AppTwins: A new approach to identify app package in network traffic

Xiang Li, Chao Zheng, Chengwei Zhang,
Shu Li, Li Guo
National Engineering Laboratory for Information Security Technoniges
Institute of Information Engineering, CAS
Beijing, China
email: {lixiang, lishu}@iie.ac.cn

Jie Xu
National Computer Network Emergency Response Technical Team
CNCERT
Beijing, China
email: xujie@cert.org.cn

***Abstract*— The smartphone applications have taken place of the web browser and became the user's primary internet entrance. One application's popularity can be measured by its downloading times, and it is valuable for commercial advertising. Identifying app installation packages from network traffic is one of the most feasible approaches to collect these data. But asymmetric routing, incomplete capture and so on make it challenging to determine app's name at large scale in network traffic. With these constraints, we proposed AppTwins, an efficient, robust and automatical approach which has the ability to determine corrupted package's name. The identification consists of three distinct steps. Step 1, identify app packages with a stream fuzzy hash fingerprint database in live network traffic. Step 2, the unprecedented ones were captured and decompiled to acquire new app's name, a fingerprint was also calculated. Step3, update the database with new app's name and fingerprint. AppTwins achieves up a recall rate of 97.63% and a precision rate of 96.44% when app packages are almost complete. Furthermore, It can also identify incomplete app packages in the real traffic where there are no name or URL.**

Keywords—app package; identify method; network traffic; incomplete capture; high concurrency

I. INTRODUCTION

Mobile apps have become extremely popular due to the accelerating rate of adoption of smartphones. For example, more than 65 billion apps have been downloaded from Google Play by May 2016 [1]. What's more, popular application stores have made it convenient for users to download apps quickly, such as App Store on iOS, Google Play on Android, and Market Place on Windows Mobile. By Jan 2016, more than two million apps are available on the App Store and have been downloaded more than 10 billion times [2]. With the vigorous development of mobile Internet, more and more people pay attention to app implementation rather than web service.

As apps have become primary internet entrance to network service, the commercial value of apps get advertising agency's attention. For advertisers, the number of app downloads is a critical factor for them to make a final offer. So it is crucial to collect the statistic of app downloading times. Due to the existence of various third party app markets and the information disclosure strategy of official app market, it is difficult to acquire an authentic result from app markets. Identifying app installation packages from the network traffic is an alternative approach to collect statistic data. There are some challenges on applying this approach in practice, that's including asymmetric routing and incomplete capture.

Previous studies have provided some approaches to solve this problem. Qiang Xu [3] identified apps by looking for app name in User-Agent. EricY.chen[12] used URL to mark an app. These approaches all depend on the information from both request direction and response direction. But, due to asymmetric routing, network flows in large ISP may only has one direction. So approach based on both direction is not suitable for every scenario. Seo [4] applied MD5 to identify apps. As a cryptographic hash algorithm, MD5 has an avalanche effect which means two files with a subtle difference will get totally different MD5 value. But two packages of same app (same version) which captured from network traffic, may not always be the binary match. That's because repackaging or incomplete capture. Stanislav Miskovic [5] proposed AppPrint, an approach which decompile and select tokens from the package as the fingerprint, and identify app with that. In a large scale network traffic, e.g. 100Gbps, decompiling each app package will consume too much computation resources.

The goal of our work is to find an efficient, robust and unsupervised solution to identify apps in large scale network traffic. Combining with previous constrains, we propose AppTwins, a new approach to solve this problem. AppTwins is designed for three explicit goals. 1) High **efficiency** means identify apps in large scale and asymmetric network traffic economically. 2) High **robustness** means the ability of identifying repackaged or incomplete app package. 3) **Automatically** means identify app name without manually specifying any samples or features.

The identification consists of three distinct steps. Step 1, identify app packages with a stream fuzzy hash [10] fingerprint database in live network traffic. Step 2, the unprecedented ones were captured and decompiled to acquire new app names, a fingerprint was also calculated. Step3, update the database with new app name and fingerprint.

In this paper, app has the same meaning as app package. The

<key>appleId</key>
<string>someone@126.com</string>
<key>bundleShortVersionString</key>
<string>qqmusic<string>
<key>purcahseDate</key>
<date>2016-11-20ST06:44:312</date>
…the rest of app data

Fig. 1. An sample of app package which is repackaged

rest of the paper is organized as follows. We overview the related work in section2. In section 3, we present challenges that app package identification faces. We describe our design detailed in section 4. In section 5 we evaluate AppTwins on a month-long traffic trace collected from a major Internet Service Provider(ISP) of China. We make some discussion about our work in section 6 and conclude in section 7.

II. RELATED WORK

The most straightforward way to identify app packages is to look for app name in HTTP User-Agent fields [3]. However, not all User-Agent domain contains app name because it is not essential. Stanislav Miskovic pointed out [5] that app name exists only in 1% of mobile traffic, while the rest is completely unknown to any characterization. Another approach is to search for identifier from auxiliary services embedded in the apps, such as ads or analytics (A&A) [6,7,8]. This may be effective in free apps but it may not work in paid apps.

EricY.chen [12] used URL to mark an app , but an industry paper of Sandvine [13] has pointed out that asymmetric routing is widespread and pervasive throughout networks. It means that a consumer application interacts with a destination server through two different routers. Due to this phenomenon, URL could not be acquired sometimes.

Seo [4] and Min Zhang [9] used MD5 to identify app packages. But it is a common observation that files captured from network flow are often incomplete due to packets loss or processing error. What's more, repackaging is widespread among app markets. MD5 does not work on these situations due to its avalanche effect.

Dai [7] developed a system that automatically runs Android apps and collects app fingerprints from the generated traffic. This may be effective , but it would be difficult to scale it to current app markets.

Choi [6] proposed an approach which installed a monitoring agent in mobile devices. The agent can collect data such as HTTP user-agent fields, HTTP hostname fields and IP subsets to identify app packages. However, installing such agents on user devices may be challenging due to privacy concerns.

Stanislav Miskovic [5] proposed AppPrint to identify app packages at a large scale. The system grabbed tokens from decompiled app packages and took token pairs as app feature. It could identify new app packages which have similar feature with collected ones. They devised a formula to evaluate the similarity. However, it will cost much time in decompiling files and grabbing tokens.

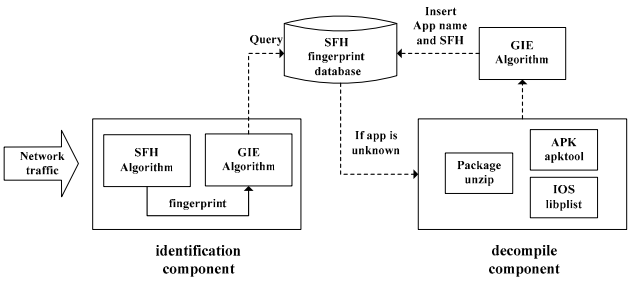

Fig. 2. Architecture of AppTwins

The approaches above are effective in most instances. However, the environment of real network traffic is complex. Asymmetric routing and incomplete capture are widespread. Thus they cannot work on some particular situations. These challenges will be discussed detailedly in next part.

III. CHALLENGE

As mentioned before, AppTwins was proposed to identify app packages in network traffic. However, running AppTwins on app packages from network traffic has several constrains.

Asymmetric routing: It has been pointed out that asymmetric routing is pervasive throughout networks. Network flows in large ISP may only has one direction. So approach based on both direction is not suitable for every scenario. Such as URL.

Repackaging: It has been observed that app repackaging is widespread. For example, App Store would add user account and downloading time to app package when users download an app. The example is shown in figure 1. In that case, packages of the same app differ from one another.

Incomplete capture: As mentioned before, it is a common observation that app packages captured from network flows are often incomplete due to packets loss and processing error. Incomplete capture makes it challenging to associate them with integral ones. Furthermore, incomplete app packages are difficult to be decompiled.

High concurrency: There are millions of app packages transmitted in network which would produce massive fingerprints. Similarity comparisons of fingerprints in massive data set is challenging. Edit distance is a general way to determine similarity between two strings. However, in large amount data the computation complexity of this algorithm is impractical.

IV. APPTWINS

In this section, some basic ideas behind AppTwins's design were explained to introduce it. Then two core algorithms were described in detail: (i)SFH, a method for producing app fingerprints, and (ii)GIE, a method for identifying app packages in the network traffic based on SFH fingerprints.

A. AppTwins overview

There are two components in AppTwins: **app identification**

0eh6jV0/QQaaH/LpfdITK5/eQYMkj9QE
pkvruLC2XjeoXNUQ0drT6vAxvX2tvmEY
+gagsl7fnRR+KfNmkppDTZZm[0:18278146]

Fig. 3. SFH sample

component and **app decompile component**. The whole architecture of AppTwins is shown in figure 2.

Firstly, the identification component captures app packages from the network traffic and calculates fingerprint with stream fuzzy hash.

Then search for similar apps in sample database to determine new app's name by GIE algorithm which is designed to do fingerprint matching. If the new app is unprecedented, the app package will be sent to app decompile component. Otherwise, the name of new app is as same as apps returned from sample database. Since the GIE algorithm can identify similar fingerprints, apps discussed in [14] which change their name to evade the detection techniques will still be matched if the packages don't change that much with original ones.

If an app package is sent to App decompile component, the component compiles it to extract its information including its size, app name, public version number and internal version number. The four basic features are chosen because they are representative for an app.

Since app packages are based on multi-national or multi-regional language version, the component acquires app information of all language versions so that users can get the specified information they want.

Then App decompile component associates these app information with its fingerprint. Finally, insert app information and fingerprint into sample database. It is a feedback mechanism to ensure that the sample dataset can be built continuously.

app identification component: this component identify apps by generating a fingerprint with stream fuzzy hash. If the app can be found in sample database, the name of this app can be identified. Otherwise, decompiling is needed.

app decompile component: this component is an important part to guarantee that AppTwins can automatically identify app without manually specifying any samples or features. App packages can be tagged automatically by decompiling instead of hand marking.

B. SFH：Fingerprint Genetation

The SFH algorithm is modified from fuzzy hash [11]. This algorithm inherits the characteristics of original fuzzy hash that it can identify similar files. In addition, it can deal with complex network environment. SFH can face the challenges of high concurrency, undetermined file length, unordered input and incomplete capture.

Fuzzy hash cut files into pieces by generating reset points. It is composed of rolling hash and a stronger hash. Reset points are produced by rolling hash. A stronger hash is used to produce hash values for each pieces between two reset points. The reset

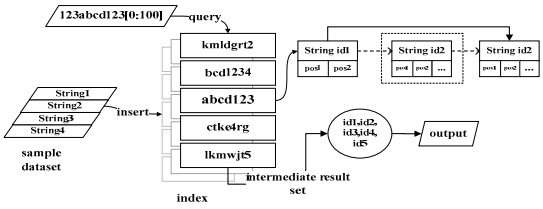

Fig. 4. GIE structure

point depends on the immediate context only. Fuzzy hash used a variable named block size b to trigger reset point, the blocksize can be calculated with (1). b_{min} is a constant minimum blocksize, s is a constant expected fuzzy hash length, n is an input file size.

$$b = b_{min} 2^{\left\lfloor log_2(\frac{n}{sb_{min}}) \right\rfloor} \quad (1)$$

A rolling hash function is composed of k window, an input sequence of k bytes:c1c2…ck when

$$\text{rolling hash}(c_1 c_2 \ldots c_k)\ mod\ b = b - 1 \quad (2)$$

A reset point will be positioned at c_k which means the smaller b is, the more reset point triggered statistically. The stronger hash based on the FNV algorithm is then used to produce hash values of the bytes between two reset points. The resulting signature comes from the concatenation of a single character from the FNV hash per reset point.

Fuzzy hash is introduced as above. In order to deal with various situations, SFH uses a context to record its intermediate state of rolling hash and strong hash. In addition, interval tree is used in this algorithm to organize one files with multiple discrete segment context.

When a segment comes, update the SFH segment context with this coming segment data. Then merge the adjacent SFH segment context when an input segment fill the gap of the interval tree.

Thus this algorithm is only a part of this system, it will not be discussed carefully, the details are elaborated in our other article [10]. The final result is a string which is called a SFH value. It is shown in figure 3. The string consists of two parts. The first part is generated from binary bits of app packages. The second part starting from '[' stands for the file offset being calculated. If an app package changes a little, only some alphabets of the first part will change.

C. GIE: App Classification

GIE algorithm is designed to do similarity comparisons of fingerprints to determine new app's name. The algorithm first builds a sample database by apps that can be decompiled and its name can be acquired.

Build sample database: Apktool was used to decompile apk and libplist was used to decompile ipa packages in the traffic. At the same time, a fingerprint of app packages is generated by stream fuzzy hash. If app name can be acquired from the decompiled package, a mapping relationship between app name and SFH would be established. Then GIE algorithm begins to build a sample database. The whole structure is represented by figure4.

In order to do fingerprint matching, an index must be built to reduce the number of strings being compared. N-gram is a common way to build index. So SFH strings are spilt into several n-grams to establish the index when building sample database. N-gram is a contiguous sequence of n characters from a given string. In order to generate n-grams, it moves one step character forward from the first character each time. For example, if n is 3, a string "abcde" will be spilt into "abc", "bcd"and"cde". The length of each gram can be any valid constant number. In practice, 7 was adopted as gram length.

Then these grams were inserted into a hash table. The key of the hash table is a string of 7-gram. The value is a link list. The app information was referred by its fingerprint's every 7-gram. What's more, gram position is also recorded in each node. If two grams are considered the same, not only do these two grams' contexts need to be the same, but also the position need to be not far apart.

One problem of n-gram indexing is that as the strings number grows, one gram's the link list may be too long. Pruning nodes was adopted to solve this problem. If the node number of a link list is bigger than a threshold *L*, nodes of this link list start to be pruned. To avoid pruning every gram of one string, those nodes that the string they referred has only *K* grams left will be preserved. *L* and *K* can be adjusted for different needs.

Query string: When a new app comes, SFH algorithm will generate its SFH value. The value is called a query string. The query string is split into several chunks. In order to speed up when querying, n-chunk is used instead of n-gram. Because the chunk number of a string is smaller than gram number. n-chunk is also a contiguous sequence of n characters from a given string. But n-chunks move n steps forward from the first character each time. Take a string as an example, if n is 3, a string "abcdef" will be spilt into "abc" and "def".

Following this, chunks are put into hash table to search if there are same chunks. As mentioned before, if the context of two chunks is the same and the position is close, the two chunks would be considered the same by our algorithm. Id of strings which contains same chunks with query string would be added into an immediate result set. Finally, string ids that appears more than C times in the set consist of the final result set. Each string in the immediate result set is called compared string.

The threshold is determined by confidence level cfds. It is calculated in (3). cfds is set by users according to their demands.

$$\text{cfds} = 10 * \left(\frac{C}{chunk\_cnt}\right)\left(\frac{gram\_cnt}{left\_gram\_cnt}\right) \quad (3)$$

C stands for the frequency of string id, chunk_cnt presents the chunk numbers of the query string, gram_cnt is gram number of the query string, left_gram_cnt presents gram number of compared string after pruning.

Cfds stands for the similarity threshold of two strings, so it has a strong effect on the precision rate and recall rate of the algorithm. The details will be discussed in the next part.

V. EVALUATION

In this section, the proposed algorithm AppTwins is evaluated together with three state-of-the-art approaches for app classification:(i)one based on the app name. (ii)one based on cryptographic hash (MD5). (iii)one based on URL.

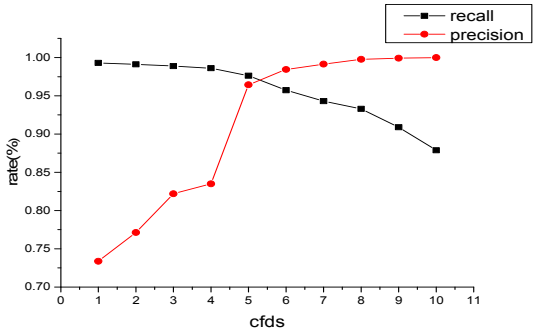

Figure.5. recall rate and precision rate via cfds

In the evaluation, there are three key parameters:(i)The longest length of link list L in the hash table;(ii)The largest remained gram number K in each link list node;(iii)Confidence level cfds. The relationship of cfds with precision and recall rate will also be shown in this part.

A. Dataset

A large dataset is collected from a major ISP in China for 30 days from 2016-10-12 to 2016-11-10. There are 236989 apps in total. The proportion of Android apps is about 77.93% with 184688 instances while the rest are iOS apps which take up of 22.07% with 52301 instances. The following two traces are scrubbed from this dataset.

Labeled Data: To evaluate the precision and recall rate, data that has a ground truth is needed. In our experiment, app name is considered as ground truth. 76038 integral app packages are picked out. their name was labeled by decompile tools automatically.

Unlabeled Data: The whole dataset including 236989 app packages is regarded as unlabeled data. 60% of these app packages don't contain names which could not provide any priori information as ground truth. So only performance and identification coverage of the algorithm are evaluated on this unlabeled data.

B. Precission and recall rate

In order to evaluate the precision and recall rate of AppTwins, the labeled data which does contain a ground truth is used. Using app name as ground truth which means that app packages with same name are considered the same.

First of all, all app packages were put into SFH algorithm to generate their fingerprint. In order to build a sample dataset, unique ones were picked out to insert into a hash table. In this experiment, 12992 strings were picked out. Following this, 76038 SFH strings were put into the hash table to do similarity comparison by GIE algorithm. Lastly, their name acquired from app packages is used to judge if the result is correct. In this part, nodes pruning is not set in GIE algorithm in order to focus on the effects of cfds. It was expected that the precision rate would be higher and the recall rate would be lower as cfds became

method	Precision rate	Recall rate
AppTwins	96.44%	97.63%
MD5	67.14%	100%
URL	60.10%	100%

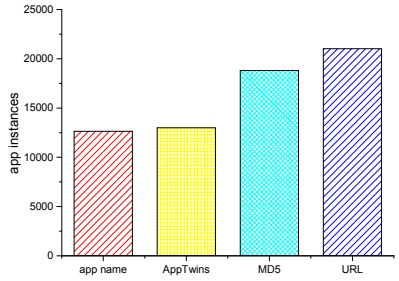

Figure.6. three methods comparing with ground truth

DEF_NAME: KuwoPlayer VERNAME: 8.2.0.0
URL:down.shouji.kuwo.cn/star/mobile/KuwoPlayerV3_ar_8.2.0.0_kwwap.apk
MD5: **81a525425c07cc3a3b5dd277089f8749**
FUZZY_HASH:393216:Dfv2bsSU6flZ3IYHlhxZDxAmkkGahx7Y5VxGdZatJ3**M**2UaaQljwvUoL37Mn0Ba**ABZ**dBxMiQ2UUuia4azCK[0:33958222]

DEF_NAME: KuwoPlayer VERNAME: 8.2.0.0
URL:down.shouji.kuwo.cn/star/mobile/KuwoPlayerV3_ar_8.2.0.0_kwwap.apk
MD5: **aa12cd5d88527169c1286b32dcfd3420**
FUZZY_HASH:393216:Dfv2bsSU6flZ3IYHlhxZDxAmkkGahx7Y5VxGdZatJ3**g**2UaaQljwvUoL37Mn0Ba**cfG**dBxMiQ2UUuia4azCK[0:33958222]

Figure.7. same apps with different MD5

bigger. The result is shown in figure 5.

As expected, recall rate decreased and precision rate increased when cfds became larger. A proper cfds value should be set in order to make a balance between these two metrics. In general, the intersection of two lines is regarded as the balance point, which is 5 in this figure for example.

In order to compare with schemes based on MD5 and URL, app name was still used as the ground truth and cfds was set as 5. App names characterized all apps into 12625 instances, our algorithm characterized apps into 12992 instances, while MD5 characterized apps into 18816 instances and URL characterized apps into 21020 instances. The result is shown in figure 6. The precision and recall rate are shown in table 1.

Table 1 summarizes precision rate and recall rate of the three method. AppTwins has the highest precision rate. It performs much better than other methods with an overpassing of almost 30%. Though AppTwins ranks last in recall rate. It can still achieve a relatively high level. So taken together, our method is much effective than MD5 and URL.

It is observed that there were many app instances which had the same names but with different MD5. It means these apps are the same app but their packages may be different. MD5 can't

DEF_NAME: QQUnrecalled VERNAME: 2.1.1.2
URL:**wechatunrecalled.sourceforge.net/apk/latest/qqu/app-release.apk**
MD5:b5c744cae53d51ef87b91bd80d457c23
SFH:12288:rlz2RMxyEXvya2727gYVanD8G+9o6ZDMB40b5+TG1IEw0ySu2gN/PERBixlWdns/o5p/UWxP5W[0:1140187]

DEF_NAME: QQUnrecalled VERNAME: 2.1.1.2
URL:**dlxda.xposed.info/modules/com.fkzhang.qqunrecalled_v35_b5c744.apk**
MD5:b5c744cae53d51ef87b91bd80d457c23
SFH:12288:rlz2RMxyEXvya2727gYVanD8G+9o6ZDMB40b5+TG1IEw0ySu2gN/PERBixlWdns/o5p/UWxP5W[0:1140187]

Figure.8. same apps with different URL

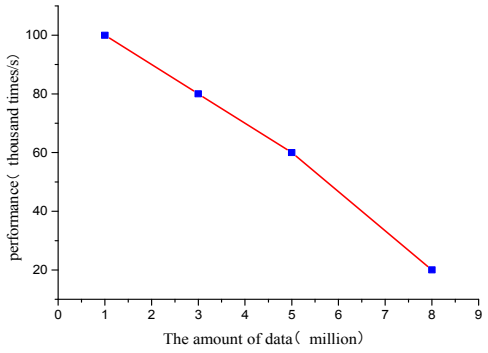

Figure.9. performance of GIE algorithm on different scale of data

solve this problem because of its avalanche effect. But AppTwins can deal with this problem because it generates similar SFH for similar files. One sample is picked to show this problem clearly in figure 7.

As shown in figure 7, the two apps are the same apps named KuwoPlayer. but they have different MD5 value. So there is a deviation if MD5 is used. But our algorithm is effective since there are just several different characters in our SFH string. Therefore our approach is more effective than MD5.

URL performs the worst in these three methods. Because it has the same problem with MD5 which lead to a low precision. The sample is shown in figure 8.

C. Identification scale

Though it is accurate to use app name to identify, app name may not exist in app packages at all. However, URL may also be missing due to asymmetric routing. If there are no name and no URL, AppTwins and MD5 are still available. But it have been proved that MD5 has a low precision rate. So AppTwins can be applied at a larger scale with a relatively high precision rate.

In order to illustrate this problem, the number of identified app instances was counted on online trace data by these methods. Firstly, app name was used to identify apps, there are 76038 packages that can be identified. It is about 32% percent. The left packages are without name. Then URL was used. 55% of the rest apps can be identified. The left ones are apps without name and URL. AppTwins can identify about 12% percent

DEF_NAME: **恋 U 夜 w 视 d 频** VERNAME: 1.3.32
URL: d1.codroider.com/dm/-e5-92-aa-e5-92-aa-e5-bf-ab-e6-92-ad_2_50311_4904190.apk
MD5: 2d94c6e6ad607e61a32869de1cdbedee
FUZZY_HASH:49152:**4K**KDWICTxKKo8e35/84BQ7XMQ44UHfh0xzo/3xuHvGHS0D1m0d+faVSt88Wt7EhUDu9rqU**3uE**/sNN**o**sNN**LIK7X1** [0:3396733]

DEF_NAME: **恋 V 夜 q 视 V 频** VERNAME: 1.3.32
URL: d1.codroider.com/dm/-e5-92-aa-e5-92-aa-e5-bf-ab-e6-92-ad_2_50320_4904341.apk
MD5: f1a5f8ae22346451ebdd6b81538be0de
FUZZY_HASH:49152:**S4**KDWICTxKKo8e35/84BQ7XMQ44UHfh0xzo/3xuHvGHS0D1m0d+faVSt88Wt7EhUDs9rqU**ASt**sNN**n**sNN**VsV2H0** [0:3405522]

Figure.10. same apps with different name

more. So take coverage into consideration, AppTwins performs better. Furthermore, Most of apps that can't be identified by AppTwins are damaged seriously.

D. performance

When the scale of dataset becomes bigger, performance becomes crucial. Assume that there may be a large number of apps in traffic, it means that our algorithm must deal with millions of SFH strings in the sample dataset. To identify app package in real time , several technics were used to speed up the process of SFH string query. In this section, the performance of our algorithm was evaluated on dataset of different scale. A mix of our online trace data and random noise data was used. As mentioned before, L stands for the longest length of link list in the hash table, K presents the largest remained gram number in each link list node. When K=50, L=50, the performance is shown in figure 9.

As plotted in figure 9, when the amount of data becomes larger the performance degrades. However, when amount reaches 8 million, AppTwins can still query 20 thousand times per second. So AppTwins can satisfy practical demands when there are tens of millions app packages.

VI. DISCUSSION

The result shows that the recall rate of AppTwins is relatively low. It is noticed that some apps with different name are identified as the same apps. Then some original app packages are checked manually. Original app packages are downloaded through URL and compared by Beyond Compare. What is surprising is that these packages are highly similar. two packages are picked out to show in figure 10.

As shown above, the name of these two app packages is different. But their SFH is similar. The original app packages are compared by Beyond Compare. About 90% of the two files are the same. It means that there is a big probability that they are the same app. So in this paper, using app name as ground truth may not be that convincing. Apps with a similar name may also be the same. some semantic analysis could be done about app's name to decide whether they are the same apps or not. The result may be more convincing.

VII. CONCLUSION

This paper proposed AppTwins, a new approach to identify app packages in real Internet traffic. The approach achieves this by its unique capability to produce app fingerprints and compare the similarity of two fingerprints.

Firstly, AppTwins captured app packages from network traffic. Following this, It calculated a fingerprint of packages with stream fuzzy hash and do a similarity comparison with strings in sample dataset. If similar ones couldn't be found, the app package was decompiled to acquire its name and add it to sample dataset.

At the end of this paper, AppTwins was evaluated on a labeled trace and an online trace data collected from a large ISP in China. The results show that AppTwins outperforms state-of-the-art approaches by achieving up to 97.63% recall rate and 96.44% precision rate on labeled trace. In addition, It can identify 12% apps more than other approaches when there are no name or URL in app packages on online trace.

REFERENCES

[1] https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/

[2] https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/.

[3] Xu, Qiang, et al. "Identifying diverse usage behaviors of smartphone apps." Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference. ACM, 2011.

[4] Seo, Seung-Hyun, Dong-Guen Lee, and Kangbin Yim. "Analysis on maliciousness for mobile applications." Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on. IEEE, 2012.

[5] Miskovic, Stanislav, et al. "AppPrint: automatic fingerprinting of mobile applications in network traffic." International Conference on Passive and Active Network Measurement. Springer International Publishing, 2015.

[6] Choi, Yeongrak, et al. "Automated classifier generation for application-level mobile traffic identification." Network Operations and Management Symposium (NOMS), 2012 IEEE. IEEE, 2012.

[7] Dai, Shuaifu, et al. "Networkprofiler: Towards automatic fingerprinting of android apps." INFOCOM, 2013 Proceedings IEEE. IEEE, 2013.

[8] Rastogi, Vaibhav, Yan Chen, and William Enck. "AppsPlayground: automatic security analysis of smartphone applications." Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013.

[9] Zheng, Min, Mingshen Sun, and John CS Lui. "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware." Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on. IEEE, 2013.

[10] Chao zheng, Xiangli, Qingyunliu,Yong sun, "SFH:Hashing Files on the Fly",unpublished.

[11] Kornblum, Jesse. "Identifying almost identical files using context triggered piecewise hashing." Digital investigation 3 (2006): 91-97.

[12] Chen, Eric Yawei, et al. "App isolation: get the security of multiple browsers with just one." Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011.

[13] Sandvine. "Applying Network Policy Control to Asymmetric Traffic:Considerations and Solutions".Industry paper, Sandvine, 2015.

[14] Zhauniarovich, Yury, et al. "FSquaDRA: fast detection of repackaged applications." IFIP Annual Conference on Data and Applications Security and Privacy. Springer Berlin Heidelberg, 2014.