DroidSD: An Efficient Indexed Based Android Applications Similarity Detection Tool *

JUNAID AKRAM, ZHENDONG SHI, MAJID MUMTAZ, PING LUO
Key State Laboratory of Information Security
School of Software Engineering
Tsinghua University Beijing 100028 China
E-mail: {znd15; szd15; maji16; luop}@mails.tsinghua.edu.cn

Android is becoming more and more popular in recent years. Meanwhile, it has been noticed that the security threats are also increasing with the passage of time. Most of the threats come by copying and pasting other applications code without knowing and evaluating it. Similar code fragments (clones) in Android applications make it very difficult to maintain these security flaws. To overcome these security problems, it is very important to discover, identify, retrieve, evaluate and recover these clones. In this paper, we propose and design DroidSD, a novel clone detection approach for android applications, that helps to detect different types of code clones from APK's source code. A prototype has been developed and implemented to detect clones in android applications. We downloaded almost 30,500 top rated android applications and decompiled those APK to get their Java source code by using reverse engineering techniques. DroidSD detects type-1, type-2 and type-3 (near-miss) clones in android applications at the source code level with high accuracy rate, which was not possible in previous Android similarity detection techniques. DroidSD can also detect the similar code fragments, that are injected into many applications, which may be an indication of spreading malware. Meanwhile it can detect full and partial level similarity between applications. We evaluate DroidSD clone detection approach on real time data-set and count the Recall and Precision on BigCloneBench, which is quite significant. Furthermore, our results show that our approach is very efficient and effective in detecting near-miss clones to check the similarity level in android applications.

Keywords: Clone detection, Maintaining APK code, Android apps re-usability, Plagiarism detection, Apps similarity detection, Information security

1. Introduction

Recently, smart phones are incredibly popular and very widely used in modern life. Android is one of the main smart phone operating system used globally. As we know that android is an open source operating system for mobile phones, that's why it has been supported by many well-featured applications. There are bundle of android app stores, which are providing the facility to download latest and updated versions of android applications[1]. For example, there are more than 3.8 million applications in Google play store [1] and more than 50 billion downloads [2]. These apps are actually providing very useful features in online payment systems, but meanwhile becoming the target for criminals to fraud. Mobile browsers also relied on sensitive security operations, such as online payments and transactions [3]. Android is very easy fraud target for attackers

Received March 11, 2014; revised June 20, 2014; accepted September 28, 2014.
Communicated by the editor.
[1]http://www.businessofapps.com/guide/app-stores-list/

because anyone can make their own apps and upload on official app stores. Even some people can download the original APK files and by using reverse engineering techniques, they can decompile the source code and rebuild app after making some changes in it [4] [5].

Because of very fast demand of new features in android applications, the developers are copying code from other apps and tries to reuse these fragments by pasting in other source code sections with or without modifying the code, this type of reuse source code approach is called code cloning, and the pasted code called as cloned code of the original code. It is very adapting approach, especially in application development activities [6]. However, during or after development process it is quite difficult to say which code fragment is the original and which was copied. Clones in source code actually bring a big trouble in applications security and maintenance [5] The previous research work shows that a significant amount of 7% - 23% of source code was actually cloned in large systems [7]. Even recent research work [8] [9] on very large systems [10] shows that 22.3% of Linux code has been cloned. A lot of research have been done in the field of open source code clone detection in large scale systems, but clone detection in android applications is still an open chapter. There are some techniques to check the similarity between android application [11] [12] [13], but these techniques check the similarity on basis of it's user interface, application signatures, application versions and uploaded patterns. But in our approach we avoid these comparing factors to check the similarity in applications and focused on analysis the source code of applications. Many researchers have explored the techniques to detect malwares in android applications [2] [14] [15]. Alam [15] and Chen [2] use clone detection techniques to detect malware in APK, Chen has used a third party component called NiCad [16] to detect malware code fragments in APK. Very rare researchers detect code clones at source code level by decompiling android applications and evaluate the source code. In our approach, we have detected code clones at very deep level of source code by extracting the main feature of APK source files.

To make sure the security and reliability of android applications, we develop DroidSD to detect the apps similarity, copy paste code from APK source files and injected code fragments, which can be a malware code. Meanwhile it can detect both partial application similarity and full application similarity. We have experimented DroidSD solution on the source code of 30,500 android applications, we get source code of these apps by using reverse engineering techniques. It detects type-1, type-2, type-3 code clones from android applications and retrieve the results in the form of similar code fragments. DroidSD worth more for application markets rather than an approach embedded in android devices or used by end users. The particular apps we actually are interested in finding code clones are those that copy code from other apps, or repackage existing apps. DroidSD is a semantic based clone detection approach, which is scalable, incrementable and can be extended to large scale android source repositories.

In this paper, our contributions are as follows:

- Decompiled and built index of 30,500 android applications.
- We developed DroidSD technique to detect similarity in android applications at source code level.
- Conduct experiments to evaluate DroidSD to identify code clones.
- It detects full and partial level similarity between applications
- DroidSD achieves significant results as compare to state-of the-art tool with high accuracy of 87%.

1.1 Types of Clones

Type-1 (Exact clones): The identical code fragments, which are the exact copies of code, except blanks, layouts, whitespaces and comments [10].
Type-2 (Parameterized/Renamed): Two Syntactically identical code fragments are similar except for variations in literals, names of variables, types, and functions [10].
Type-3 (Near miss clones/Gapped clones): Two copied code fragments with further modifications such as added or removed statements, the use of different literals, identifiers, types, whitespaces, comments and layouts [10].
Type-4 (Semantic clones): Two code fragments that perform the same computation but implemented by different syntactic variants. Or they are semantically similar, without being syntactically similar [10].

The cloned code is actually the similar source code fragment between two applications, where cloned type describes the degree of similarity between code fragments. Type-1 clones are the totally similar code fragments with 100% similarity because the code is fully identical. Type-2 have the similar source code with 90% similarity because in type-2 clones there are little variations in literals, names of variables, types, and functions but the source code is also similar. In case of type-3 clones, where new lines added in the original source code or maybe deleted, it is still suspected code which shows the similarity of 60% to 80% between source code fragments, which have been further categorized and explained in case study of section 6. So, the similarity between Android source code varies with the detected code clone types. Although two application share same source code but through DroidSD, we can find the degree of similarity between the source code.

2. Related Work

Detecting and evaluating code clones in applications or code bases have been very useful to many software engineering techniques such as refactoring and bug detection [17]. Researchers have explored and proposed different ways to identify and detect similarity between android applications [11] [12] [13], similarity between different documents [18] [19] and in open source files [10] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30]. Their results and methods have been employed by code clone management tools for open source projects [31] [32]. Prior research has shown that there are many cloned and fake applications in android markets [11] [14] [33]. Recently similarity detection in android applications is very hot research topic in software reliability and security [10], because a lot of applications are having vulnerable source code [34]. Haofei [12] performed a semantic analysis over android applications to detect clones, which compute similarity value of original apps. Mostly signature based techniques have been used for malware detection in source code [2]. Juxtapp is a system used to detect the similarity of code based on feature hashing in android applications [35]. It extracted the DEX files to analyze the code similarity in different applications. In this technique, the basic blocks are generated from XML files, which are derived from DEX file. Then the hash function is applied to k-grams into vectors. The similarity between two-bit vectors counted to check the similarity between two android applications. DroidMOSS [33] detect the similarity of two applications on the basis of fuzzy hashing, which can identify the repackaged or rebuild applications after decompilation. This method divides the entire package into small parts to compute the hash value of the individual part, and then combine all hash values into final fingerprint value of an application. Zhou [14] collected almost 1200 code malware samples to detect android malware in other applications. Jian Chen [2] collect the android applications which were known to be malware, then by using NICAD [16] near miss clone detection technique to detect the same code fragments in other applications [9]. He decompiled almost 1000 applications for clone evaluation. Sungmin Kim [36] proposed a method to detect illegally copied android application on the network. He extended data objects which were being transmitted from the network through sniffing, assembling and analyzing the packets. He made an analysis on extracted features of data objects are APK files or not. DroidClone [15] exposes the code clones in android applications by using MAIL (Malware Analysis Intermediate Language) technique. This technique uses a specific flow pattern to reduce the obfuscations effect. It can detect the byte code and native code simultaneously. Charlie Soh [13] proposed a clone detection technique based on applications UI (user interface) information. UI information can be collected very easily at run time without affecting it's behavior. It can also detect the repackaging attacks of android applications. In this approach, it runs the android Emulator to get the XML files then by applying hashing filters, it finally gets the similarity indexes of clones. AnDarwin [4] uses PDG (Program Dependence Graph) approach to detect the similarity of two applications. It only analyzes the application at the level of Java byte code. But in our method we check and identify the similarity at source code level, we use our own developed method to detect code clone fragments in two different applications. Svajlenko [37] have proposed an efficient way to detect large-scale near-miss clone in large scale systems. Recently SourcererCC [38] tool performs code clone detection in big code and it's extended version SourcererCC-1 [39] is an Eclipse plug in, which is using SourcererCC to detect, identify and navigate all clones during software development.

3. Problem and Proposed Solution

It is very easy to apply reverse engineering techniques on android APK files and rebuild them by adding some malicious code fragments inside. Mostly previous techniques have focused on similarity detection between application by consider it's user interface, uploaded information, signatures and patterns. But there are very less techniques, which checks the applications similarity at source code level. Even some developers copy some code from other applications and paste into their own application without testing that code, this code fragment can be the malicious code of original applications. So if we can detect that malicious code fragments from APK files we can use that code fragment as malware pattern to check in other applications. Hence, to perform that task code clone detection technique can be best solution. To make sure the security level and reliability of android APK files, we propose a scalable solution in the form of DroidSD for clone detection in APK source code files. Our purpose is to build a tool, where we can detect the cloned android applications. DroidSD has been tested on almost 30,500 android applications, those downloaded from AppChina market. It detects similarity between applications at different granularity. Our proposed solution can be defined by two main steps:

Step 1: (APK Source Code Collection): All 30,500 applications decompiled to get their DEX files by using reverse engineering techniques. From DEX files we retrieved JAR files, after that these JAR files further decompiled to a Java source files as shown in Fig 3.

Step 2: (APK Code Clone Detection): We developed DroidSD, a code clone detection tool to detect the same code fragments in different applications. We use Hadoop file system environment to build an index of 30,500 application's source code and save into

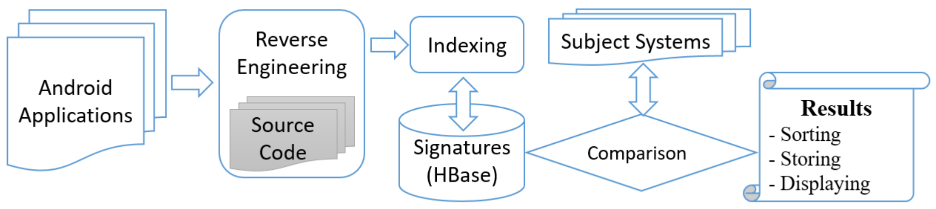


Fig. 1: Top level view of whole system.

HBase. Our code clone detection approach is hybrid (semantic), which detects type-1, type-2, type-3 code clones from the source code of android applications.

4. APK Source Code Collection and Decompilation

Google Play Store is the largest distributed android channel for android applications, which is actually offering global coverage for the big audience [5]. But we used AppChina[2] platform to download APK files. The reason we use AppChina store is that it has the application's review criteria [2]. Every application would be checked from developer's side before it released. AppChina has more than 30 million users and about 600 million applications have been downloaded every month[3].

4.1 APK Crawler

For the collection of APK files, we wrote a web crawler, which was downloading the top ranking applications from market including basic information, i.e Name, Path, Size, Downloaded Frequency and Category).

4.2 Reverse Engineering (.apk to .java source)

We used reverse engineering techniques to get the source code from APK files. The development process of an android app is shown in Fig 2.

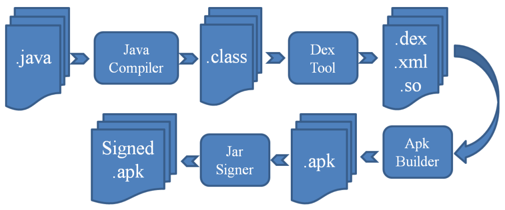


Fig. 2: Building structure of android applications.

Our most concern files in apk is DEX files because these are the files holding all source code of Java class files. Our performed reverse engineering steps have been shown in Fig 3. Reverse Engineering on 30,500 android applications was done in first part through the decompilation of all DEX files to Java source code files. We wrote Java scripts for every reverse engineering step to perform decompiling automatically. The following main steps and resources which were used in decompiling the APK files to source code.

[2]http://www.appchina.com

[3]http://www.businessofapps.com/the-ultimate-app-store-list/

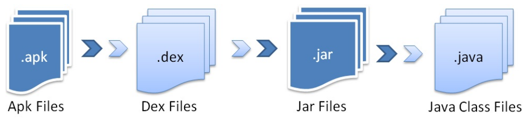


Fig. 3: Reverse engineering for android applications.

1. Unzip .apk files to get all .dex files.
2. Through scripts, we used Dex2jar[4] tool to get .jar files
3. We used JD-CORE[5] decompiler to get .java source files from .jar files

4.3 Excluding Third-Party Libraries and Obfuscation Handling

Third-party libraries may affect the accuracy rate of malware detection and may also slow down the detection process. So, we excluded them during signature generation and detection process of malwares. We uses a Whitelist to filter all third-party libraries. Although, it was not possible to build a complete Whitelist effectively because sometime obfuscation may change the name of packages. During experiments, we find many library files named ”com/d/c/b”, where come is the root folder. Which was quite hard to filter using Whitelist technique. So, We apply some filters against packages and classes names to ignore third-party library files. We extract the semantic features of different API calls and generated a vector sequences of directories. Then we perform comparison between these sequence vectors during malware detection and ignored these library files.

The obfuscation process make the source code difficult to explore, analyze and understand. Obfuscation can remove the literals, strings, variable names and sometimes it prevent decompilation of APK files. So, to overcome this problem, we used Annotated Control Flow Graph (ACFG) and Sliding Window of Difference (SWOD) techniques [40], which help us to detect the code similarity more effectively.

5. APK Code Clone Detection

In this section, we explain how our proposed DroidSD clone detection tool works. Our proposed approach is a hybrid (semantic), which performs several screening filters to detect absolute and exact code clones in APK source code files. The clone detection method runs in a pipeline, where every process fully depends on the output of the previous step as shown in Fig 4.

5.1 Preprocessing and Normalization

This is a first but very important process in APK code clone detection, which consists of several steps to make code ready for further procedures of detection. The source derived from APK files by using different reverse engineering techniques places in a separate repository to perform preprocessing and normalization on it. In preprocessing code retrieved from source repository and split into tokens. We replace all integers, variables, functions, methods etc. into specified ids and number, meanwhile all import utilities, comments and spaces ignored as shown in first part of Fig 5. Preprocessing of APK source code includes many steps, i.e. loading source code, verify and clean the source

[4]https://github.com/pxb1988/dex2jar
[5]https://github.com/nviennot/jd-core-java

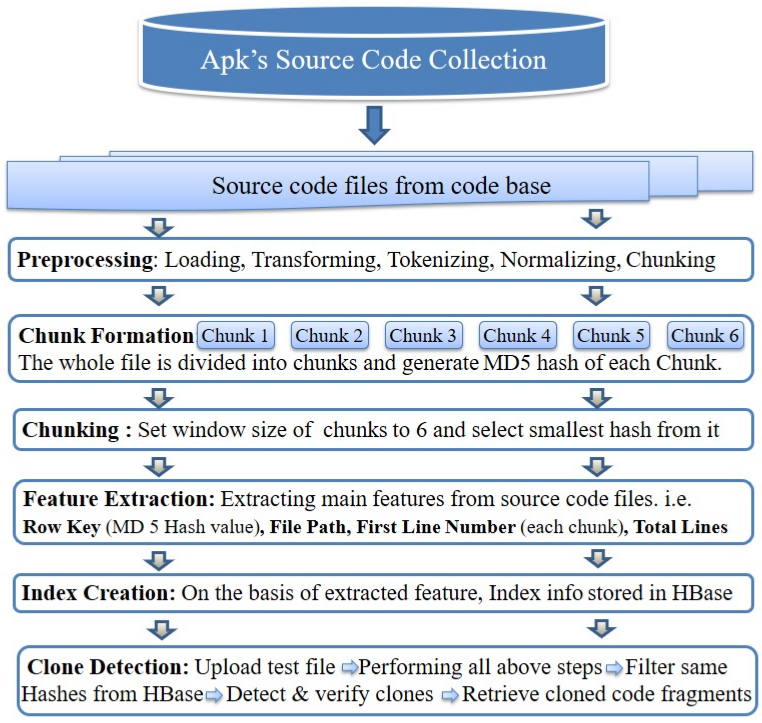

Apk's Source Code Collection
Source code files from code base
Preprocessing: Loading, Transforming, Tokenizing, Normalizing, Chunking
Chunk Formation Chunk 1 Chunk 2 Chunk 3 Chunk 4 Chunk 5 Chunk 6
The whole file is divided into chunks and generate MD5 hash of each Chunk.
Chunking : Set window size of chunks to 6 and select smallest hash from it
Feature Extraction: Extracting main features from source code files. i.e.
Row Key (MD 5 Hash value), File Path, First Line Number (each chunk), Total Lines
Index Creation: On the basis of extracted feature, Index info stored in HBase
Clone Detection: Upload test file Performing all above steps Filter same Hashes from HBase Detect & verify clones Retrieve cloned code fragments

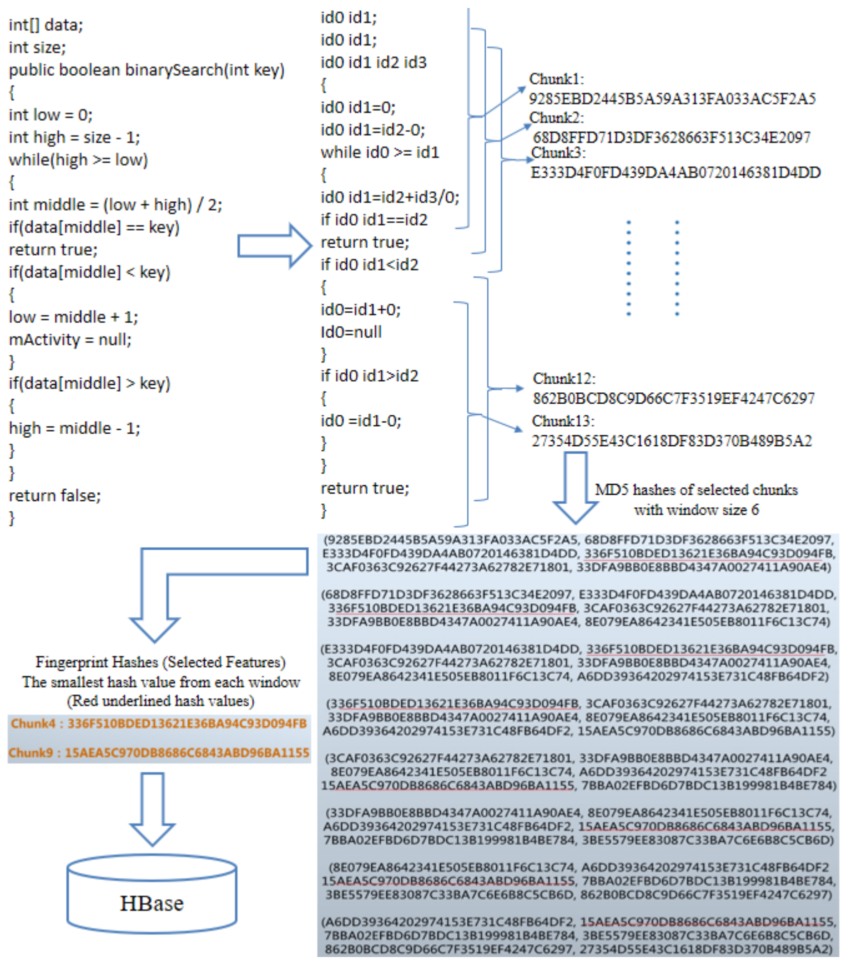
Fig. 4: DroidSD feature extraction and indexing process

code, tokenization for lexical analysis as shown in Algorithm 1, normalization,chunk formation, MD 5 hash generation, indexing, feature extraction and fingerprint generation as shown in Fig 4.

The ConQAT [27] lexical analysis tool was used for preprocessing and normalization purposes. This tool mainly filters the source code on the basis of programming language characteristics, i.e. preprocessing commands (import, include, package); visibility modifiers (static, public, private and so on); name space qualifiers; number all the identifiers, types, and constants. Meanwhile, identifier replaced by id+ numbers, replace ”empty” to string, integer replaced by 0, floating-point is also replaced by 0 and boolean replaced by true. In the ConQAT preprocessing section, we have made some modifications to its source code to implement the reserved converter operation so that the subsequent generation of chunk is easily generated.

5.2 Feature Extraction

This is very basic but the core phase of code clone detection in android applications, main features from the source code of APK files would be extracted to transform source code into representation form for code comparison. These features include MD5 hash values of every chunk, path of source code file, first line number of every chunk and total lines of code in each file. After preprocessing and tokenization is done, the chunk formation performed on every file by putting 10 lines of code into a single chunk. The MD5 hashing algorithm used to get the hashing value of each chunk. To select some of these hashes as a fingerprint from every source code file, we divide these hashes into groups (windows) and fixed the window size to 6 by choosing at least one fingerprint



```
int[] data;
int size;
public boolean binarySearch(int key)
{
int low = 0;
int high = size - 1;
while(high >= low)
{
int middle = (low + high) / 2;
if(data[middle] == key)
return true;
if(data[middle] < key)
{
low = middle + 1;
mActivity = null;
}
if(data[middle] > key)
{
high = middle - 1;
}
}
return false;
}
```

```
id0 id1;
id0 id1;
id0 id1 id2 id3
{
id0 id1=0;
id0 id1=id2-0;
while id0 >= id1
{
id0 id1=id2+id3/0;
if id0 id1==id2
return true;
if id0 id1<id2
{
id0=id1+0;
Id0=null
}
if id0 id1>id2
{
id0 =id1-0;
}
}
return true;
}
```

```
Chunk1:
9285EBD2445B5A59A313FA033AC5F2A5
Chunk2:
68D8FFD71D3DF3628663F513C34E2097
Chunk3:
E333D4F0FD439DA4AB0720146381D4DD

Chunk12:
862B0BCD8C9D66C7F3519EF4247C6297
Chunk13:
27354D55E43C1618DF83D370B489B5A2
```

```
(9285EBD2445B5A59A313FA033AC5F2A5, 68D8FFD71D3DF3628663F513C34E2097,
E333D4F0FD439DA4AB0720146381D4DD, 336F510BDED13621E36BA94C93D094FB,
3CAF0363C92627F44273A62782E71801, 33DFA9BB0E8BBD4347A0027411A90AE4)

(68D8FFD71D3DF3628663F513C34E2097, E333D4F0FD439DA4AB0720146381D4DD,
336F510BDED13621E36BA94C93D094FB, 3CAF0363C92627F44273A62782E71801,
33DFA9BB0E8BBD4347A0027411A90AE4, 8E079EA8642341E505EB8011F6C13C74)

(E333D4F0FD439DA4AB0720146381D4DD, 336F510BDED13621E36BA94C93D094FB,
3CAF0363C92627F44273A62782E71801, 33DFA9BB0E8BBD4347A0027411A90AE4,
8E079EA8642341E505EB8011F6C13C74, A6DD39364202974153E731C48FB64DF2)

(336F510BDED13621E36BA94C93D094FB, 3CAF0363C92627F44273A62782E71801,
33DFA9BB0E8BBD4347A0027411A90AE4, 8E079EA8642341E505EB8011F6C13C74,
A6DD39364202974153E731C48FB64DF2, 15AEA5C970DB8686C6843ABD96BA1155)

(3CAF0363C92627F44273A62782E71801, 33DFA9BB0E8BBD4347A0027411A90AE4,
8E079EA8642341E505EB8011F6C13C74, A6DD39364202974153E731C48FB64DF2
15AEA5C970DB8686C6843ABD96BA1155, 7BBA02EFBD6D7BDC13B199981B4BE784)

(33DFA9BB0E8BBD4347A0027411A90AE4, 8E079EA8642341E505EB8011F6C13C74,
A6DD39364202974153E731C48FB64DF2, 15AEA5C970DB8686C6843ABD96BA1155,
7BBA02EFBD6D7BDC13B199981B4BE784, 3BE5579EE83087C33BA7C6E6B8C5CB6D)

(8E079EA8642341E505EB8011F6C13C74, A6DD39364202974153E731C48FB64DF2
15AEA5C970DB8686C6843ABD96BA1155, 7BBA02EFBD6D7BDC13B199981B4BE784,
3BE5579EE83087C33BA7C6E6B8C5CB6D, 862B0BCD8C9D66C7F3519EF4247C6297)

(A6DD39364202974153E731C48FB64DF2, 15AEA5C970DB8686C6843ABD96BA1155,
7BBA02EFBD6D7BDC13B199981B4BE784, 3BE5579EE83087C33BA7C6E6B8C5CB6D,
862B0BCD8C9D66C7F3519EF4247C6297, 27354D55E43C1618DF83D370B489B5A2)
```

```
Chunk4 : 336F510BDED13621E36BA94C93D094FB
Chunk9 : 15AEA5C970DB8686C6843ABD96BA1155
```

MD5 hashes of selected chunks with window size 6
Fingerprint Hashes (Selected Features)
The smallest hash value from each window
(Red underlined hash values)
HBase

Fig. 5: Preprocessing, normalization, feature extraction.

from each window as shown in Fig 5 in red underlined hashes. By choosing at least one fingerprint from each window, we bound the maximum gap in fingerprints. Given a set of subject files, we want to find chunks matches between them should be satisfy two things.

1. If a chunk match, it verifies the actual threshold: t, then this chunk is detected.
2. There is no need to detect any chunk, who's threshold is smaller than threshold: u.

Where the t and u are constants and can be set by the user. The larger value of u makes us confident that the detected chunks are not coincidental.

Given a sequence of generated hash values $h_1, h_2 \dots h_n$, if $n > t - u$ then at least one hash h_1 must be selected to guarantee the detection of all clone fragments of at least length t. By considering a simple approach, let the window size $w = t - u + 1$. Now consider a sequence of MD5 hashes $h_1, h_2 \dots h_n$, which represents the chunks of a source code file. Every position $1 \leq i \leq n - w + 1$ in this kind of sequence, defines a window of hash values $h_i, \dots h_{i+w-1}$. So, it's mean to keep the guarantee of clone detection, it is very essential to select at least one hash value from every window size to be a fingerprint of the source code file. In our case we select the smallest hash value from every window as a fingerprint.

Algorithm 1 APK Feature Extraction Algorithm

Input: A is the lexical analyzer that takes filepath, token streams, normalized tokens and window size of chunks.
Output: Generate chunks and extract main features from the source file, i.e. MD5 hash values, list of chunks, first & last line number of every chunk and size of chunk. Then add all these features in HBase.

```
for $i \Leftarrow 0$ $to$ $unitsStandard.size() - 1$ do
if $(units.get(i).hashCode() == 10)$
   then $rowNum++$
   $rowRecords.add(i)$
 end if
end for
if $(rowActualNum \geq leastRow)$
   then $chunkList \Leftarrow \emptyset$
  $digest \Leftarrow buildHash(units, 0, row.get(size-1))$
    $chunk \Leftarrow$ new $Chunk(element.getPath(), rowNum)$
   $chunkList.add(chunk)$
  for $i \Leftarrow 0$ $to$ $row.size() - sizePerChunk - 1$do
     $digest \Leftarrow buildHash(units, row.get(i)+1, rowRecords.get(i+sizePerChunk))$
     $chunk \Leftarrow newChunk(element.getPath(), digest, i+1, rowNum)$
     $chunkList.add(chunk)$
   end for
end if
$featuresList \Leftarrow \emptyset$
$windowSize \Leftarrow 6$
for $i \Leftarrow 0$ $to$ $(chunkList.size() - windowSize + 1)$ do
  $minChunk \Leftarrow chunkList.get(i)$
    for $j \Leftarrow i+1$ $to$ $i+windowSize$ do
if$(chunkList.get(j).getHash() < min.getHash())$
      then $min \Leftarrow chunkList.get(j)$
end if
if$(!featuresList.contains(min))$
      then$featuresList.add(min)$
end if
   end for
 end for
$/*insertthefeaturesListintoHBase*/$
$insertChunks(featuresList)$
```

The main reason behind selecting the minimum hash value is that the minimum hash value in one window is probably remain the minimum hash value in nearby windows, so the probability is that the minimum value of w (random numbers) is quite smaller than one supplementary random number. Therefore, many overlying windows select the same hash value and meanwhile the number of fingerprints selected values are very smaller than the total number of windows while still sustaining guarantee as shown in Fig 5 in red underlined hashes.

The main idea of fingerprint used to set the window size 6 of chunks sequence and select the smallest hash value from each window. The repeated hash values ignored by keeping it's information i.e. file name, starting line, ending line of that specific chunk. Each window size resulted in a term as a fingerprint, which later used for comparison of different source code files to detect code clones. Fingerprint value is directly proportional to code clones; it means that if the file has bigger fingerprint value, it has much affected by clones. The same method also used during code detection process, where hashes values of chunks later on compared with the index values stored chunks in HBase. Normalized source code files from repository pass through some core steps to form feature extraction as shown in Algorithm 1.

Source code divided into chunks of equal sizes, each chunk consists of 10 lines of code, less than 15 LOC files discarded and ignored automatically. Chunk size is adjustable and can be varied any time, but for APK source code we set it at 10 LOC. In Chunk formation of a file, line from 1 to 10 would be formed as chunk1, line from 2 to 11 would be formed as chunk2, line from 3 to 12 would be formed as chunk 3 and so on. We add all these chunks into a chunk list and apply an MD5 hashing algorithm to get their hash values, which help to compare code fragments during clone detection. After getting all hashing values against all chunks, the least values from each window abstracted and saved in HBase as an index value. Here are the main extracted features.
MD5 Hash: This is the hashing value of each chunk.
File Path: Location where source code located in repository.
First Line Number: First line number of every chunk, to keep a record that from where this chunk starts. It helps us to retrieve clone fragments from the source code.
Total Lines: This feature is used to further verify that the cloned files have the same LOC or not. In case of both source files have the same number of LOC, it shows that file was fully copied from an original file.

5.3 Index Creation (Fingerprint Generation)

To minimize the indexing information and use less storage for index, there are two very important parameters considered in DroidSD: the fixed size of each hash value and the size of each window. These two parameters determine the eigenvalue density and the accuracy of clone detection. In order to reduce the storage cost and to improve the detection accuracy, we have carried out several rounds of tuning the two parameters, and finally achieved satisfactory results. Since code cloning is typically done by rows, the first parameter is the size of the chunk, which computed by lines. Based on your own writing code, or by referring to the experience of open source code, the general 10 line can complete a simple functional unit, or a basically complete logical structure. So, we set the size of chunk to 10 lines. The second parameter is the size of the window (set of chunks) determines the shortest possible number of cloned lines of code, which set as 6.

All the extracted features from every source code file of APK saved in HBase as an index information. Our proposed index creation method is very fast and reliable. It just took 26 hours to build an index of 41 million files & 983 million LOC accurately.

It considered takes very less time for large files as compare to small files of same size. Index function gets the source files path, total lines, chunks, MD5 hash values and stored in HBase. There were almost 83 million hash values were saved in HBase index table. The proposed index creation process is very fast, accurate, flexible and easy to maintainable.

5.4 Clone Detection and Retrieval

The proposed approach in this paper is a novel way to detect code clones in android applications. In this section, we explained the final task of DroidSD, in which it filters the clone fragments from source code and perform the evaluation. The concept of MapReduce has been used to detect and retrieve the cloning files from repository against every subject system (to be detected). All the subject systems uploaded to the HDFS file system and become the input of map() in MapReduce. Then pass through certain criteria of preprocessing, normalization, feature extraction and chunk formation to detect clones in it as shown in Fig 6.

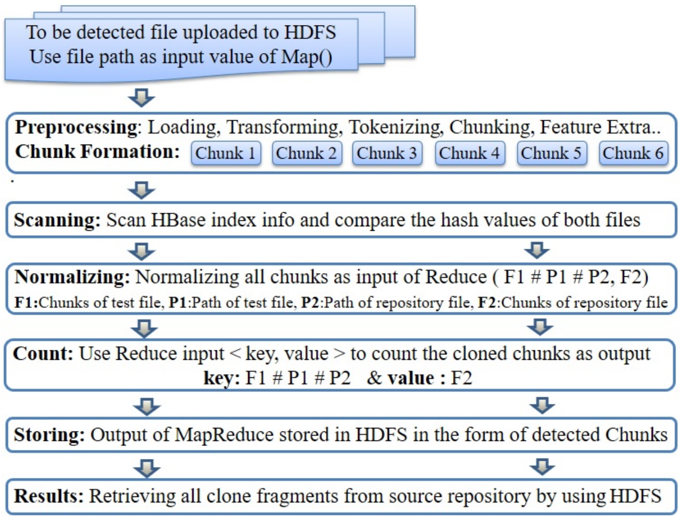

To be detected file uploaded to HDFS
Use file path as input value of Map()
Preprocessing: Loading, Transforming, Tokenizing, Chunking, Feature Extra..
Chunk Formation:
Chunk 1
Chunk 2
Chunk 3
Chunk 4
Chunk 5
Chunk 6
Scanning: Scan HBase index info and compare the hash values of both files
Normalizing: Normalizing all chunks as input of Reduce (F1 # P1 # P2, F2)
F1:Chunks of test file, P1:Path of test file, P2:Path of repository file, F2:Chunks of repository file
Count: Use Reduce input < key, value > to count the cloned chunks as output
key: F1 # P1 # P2 & value : F2
Storing: Output of MapReduce stored in HDFS in the form of detected Chunks
Results: Retrieving all clone fragments from source repository by using HDFS

Fig. 6: Clone detection architecture

At the initial stage of the detection process, all the preprocessing and normalization steps would have performed after uploading a subject system. After forming chunks of the subject system, the scanning process takes over the next procedure, which scans all the hash values of the index in HBase and compares them with the hash values of subject system. During the scanning process, it starts retrieving the chunks against similar hash values from HBase repository. After scanning and retrieving all detected chunks, DroidSD performs normalization on all chunks through reduce(). Inputs of reduce (F1# P1# P2, F2) function are the values of test files, where F1 is chunks of the subject file, P1 is the path of the subject file, P2 is the path of repository files, and F2 is the chunks of repository files. Chunks of subject files and repository files further compared and evaluated by considering their path value. All detected chunks counted one by one to find their occurrence in different repository files. After performing count step, MapReduce retrieves all the chunks, which are detected as clones and stores into HDFS for further process of retrieving code segments. The algorithm of APK clone detection part has been shown in Algorithm 2. In this algorithm the line 1-6 shows the initialized variables and functions. Lines 10-13 scans the stored hashes from big table against subject system hash values. Lines 14-17 MapReduce retrieves the detected results and normalised them for evaluation. Finally, all results saved in HDFS file and basic info stored in mysql.

Algorithm 2 APK Clone Detection Algorithm

Input: Subject System uploaded into Hadoop file system. preprocessing, normalization, formation of chunks performed.
Output: Retrieve all clone fragments from source code repository, sort them and count the similarity value between them.

```
1:  $Initialize\ some\ Variables$
2:  $hdfs\ as\ fs,\ hbasetable\ as\ ht\ and\ mysql\ as\ mq$
3:  $/*Consider\ cks1, cks2\ are\ chunks\ and\ \ ck : chunk,\ cks : chunks*/$
4:  $function\ detector(filepath)$
5:  $map(k,v) \Leftarrow filepath(k\ is\ a\ filepath\ value,\ v\ is\ 0)$
6:  $cks1 \Leftarrow preProcess(filePath)$
7:  $cks2 \Leftarrow chunkExtract(cks1)$
8:   **for** $i = 0\ to\ length(cks2)$ **do**
9:     **loop**
10:    $ck \Leftarrow cks2.get(i)$
11:     **If** $ck\ is\ not\ null$ **then**
12:     $scan\ hbase\ chunks,\ according\ ck's\ md5\ hash$
13:     $cks \Leftarrow chunks$
14:     **end if**
15:     $Let\ result\ 1\ is\ a\ (k,v)$
16:     $result1 \Leftarrow normalizeResult(cks)$
17:     $Store\ result\ 1in\ mapReducecalculate\ model$
18:     $reduce(k,v) \Leftarrow result\ 1$
19:       **for** $i = 0\ to\ length(k)$ **do**
20:         **loop**
21:         $countChunk(v)$
22:         $updateValue(v)$
23:        **end loop**
24:      **end for**
25:    $let\ Initialize\ hdfs\ file\ hf1,\ hf2$
26:    $hf1 \Leftarrow reduceOutput(k,v)$
27:    $insert(k,v)\ into\ mq$
28:    $hf2 \Leftarrow locateResults(k,v)$
29:    **end loop**
30: **end for**
```

The detected types of clones depend on the similarity measures between the subject file of subject system and the files in the repository. If the value of similarity is 1, then it's mean that there are type-1 & type-2 clones in subject file, and if the value varies from 1, then the clones would be considered as type-3.

5.5 Near-Miss (Type-3) Clones Detection

This is the distinguish characteristic of DroidSD to detect near-miss clones in Android application, where the source code have been changed by developers. These clone fragments have been changed by adding, modifying or removing statements during copying the source code from other applications. Many type-3 clones have modifies by swapping statements in a source file or by combing multiple condition statements into one. Detecting such kind of similar code fragments from apk is very challenging for other Android clone detection approaches. So, we focused on detecting near-miss clones. Near-miss clones were further divided into different groups, i.e very strong type, strong type, medium type and weak type and their detection results have been shown in Table 2. Fig 9 shows one of the detected near-miss code fragment, which was detected in WhatsApp application. The other near-miss clone fragments have been discussed below in case study section and the near-miss clones results have been shown in Table 4, which are quite significant.

6. Case Study of Clone Detection

In this part, the results of DroidSD approach have been shown and discussed. By downloading a big amount of android applications from AppChina store as shown in Table 1, we transform all .apk files to .java source code files by using different reverse engineering techniques. We preprocessed and normalized almost 41 million files to build and save their index information in HBase. All android application files decompiled in the same way and with the same tools, that's why the source code recovered by reverse engineering process was the same every time.

Table 1: Source Code and Index Info

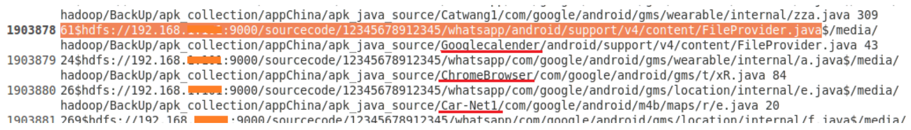
Downloaded Apk	Decompiled Source files	Lines of Code	MD5 Hash Values
30,500	41,261,694	983,975,411	82,998,573

System Specification: Our APK clone detection approach was performed on Linux operating system. All process till clone retrieval was performed on a single machine (Intel core-i7, 3.60GHz*8 & 24 GB of RAM) i.e. downloading android apps, decompiling source code, preprocessing, normalizing, indexing and clone detection.

We extracted the main features of source code files and built an index of almost 30,500 android applications to perform the clone detection based on index info comparison. For the evaluation of DroidSD, we took top android application named *Chrome, Firefox, Gmail, WhatsApp, GoogleMap, GooglePlayStore, Baidu* as a subject system to detect clones in them. These APK files were first decompiled through reverse engineering techniques to get their source code. HDFS has been used for uploading these applications decompiled source code to Hadoop file system for clone detection. Preprocessing, normalization, feature extraction were performed to make code ready for detection. Indexed

Table 2: Detection Results of Chrome, Firefox, Gmail, WhatsApp, GoogleMap, GooglePlayStore, Baidu and BigCloneBench

App Name	Total Files	LOC	Detection Time	Type-1 Clones	Type-2 Clones	VST3 Clones	ST3 Clones	MT3 Clones	WT3,T4 Clones
Chrome	8,291	618,682	2 h 53 min	230,305	142,137	82,949	5,300	192,604	266,316
Firefox	3,614	578,124	2 h 38 min	286,850	212,861	294,305	7,938	543,236	610,174
Gmail	10,796	874,888	4 h 36 min	172,784	214,676	129,893	16,027	250,949	381,531
WhatsApp	3,895	628,426	3 h 10 min	484,323	169,346	277,656	190,956	512,977	256,093
GoogleMap	12,794	1,115,701	5 h 57 min	110,209	146,753	120,458	16,328	226,588	318,798
GooglePlayStore	11,133	899,689	5 h 3 min	205,095	234,390	129,430	13,409	219,788	385,042
Baidu	3,326	631,469	3 h 8 min	22,755	15,596	7,071	2,309	12,650	20,800
BigCloneBench in (IJaDataset)	51,499	10,431,956	7 h 15 min	91,387	1,572,672	451,319	103,827	759,867	1,693,033



```
        hadoop/BackUp/apk_collection/appChina/apk_java_source/Catwang1/com/google/android/gms/wearable/internal/zza.java 309
1903878 61$hdfs://192.168.[redacted]:9000/sourcecode/12345678912345/whatsapp/android/support/v4/content/FileProvider.java$/media/
        hadoop/BackUp/apk_collection/appChina/apk_java_source/Googlecalender/android/support/v4/content/FileProvider.java 43
1903879 24$hdfs://192.168.[redacted]:9000/sourcecode/12345678912345/whatsapp/com/google/android/gms/wearable/internal/a.java$/media/
        hadoop/BackUp/apk_collection/appChina/apk_java_source/ChromeBrowser/com/google/android/gms/t/xR.java 84
1903880 26$hdfs://192.168.[redacted]:9000/sourcecode/12345678912345/whatsapp/com/google/android/gms/location/internal/e.java$/media/
        hadoop/BackUp/apk_collection/appChina/apk_java_source/Car-Net1/com/google/android/m4b/maps/r/e.java 20
1903881 269$hdfs://192.168.[redacted]:9000/sourcecode/12345678912345/whatsapp/com/google/android/gms/location/internal/f.java$/media/
```

Fig. 7: APK Clone Detection results (HDFS file)

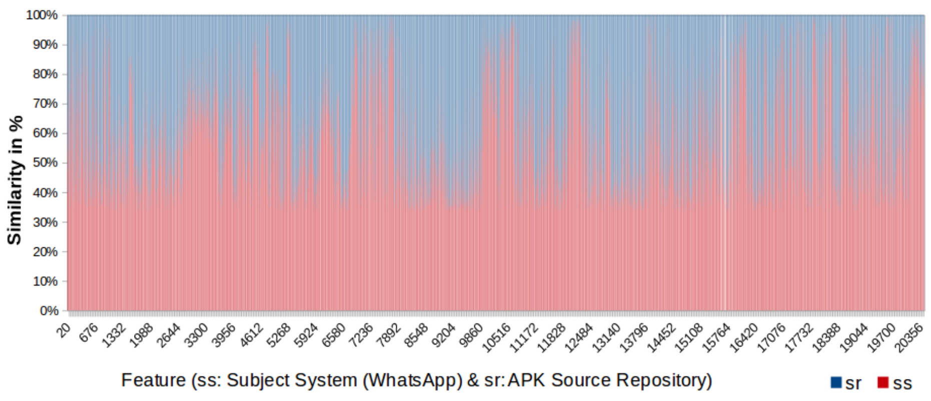
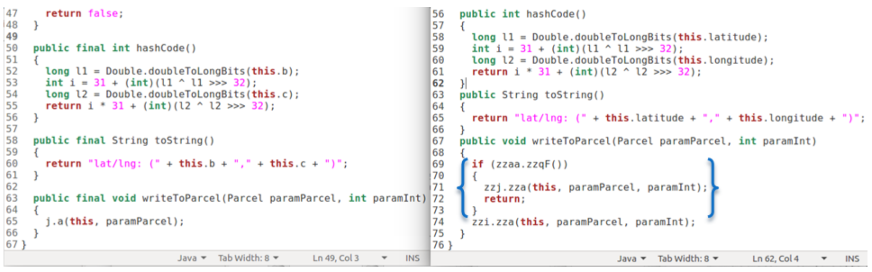


Fig. 8: Features similarity of WhatsApp in APK source repository

MD5 hash values of approximately 83 million from HBase were scanned and comparison between these hashes was performed through MapReduce. The detection time of each subject system is displayed in Table 2. Results of clone detection were retrieved from Reduce, which consists of feature similarity ratio in two files and their local path. Fig 7 shows of only 3 selected results of subject system (*WhatsApp*), it is actually the HDFS file of size 417 MB, which consists of 1,903,894 retrieved clone results. Each result in it gives the actual path of detected cloned subject file (*WhatsApp*), repository files and the number of similar features in clone files. The 1903878 fully highlighted line in Fig 7 is the file from WhatsApp, which was cloned in other android applications in our source repository, names of detected clone android applications in our source repository have been underlined with red color. The graphical representation of selected results of Fig 7 has been shown in Fig 8, which displays the actual similarity against different retrieved files. In Fig 8, the files of subject system (WhatsApp) are in red color and the files of source repository are in blue color. The white lines shows that the features in these files are very less, i.e the source code in these files is less than 10 LOC.

Table 2 shows the detection results of the subject systems against their total number of files, LOC, time and number of detected clones of type-1, type-2, type-3. Types of clones actually depends on features similarity of files, i.e. similarity: 1.0 shows type-1 clones, similarity: 0.9 - 1.0 shows type-2 clones, similarity: 0.5 - 0.8 shows type-3 clones. There is no hard and fast rule on when a clone is not syntactical similar, so it's quite hard to separate type-3 and type-4 clones, instead we divide and categorise them on the basis of their similarity measure, i.e. very strongly type-3 (VST3) clones have the feature similarity of 80-90%, strongly type-3 (ST3) clones have the feature similarity of 70-80%, moderately type-3 (MT3) clones have the feature similarity of 60-70%, weakly type-3/4 (WT3/4) clones have the feature similarity of 50-60%. The clones of less than 50% similarity can be type-4 clones, but not surely evaluated and considered in this paper.

The result of type-3 (near-miss) clone of our similarity detection approach has been



```java
    return false;
  }

  public final int hashCode()
  {
    long l1 = Double.doubleToLongBits(this.b);
    int i = 31 + (int)(l1 ^ l1 >>> 32);
    long l2 = Double.doubleToLongBits(this.c);
    return i * 31 + (int)(l2 ^ l2 >>> 32);
  }

  public final String toString()
  {
    return "lat/lng: (" + this.b + "," + this.c + ")";
  }

  public final void writeToParcel(Parcel paramParcel, int paramInt)
  {
    j.a(this, paramParcel);
  }
}
```

```java
  public int hashCode()
  {
    long l1 = Double.doubleToLongBits(this.latitude);
    int i = 31 + (int)(l1 ^ l1 >>> 32);
    long l2 = Double.doubleToLongBits(this.longitude);
    return i * 31 + (int)(l2 ^ l2 >>> 32);
  }
  public String toString()
  {
    return "lat/lng: (" + this.latitude + "," + this.longitude + ")";
  }
  public void writeToParcel(Parcel paramParcel, int paramInt)
  {
    if (zzaa.zzqF())
    {
      zzj.zza(this, paramParcel, paramInt);
      return;
    }
    zzi.zza(this, paramParcel, paramInt);
  }
}
```

Left picture: $hdfs://192.168..../whatsapp/.../gms/maps/model/LatLng.java
Right picture:$/.../appChina/apkSource/HellRider/unity/maps/ads/Unity.java

Fig. 9: Near-Miss clone detection result of APK

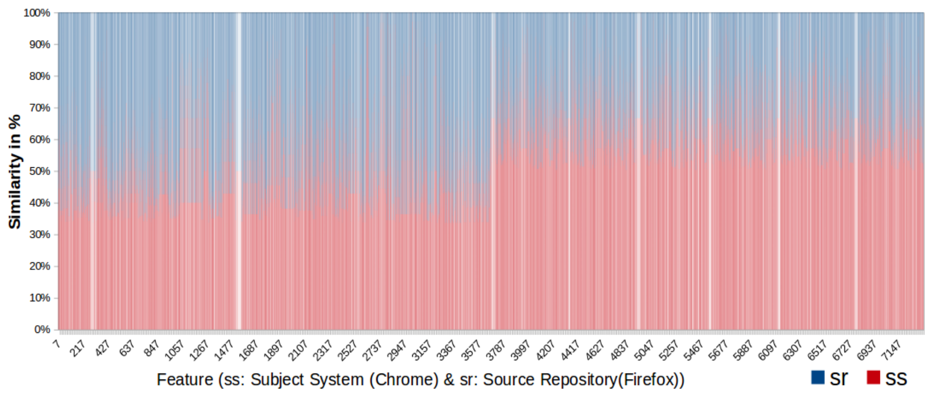


Fig. 10: Features similarity between Chrome and Firefox

shown in Fig 9, which describe that new lines have been added to the source code of the original file, or may be the lines of source code were deleted from the original file. Furthermore, there were some function & method names have been changed in cloned file. Left picture in Fig 9, is one of the subject file named *LatLng.java* from 3,895 files of *WhatsApp* application. The picture on the right is the file from another APK named *Hell-Rider*, which was detected and retrieved from our android source code repository.

Another case study has been done for Batch clone detection (one to one), in which two systems have been used, i.e Chrome and Firefox. The clone detection was automatically performed on each system one by one by considering one system as a subject system and another as a source repository and vice versa, the results are listed in Table 3. Through this experiment we can see the source code similarity between these two browser applications. It have been seen that many files of these application have been sharing the source code fragments. Through DroidSD we can even display those code fragments very effectively. The features similarity graph of Chrome over Firefox has been shown in Fig 10. Table 3 shows the results of type-1 type-2 and type-3 clones Chrome over Firefox (Chrome as subject system) and Firefox over Chrome (Firefox as subject system).

Table 3: Clones detected between Chrome and Firefox

App Name	Detection Time	Type-1 Clones	Type-2 Clones	VST3 Clones	ST3 Clones	MT3 Clones	WT3,T4 Clones
Chrome over Firefox	13 min	301	5085	670	221	1020	4820
Firefox over Chrome	11 min	67	2245	333	138	569	2181

7. Evaluation

In this section, we have evaluated the performance of DroidSD. We have implemented our approach in real time environment to evaluate it's accuracy. To get the value of false negative and to make sure, if our APK code clone detection method is better enough, we supposed to count how many clones were not identified by DroidSD. So, we count the Recall and Precision. A perfect code similarity detection approach supposed to have Recall and Precision values both 100%. Recall is actually the fraction of all related files, which have been retrieved through a query. High Recall means that most of the clones in that application have been found. In our approach, relevant clone files are those files which are supposed to be retrieved as cloned files, and retrieved files are those which were detected and retrieved as clones. In Precision, the relevant files retrieved by the query and it measures that how many irrelevant files were retrieved as a clone. High precision means that candidate clones are mostly actual and real clones. However, we could not able to find any benchmark for android applications. Neither we could evaluate all the retrieved results. Because there were million clone results, which were retrieved from the source code of 30,500 million applications as shown in Table 2. So, we used an unbiased way to evaluate DroidSD through two ways. The first way by mutation framework and the second way BigCloneBench[6].

Mutation Framework: The first way to evaluate DroidSD is creation a data-set by manual inserting different type of clone fragments from BigCloneBench in the source code files. We create our own data-set of 100 android applications by inserting test code fragments of type-1, type-2 and type-3 in these applications. We inserted 30 type-1 clone fragments in first 30 applications, 30 type-2 clone fragments in next 30 applications and 40 type-3 clone fragments in last 40 applications. The data-set repository formed and ready for DroidSD evaluation. Then we extract features and build index of this data-set repository by using DroidSD. We take these inserted code fragments of BigCloneBench as a subject system and detect code clones in data-set repository. DroidSD detects clones at different threshold values to differentiate different types of clones. All retrieved clone fragments from data-set further manually evaluated by comparing them with the subject code fragments those inserted in data-set. Fig 11 shows the clone detection results from data-set at different threshold values. As we minimize the threshold value, we able to detect type-3 clones. The graph in Fig 11 shows that DroidSD detects almost all inserted code fragments of type-1 and type-2 clones. But there is a small difference in detection of near-miss clones because of threshold value, if we minimize that value, we could detect almost all inserted fragments.

BigCloneBench: BigCloneBench is a benchmark of manually collected and evaluated clone pairs in IJaDataset[7]. IJaDataset consists of almost 25,000 open source Java projects, 3 million files and 250 MLOC. BigCloneBench was built on the basis of IJaDataset and consists of almost 8 million validated clone pairs.

We executed DroidSD for IJaDataset and evaluated Recall with BigCloneBench

[6]https://github.com/clonebench/BigCloneBench

[7]https://jeffsvajlenko.weebly.com/bigcloneeval.html

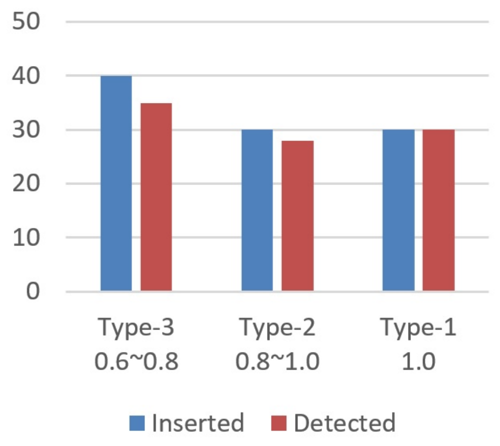
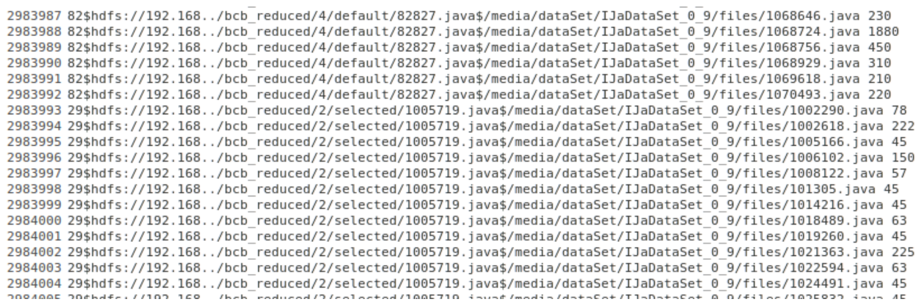


Fig. 11: Manually evaluated results of data-set



```
2983987 82$hdfs://192.168../bcb_reduced/4/default/82827.java$/media/dataSet/IJaDataSet_0_9/files/1068646.java 230
2983988 82$hdfs://192.168../bcb_reduced/4/default/82827.java$/media/dataSet/IJaDataSet_0_9/files/1068724.java 1880
2983989 82$hdfs://192.168../bcb_reduced/4/default/82827.java$/media/dataSet/IJaDataSet_0_9/files/1068756.java 450
2983990 82$hdfs://192.168../bcb_reduced/4/default/82827.java$/media/dataSet/IJaDataSet_0_9/files/1068929.java 310
2983991 82$hdfs://192.168../bcb_reduced/4/default/82827.java$/media/dataSet/IJaDataSet_0_9/files/1069618.java 210
2983992 82$hdfs://192.168../bcb_reduced/4/default/82827.java$/media/dataSet/IJaDataSet_0_9/files/1070493.java 220
2983993 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1002290.java 78
2983994 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1002618.java 222
2983995 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1005166.java 45
2983996 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1006102.java 150
2983997 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1008122.java 57
2983998 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/101305.java 45
2983999 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1014216.java 45
2984000 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1018489.java 63
2984001 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1019260.java 45
2984002 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1021363.java 225
2984003 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1022594.java 63
2984004 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1024491.java 45
2984005 29$hdfs://192.168../bcb_reduced/2/selected/1005719.java$/media/dataSet/IJaDataSet_0_9/files/1025832.java 45
```

Fig. 12: BigCloneBench results from IJaDataSet (HDFS file)

(51,499 files, 10 MLOC). The results measured BigCloneBench are summarized per clone type in Table 2 and the HDFS result file have been shown in Fig 12. DroidSD has perfectly detected type-1 and type-2 clones in BigCloneBench. The VST3 clones are also have excellent detection rate as shown in Table 4. To detect near-miss code fragments of type-3 was not possible in previous Android similarity detection techniques. The weak type-3 clones which can be type-4 clones were not evaluated in this paper, So we consider it as a 0 Recall and Precision.

As for Recall, there exist a high quality benchmark, but to count Precision, it was still an open problem. So, we used manually evaluation method to count Precision for DroidSD. We randomly selected 200 files from the results and for fair evaluation, we divide them on three judges, who were having the knowledge of source code similarity measures. After combining their results, we found that DroidSD has high Precision (97%) for type-1 clones, (91%) for type-2, (83%) for very strong type-3 clones.

Table 4: Recall and Precision (BigCloneBench)

Clone Types	Type-1	Type-2	VST3	ST3	MT3	WT3
Recall (%)	98	93	87	63	30	0
Precision (%)	97	91	83	49	17	0

During evaluation process, we have performed full application similarity detection and partial application similarity detection. Using full application similarity detection, we detected all clone files from dataset repository against our all subject systems as the results are shown in Table 2. It has been detected that two same applications with different versions have much more common features than different application. In the case of partial similarity detection, our approach has successfully found the applications which share some source files or part of their source code. our approach also retrieves and count all features in the subject file and the clone file, if the total number of features in both files are same, it's also the symbol of 100% similarity in files. Both full and partial similarity detection require finding similar code fragments in application's source code.

For further evaluation and to check the accuracy, we perform manual analysis of clone fragments. We manually evaluated almost 200 code pairs from Chrome (subject system), which were reported to be cloned. To manually evaluate every clone fragment, we opened both files side by side and verify if they were similar. Manually evaluation on clone fragments performed for type-1, type-2 and type-3 code clones. Meanwhile, Recall and Precision counted against every clone type as shown in table 5.

Table 5: Recall and Precision Measurement (Chrome apk)

Clone Types	Type-1	Type-2	VST3	ST3	MT3
Evaluation Files	20	20	20	20	20
Recall (%)	93	86	75	54	23
Evaluation Files	20	20	20	20	20
Precision (%)	96	89	81	32	17

Scalability: The scalability of DroidSD was evaluated by using different inputs with varying the size of LOC, at different level of granularities. In default execution time scales with the size of input (LOC). As our approach is based on Hadoop, HBase and MapReduce, so the scalability is not an issue. DroidSD can handle million of files and billion of LOC very effectively. IJaDataset and BigCloneBench are considered two big data-sets, which we used for the evaluation of DroidSD. For much large scale systems, our technique can be extended to multiple clusters which can handle any size of input.

The results in Table 2, 3, 4 and 5 proves the performance ability of our approach. The results were retrieved and calculated from 30,500 APK's source code repositories and IJaDataset[8]. We know that type-1 is exact clones, type-2 are renamed and type-3 (near-miss) are modified clones. Near-miss clones were a little bit difficult to detect but our approach has detected them with high accuracy.

The main mechanism behind the evaluation of results is the formation of chunks and then the comparison of hash values of these chunks. The results were evaluated through two unbiased ways, manually and be using the code cloning data-set. In manually, we create our own data-set of 100 android applications by inserting test code fragments from BigCloneBench. The retrieved results were divided on three different judges, who has the knowledge of code clones. During evaluation process they opened cloned file and the test file on both side to match the code against its clone type. In case of with data-set evaluation, we executed DroidSD for IJaDataset and evaluated Recall with BigCloneBench (benchmark of manually collected and evaluated clone pairs). So, the results are very satisfactory in the sense of similarity detection between different applications.

[8]https://jeffsvajlenko.weebly.com/bigcloneeval.html

8. Comparisons with Existing Approaches

Most of the application's clone and similarity detection approaches are simple hashing, semantic feature based, PDG based, API method based and UI based and even they were unable to detect near-miss clone fragments from apk source code. We considered a few detection approaches to compare with DroidSD, i.e. PDG based approaches are not scalable and can not be implemented on clone detection on a market with over million applications. Kim's technique (API method) executes applications and then collect all API call sequences as birthmarks. API method can not work when apps are encrypted by Ijimi because API method would not be able to get exact API traces of the application. Soh (UI method) uses the user interface information to check the similarity. If some fake activities or code fragments inserted in apps, which does not influence or make any change in user interface, UI method can not detect these kind of similar code fragments. DroidSD have some benefits over Wukong approach also.

WuKong [41] detects code clone by using feature matrices for each app by building n*m Characteristic Matrix, which is an abstraction of code segments. DroidSD detects code clones and similarity of android applications at source code level by extracting it's main features. DroidSD have few advantages over Wukong

- If the classes, methods and API are encrypted or obfuscated, the Wukong approach can not detect these kinds of code clones but DroidSD can do.
- In case of an application contains many small applications embedded inside, Wukong would not be able to detect this as an application clone. But in case of DroidSD it's quite simple, because we consider every decompiled file of an application as a subject file for clone detection.
- As long as it concerns to usage and implementation, Wukong technique is not ready to be implemented as an app clone detection platform for an enterprise level. But DroidSD approach is extensive and scalable to be implemented on enterprise level, and it can be adopted by mobile app stores.

9. Limitations

DroidSD can detect code clones in android applications very effectively but it's still hard to differentiate between the clone code and the original code. DroidSD can detect type-1,2,3 code clones with high accuracy rate but it's very hard to detect type-4 clones, because it's a token based clone detection approach and up to our knowledge there is no any token based approach which can detect type-4 clones but we considered it as a probability that the clones of less than 50% similarity can be type-4 clones, but not surely evaluated and considered in this paper.

10. Conclusion and Future Work

In this paper, we have proposed DroidSD a novel approach for clone detection in android applications at source code level. Reverse engineering has been used for decompiling the source code of android applications. Several tasks have been performed to preprocess the source code i.e. cleaning, transforming, normalizing and tokenizing etc. Index of each source code file was built by extracting their main selected features. In total, index of 982 million LOC source code was built just less than 3 days. DroidSD can detect type-1, type-2 and type-3 code clones from any kind of android application, which helps us to check the fake, vulnerable and malware application. Our clone detection approach is incrementable and can be extended to distributed clone detection approach, where we can build an index of a large amount of android applications and detect code clones from apps on large scale. DroidSD can be used by android app stores to check the accuracy, validity and repackage similarity in every application before upload into app stores. For future concerns, we are planning to develop an algorithm, which can help us to detect type-4 clones effectively, and meanwhile detect the malicious or vulnerable code segments from android applications to overcome the security threat.

Acknowledgement

This research done in Key State Laboratory of Information Security, School of Software Engineering, Tsinghua University Beijing, China and it was supported by the Nationnal Natural Science foundation of China under Grant No: 61540020.

REFERENCES

1. K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on.* IEEE, 2016, pp. 468–471.
2. J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, “Detecting android malware using clone detection,” *Journal of Computer Science and Technology*, Vol. 30, no. 5, 2015, pp. 942–956.
3. C. Amrutkar, P. Traynor, and P. C. Van Oorschot, “An empirical evaluation of security indicators in mobile web browsers,” *IEEE Transactions on Mobile Computing*, Vol. 14, no. 5, 2015, pp. 889–903.
4. J. Crussell, C. Gibler, and H. Chen, “Andarwin: Scalable detection of android application clones based on semantics,” *IEEE Transactions on Mobile Computing*, Vol. 14, no. 10, 2015, pp. 2007–2019.
5. Y. Y. Ng, H. Zhou, Z. Ji, H. Luo, and Y. Dong, “Which android app store can be trusted in china?” in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual.* IEEE, 2014, pp. 509–518.
6. M. Mondal, C. K. Roy, and K. A. Schneider, “Identifying code clones having high possibilities of containing bugs,” in *Proceedings of the 25th International Conference on Program Comprehension.* IEEE Press, 2017, pp. 99–109.
7. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?” in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.* IEEE, 2009, pp. 485–495.
8. C. K. Roy and J. R. Cordy, “An empirical study of function clones in open source software,” in *Reverse Engineering, 2008. WCRE’08. 15th Working Conference on.* IEEE, 2008, pp. 81–90.
9. C. K. Roy and J. R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on.* IEEE, 2008, pp. 172–181.

10. A. Sheneamer and J. Kalita, “A survey of software clone detection techniques,” *International Journal of Computer Applications*, 2016, pp. 0975–8887.
11. K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 175–186.
12. H. Niu, T. Yang, and S. Niu, “Clone analysis and detection in android applications,” in *Systems and Informatics (ICSAI), 2016 3rd International Conference on*. IEEE, 2016, pp. 520–525.
13. C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, “Detecting clones in android applications through analyzing user interfaces,” in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 163–173.
14. Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.
15. S. Alam, R. Riley, I. Sogukpinar, and N. Carkaci, “Droidclone: Detecting android malware variants by exposing code clones,” in *Digital Information and Communication Technology and its Applications (DICTAP), 2016 Sixth International Conference on*. IEEE, 2016, pp. 79–84.
16. J. R. Cordy and C. K. Roy, “The nicad clone detector,” in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 219–220.
17. Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, “Transferring code-clone detection and analysis to practice,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2017, pp. 53–62.
18. S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: local algorithms for document fingerprinting,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 76–85.
19. Q. Li, S. Wang, H. Mao, Q. Han, and X. Niu, “An adaptive improved winnow algorithm,” in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, Vol. 3. IEEE, 2015, pp. 303–306.
20. D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, Vol. 55, no. 7, 2013, pp. 1165–1199.
21. P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, “Ccaligner: a token based large-gap clone detector,” in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1066–1077.
22. C. K. Roy and J. R. Cordy, “Benchmarks for software clone detection: A ten-year retrospective,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 26–37.
23. M. A. Nishi and K. Damevski, “Scalable code clone detection and search based on adaptive prefix filtering,” *Journal of Systems and Software*, Vol. 137, 2018, pp. 130–142.
24. A.-F. Mubarak-Ali, S. Sulaiman, S. M. Syed-Mohamad, and Z. Xing, “Code clone detection and analysis in open source applications,” *Computer Systems and Software Engineering: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2018, pp. 1112–1127.
25. A. Ghosh and Y. Lee, “An empirical study of a hybrid code clone detection approach on java byte code,” *GSTF Journal on Computing (JoC)*, Vol. 5, no. 2, 2018.

26. A. Gupta and B. Suri, "A survey on code clone, its behavior and applications," *Networking Communication and Data Knowledge Engineering*. Springer, 2018, pp. 27–39.
27. E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective-a workbench for clone detection research," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 603–606.
28. B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–9.
29. H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, Vol. 27, no. 6, 2015, pp. 402–429.
30. R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 309–318.
31. X. Cheng, H. Zhong, Y. Chen, Z. Hu, and J. Zhao, "Rule-directed code clone synchronization," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.
32. C. Kapser and M. W. Godfrey, "Improved tool support for the investigation of duplication in software," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 305–314.
33. W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 2012, pp. 317–326.
34. R. Dhaya and M. Poongodi, "Detecting software vulnerabilities in android using static analysis," in *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*. IEEE, 2014, pp. 915–918.
35. S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 62–81.
36. S. Kim, E. Kim, and J. Choi, "A method for detecting illegally copied apk files on the network," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*. ACM, 2012, pp. 253–256.
37. J. Svajlenko and C. K. Roy, "Cloneworks: a fast and flexible large-scale near-miss clone detection tool," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 177–179.
38. H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.
39. V. Saini, H. Sajnani, J. Kim, and C. Lopes, "Sourcerercc and sourcerercc-i: tools to detect clones in batch mode and during software development," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 597–600.
40. S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *computers & security*, Vol. 65, 2017, pp. 230–246.

41. H. Wang, Y. Guo, Z. Ma, and X. Chen, “Wukong: A scalable and accurate two-phase approach to android app clone detection,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 71–82.

Junaid Akram is a member of IEEE Computer Society. He received his 1st Master degree in major of ”Information Technology” from Pakistan in year 2009, and 2nd Mater degree in major of ”Communication and Information System” from China in year 2015. Recently he is PhD scholar in Department of Software Engineering at Tsinghua University China. His current research includes: software reliability and testing, system security, vulnerability detection and code cloning detection.

Zhendong Shi received the BS degree in software engineering from Yunnan University China, in year 2015. Currently studding a master program in Tsinghua university China of major entitled ”Software Engineering”. Research interests include big data, software verification, vulnerability analysis and parallel computing.

Majid Mumtaz is a faculty member of COMSATS Institute of Information Technology Pakistan. He received Bachelor degree in Computer Science, MS in Information Security from Royal Institute of Technology, Sweden. He is doing PhD in Tsinghua University Beijing China. His primary research interests are Information and communication systems security, cryptography, algebraic cryptanalysis and mobile application security.

Luo Ping is a faculty member in Tsinghua University, China. He received PhD degree in Applied Mathematics from Chinese Academy of Sciences Institute of Systems Science in year in 1996. He joined department of Computer Science and Technology, Tsinghua University in year 2008. Later on he started offering his duties as Professor in Key Laboratory of Information System Security, School of Software, Tsinghua University, Beijing, 100084, China His area of research is Information security, Including: cryptography, vulnerability analysis and attack, database vulnerabilities and security.