadcastReceiver* must be registered and unregistered.

Another issue which we must take care of is returning proper pseudo-objects in cases where we wish to spoof information such as contact lists, instead of simply blocking a method and returning null, which may cause the program to crash.

Listing 3, is a snippet of aspect code which blocks invocations to the method ”loadNewAd” within the ”com.inmobi.androidsdk” package, while allowing other method calls for the package to pass through. The snippet takes into account indirect calls via the reflection API, by wrapping an advice around calls to the ”java.lang.reflect.Method.invoke” method.

```
Object around() : execution(* com.inmobi.androidsdk.*.loadNewAd(..)) {
  return null;
}

Object around(): call(Object java.lang.reflect.Method.invoke(..)) {
  java.lang.reflect.Method target =
    (java.lang.reflect.Method)(thisJoinPoint.getTarget());
  Object[] args = thisJoinPoint.getArgs();
  if (args != null && args.length > 0 && args[0] != null) {
    String receiver = args[0].getClass().getName();
    if (target.getName().compareTo("loadNewAd") == 0
    && receiver.startsWith("com.inmobi.androidsdk"))
    return;
  }
  return proceed();
}
```

Listing 3: Shortened example of aspect code for blocking inmobi ads.

3.2 Amending the Application

In this section we describe the steps taken by our implementation for suppressing the advertisements of an input application (see Fig. 7).

ⓘ The application’s APK file is passed to the Ad Analyzer. The Ad Analyzer searches through the libraries used by the application and compares them to a list of known advertisement network libraries. Using this list the Ad Analyzer determines the set of aspects to use for ad blocking.

ⓘⓘ The application’s APK file, and the set of aspects required for blocking the advertisements specific to it are passed to Weave Droid, which may be remote or local. Weave Droid handles weaving of the aspects into the application.

ⓘⓘⓘ Finally, Weave Droid outputs a new APK. The output APK contains the ad blocking behavior in it, and therefore fail to display ads.

4 Implementation: miAdBlocker

miAdBlocker, is a user-friendly Android application based on the methodology described in Sec. 3. miAdBlocker is implemented using Java in 7,260 lines of code (LoC),

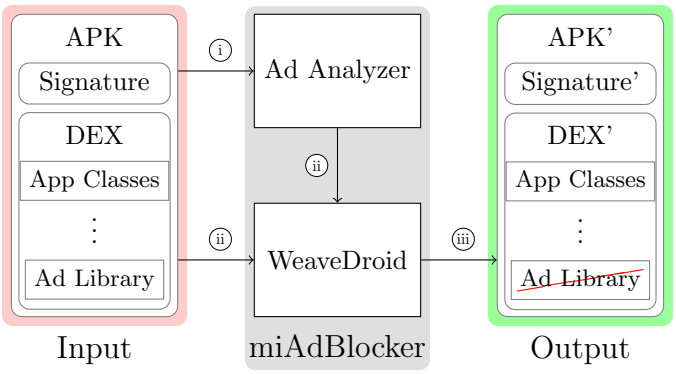


Fig. 7. Pipeline of advertisement suppression process

and uses a remote Weave Droid server for weaving enforcement monitors inside applications. The application allows users to selectively disable the advertisements of applications installed on their device. The implementation supports devices running on Android version 2.3.3 and higher. It uses a 2,190 LoC aspect library capable of disabling over 30 different ad network libraries.

At startup miAdBlocker scans all the applications installed on the system; detecting for the presence of advertisement libraries. Then, a list is populated with all the applications that are candidates for ad-blocking, as seen in Fig. 8(a). The user may select the application for which he wants to block ads. Once the user has indicated that he wants ads to be removed from the application, a confirmation dialog window is displayed as seen in Fig. 8(b).

While weaving directly on the device is possible, due to the inherent limitations and performance issues of weaving aspects directly on android devices [10], miAdBlocker defaults to querying a Weave Droid server, to handle the process.

5 Evaluation

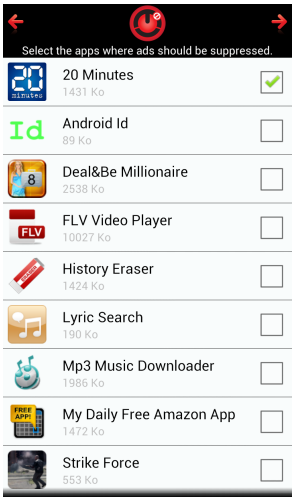
This section presents our evaluation of miAdBlocker with "off-the-shelf" Android applications retrieved from Google Play.

5.1 Case Study

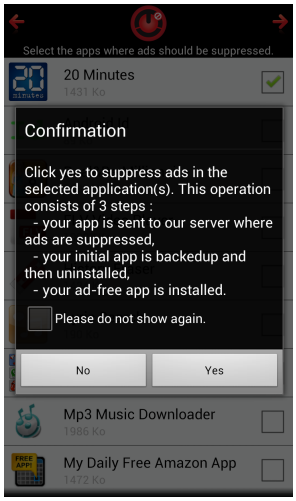
We ran and experiment focused on analyzing the reliability of miAdBlocker to amend applications with ad-blocking enforcement monitors, while preserving the features of the target application (the application should remain functionable and its performance should not be degraded).

To evaluate the proposed method of ad-blocking in Android applications, we tested a sample of 860 popular applications from different categories (games, utilities, misc), and recorded the success or failure of an application.

There are three phases to our testing process. If any error occured during a phase, the application was considered to have failed the current tests, and the tests after it.



(a) Showing the list of applications harboring advertisement libraries.



(b) Confirmation before processing an application.

Fig. 8. miAdBlocker in action.

Applications that were successfully modified, had their modified versions put through the execution test. Due to the time consuming nature of thoroughly testing applications, for the third test a randomized sample of applications from those that were successful in the execution test were selected for this stage.

Modification Amending the application with ad-blocking aspects and repackaging it.
Execution Installing, initial launch, and uninstallation of the amended application.
Thorough All activity windows of an application were checked to ensure proper functioning.

	Modification		Execution		Thorough	
Games	341	96.19%	328	85.98%	52	77.61%
Utility	95	98.95%	94	96.81%	52	94.12%
Misc	424	97.41%	413	93.22%	30	100%

Table 1. Number of applications tested and their success rates at each of the stages.

Table 1 shows the results of the three test phases. For each phase we show the number of applications tested, and their success rate. In each of the phases we encountered errors which we explain below.

Modification involves invoking the Weave Droid pipeline which consists of several sub-steps as seen in Fig. 5. In the first step we have to perform DVM→JVM bytecode

Fig. 9. An application before (left) and after (right) being processed by miAdBlocker.

retargeting, this process is error prone and discussed in Sec. 6.1. We also encountered applications exploiting limitations in the retargeting tool dex2jar, for example by using method and field names obfuscated with unicode characters dex2jar crashes. During the weaving phase AspectJ encountered "missing type" errors due to the presence of calls, and type references that do not have their corresponding libraries bundled with the application.

In the execution and thorough testing phase, we had crashes due to the introduction of errors in the modification phase. We also encountered errors possibly due to the presence of anti-tampering code, and found that game applications in particular had a much higher rate of failure at execution than other application categories.

Remark 1 (A note on performance and file size overhead). The aspects applied to the programs affect their performance. However as with the work done previously we found that the types of aspects developed for ad-blocking had a negligible effect on performance degredation [10]. Rather we observed performance improvements, bandwidth savings, and energy savings. Indeed, our aspects are woven at compile time, and disable a significant amount of code from being run.

The amendment process requires the inclusion of a fixed 117KB library, as well as the newly woven classes/aspects bytecode. A review of the APK sizes before and after the ad-blocking transformation, showed that there was a negligible increase in APK size in the range of 0-5%, and overall averaged near 0%.

6 Discussion

While there are currently many challenges faced from a bytecode weaving approach, there remain advantages, in that it is a more targeted approach and unlike other approaches it does not require alteration of the host operating system, nor use of superuser permissions, both of which may result in voiding of the device warranty. This section discusses some of the challenges faced (and possible counter measures) when applying runtime monitoring and miAdBlocker to Android applications.

Analysis of 100 applications on the market revealed that the majority of those tested used features such as encryption and classloaders. These features may be used to circumvent methods relying on static analysis and bytecode weaving. Further implications

Feature	Percent of Apps Tested
Reflection	99%
Encryption	96%
ClassLoader	99%
Native Code	0%
Calls External Executable	88%

Table 2. Some features used by 100 free applications with advertisements from the top 200 on Google Play.

of using these features are discussed in the rest of this section. We believe these issues were not raised by previous monitoring frameworks applied to academic benchmarks. These issues are to be considered when using a monitoring framework for third-party applications. See Table 2 for the results of the analysis.

6.1 DVM to JVM Retargeting

There are significant differences between the register-based Dalvik Virtual Machine and the stack-based Java Virtual Machine, these differences result in information loss when converting bytecode from one format to the other. The information loss is a cause for some of the errors we encounter when running our tool-chain. As resolving this issue is an active area of research [14], in time we expect success rates to increase.

Malware has also been known to take advantage of bugs in present in retargeting software, preventing proper conversion [15]. Modifying the AspectJ compiler to target Dalvik bytecode directly is a possible solution for avoiding the problems introduced by intermediate retargeting software.

6.2 Native Code

Our method is limited to the modification of Java-based applications, and may be bypassed in applications using native code. But as stated earlier, due to the difficulties of developing apps using native code, only a small percentage of the available applications available use it, and even the ones that use it only use it in small critical performance areas.

6.3 Tamper Detection

Applications using tamper detection can detect unauthorized modification. Developers can integrate this detection using tools such as Arxan [16] and Google LVL [17]. Upon the detection of tampering, applications may be designed to exit, or behave improperly.

Detection typically revolves around signature verification. Application modification as a side-effect results in different signatures compared to the original application.

As miAdBlocker and Weave Droid by design modify the application package, they fall prey to this detection; contributing to the failure rates seen during application execution. Bypassing this mechanism would allow for higher success rates. We will briefly

discuss countermeasures for two basic common techniques, of implementing this detection.

Package signature verification. Applications are signed by the developers using a private key that is only accessible by them. When an application is modified, the original signature will no longer correspond to it. An application without a valid signature will fail to run. Thus, to have a usable application the tamperer must sign it with their own key. A detection mechanism could be for the application to compare its current signature against a copy of the signature known to be authentic.

```
Signature[] sig =
  getPackageManager()
  .getPackageInfo(app, PackageManager.GET_SIGNATURES)
  .signatures;

if (sig[0].hashCode() != authenticSignature) fail();
```

Listing 4: Example implementation of tamper detection

A countermeasure could be to store the valid package signature before transformation, and to intercept package manager calls in the modified application. The modified application would return the original recorded application's signature, thus passing this test.

File signature verification. File signature verification is another form of protection used by application developers. The method involves computing a checksum value of the application files, using a hash function. Detection can be performed at application startup by recomputing the checksum values of the current files and comparing them against the previously computed authentic hashes.

Counter-measures could be to:

- Keep a copy of the unmodified application, and intercepting Java's file system libraries. When an application wishes to access its own files, the interception method redirects access to the original versions that would pass the signature checks.
- Intercept common hash functions used for signature checking, and return a precomputed correct hash upon request.

Amending applications with more complicated verification systems can be done by integrating verification library subversion tools into the Weave Droid pipeline.

6.4 Dynamically Loaded Code

Java allows for code not present in the application to be loaded at runtime from either a local path, or an online location, using a *ClassLoader*. As this code is not present for Weave Droid to perform transformations on, the dynamically loaded code is free of the behaviors enforced upon the rest of the application. Developers may use this mechanism to dynamically load advertisements, bypassing ad blocking utilities based on static analysis and bytecode transformation.

Our technique can be extended to handle such cases via interception of calls to the ClassLoader. A custom ClassLoader can then analyze and send the code to a Weave Droid server. There, the desired behavior is enforced on the code, then returned to the device for execution.

6.5 Obfuscation

Obfuscation is a technique employed by developers to protect their applications against reverse engineering and analysis. Through obfuscation, the names of methods, classes and packages are rewritten while preserving the functionality of the app. Tools such as ProGuard fulfill this purpose. This technique renders aspects that would have worked, targeting specific pointcuts based on names unusable.

However, we did not encounter much problems in this regard when blocking ad libraries, as it is common practice to preserve the public APIs of said libraries due to issues arising from their obfuscation.

Another type of obfuscation can be performed involves storing the code in encrypted form, this code is then decrypted and loaded by a ClassLoader at runtime. Our solution of intercepting the ClassLoader would properly account for this problem, as the ClassLoader must take in the unencrypted bytecode.

6.6 Signature Modification

A side-effect of this modification is that market updates are not properly detected for the modified application, and if one wishes to update directly from the market, they would have to first uninstall or restore the application, before they can move to a newer version. To solve this issue a separate mechanism must be used to check and handle updates.

7 Related Work

7.1 Comparison with Ad-Blocking Software on the Market

We made a survey of the ad blocking solutions found on Google Play and compiled the results (see Table 3). Compared to the other tools on the market that were surveyed, miAdBlocker had less requirements for enforcement of ad blocking, making it more user-friendly.

7.2 Comparison with Similar Research Projects

miAdBlocker [10], was extended upon. The Weave Droid engine is now reimplemented in 6,130 lines of Java code, with a focus on robustness. Originally Weave Droid only handled Google ads; with miAdBlocker we can handle over 30 different advertisement networks.

Aurasium [18] is a policy enforcer that intercepts Android applications via a native library layer. Unlike their method, our method has the benefit of using monitors with awareness of the call context in Java, giving us the advantage of selectively enforcing monitors on a finer-grained level (such as per library level).

Bartel, et al. [8] present a method and implementation that performs static analysis. Our system differs in that we can make decisions dynamically with the awareness of context and avoids the false positives usually induced when using static analysis. For

	Requirements		
	Root	Proxy	Reboot
Adblock plus	✓	✓	✗
AirPush Block	✓	✗	✗
Adway	✓	✗	✗
MyInternetSecur	✗	✗	✓
MiAdBlocker	✗	✗	✗

Table 3. Comparison of the requirements of several ad-blocking applications

instance, our tool can detect specific (dynamically computed) URLs instead of only detecting that an HTTP connection is made in the application.

8 Conclusion and Future Work

8.1 Conclusion

This paper studies the use of monitoring techniques on a real application scenario: blocking advertisement on third-party Android applications retrieved "off-the-shelf" from public repositories such as Google Play. Our purpose was to produce a tool that goes beyond usual research prototypes in runtime verification as we aimed to reach a level of maturity allowing our tool to be publicly released and delivered to users without any computer-science background. During our case studies we encountered many challenges such as the number and heterogeneity of Android applications on which our technique has to be tested, the diversity of the possibilities for developers to displaying advertisements, the discrepancy between the Dalvik format (executable) and the bytecode format (instrumentable). The challenges stem from the facts that we target third-party applications, from many developers, in a domain where a strong competition exists between applications.

We use good practices obtained from research endeavors in the runtime verification community and encode monitoring using Aspect-Oriented Programming. Our experiments show that using AOP via AspectJ is an effective technique to modify existing closed source applications to incorporate ad-blocking monitors. Unlike tools relying on low-level bytecode analysis and transformation, AOP allows for easier targeting and modification of existing application code; through specification of transformation sites via a pointcut matching system. Our method also has the benefit over other solutions in that it has been implemented and tested to work embedded from an android device, and as a cloud-based service.

Our experiments showed a good success rate overall, with better success rates depending on the category of the application. Analysis of applications on the market however, showed the heavy presence of features that could be used for circumvention of the enforcement mechanism. These features are targeted in a newer implementation.

8.2 Future work

Even if our approach is dedicated to blocking advertisement on Android applications the challenges encountered in this paper will remain when applying monitoring in other application domains sharing features (e.g., third-party applications). Thus, we believe that future research endeavors in the runtime verification community should consider extending monitoring techniques to deal with the issues of dynamically loaded code, obfuscation, and tamper-resistant code, for the purpose of yielding a higher success rate at integrating effective monitors.

References

1. Google Inc.: Android (2014) www.android.com, developer.android.com.
2. Gartner: Market share analysis: Mobile phones, worldwide, 4q13 and 2013 (2013)
3. Pearce, P., Felt, A.P., Nunez, G., Wagner, D.: Addroid: Privilege separation for applications and advertisers in android. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ACM (2012) 71–72
4. Pathak, A., Hu, Y.C., Zhang, M.: Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In: Proceedings of the 7th ACM European Conference on Computer Systems (2012). (2012) 29–42
5. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: Proceedings of Mobile Security Technologies Workshop (MoST). (2012)
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
7. Backes, M., Gerling, S., Hammer, C., Maffei, M., Styp-Rekowsky, P.: Appguard – enforcing user requirements on android apps. In Piterman, N., Smolka, S., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 7795 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 543–548
8. Bartel, A., Klein, J., Monperrus, M., Allix, K., Traon, Y.L.: Improving privacy on android smartphones through in-vivo bytecode instrumentation. CoRR **abs/1208.4536** (2012)
9. Xerox Corporation: Aspectj programming guide. http://www.eclipse.org/aspectj/ (2014)
10. Falcone, Y., Currea, S.: Weave Droid: aspect-oriented programming on Android devices: fully embedded or in the cloud. In Goedicke, M., Menzies, T., Saeki, M., eds.: ASE, ACM (2012) 350–353
11. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: NDSS, The Internet Society (2012)
12. Shekhar, S., Dietz, M., Wallach, D.S.: Adsplit: Separating smartphone advertising from applications. In: USENIX. (2012)
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP, SpringerVerlag (1997)
14. Octeau, D., Jha, S., McDaniel, P.: Retargeting android applications to java bytecode. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM (2012) 6
15. Chenette, S.: Building custom android malware (2013) BruCON.
16. Arxan: Ensureit® for android on arm (2013)
17. Google Inc.: Licensing overview - android developers (2014)
18. Xu, R., Saïdi, H., Anderson, R.: Aurasium: Practical policy enforcement for android applications. In: Proceedings of the 21st USENIX Conference on Security Symposium. Security'12, Berkeley, CA, USA, USENIX Association (2012) 27–27