2017 IEEE Symposium on Security and Privacy Workshops

Seamless In-App Ad Blocking on Stock Android

Michael Backes
CISPA, Saarland University & MPI-SWS
Saarland Informatics Campus

Sven Bugiel, Philipp von Styp-Rekowsky, Marvin Wißfeld
CISPA, Saarland University
Saarland Informatics Campus

***Abstract*—Advertisements in mobile applications have been shown to be a true danger to the users' privacy and security. Unfortunately, all existing solutions for protecting users from this threat are not simultaneously satisfying in terms of effectiveness or deployability. Leveraging recent advances in app virtualization on Android, we present in this paper a solution for in-app ad-blocking that provides a high level of effectiveness in blocking ads while at the same time being favorable for end-user deployment by abstaining from OS modifications or any elevated privileges. We discuss the technical challenges and their solutions for robustly stripping ads from apps while preserving the apps' functionality.**

I. Introduction

In-app advertising is a common monetization strategy for free mobile applications. Yet, various studies have raised [6], [7], [15] and re-raised [12], [5], [9] severe privacy concerns about in-app advertisements (ads). First of all, in-app ads are not subject to the same level of privilege separation as advertisements in browsers, e.g., in-app ads inherit the same privilege as their host app for communicating with the system and other apps, giving them access to a plethora of detailed user information. This privilege is being actively exploited by in-app ads to exfiltrate sensitive user information [7], [15], [5], [9] that allow tracking of the user and drawing a pretty accurate user profile. Second, as for in-browser advertisements, in-app ads are an efficient channel for "malwaretisement" and phishing to lure users into malware and scam campaigns [12].

Status quo of in-app ad-blocking: To help mobile users to defend themselves against those threats of in-app advertisements, it would be highly desirable to have ad-blockers as in the browser domain, which remove a potentially harmful advertisement in order to protect the user. However, crucial technical differences exist between the distribution and presentation of in-browser and in-app advertisements, most importantly that in-app ads' bytecode is merged with that of the app displaying the ad and thus is protected by the system's app-sandboxing as part of its host app from tampering (such as ad-blocking) by other installed apps. To tackle this challenge, the research community has proposed different solutions based on *app rewriting* [16], [8], *network-based* [14], [1] filters, or *operating system support* [13], [10], [17] to remove or constrain in-app ads. Unfortunately, none of those existing approaches to in-app ad-blocking is simultaneously satisfying for the end-user in terms of effectiveness or accessibility, which has yet prevented a wide-spread proliferation of efficient and usable in-app ad-blockers.

App virtualization for ad-blocking: With the advent of app virtualization on Android [4] a new, powerful tool has been created that in principle allows effective, highly efficient ad-blocking without adversely affecting the deployability of the solution. In this paper, we report on the technical challenges and their solutions in creating an ad-blocker for in-app ads on Android. While app virtualization inherently provides great benefits for deployability of security solutions, the primary challenges for this work were a) effectively identifying the advertisement code in application packages and b) robustly stripping that code from applications without breaking the app's functionality (e.g., breaking control flows that involve advertisement code).

II. Background on In-App Ads

App developers that want to monetize their app with in-app ads have to add advertisement libraries to their apps in the form of separate code packages and link those libraries' lifecycles with that of their app, e.g., to load and display ads. This linking is done via well-documented public interfaces of the ad library.

A. Types of in-app advertisement

By studying various popular advertisement libraries, we found two basic types of advertisements used on the Android platform: Banner ads and interstitial ads.

Banner ads are part of the program UI design and always occupy a certain dedicated screen region of the app's UI (e.g., the bottom quarter of the screen). Banner ads are usually implemented by providing an implementation of the *View* class[1] that the app developers have to include in their apps' UI declarations.

Interstitial ads are best compared to pop-up advertisements on the web: On certain navigation actions, instead of the desired screen, a full-screen ad is shown. After closing the fullscreen ad, the user is sent back to the application. To trigger displaying interstitial ads and resuming the host application's control flow after closing the ad, host app and ad library have to be linked through a simple event-driven protocol. Commonly, the app will call the ad lib to display the ad after the library has signaled that the ad has been loaded and the ad lib will issue a callback to the host app to inform it that it can continue execution.

[1] View is the base class for all visible content on Android.

DOI 10.1109/SPW.2017.24

B. Advertisement library inclusion

Advertisement libraries are usually provided in one of the two formats supported by the official Android SDK: JAR-files and Android Archives (AAR). While JAR-files are just a bundle of Java classes, AAR-files may also provide native code and Android specific entrypoint annotations, which is a requirement to add new a *Activity*[2] or background service to an app. When JAR-files are provided, the developer is often required to add relevant metadata to the App's sources, because it's impossible to inject these from JAR-files.

All Java classes from external libraries that are specified to the build process in the Android SDK, will be bundled together with the application logic code into a single *classes.dex*-file containing the compiled code of both app code and all included library code. Similarly metadata and resources from the libraries will be merged into the app's original definitions and this will be bundled together with the *classes.dex* into a single APK-package-file, which is then published for installation on the device. This means that the ad libraries cannot be detected trivially after the compilation step, especially as the Android SDK tries to minimize the size of the resulting file by removing metadata not required for execution.

III. Existing Solutions

A. Challenges in comparison to in-browser ad-blockers

Ad-blockers for browsers operate by filtering content from the web content when this content is being loaded (e.g., based on blacklisting advertisement network domains) or removing undesired elements from the loaded web content (e.g., Java-applets or Flash content). Such in-browser ad-blockers require the necessary privileges to observe and modify the web content loaded into the tabs of the browser program, usually in form of browser extensions and plug-ins that request those privileges from the user. In contrast, as explained in Section II, in-app ads' bytecode is merged with that of the app displaying the ad. Since mobile operating systems, such as Android, sandbox applications and isolate them from each other, an ad-blocker app cannot acquire the necessary privileges to tamper with another application, including removing ad content from another application.

B. Blocking or isolating in-app ads on Android

To tackle the challenge of removing or privilege-separating advertisements on Android, the research community has proposed different techniques that will be compared according to functional requirements in the following.

Functional requirements: While blocking or privilege separating advertisements is highly desirable for protecting the users' privacy, an effective solution should also consider different functional requirements that affect the deployability (and hence widespread adoption) as well as minimal required ad-blocking in light of non-malicious applications. We summarize those requirements in Table 1 as follows:

[2]Android's interactive user interface base class, similar to a window on a desktop

O1 No system modification: The solution should abstain from customized Android firmwares, such as extensions to Android's middleware, and is able to run on stock Android versions.

O2 No application modification: The solution does not rely on or require any modifications to the applications from the advertisement is blocked or separated, such as rewriting existing code.

O3 Blocking cached/pre-packaged ads: The solution is able to block (or separate) advertisements that were already pre-packaged with or cached by the application and thus does not solely depend on monitoring and modifying the applications network I/O.

OS extensions: Different security extensions to the Android software stack for isolating and privilege separating advertisement libraries have been proposed, such as [13], [10], [17] to name a few. Generally, those approaches build on privilege separating the advertisement library into a separate process with a distinct UID and hence different privileges from the app showing advertisements, but focusing on different re-integration techniques of the separated app, such as introducing a new advertisement API [10], authenticating user input and visual fidelity [13], or iframe-styled display and input isolation [17]. As such, those solution form a robust solution to isolating advertisements (O3: ✔) and are mostly backwards compatible for apps (O2: ✔). However, operating system support for isolating advertisements (O1: ✘) is highly unlikely to be adopted by vendors (e.g., Google, Samsung, etc.), thus forcing the user to resort to aftermarket firmwares, whose installation forms a technical barrier for most end-users.

App rewriting: To be independent from operating system support (O1: ✔), alternative solutions, such as [16], [8], build on top of app rewriting techniques. Usually, those solutions identify the ad lib code within the application packages and then, for instance, remove this code and its call-sites within the app code from the application package [16] or inline a reference monitor that enforces separate privileges on access to the application framework API by ad lib code [8]. Thus, like OS extensions, those approaches can block or privilege separate advertisements efficiently (O3: ✔), however, rewriting applications (O2: ✘) breaks the same-origin of the application package and prevents the default update mechanisms, forcing the user to rely on "out-of-band" updates for rewritten apps.

Network-based filters: Lastly, network-based filters have proposed, which do not rely on any OS security extension (O1: ✔) or application modification (O2: ✔), but instead rely on removing advertisements from the applications' network I/O streams, thus preventing them from being loaded and displayed. Usually, those solutions make use of Android's VPN API [3] to act as *man-in-the-middle* that can monitor and filter the apps' network traffic. However, filtering network traffic is inefficient in light of cached or pre-packaged ads (O3: ✘) and additionally has to surrender (or compromise) encrypted connections to be able to filter traffic. In particular encrypted network connections are a limitation of those solutions, since advertisements should

Fucntional Objectives	OS extension	App rewriting	Network filter	App virtualization
O1: No system modification	✘	✔	✔	✔
O2: No application modification	✔	✘	✔	✔
O3: Blocking cached/pre-packaged ads	✔	✔	✘	✔

✔= applies; ✘= does not apply.

Table 1: Comparison of deployment options for Android advertisement blockers/containers based on desired functional objectives.

be and are loaded over secure channels in order to prevent easy code injection attacks against the ad libraries' host apps [11].

IV. Stripping In-App Ads

For our solution to strip in-app advertisements from Android applications, we leverage our observations on how in-app advertisements are deployed. We use the fact that ad libraries are exclusively used via public interfaces to strip ad libraries—and potentially any library that is integrated into host apps in a similar way—by replacing on-device the bytecode implementations of these public interfaces with dummy code. By preserving the callsites in applications to valid but dummy call targets in the libraries, we ensure that we do not interrupt control flows between host app and library, which could adversely affect the app's functionality and stability. Moreover, a particular benefit of limiting ourselves to the well-documented public APIs of the advertisement libraries is that we abstract from the libraries' internals, easing the task of identifying and stripping the relevant code fragments.

A. Identifying Call Targets

The first step in our solution is identifying the call targets within advertisement libraries that have to be replaced with dummy logic. This identification predominantly depends on the type of advertisement that is included in the app and hence the interface between the library and the host app. Moreover, dead code elimination and code obfuscation can complicate robust identification of the call targets.

1) Technical challenges: Apps are more and more commonly obfuscated with the ProGuard tool of the Android SDK. Apps that have been processed with ProGuard impose two additional challenges for identifying call targets in ad libraries:

Identifier obfuscation: To obfuscate the app's code, ProGuard renames identifiers, e.g., of methods and classes, with short strings like *a()* or *b.a.c*. In its default configuration, ProGuard will also apply this renaming to external libraries. Thus, when identifying ad libraries or modifying their code we cannot rely on those identifiers.

Dead code elimination: In addition to obfuscating the code, ProGuard also optimizes the resulting bytecode size by eliminating classes and methods that are unreachable from any of the entry points. This removal of dead code makes proper identification of advertisement libraries more challenging when parts of the library have been removed.

2) Class fingerprinting and filtering rules: To ensure that we only block components belonging to advertisements and not those of the host applications or other 3rd party libs, we need to create filter rules that clearly define the content to be stripped from an application's codebase. Clearer filter rules directly result in a lower false positive rate. This problem of defining clear filter rules is already known for browser-based ad blockers where rules define HTTP requests and HTML elements to be blocked, but is a lot more complex for in-app advertisements where the rules have to refer to classes and methods instead, which can be subject to dead code elimination and obfuscation. In presence of code elimination or obfuscated identifiers, the public API of advertisement libraries becomes unreliable as an identifier, since there is no guarantee that it is present in the code in the same form as in the API specification.

Thus, our approach instead relies on a set of class structure information (e.g., class hierarchy and method signatures) of the API classes including their cross-references to fingerprint ad libraries more robustly. This set of API classes consists of the smallest set of classes that has to be always present for the ad library to be functional, while the classes outside this set can be subject to code elimination. Using the class structure for fingerprinting ad libraries removes the need to rely on method and class identifiers and increases robustness against simple obfuscation techniques. In rare conditions, it is also required to add non-API classes to the set to draw the line between API classes with extremely similar structure and hence uniquely identify all relevant classes.

To further reduce the risk of false positives of our class fingerprint, we additionally use the Java package name as a criteria for fingerprinting. We found that ProGuard and other obfuscation tools often refuse to fully obfuscate the package name. We discovered this to be particularly the case when ProGuard detects references to the package name that are used in reflection calls or in references from XML resources of the application (e.g., referencing Views). Moreover, the package name tree structure is preserved by some obfuscation tools and can be leveraged for fingerprinting ad library code within the application code.

The package name, class structure information as well as annotations on how to further proceed with the class and methods are then written down in a domain specific language in a structured filter definition file. Listing 1 presents an example excerpt from such a filter file, which defines a final class in package "com.google.android.gms.ads" that has the class "android.view.ViewGroup" in its inheritance tree, defines a one-argument, void-returning "loadAd" method, and should be filtered such that it becomes an empty View. The method's argument type ".AdRequest" of said method as well as other names starting with a dot are cross-references to other classes defined in the same package.

```
1  package com.google.android.gms.ads
2      [...]
3      class .AdView extends* android.view.ViewGroup
4          set filter-action empty-view
5          flag final
6          property define .AdListener listener
7          method exists,replace void loadAd .AdRequest
8      end class
9      [...]
10 end package
```

Listing 1: Example excerpt of filter definition file

B. Applying Filter Rules

We manually created a set of filter rules for various libraries. With the class fingerprint from the filter definitions we are now able to identify ad library classes that match the fingerprint and define criteria on how to filter those classes.

Although automatic creation of filter rules would be desirable, this forms a technical challenge for future work and is also an open problem in other ad blocking domains such as in-browser ad blockers.

1) Stripping advertisements: After we identified the ad library code, we need to strip it in a way that the API functionality is preserved. If stripping would lead to a non-functional library API implementation, this would result in applications crashing or being rendered unusable for the user. However, we noticed that most apps only use a small fraction of the functionality provided by ad libraries or sanitize return values by the ad library to ensure proper functionality. It is therefore not necessarily required to provide perfect compatibility of a library for keeping the app functional after stripping relevant public methods.

To avoid the highly involved case of having to handle cross-references and undocumented private methods inside classes that we want to strip from the library, we rewrite all public methods and abandon private methods in all classes we decided to strip and classes along the inheritance path of those. As this removes all of the original code of said class, we can be sure that the part of the class that caused the ad to be shown to the user is no longer present.

We distinguish different tactics of rewriting: For simple getter/setter method pairs we use a heuristic approach to always return the last set value (or a default value) in the get method. Functions that necessarily require an implementation can be redirected to any public method including those of custom, injected methods. Lastly, if neither of those two tactics applies, we simply return values defined in the corresponding filtering rule (or a type-dependant default value like empty string, 0, or false).

2) Type-specific behaviour: **Banner ads:** To ensure that we do not violate the visual fidelity of the app after removing banner ads, we inject a custom implementation of View's `onMeasure` method[3] and constructor for layout inflation[4]. Our implementation will force the View to not take up any space on screen, thus avoiding "holes" in the app's UI where the banner ad was previously placed.

[3]Called to determine the View's size on screen.

[4]Instantiating actual View classes from XML definitions.

Table 2: Stability and effectiveness of our solution for different in-app advertisement types.

App	Stable	Banners	Interstitials	# Classes Total	# Classes Patched	Time (ms)
2048	✔	✔	✔	4,661	2	656
4 Pics 1 Word	✔	✔	✔	1,997	5	611
Color Switch	✔	✔	●	7,377	3	651
wetter.com	✔	✘	●	7,998	2	601
Flashlight	✔	✔	●	7,951	4	577
QR & Barcode Scanner	✔	✔	●	4,084	2	379
Plague Inc.	✔	✔	●	7,842	2	630
RegenRadar	✔	✔	●	7,759	4	497
Solitaire	✔	●	✔	7,457	5	556
Stack	✔	✔	●	7,127	5	715
Unblock Me FREE	✔	✔	●	5,639	2	586
Accuweather	✔	✘	●	7,226	1	406
Alarm Clock	✔	✔	✔	8,303	4	510
LED Flashlight	✔	✔	●	1,329	3	384
Glow Hockey	✔	✔	✔	3,424	2	427
ZigZag	✔	✔	●	8,104	3	687
AndroZip File Manager	✔	✔	●	2,709	2	368
Calories in Food	✔	✔	●	4,016	2	449
Notes	✔	✔	●	2,959	2	383
File Commander	✔	✔	●	8,611	3	532
Alarm Clock Xtreme	✔	✔	●	4,489	2	426
TV Remote	✔	✘	✘	6,985	2	687

✔ Blocked; ✘ Not blocked; ● Type not present

Interstitial ads: Removing interstitial ads may require invoking an appropriate callback method to not block the host app (cf. Section II-A). By analyzing the Binder IPC interfaces defined for the host application, we can identify the methods of the application that match the required signature for the callback method and then rewrite the ad library such that it always immediately invokes the callback methods, thus mimicking the behavior of a finished and closed interstitial ad.

C. Deployment

We implemented a prototype of our approach using the *Boxify* [4] app virtualization solution. While installing an application into the virtual environment, we analyze the app to identify the library and call targets, create a new bytecode file containing the relevant stripped classes and custom method implementations for stripping this library, and inject this file into the application by prepending the class path with this new file. This way we can overwrite the original library implementation without any operating system support and without modifying the application code, i.e., we do not break the original application signature as done by other rewriting based approaches [16], [8].

V. Evaluation and Discussion

We evaluate the prototypical implementation of our system in terms of effectiveness and efficiency and discuss ethical considerations and limitations of our approach. For our

evaluation, we randomly selected 22 apps from the Google Play Store that display advertisements according to the store description (Table 2). Our filter ruleset contained 29 rules covering 7 different advertisement libraries. All tests described in the following were performed on an LG Nexus 9 running Android 6.0.1.

A. Effectiveness

To assess the effectiveness of our approach, we first manually verified the presence of advertisements and identified the types of ads used in each app. From the 22 apps in our testset, 21 apps contained banner ads and 6 apps showed interstitials. After applying our adblocking technique, all apps in the testset were still functioning normally and no more advertisements were shown in 19 apps (86%). In only 3 cases, some ads were still visible; manual investigation revealed that these apps are using advertisement libraries that are not yet covered by our filter definitions.

B. Efficiency

We measured the runtime of our ad stripping algorithm for every app in our testset. On average, the algorithm took ≈533ms to complete. Since ad stripping takes place only during the installation of an app, no additional runtime overhead is incurred (besides the base cost for virtualization [4]). Moreover, only a minimal fraction of the library classes has to be patched for stripping the advertisements.

We also evaluated the robustness of our filter definitions against changes in the advertisement libraries themselves. To this end, we compiled a synthetic test app with 10 different versions of the Google Play Ads library released over a period of two years. Without changes to the filter definitions, ads were successfully blocked for all versions of the library.

C. User feedback

To assess how our approach performs under real-world conditions, we implemented an end-user version of our system in collaboration with *Backes SRT GmbH*[5] and made it publicly available via [2]. The published app offered the users the possibility to send anonymized telemetry data back to *Backes SRT*. From this telemetry data, we observe that over a two month period, our app was downloaded 5,718 times and installed on 470+ different device models from 16+ distinct manufacturers. Users applied ad blocking to 15,000+ different apps, most commonly games, news and weather apps, and social network apps. Interestingly, a significant number of users tried to block ads within web browsers, with Google's Chrome browser topping the list with over 1,000 installations. Users engaged with ad-blocked apps 24,000+ times, of which 1819 app sessions resulted in the target application crashing. Only about 25% of these crashes could be attributed to our ad blocking technique and were caused by incomplete or incorrect filter definitions, while the majority of crashes was due to Boxify app compatibility issues. We argue that those numbers underline the deployability of our solution in terms of device and app diversity.

We also asked users to rate their experience with our app and to provide suggestions for improvement. Rating scores ranged from zero to five; we received a total number of 401 ratings and achieved an average score of 3.9. Users suggested improvements 450 times: 42% suggested to support more apps, 22% asked to improve the stability of the system, the remainder of suggestions concerned user interface (19%) and performance (17%). Based on our users' feedback, we argue that our ad blocking is effective, but requires at the same time further improvements of the filtering rules to support a higher number of advertisement libraries—something that would in future greatly benefit from an automatic creation of filtering rules.

D. Ethical Considerations

Showing in-app ads is usually part of the monetization model of an application. Blocking ads will reduce the developers' income and might render their business not profitable. This issue has been discussed in detail for web-based advertisements. However, major mobile platforms offer alternative monetization models. The platforms already have support for in-app micropayments, allowing users to easily pay for additional features or virtual in-game currencies. This *freemium* model is already widely deployed in top apps and has been shown to yield high profits.

A future version of our ad blocking approach might be extended to use more fine-grained, policy-driven ad blocking and allow the user to only block intrusive ads.

E. Limitations for ad blocking

In order to identify advertisement libraries within an app, the app's (full) bytecode must be available at installation time. However, Android apps may use dynamic class loading to load additional code at runtime, which could be missed by our analysis (e.g., code retrieved from a remote server). A potential solution would be to instrument the ClassLoader to analyze code fragments when they are first loaded. However, our approach is not alone in this aspect, since also app rewriting techniques [16], [8] depend on instrumenting all potentially executing code.

More advanced obfuscation techniques than identifier renaming exist, e.g., control flow obfuscation. These obfuscation tools can currently thwart our class and method identification, rendering our approach ineffective. However, stronger obfuscation would need to be applied on a per-app basis by the app developers themselves, as advertisers need to leave the public APIs of their libraries unobfuscated. Thus, unless ProGuard is superseeded by a more advanced obfuscation tool in the standard Android build toolchain, we do not expect these methods to be employed on a larger scale anytime soon. In comparison to related work, again app rewriting techniques [16], [8] suffer the same drawback, since they also depend on identifying ad library call-sites within application

[5]https://www.backes-srt.com

code in order to safely remove the ad lib [16] or inline a reference monitor [8].

Advertisement libraries might also be shipped in form *native* libraries, i.e., C/C++ code. While instrumenting C/C++ code is generally considered a hard problem and is currently excluded by most solutions, such native code has to be integrated into the usually Java-based host app (e.g., life-cycle management). Thus, a future extension of our solution could be to identify the native code call-sites (i.e., *native*-flagged methods) and redirect them to injected dummy stub methods, which are not native.

Lastly, our approach, like all previous approaches, is concerned with *third party* advertisement *libraries*, which are included into apps by the app developers to show banner or interstitial advertisements in the apps' GUI. Advertisements that are *built-in* to the apps' content, e.g., in form of showing promoted tweets or posts or advertisements shown within loaded web content (e.g., in browser apps or within WebView components), are currently out-of-scope for our solution and the other approaches.

VI. Conclusion

We presented an adblocking solution for Android based on app virtualization, which combines deployability with efficient privacy protection. A particular challenge to be solved was the identification of ad libraries within apps to be able to effectively strip the ads from the apps.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345).

References

[1] Adguard. https://adguard.com, 2016. Last visited 07/25/16.
[2] SRT AdVersary. https://www.backes-srt.com/de/solutions/srt-adblocker/, 2017. Last visited 03/08/17.
[3] Android Developer Reference. Vpnservice. https://developer.android.com/reference/android/net/VpnService.html.
[4] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *Sec'15*. USENIX Association, 2015.
[5] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter. Free for all! assessing user data exposure to advertising libraries on android. In *NDSS'16*. Internet Society, 2016.
[6] W. Enck, D. Octeau, P. McDaniel, and C. Swarat. A study of android application security. In *USENIX Security'11*. USENIX, 2011.
[7] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WISEC'12*. ACM, 2012.
[8] B. Liu, B. Liu, H. Jin, and R. Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys'15*. ACM, 2015.
[9] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *NDSS'16*. Internet Society, 2016.
[10] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ASIACCS'12*. ACM, 2012.
[11] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS'14*, San Diego, CA, 2014.
[12] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS'16*. Internet Society, 2016.
[13] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Sec'12*. USENIX Association, 2012.
[14] Y. Song and U. Hengartner. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *SPSM'15*. ACM, 2015.
[15] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *MoST'12*. IEEE, 2012.
[16] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu. Apklancet: Tumor payload diagnosis and purification for android applications. In *ASIACCS'14*. ACM, 2014.
[17] X. Zhang, A. Ahlawat, and W. Du. Aframe: Isolating advertisements from mobile applications in android. In *ACSAC'13*. ACM, 2013.