RmPerm: A Tool for Android Permissions Removal[*]

Simome Aonzo, Giovanni Lagorio and Alessio Merlo
DIBRIS, University of Genoa, Italy

Keywords: Android Security, Mobile Security, Privacy and Data Protection.

Abstract: Android apps are generally over-privileged, i.e., they request more permissions than they actually need to execute properly. Prior to version 6 users can install an app only by accepting *all* its requested permissions, while newer Android versions allow users to dynamically grant/deny *groups* of permissions. Since some them impact on users' privacy, we argue that users should be granted control at the granularity of the single permission. We propose a novel approach, which does not require any change to the underlying OS, allowing users to selectively remove permissions from apps before installing them, and with a finer granularity. We developed RmPerm, an open-source tool, that implements our methodology, and we present the viability of our approach via an empirical assessment on 81K apps, underlining that, in the worst case, up to 86% of the apps can execute without crashing when *none* of the requested privacy-related permissions are granted.

1 INTRODUCTION

Apps are the main attack vector for Android devices; therefore, they should require the minimum set of permissions to work properly, while satisfying the least privilege principle to reduce the attack surface. However, apps are generally over-privileged (Felt et al., 2011a) since developers tend to require more permissions than necessary to reduce the probability that their app crashes. Furthermore, it is worth noticing that some permissions are particularly important for the privacy of the user, like, for instance, those that allow apps to profile the user by accessing her contacts, messages and call logs. These permissions are called *dangerous* by the Android documentation, and we argue that users should be granted more control over them.

Android versions prior to 6, that have still an adoption of 71.5% (and, 2017b), grant a very coarse-grained control over permissions, i.e., the user cannot remove permissions from apps, and should grant *all* permissions requested by any app in order to install it. Newer versions (i.e., 6 and later) support dynamic management of groups of permissions, but they do not allow the user to grant/deny single permissions. To overcome these limitations, we put forward an approach allowing to selectively remove permissions from apps that does not require any modification to the underlying operating system and is compatible with all Android versions. A key strength of our approach is that, when a user decides to remove certain permissions from an app, we can guarantee, by design, that no Java nor any native code could ever exploit such permissions, no matter what. The worst case scenario is a crash of the less-privileged app, but not a privacy leak. We have implemented our methodology in a tool, `RmPerm` (rmp, 2017), that we use to extensively assess the viability of the approach on a set of $81,000$ apps.

RmPerm is an *open-source* project implemented in Java, and consists of a console application and a library. In this respect, we also implemented an app, `ApkMuzzle` (apk, 2017), using RmPerm as external library. We argue that releasing a tool like RmPerm as open-source is a liability, as any tool that repackages apps can subtly add malicious code. By releasing RmPerm as open source we grant anyone the possibility to verify its behavior, by inspecting the source code.

The rest of the paper is organized as follows. Section 2 describes our methodology for permission removal, while Section 3 assesses its viability and performance. Section 4 sets our proposal within the current state-of-the-art. Finally, Section 5 concludes and points out some future work.

[*]Partially supported by CINI Cybersecurity National Laboratory within the project FilieraSicura funded by CISCO Systems Inc. and Leonardo SpA

Aonzo, S., Lagorio, G. and Merlo, A.
RmPerm: A Tool for Android Permissions Removal.
DOI: 10.5220/0006420203190326
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 319-326
ISBN: 978-989-758-259-2

2 PERMISSION REMOVAL

A quite direct way to remove a set of permissions $P = \{p_1, \ldots, p_n\}$ from an app A is simply to modify its manifest, contained in the APK of A, obtaining A'. This action has two consequences:

1. the digital signature s, for the APK of A, cannot be reused for A' since its manifest, and so the resulting APK, differs from the original;
2. A' may crash due to unexpected exceptions, thrown by the invocation of some API method that needs some permission p_i to run. Indeed, prior to Android 6, an app asking for the set of permissions P gets installed only if the user grants the whole set P, so apps could assume to be granted all asked permissions.

Since the signature s has been produced using an unknown secret key, we have no choice but to sign A' with another (secret) key. The only user visible effect is that Android will consider A and A' two different apps; however, since they have the same name, only one of them can be installed at any time. We do not consider this a problem; in some sense they *are different*: A' is probably safer than A! To avoid that A' crashes because of an unexpected exception, due to a missing permission p, we carry out a customization to all invocations of API methods that need the permission p. This, in turn, means that we need a mapping between permissions and the API methods that require such permissions. Surprisingly, this mapping is not provided by the official documentation. However, we were able to obtain the mapping from the Androguard Project (and, 2017a).

```
@CustomMethodClass
public class CustomMethods {
  @MethodPermission(
   permission="android.permission.INTERNET",
   defClass="java.net.URL")
  public static InputStream openStream(URL u)
    { return new FakeInputStream(); }

  @AuxiliaryClass
  public static class FakeInputStream
                extends InputStream {
      @Override
      public int read() throws IOException
        { return 0; }
  }
}
```

Figure 1: Example of method redirection.

Using *Dexlib2* (dex, 2017) we have implemented RmPerm, a tool to *redirect* selected API method invocations to our own alternative implementations.[2] Custom methods typically just return some fake data to the app, to let it proceed. Consider, for instance, the method `execute`, declared in `org.apache.http.impl.client.DefaultHttpClient`: it needs the `INTERNET` permission and returns a `org.apache.http.HttpResponse`; in this case, we cannot just remove the invocation or return a `null` reference, because that would likely make the app crash. In such cases we must mock a “reasonable” return value. In the general case, when we want to redirect the invocation for an instance method m of class C, with the signature $T_r\ m(T_1, \ldots, T_n)$, we define a new static method $T_r\ m(C, T_1, \ldots, T_n)$, defined in a class X that we add to the APK. The first parameter, of type C, plays the role of the *receiver* of the corresponding instance method. This kind of setup is similar to C# extension methods. Then, we rewrite all the invocations of the form $e_0.m(e_1, \ldots, e_n)$, where e_0 has static type C, with $X.m(e_0, e_1, \ldots, e_n)$[3].

To make writing custom replacement methods as easy as possible, our tool reads a DEX file and searches for classes and methods that are annotated with Java custom annotations, that we have defined: `@CustomMethodClass`, `@MethodPermission` and `@AuxiliaryClass`. The annotation `@CustomMethodClass` simply marks classes that contain replacement methods; this is simply an optimization to avoid to process each and every method of the input file. The annotation `@MethodPermission(p, defClass)` indicates the involved permission `p` and the defining class (in the Android API) `defClass`. Finally, `@AuxiliaryClass` marks the classes that must be copied into the repackaged app, because they are needed by the custom methods. For instance, Fig. 1 shows class `CustomMethods`, which contains a redirection for method `openStream` defined in class `java.net.URL`.

3 EXPERIMENTAL ASSESSMENT

To assess the effectiveness and efficiency of the proposed methodology, we have carried out an empirical assessment by executing RmPerm on a dataset of 81,000 APKs randomly downloaded from three different markets, namely Google Play (70,000 APKs),

[2] As an optimization, RmPerm can automatically remove the invocations of `void` methods, when such methods do not have an explicit custom replacement.

[3] Here, to make the explanation simpler, we use a source-like syntax but, of course, we work directly at DEX byte-code level.

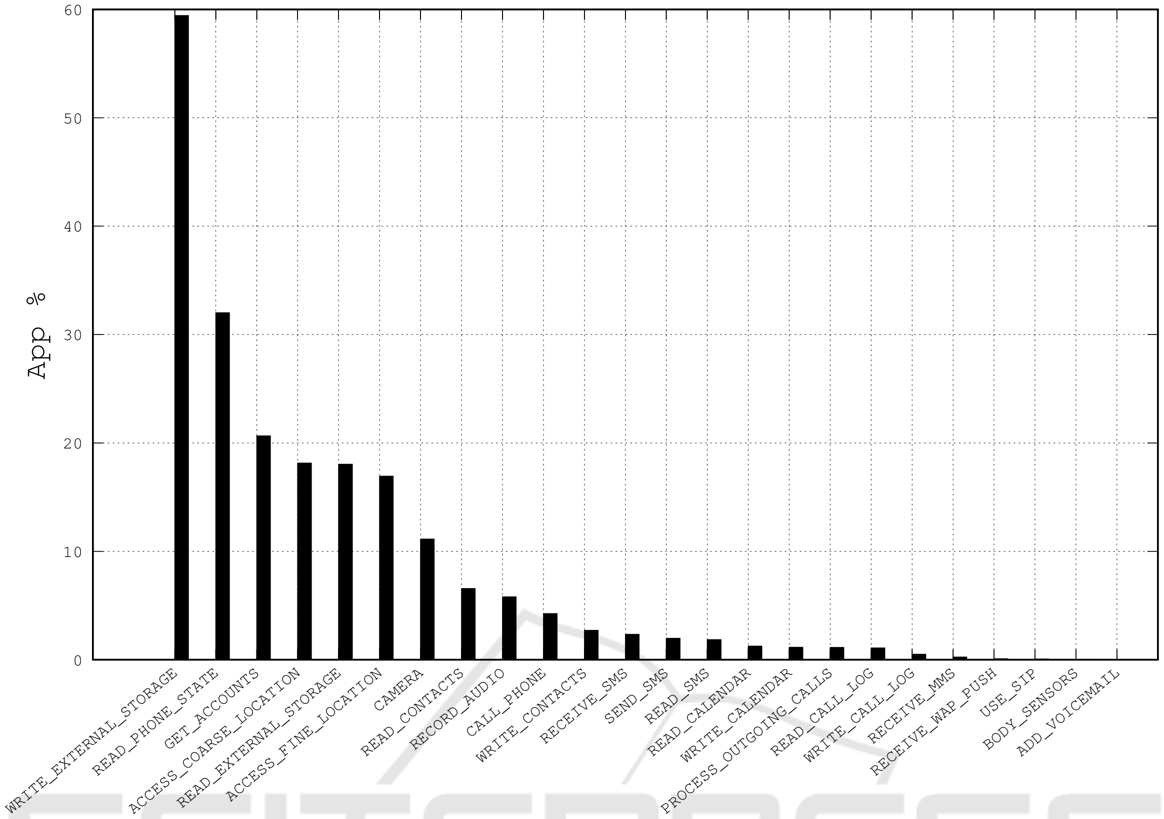


Figure 2: Distribution of dangerous permissions in the dataset.

Aptoide (5,500 APKs) and Uptodown (5,500 APKs).

Since we could not conceivably choose, for each app of the dataset, a different set of permissions to remove, we have decided to assess the worst-case scenario, that is, to remove *all dangerous permissions* from each app A, producing a new app A'; then, we have checked whether the less-privileged app A' could be installed and execute properly. This process is detailed below.

Dangerous permissions refer to the Android classification (req, 2017); there currently are 24 dangerous permissions. They are strictly related to the user's privacy as they allow app to access storage, camera, GPS coordinates, user's calendar and contacts, just to cite a few.

We extracted all permissions requested by the apps in our dataset by systematically parsing the app manifest file, that contains all permissions required by an app. Fig. 2 shows the distribution of dangerous permissions in the dataset. Intuitively, the x-axis shows the 24 permissions ordered accordingly to their frequency on the dataset. The y-axis indicates the percentage of apps requesting the permission. Fig. 3 plots the top 24 permissions requested by apps in the same way.

Testing RmPerm. A single automated test, for each app A, runs as follow:

1. RmPerm gets the APK of app A, and builds up a new app A' that does not require any dangerous permission.
2. A' is installed on an actual Android device.
3. If this step fails, the original A is installed, in order to verify whether the failure is due to the modification carried out by RmPerm or it is independent from the permission removal.
4. If the installation of A' has been successful, its behavior is tested by generating a stream of 512 pseudo-random user events with *Monkey* (mon, 2017), seeded by a random number n. Using different seed values leads to generate distinct sequences of user events. If A' fails, this can be due either to the removal of permissions or to the presence of bugs in the original app A.
5. To ascertain this, we stimulate A with the same stream of events, generated by seeding *Monkey* with the same seed n.

Through previous steps, we can empirically assess whether the removal of dangerous permissions through RmPerm leads to failures. We carried out the

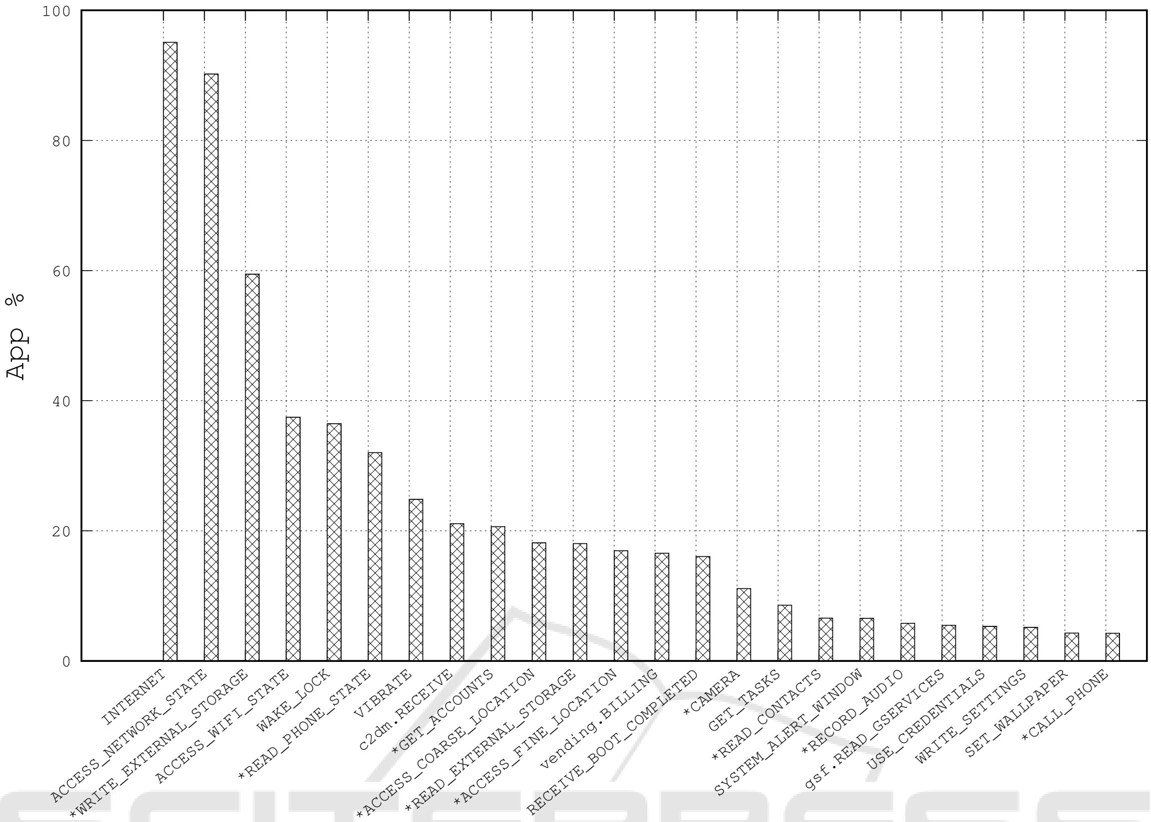
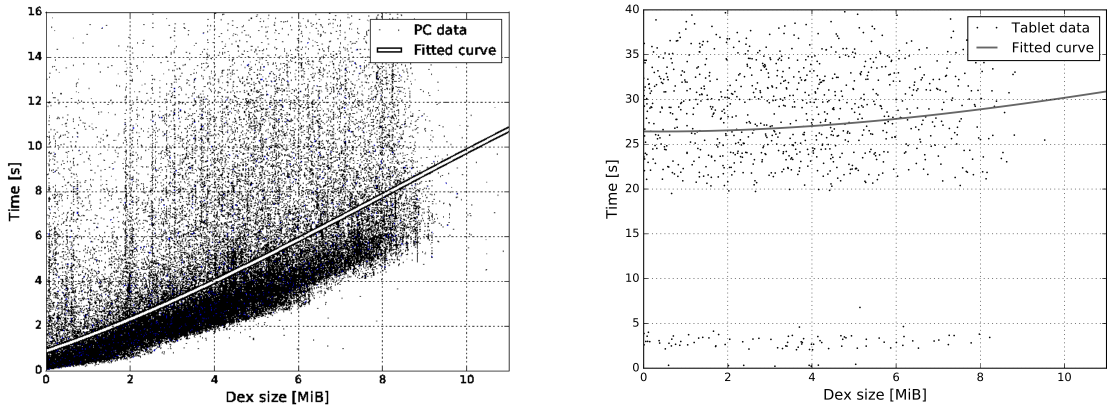


Figure 3: Top 24 requested permissions in the dataset. Dangerous permissions are labeled with*.

Table 1: RmPerm: Performance and size statistics.

	#Permissions			**SizeRatio**		**Exec. time[s]**		**Test results**	
AppName	**original**	**new**	**Δ**	**APK**	**DEX**	**PC**	**Tablet**	**Inst.**	**Monkey**
AdobeReader	6	3	3	1.16	1.00	5.70	27.41	✓	✓
CandyCrushSaga	10	8	2	1.08	0.99	6.17	30.83	✓	✓
Facebook	48	33	15	1.03	0.94	4.04	27.73	✓	✓
Instagram	25	17	8	1.19	1.01	7.13	30.15	✓	✓
LedFlashLight	9	8	1	1.06	1.00	6.60	27.47	✓	✓
MGuard	35	33	2	0.97	1.00	8.46	32.57	X	n. a.
Shazam	20	13	7	1.04	1.00	7.25	35.49	✓	X
Skype	45	33	12	1.13	1.00	11.49	44.82	✓	✓
Snapchat	15	7	8	1.07	1.00	5.14	25.17	✓	✓
Spotify	23	19	4	1.11	1.00	8.49	35.50	✓	X
SwiftKey	14	11	3	1.18	1.00	9.62	31.90	✓	X
Telegram	32	21	11	1.12	1.00	5.94	23.94	✓	X
Twitter	31	22	9	1.45	1.00	8.76	28.19	✓	✓
Viber	56	41	15	1.21	1.00	7.65	29.35	✓	✓
Whatsapp	50	38	12	1.45	1.00	7.38	26.95	✓	X
...	...	...	...	...	...	...	...	...	...
Average	6.8	4.9	2.1	1.1	1.1	3.9	27.1		
Std deviation	5.9	4.4	2.4	0.2	3.1	3.3	9.8		



(a) PC.

(b) Tablet.

Figure 4: Time required to repackage an app.

experimental assessment on a Dell XPS 9530 (Ubuntu 16.04, Intel i7-4712HQ @ 2.30GHz, 16GB RAM), as well as on two Asus Z170CG (Android 5.0.2, Intel Atom x3-C3200 @ 900MHz, 1GB RAM) that we used to install and (automatically) stimulate the apps.

Our results indicate that on $81,000$ samples, $2,358$ repackaged apps failed in step 2; that is, they could not be installed successfully. Among these, 572 failed the step 3 too; this means that the corresponding original APKs were already broken in some way. Therefore, we discarded them and we considered a new set, consisting of the original samples except for the already broken APKs; that is, $81,000-572=80,428$ apps. On this set, the $98\% = 78,642/80,428$ of repackaged apps have been successfully installed. Then, we stimulated the installed apps according to step 4. Among these, $66,051$ were repackaged and stimulated without crashes, while $14,377$ $(= 80,428-66,051)$ failed and required further analysis. Therefore, we applied step 5 to such apps obtaining that $3,633$ original apps crashed, thereby proving that the same problems affected the original app, too. For this reason, we also discarded these APKs. Summing up, the $86\% = 66,051/76,795$ of working apps have been successfully modified, installed and executed properly after removing all dangerous permissions.

Discussion on Global Statistics. We analyzed the average values for the whole dataset in terms of permission removal, size of APKs and DEX after repackaging, as well as the time required for the whole process. To this aim, Table 1 summarizes global and some per-app statistics; the database containing all the data can be freely downloaded (sql, 2017). Regarding permissions, removing dangerous ones has led to the removal, on average, of 38% of all app permissions, as original apps have on average 6.8 permissions while repackaged ones have 4.9, with a large standard deviation value in both cases.

Repackaging phase does not alter the size of the APKs significantly; on average, repackaged APKs are smaller by a factor of 1.1, with a standard deviation of 0.2. Modified DEX files are smaller on average, with a factor of 1.1 and a large standard deviation of 3.1. This is due to the fact that, when removing a set of permissions P, we also remove all invocations to `void` API methods, requiring some permission $p \in P$, for which we do not have explicitly defined a redirection. Furthermore, the size of the original manifest file is decreased, but the size of DEXs is increased due to the addition of custom classes/methods.

RmPerm Performance. We have measured the time needed to remove all dangerous permissions from an app, in two different use cases: when RmPerm is running on a PC, and when RmPerm is running on an Android device. We have measured the running time on all APKs of our sample set when running on the PC, while we have randomly picked $1,000$ apps when RmPerm was run on the Android device. Indeed, we were only interested to check whether running RmPerm on an Android device was practical and if the running time was still linear in the size of the DEX file. Fig. 4 shows the performance results: the DEX size, say s, is generally a sensible parameter for predicting the running time t; indeed, as shown by the fitted curve, $t(s)$ is roughly a linear function. However, there are cases where a relatively small DEX file is contained in a large APK; for instance, this is the case for apps containing graphical/multimedia resources, like games. In these cases, the time to copy the resources from the original APK into the new one may prevail the time needed to process and rewrite the bytecode. This is the reason for the jitters in both graphs. Results have been positive in both regards:

while obviously slower, running RmPerm on the device requires less than 30 seconds on average, and the times are still linear in the size of the DEX file, even though the slope is less steep and there is a constant cost, presumably due to the start-up time of RmPerm on Android.

4 RELATED WORK

As shown by previous work (Felt et al., 2011b; Felt et al., 2011a) , many Android apps are *over-privileged*, that is, they request more permissions than they actually need, thereby making the built-in permission system rather inadequate to protect the users and their privacy. To address these concerns, some authors have proposed to enrich the built-in security framework, by modifying the underlying operating system, and requiring changes to the app sources, in order to exploit the new features.

One of the top problems related to Android permissions is the fact that, up to Android 6, the protection offered by the system was an *all-or-nothing* choice at installation time, when the user was asked to accept *all* permissions requested by the app, or to abort the installation altogether. Moreover, with the current permissions management, if an app requests a dangerous permission, belonging to a certain permission group, and the user agrees on its usage, then the user is actually agreeing on accepting *all* permissions of the same group. That is, any subsequent update of the app can request, and be silently granted, any other permission belonging to an accepted group. Clearly, a fine-grained control was needed. For this reason, many proposals, including ours, tackle the built-in permission system directly. *Apex* (Nauman et al., 2010) is a policy enforcement framework that allows users to selectively grant permissions to apps, as well as impose constraints on the usage of resources. This implementation requires some changes to the Android code base so, while we share a similar goal, the striking difference is that we require no changes to the underlying system.

Some work addresses privacy concerns directly: *AppFence* (Hornyack et al., 2011) retrofits the Android operating system to protect private data from being exfiltrated, by replacing *shadow data*, in place of data that the user wants to keep private, and by blocking network transmissions that contain data the user marked for on-device use only. *MockDroid* (Beresford et al., 2011) modifies the Android operating system to allow users to *mock* the app's accesses to a resources . We use a similar *trick* to avoid that apps crash due to unexpected exceptions, once we have removed some of their permissions. However, we repackage apps and leave the operating system untouched. Finally, *TISSA* (Zhou et al., 2011) is a privacy-mode implementation in Android. All the above proposals allow to run unmodified apps more safely, at the cost of modifying the underlying Android operating system, which severely hampers the widespread adoption of these solutions.

Other proposals (Xu et al., 2012; Jeon et al., 2012; Davis et al., 2012; Backes et al., 2013; Davis and Chen, 2013; Reddy et al., 2011) bypass the need to modify the underlying operating system by repackaging arbitrary apps to attach user-level sandboxing and policy enforcement code. These proposals, like ours, use static analysis to identify the usage of API methods and instrument the bytecode to control the access to these invocations. However, with the exception of (Jeon et al., 2012), discussed below, all of these do not remove permissions from the manifest of the original app A, when creating the repackaged app A'; thus, the underlying OS process that runs A' retains all permissions of the original app A. This means that incomplete/flawed implementations of bytecode rewriting can lead to bypassing access control mechanisms, e.g., by using Java reflection and/or native code (Hao et al., 2013).

Dr. Android (Jeon et al., 2012) is a tool that uses bytecode rewriting to replace Android permissions with a specified set of fine-grained versions, that are accessed through a separate service, called *Mr. Hide*. In this case bytecode rewriting is adopted for replacing API calls, used by the original app, with interprocess communication primitives to query *Mr. Hide* service. These primitives are rather expensive, so there is a significant slowdown on API invocations. With our approach, instead, API invocations are "short-circuited" or removed altogether, making repackaged apps slightly *faster* than the original.

Finally, *Boxify* (Backes et al., 2015) has introduced a concept of app sandboxing on stock Android, based on app virtualization and process-based privilege separation. While this approach eliminates the need to repackage apps, it requires a lot of additional code (about 12 K lines of Java code, plus 3.5 K LoC of C/C++, according to the paper), which should be carefully audited[4]. On the contrary, our approach simply requires to customize 57 trivial Java methods. Moreover, Boxify requires the presence of a fully privileged controller process, called *Broker*, which is an attractive target for privilege escalation attacks.

[4] Authors promised to make the source code available, but at the time of writing, more than a year later, it is still unavailable.

5 CONCLUSION AND FUTURE WORK

We have presented a novel approach, and its supporting tool, to enable Android users to better protect their privacy by selectively removing permissions from any app, on any Android version. Indeed, our approach, which removes permissions from the set requested by an app by rewriting its APK, does not require any modification to the underlying operating system and is thus compatible with all versions and hardware architectures. The core idea is to remove the (user-selected) permissions from the app manifest, while redirecting API invocations, which need those removed permissions, to custom methods that return properly-crafted objects. These redirections, that we introduce by rewriting the DEX bytecode, aim at avoiding the throwing of (unexpected) security exceptions at runtime, which would make the app crash. We have implemented this approach in an open-source tool, RmPerm, that we have used to assess the effectiveness of our idea on a set of $81,000$ real-world samples. The experimental results have been encouraging; indeed, we have blindly removed, from these apps, *all* dangerous permissions obtaining that 86% of these rewritten apps can be installed and executed without crashing. We plan to run more fine-grained tests to assess which subsets of dangerous permissions are more problematic. Obviously, we could not expect a 100% success-rate: some apps do need some of the permissions they request. However, as we expected, the majority of them can be run equally fine with a strict subset of the permissions they originally requested. By using RmPerm, users can freely decide where to draw the “privacy line” and can run virtually any app without disclosing more personal information than they want to.

A limitation of our current implementation is that apps using anti-tampering techniques can detect that they have been rewritten. However, our experiments indicate that, for the time being, Android apps very seldom adopt anti-tampering techniques. Another limitation is that RmPerm currently redirects only “direct” API invocations, that is, we do not even try to redirect API invocations executed through the use of Java reflection or native code. We are considering how to extend our approach to intercept those reflective invocations too; however, this limitation is not severe as it may sound. With our approach, no Java nor any native code could ever exploit a removed permission, no matter what, since involved permission requests are actually removed from the app manifest. As remarked, the worst case scenario is a crash of the less-privileged app, but not a privacy leak.

Android 6 has introduced the possibility of both installing apps without granting all requested permissions at once, and toggling permissions at a later time. So, the usefulness of RmPerm could appear as dramatically reduced through the growing adoption of the latest Android versions. However, RmPerm offers a finer grained permission selection, which is unavailable in the Android user interface. In fact, Android 6 allows the user to only grant/-deny *groups* of permissions. For instance, because the user-level permission group *contacts* consists of the set of permissions `READ_CONTACTS`, `WRITE_CONTACTS` and `GET_ACCOUNTS`, an user cannot grant an app the ability to read his/her contacts, without granting the ability to write them too. While, by using RmPerm, such a policy is easily enforceable.

REFERENCES

(2017a). Androguard permission mapping. https://github.com/androguard. Accessed 2017-06-02.

(2017b). Android: distribution of market shares at march 2017. https://developer.android.com/about/dashboards/index.html. Accessed 2017-06-02.

(2017). Apkmuzzle app. https://play.google.com/store/apps/details?id=it.saonzo.apkmuzzle.

(2017). dexlib2. https://github.com/JesusFreke/ smali/tree/-master/dexlib2. Accessed 2017-06-02.

(2017). Empirical assessment with rmperm. https://github.com/CSecLab/BatchRmPerm/tree/master/dbDump. Accessed 2017-06-02.

(2017). Monkey. https://developer.android.com/studio/test/monkey.html. Accessed 2017-06-02.

(2017). Requesting permissions. https://developer. android.com/guide/topics/permissions/requesting.html. Accessed 2017-06-02.

(2017). Rmperm tool. https://github.com/CSecLab/ RmPerm. Accessed 2017-06-02.

Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. (2015). Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium*.

Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. (2013). AppGuard – enforcing user requirements on Android apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. (2011). Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM.

Davis, B. and Chen, H. (2013). Retroskeleton: retrofitting Android apps. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 181–192. ACM.

Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. (2012). I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. *Mobile Security Technologies*, 2012(2):17.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011a). Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM.

Felt, A. P., Greenwood, K., and Wagner, D. (2011b). The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7.

Hao, H., Singh, V., and Du, W. (2013). On the effectiveness of api-level access control using bytecode rewriting in android. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 25–36. ACM.

Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. (2011). These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM.

Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. (2012). Dr. Android and Mr. Hide: fine-grained permissions in Android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM.

Nauman, M., Khan, S., and Zhang, X. (2010). Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM.

Reddy, N., Jeon, J., Vaughan, J., Millstein, T., and Foster, J. (2011). Application-centric security policies on unmodified android. *UCLA Computer Science Department, Tech. Rep*, 110017.

Xu, R., Saïdi, H., and Anderson, R. (2012). Aurasium: Practical policy enforcement for Android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552.

Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. (2011). Taming information-stealing smartphone applications (on Android). In *International conference on Trust and trustworthy computing*, pages 93–107. Springer.