Identifying Previously Installed Apps on Android Devices

Kole W Swesey
Iowa State University
kwswesey@gmail.com

Abstract— **Determining if an app has been uninstalled from an smartphone could be useful in an investigation. Currently, there has been little work done on determining what an investigator needs to look for when doing so. In this paper, we propose a system of cataloging the files created by an app that a researcher could search a device image for into three different categories: files created when the app is installed, files created after the app is opened, and files that are left on the device after the app is uninstalled. and examining how consistent the contents of these files are and if they are truly unique to the app in question. We found files created on install are good for being consistent and unique to the app in question. Files created after the app is opened are often greater in numbers than files created during install and can be more consistent across installs, but are often not app-specific. Files left behind on the device after the app is uninstalled are rarely unique to the app, but this isn't as much of an issue since they will likely never be overwritten. This approach shows good results overall, but needs further work is needed to determine how truly unique some of these files are.**

Keywords—Android, artifact, checksum/hash, linux

I. INTRODUCTION

With over 1.3 billion devices in use as of 2020, Android phones make up over 86% of the smartphone market [1]. The fact that there are this many devices out there would mean that the ability to perform forensics on an Android device could prove critical to law enforcement. One thing that might be important in an investigation would be determining if certain apps are or were previously installed on the device. For example, if a suspect was accused of sending death threats over social media, determining that they uninstalled the app from their phone might be meaningful. In this paper, we will discuss a method for cataloging the files created by apps in order to develop a variety of searchable artifacts for an individual app. When performed at a large scale, this could lead to researchers finding evidence that multiple apps were installed on a device but deleted.

II. PRIOR WORK

Android devices have been available to the public since 2008, and it did not take long for groups to start developing forensic methodologies for the platform. One of the first comprehensive handbooks for forensic analysis of Android devices was released in 2011 [2]. The handbook covered areas including virtualization, file system structure, steps for imaging a device, and file carving methods. An investigator can image a device and carry out forensic steps as they normally would from a technical perspective. Effectively, Android forensics had all the key features that general computer forensics had.

But Androids aren't typical computers, and the main area that separates them apart is the concept of app forensics, since most interactions occur on dedicated apps instead of over a web browser. The initial contributions to app forensics were primarily for detecting privacy issues in an app, either so developers can fix them or so users can determine if apps they are using are leaking information. An early example of this would be CHEX, which was released in 2012 [3]. The purpose of CHEX was to find vulnerabilities through static data flow analysis, which had the main goal of finding entry points into the app. While not directly related to forensics, it did lay the groundwork for future work.

The concept of cataloging individual apps for forensic purposes is also not a new one. One early approach to making an App Evidence Database (AED) was accomplished through EviHunter, which was designed to search for files that an app creates that would contain useful evidence to an investigation [4]. It works by using static data flow analysis to track data as it moves from a sensitive source to a file sink, adding the path of any files that it detects doing this to the database. This process is automated and can quickly be performed across thousands of apps to fill the database. A forensic investigator analyzing or file carving an image can then use a matcher to see if any of the file paths stored in the database are present on the image. If so, the contents can be viewed and sensitive data can be found. EviHunter has shown to be effective at what it sought out to do, achieving nearly a 90% success rate per app, with thousands of apps tested automatically.

While this approach is good, it is purposefully narrow in scope, and is intended for only looking for certain files that contain information that might be considered evidence in a case. There are times where it might be useful to investigators to simply determine that an app was installed but has now been deleted. That alone might be useful evidence, but it could also lead to them looking for more critical artifacts once that has been discovered.

Not a lot of research has been done on looking for uninstalled apps on Android devices. One of the few examples of this is the ransomware forensics tool called RansomwareElite, which is designed to look for ransomware in Android apps, even if they are uninstalled [5]. However, RansomwareElite only looks for uninstalled apps by looking for .apk files on the system. It does not do any sort of file carving to look for artifacts of previously installed apps that might be ransomware. The scope here is limited since it's looking for apps that the user might be about to install or reinstall, and isn't worried about attempting to find every possible app that they can that the user has uninstalled. This approach would not be nearly sufficient for what we are trying to accomplish.

The straightforward approach to what we are trying to accomplish would be to just download an app, see what files it creates in the /data/data/<appname> directory, and search

for those files or parts of those files on the target device. Or even simpler, just search for the app's .apk file. However, files might be created in other parts of the file system by another process that is doing it in response to the app being installed or run. Some files might be created by the app while it is being run or when it is first opened. These files might significantly increase the number of files that a researcher would be able to look for. It would also be beneficial to look at what files are left behind after the app is uninstalled. Not only would these files be more likely to be discovered during an investigation since they might never be overwritten, it's possible that the contents of the file might not even be important, just the file path alone would be evidence that the app has been installed.

III. Our approach

We have developed an approach for testing Android apps in order to determine what artifacts could be searched for on a device image that could reveal if that specific app has been previously installed on the device. This testing process involves determining what files the app created on the device at three phases:

1. Immediately after an app has been installed
2. After the app has been opened and used for a period of time
3. What files have been left behind after the app was uninstalled

The first step in this process would be the simplest to perform and likely detect the artifacts that are most likely to be the same across all installs, since it leaves little room for differences between users. Checking for files created after the app is used is something that isn't as obvious, but it is possible that certain paths in the code would result in more files being created. There are multiple things that could cause these files to be created, such as stored login credentials, game saves, or images / videos that have been downloaded from the app. Finally, finding any files that are not removed when the app is uninstalled could prove to be very beneficial, as we might not have to worry about those files ever going away and could be much more likely to be found during an investigation. To perform these tests, we used LDPlayer as our Android emulation platform [6]. We chose LDPlayer because there were three main things we wanted to make sure our emulation platform allowed for. The first is easy access to the Android play store so we can download apps from there like a normal user would, which was included on the emulator in the exact same way as it would be on an actual phone. The second is the ability to quickly create new virtual machines. To avoid any sort of data contamination caused by other apps, we made sure to only test one app per virtual machine before deleting it. LDPlayer allows for us to clone a base virtual machine with everything setup how we need it to be. Finally, we chose LDPlayer because it allows for one click rooting of the device, which helps get around any issues we would have with file permissions when it comes to viewing parts of the file system. The base virtual machine that we used for each test only had one additional app installed, the terminal emulator termux. Termux was installed because it was the tool we used to track file writes by using the unix command tree. To start each test, we would run the tree command with the switches -afi, which resulted in the full file path of all files or directories printed without any additional syntax, creating a full list of files. This command was ran with a working directory of /data, since that is the only area that the app should be able to write to, and because including areas outside of /data in these tests would likely just cause files that are created by other processes running on the operating system to be counted. This command was run before the app was installed to determine what files were on the device before the app was installed. After that, the app was installed and tree was run again, comparing the results with the previous run to find all the files that are created immediately after the app is installed. This was done by using the linux diff command to compare the output of each run of tree. We then collected sha256 checksums on all of these new files as well, which helps determine if the files' contents are consistent across different users installing the app. Next, we opened the app and made a best effort to mimic typical user behavior before again running tree, comparing that output to the previous output of tree, and collecting checksums on the newly found files again. Finally, we uninstalled the app and ran tree one final time,

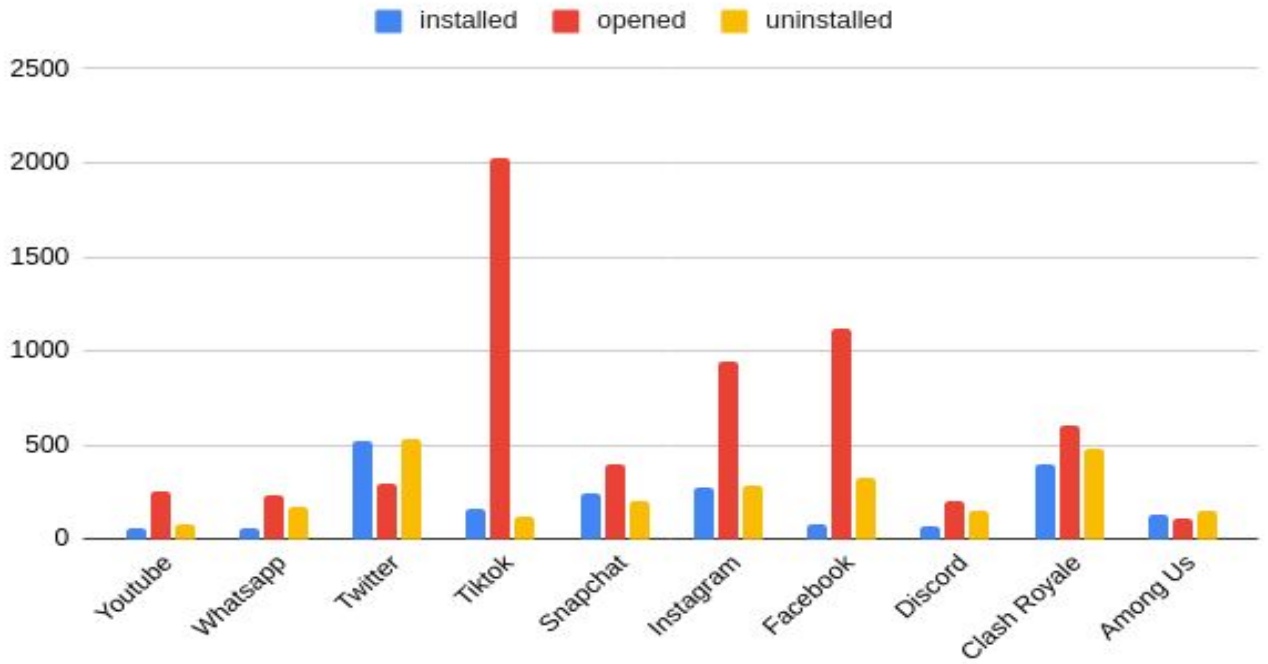

Figure 1: The number of files created by each app per file group

this time comparing the output to the original run of tree in order to see which files that were created by the app remained on the file system. Afterwards, we stopped using the virtual machine, created a new one, and ran through the steps again with the same app. Once that was done, we compared the hashes gathered between the two runs to figure out which files are the same on both installs, in order to determine which artifacts are most likely to be present across all instances of the app. We also compared the file names of the files left behind after the app was uninstalled, since that might be all that is needed in that case.

In total, we tested 10 different apps that we selected from among the most downloaded apps on the Google play store at the time of testing. We put an emphasis on selecting social media apps, since they are more likely to be relevant to an investigation than, for example, a gaming or productivity app. The apps tested were Discord, Twitter, Facebook, Among Us, Instagram, Clash Royale, Whatsapp, Youtube, Tik Tok, and Snapchat. Our two tests of the app were performed in quick succession in order to ensure that the same version of the app was being tested in each of the tests.

IV. RESULTS

After collecting the data on all of the apps, we analyzed the data in multiple ways. The first way we examined the data was by simply looking at how many files fall under each category, (added on install, added after use, left behind after uninstall) for each app, with the files in the final category also being present in one of the first two categories. The main purpose of this is to show the relative amount of the files that are only created after the app has been opened and the files that still exist on the device after the app is uninstalled. We noticed while doing our tests that the number of files present in for each app in each step would slightly fluctuate between the two tests that we ran, which poses a problem. To remediate this, we decided to include in our data the lower of the two values, because we figured that the lower value would be less likely to have files that were created by processes other than the app that is being tested, and will also likely have fewer files that are unique to the user's behavior, which therefore should give us more reliable data.

This data can be seen in figure 1, which includes a breakdown of the amount of files that fall into each of the three categories for each app, with the file count being on the y axis. It is important to note that while the 'installed files' and 'opened files' groups are disjoint, the 'left behind fies' group is not, and is made up of files from both groups. One takeaway from looking at this graph is that the amount of files that fall into both the 'opened files' and 'left behind files' groups are both relatively large, with the cumulative opened files count being approximately 70% greater than the installed files group. This shows that there is a lot of potential to come out of looking at the opened files group for artifacts based on the sheer size of the group. The cumulative percent of observed files that were not removed from the device after the app was uninstalled was calculated to be 39%, which also shows that looking at files in this category could lead to effective results.

We also tested to see how much overlap there was with the checksums of the files from each of the three categories between the two tests for each app. It's possible that full file names and paths might be lost when a file was deleted, so just knowing the names of these files wouldn't necessarily be enough when investigating a device. We would also want to know what files are going to be constant for all users, so we know what file contents are more reliably going to be discovered. Some file names might also be randomly generated, such as files for cached content or file names based on a UUID. To calculate the percent overlap, we calculated the number of hashes that appeared in the respective file category in both tests, while making sure to exclude the hash value of e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855, since that that hash is indicative of an empty file. This value was then divided by the lesser of the two test's total file count, again as an attempt to ensure that we are avoiding any files that were created by a task being run by some program other than the app that we are looking at.

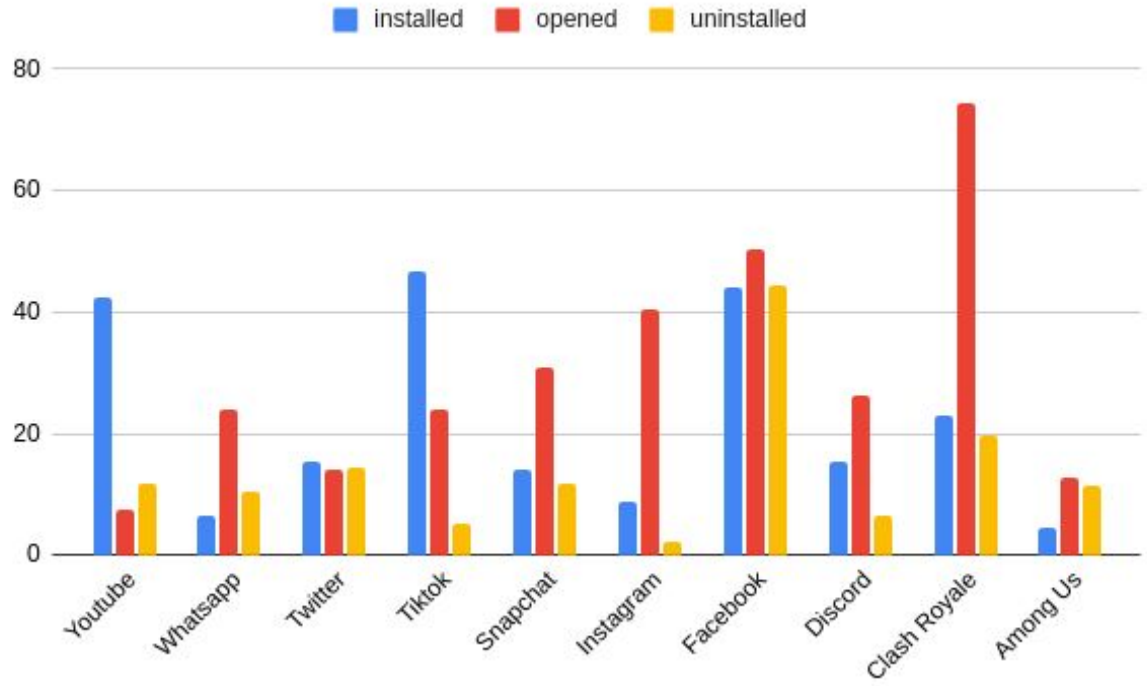

Figure 2: Percentage of file overlap between both tests of the app

This data can be seen in figure 2, with the percentage values being on the y axis. This test yielded results that were not consistent across the apps, having percentages that would range from less than 1 percent all the way up to 74 percent. On average, the hashes of the files created after the app was installed would overlap approximately 19% of the time, the files created after the app was opened and used would overlap approximately 38% of the time, and the files that remained on the device after the app was uninstalled would overlap approximately 16% of the time. The main takeaway from these results is that they show that the files created while the app was opened are more likely to be exactly the same across installs than the files that are created during installation. That coupled with the fact that the files created when the app is opened are more numerous than the files created during install further implies that these files are important to look for when determining if an app was previously installed on a device.

Youtube	Whatsapp	Twitter	Tiktok	Snapchat
50	59	227	29	77
Instagram	Facebook	Discord	Clash Royale	Among Us
28	216	49	289	30

Figure 3: the number of shared file paths between the two tests for each app

While the consistently high numbers we observed from doing this test were promising, manually looking over the files that were similar between the two installs revealed a serious issue. We noticed that there was an overwhelming amount of file names that were being repeated across the ten apps that we were testing. We believe that this is related to how our testing was conducted, since each app was installed on a fresh emulated Android system with only one additional app in it. The files that were left behind are probably files that contain information about multiple apps that meet some certain criteria, and the apps we installed met that criteria and caused the files to be created. In response, we did an additional test in order to determine what left behind file names are unique to each app that was tested. The total count of those files for each app can be seen in figure 4.

Youtube	Whatsapp	Twitter	Tiktok	Snapchat
7	15	9	6	11
Instagram	Facebook	Discord	Clash Royale	Among Us
0	178	1	72	2

Figure 4: the number of shared file paths between the two tests for each app that are unique to the app in question.

This greatly reduced the number of files present for almost every app that was tested and now puts us in a position where we can quickly analyze the remaining files.

This discovery led to us also double checking to make sure that the hashes that we collected were unique to the individual apps and weren't being found across multiple apps, which would potentially make them useless as evidence. We decided to perform the same tests that we did on the file paths for files remaining after the app is uninstalled to the shared hashes in the apps in each of the three groups of files.

The results of this test can be shown in figure 5. For the most part, the results show that a strong majority of the hashes of files created when the file is installed and after it is opened are unique to the apps being tested. In total, this data shows that approximately 67% of the hashes from files shared between the two tests of the app created during install are unique, 93% of the files created after the app is opened are unique, and 49% of hashes from files left behind after the app was uninstalled are unique. However, we did

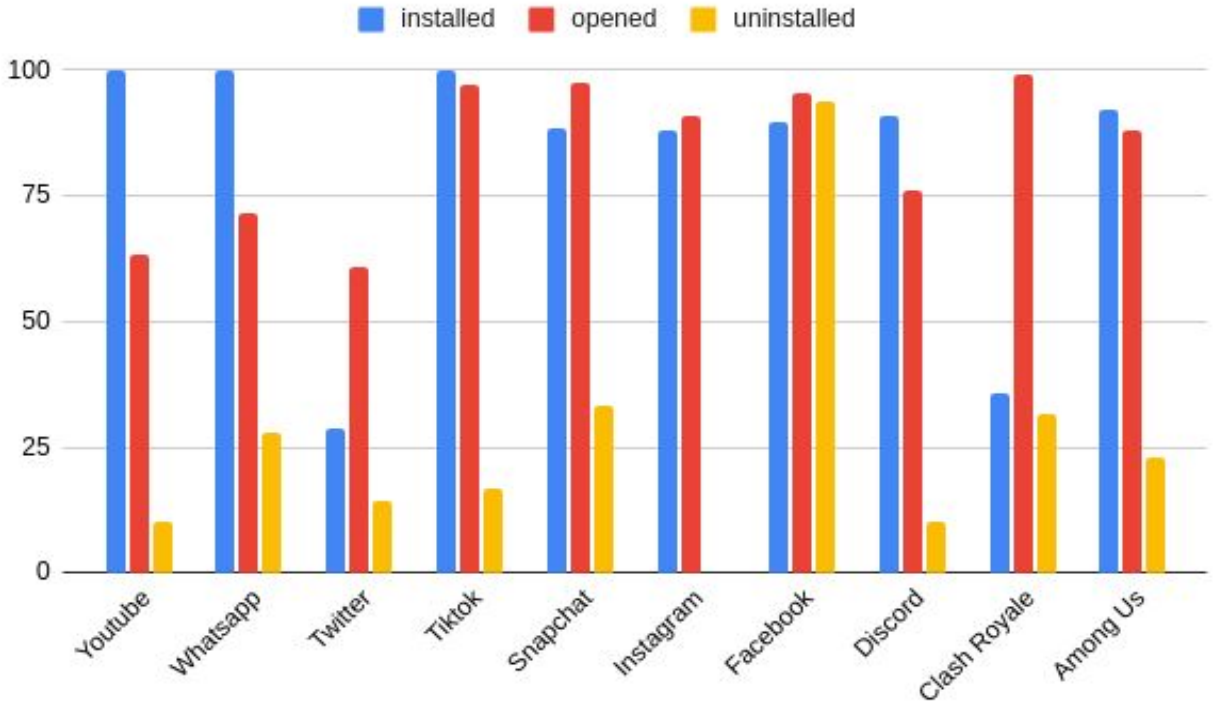

Figure 5: Percentage of files that were found in both tests that are unique to the app

notice a couple outliers here. The first outlier was in the area of files created when the app was installed. Both Twitter and Clash Royale ended up with percentages of 28.75% and 35.87% respectively, which is the result of approximately 50% of their files created during install having shared hash values between each other. Looking at these overlapping files between Clash Royale and Twitter revealed that the majority of them are located in the com.google.android.gms directory and appear to be a random assortment of .so and .apk files, which might be the result of another process running in the emulator. The other outlier that we noticed was in the files that were left on the device after the Facebook app was uninstalled. 93.93% of the file hashes were not found in any of the other apps, which greatly skewed the average. While it is possible that these files are actually unique to Facebook, it's possible that we simply did not compare it to enough other apps to figure out which ones are actually unique to the Facebook app. If the Facebook outlier was removed, the average percentage would actually be 23.17%.

At this point, we felt like we had collected all the data we would need to determine what artifacts left behind by an app can indicate that it was previously installed on the device, so the only thing left to do was to look at the artifacts and determine what patterns we can find from them.

Looking at what types of unique, shared files we found that were created on install, we did not see a whole lot of surprising data. This category was primarily made up of .so libraries stored in each app's respective /lib/ directories, as well as apk files found in their directories. Twitter also created non-empty log files upon installation, while facebook created cached content.

For unique, shared files that were created after the app was opened, we noticed more variety. Across all apps we observed .xml and .db files that appeared to be local databases or configuration files being created. While these files will likely be modified as the app is used, the basic structure of the dbs/xml files can still be searched for. On Whatsapp, Twitter, and Clash Royale we noticed log files being created, with clash royale's log files being located in play.games. Whatsapp, Instagram, Facebook, Discord, Clash Royale, and Among Us all created a significant amount of cached content, including a cached video from Instagram. Some apps created entire new directories in the /data/data directory that had essentially an entire other app worth of different files. The examples of this were Tik Tok creating com.zhiliaoapp.muscally (Tik Tok's predecessor), snapchat creating com.snapchat.android (not to be confused with com.android.snapchat), instagram similarly creating com.instagram.android, and facebook creating gkbootstrap, lib-xzs and com.android.tts. Clash Royale created a directory called 'update' in its folder that contained hundreds of .sc files.

A common trait shared amongst the files that were left behind after the app was uninstalled, whether we are looking at shared unique file names or shared unique hashes, is that all apps left behind files that existed in the com.android.gms (google mobile services, a collection of APIs for apps to interact with google owned apps) or com.android.vending (which is Google Play Store related) [8] directories that seemed to all have generic names with no relation to the app. Further work might be needed to ensure that these filenames or file contents are actually unique to these tested apps and aren't just the result of a small sample size. Some apps did leave behind notable things outside of these directories, however. WhatsApp left behind a directory called /data/media/WhatsApp, which also contained an empty .shared directory, and in that an empty .nomedia file based on their respective hashes. .nomedia files are commonly created by apps to indicate that any photos/videos in the directory shouldn't be viewable by the image gallery [8], but the directory name reveals which app created this one. Among Us also leaves behind a .nomedia file but it is in the media/obb directory. An image downloaded from the Twitter app created a directory called "Twitter" in the IMAGES folder. Clash royale left behind a play log file in play.games, as well as game images and databases that appear to be named after various levels in the game. Facebook left behind the entirety of the .com.android.tts directory that it installed, which would also explain the outlier in figure blah. Facebook also created a ./media/0/DCIM/Facebook directory to sure downloaded images and videos.

V. Critiques

As a whole, this approach appears to be successful at determining if an app was previously installed on an Android device. There are shortcomings however that would require further research to find solutions to. One issue is that apps are often updated, and as a result, artifacts that are produced by one version of the app might not be present in future or previous versions. While these changes might not be significant between individual versions, they will likely get bigger overtime and might make the artifacts used in a search critically out of date if they aren't updated. Our approach also does not consider pre existing files that might be modified by an app being installed. It could be possible that an app install might cause an existing file to be modified, with those modifications persisting beyond an uninstall. This could be great evidence, but would be undetectable by this approach. We are also making a potentially incorrect assumption that every new file is directly related to the file being installed. It is possible that background tasks we aren't controlling are creating new files that are completely unrelated to the app we are testing. While doing multiple tests might cause these files to be discarded, assuming they are different for different users, we have no way of identifying them in our current methodology, however all the files located in /data/data/<appname> should be created as a direct result of the app being installed or ran. The practicality of our testing methodology that we used is diminished by the fact that it is very time consuming, since everything was done manually in an attempt to accurately mimic user behavior. If this testing wants to be performed on apps at a large scale, methods of automating the testing process are critically needed. The timing of when we take the checksums also might be an issue. For example, we gather checksums of the files that are written on install before ever opening the app, which means we are making a large assumption that these

files would only potentially change when the app is updated, and not by the actions of the user. Rerunning the checksums on those files in particular after the app has been used might've provided us with useful data on what files can't actually be trusted to stay the same. This is especially true for any of the files that appear to be databases. Finally, since we are only looking at files in the /data directory this approach pays no attention to the contents of /proc. Due to the rapidly shifting nature of /proc, it would be difficult to determine what files in /proc are related to the app from simply looking at file creations. An effective way of doing so could potentially provide investigators with meaningful data. There also might be file writes going on in other directories located in /, but /data and /proc are the two most likely to be meaningful. Some observations we made indicated that our decision to test each app on a mostly clean slate of Android might have led to us collecting information on more files than we should have, which is what we were specifically trying to avoid with that decision. It might be a better idea to run these tests on a device that already has a random assortment of apps installed on it to gather more accurate data. The sample size of 10 apps also could be causing our count of unique file paths and hashes to be off, which likely lead to some of the outliers that we experienced in these areas. Despite these shortcomings, we believe that the data collected is in a large enough quantity that it can be used by forensic investigators effectively, but determining how to scale these tests needs to be done meticulously to ensure that these issues are corrected.

VI. CONCLUSION

This paper set out to look at ways to collect artifacts on apps that could be used to determine if an app has been previously installed on a device. Dividing files into three different categories demonstrated how there are advantages and disadvantages to looking for certain types of files based on when they are created and if they aren't removed when the app is uninstalled.. Looking at files created when the app was installed naturally showed that searching for the apps .apk file and the .so libraries the app uses would be successful if they haven't been overridden yet, and that some apps might consistently create files beyond those during install. Files created after the app was opened showed to be a much more diverse area with a lot of potential, with their high quantity and high consistency between apps, coming from a mix of cached content, databases/xml files, and in some cases entirely new directories in /data/data. The files that remained on the devices after the app was uninstalled showed to be, in many cases, potentially not unique to the app being tested. Some files left on the device have file names and paths that make it obvious what app they were created by, while others show up with more ambiguous names and locations that might also be related to other apps that we did not include in our tests. Some of these files, however, were clearly unique to the app being tested, and since they might never be deleted, only finding one file per app could be all that is needed. While there may be too many different things to consider in each test for this concept to be automated and used to fill a large database, we do believe that concepts that this paper introduced show enough potential that they still could be at least used by investigators on a case by case basis as they see necessary.

REFERENCES

[1] “Smartphone Market Share - OS,” *IDC*. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os. [Accessed: 15-Nov-2020].

[2] A. Hoog and J. McCash, *Android forensics: investigation, analysis and mobile security for Google Android*. Waltham, MA: Syngress, 2011.

[3] Lu, Long, et al. "Chex: statically vetting Android apps for component hijacking vulnerabilities." *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012.

[4] Cheng, Chris Chao-Chun, et al. "EviHunter: identifying digital evidence in the permanent storage of Android devices via static analysis." *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018.

[5] Shivangi, G. Sharma, A. Johri, Akshita, A. Goel and A. Gupta, "Enhancing RansomwareElite App for Detection of Ransomware in Android Applications," *2018 Eleventh International Conference on Contemporary Computing (IC3)*, Noida, 2018, pp. 1-4, doi: 10.1109/IC3.2018.8530614.

[6] “Fastest Android Emulator for PC, Free Download,” *LDPlayer*. [Online]. Available: https://www.ldplayer.net/. [Accessed: 15-Nov-2020].

[7] “Google Mobile Services,” *Android*. [Online]. Available: https://www.android.com/gms/. [Accessed: 15-Nov-2020].

[8] M. Kun, “Getting Know About .nomedia Files on Android,” *Medium*, 17-Feb-2020. [Online]. Available: https://medium.com/@mutiakun/getting-know-about-nomedia-files-on-android-a51068430e38. [Accessed: 15-Nov-2020].