國立政治大學資訊管理學系

碩士學位論文

指導教授:郁方博士

Gnafuy:基於行動裝置下的分散式運算研究

Gnafuy: a framework for ubiquitous mobile computation

研究生：陳晉杰

中華民國一零五年六月

摘要

隨著科技日新月異的發展，智慧型手機本身通訊與運算能力也隨著軟體和硬體的改善而不斷地增強，其便利性與高機動性的特色使得越來越多人持有智慧型手機，最後成為人們生活中不可或缺的部份。總觀來說，持有與使用率的上升，不知不覺的形成一種共享經濟與無所不在的行動運算網絡。

基於普及性與相對優秀的運算效能，我們設計與實作出 Gnafuy，一個基於行動裝置下的分散式運算框架，希望借用世界上所有閒置行動運算裝置的資源來實行無所不在的運算。

我們發展出一套應用程式介面(API)供開發者依照自己的需求來撰寫自己的分散式運算程式，藉由遵循 Gnafuy 所制定的應用程式介面，開發者可只專注在演算法本身的開發，而不需要在意其演算法如何被分配到手機上以及待處理資料的分配情形。本篇文章還討論了 Gnafuy 所採用的分散式運算的程式模型，以及我們如何藉由一個手機應用程式將任務部署至自願者的智慧型手機中，我們發展出一套伺服器端的機制來增加訊息傳遞的成功率，以及偵測計算後回傳結果是否正確，排除被惡意程式污染的客戶端結果。

除了分散式運算框架與基本運算之外，本篇文章還介紹了如何使用 Gnafuy 取得人類使用者的輸入來解決 Google reCAPTCHA 所提出的圖像辨識問題。

關鍵詞：行動運算、雲端運算、分散式運算、群眾外包

ABSTRACT

Along with the clipping evolution of technology, the bud of smartphone has germinated and sprang up in this decade. The increasing popularity and the improving computation and communication power of smart mobile devices facilitates shared economics of ubiquitous mobile computation.

We present the design and implementation of Gnafuy, a framework utilizing crowd-smartphones to fulfill ubiquitous distributed computation, offering a novel crowd-based computation service platform and programmable APIs for developers to take leverage of the spare capacity of smartphones among the world.

We discuss programming models of parallel computation, detail how tasks can be deployed on massive smartphones via mobile applications, and propose a server side mechanism to increase the probability of successful delivery and detect the corrupt results from the viciously slaves.

The demonstration on google scholar paper reference construction solicits smartphone users to solve the image recognition challenged by reCAPTCHA, showing the unique advantage of leveraging crowdsourcing ability to distributed mobile computation.

Keywords: Mobile computing, Cloud computing, Distributed computing, Crowdsourcing

Contents

1 Introduction

Batch processing systems for huge volume of data have become an indispensable part of modern business and scientific research [1, 2]. To provide promising capability to handle massive datasets, these systems often featured with scale-out architecture that makes it easy to scale both capacity and performance beyond the physical limitations [3]. Frameworks and corresponding APIs they provided is dedicated to lowering the threshold for developers to launch clusters for distributed computation and design their own applications [4, 5].

On the other hand, the significant progress of hardware and software improvement on mobile devices in recent generations allow developers to design delicate applications from cloud services to smartphone platforms [6, 7]. The derived application could be classified into three categories [8]. In the early stage, due to the capability of smartphones, one application category falls in the extension of the existing cloud service, such as Gmail, Facebook and YouTube. To amplify the capability of mobile devices, another application category falls in applications with complicated computation but submitting resource-consumption tasks to cloud service providers to exceed the computation power of mobile device [9, 10]. As mobile device technology evolves, the last category of mobile applications is to consider the mobile device as part of the cloud service provider which can be an individual computation node to solve problems without exceeding the limitation [11, 12]. Our work facilitates the last category mobile applications by proposing a general purpose framework *Gnafuy*) that adopts common distributed APIs for developers to develop their applications and distribute their computations to mobile devices.

We aim to provide programmable interfaces for developers to design distributed computing applications on mobile devices and take the efforts on porting their applications into mobile devices. The ability to port applications to mobile devices provides developers the advantage on developing crowd-sourcing applications. Problems like Chinese word segmentation and image recognition are easy to solve for humans, but are relatively difficult for computers without large inputs. These features are commonly used to distinguish humans and programs in web services such as google to prevent systematic queries or crawling. With Gnafuy, developers may take advantage of crowd resources to develop applications that cross the barriers. In our demonstration, by importing web browser automation tool such as Selenium [13] to Gnafuy, we develop an application that ports to smartphone users to solve the image recognition challenged by their web browsers. Revealing the problem to be solved on their screen with the resultant of crowd-sourcing, we are able to bypass google checking.

For encapsulating developers' algorithm and porting applications to mobile devices, we adopt the reflection mechanism in the contemporary programming languages that offers the capability to load library and invoke instance dynamically in the run-time. The approach reduces the burden of learning a new framework and retains the effort for developers to reinvent the wheel.

Gnafuy enables mobile volunteers to be part of computation resources in the framework without additional certificate. It is hence essential to ensure correctness of computation results from end devices. We further apply a fault correction mechanism based on Condorcet's jury theorem [14, 15] to improve the correctness of collected computation results from the end users.

2 Related Work

We summarize previous works on distributed framework design in mobile computing, crowd-sourcing applications and resource correction techniques in this section.

On the topic of distributed framework design, the famous MapReduce [16] and Hadoop [1] both provide high level programming interface to simplify the complex problems by dividing them into several map/reduce block and keep developers from miscellaneous details of parallel and distributed computation like job scheduling or compute node management. Apache Hive [4] and Apache Phoenix [17] both offer a mechanism to transform the SQL-like (HiveQL in Hive and standard SQL query in Phoenix) statement into Hadoop map/reduce jobs. This approach facilitates the development of some data analysis in parallel. Compared to MapReduce, Spark [2] features in-memory computing which has better performance than Hadoop MapReduce. Moreover, Spark's resilient distributed datasets (RDD) [3] offers a concise view for developers to manipulate huge data sets in parallel unconsciously since it imports functional programming model as operations on RDD. The developers who familiar with Scala [18] would learn Spark without much pain since it's almost the same thing on manipulating both RDDs and data structures in Scala. Dryad [19] is another distributed computing platform that aims for providing a flexible programming interface. It allows developers to compose complex computation instead of the procedure of a map function followed by a reduce function in Hadoop. Meanwhile, DryadLINQ [5] is the programming model provided by Dryad which offers developers to write parallel applications in SQL-like style. The developers could easily catch up DryadLINQ if they were familiar with .NET Language Integrated Query. These frameworks accepts functional programming model to a certain extent which offers concise expressive style and exempt developers from suffering concurrency problems.

Our work is also related to the mobile computation field especially on considering smartphones as cloud service providers [20]. BOINC [21] is the pioneer of volunteer computing which uses the idle time of computers to perform scientific researches. It provides tools to build application for several platforms (Windows, Mac OS X, Linux). Additionally, the support on Android platform extends the ability of volunteer computing to mobile phones. The other distinguished projects using the BOINC platform are SETI@home [22] and Rosetta@home [23]. The Ibis framework [12] also provides a room for integrating Android devices into the grid to intensify the overall computing power. Hyrax [11] reveals the capability in porting Hadoop to the Android platform which offers reasonable performance in data sharing. Serendipity [24] advocates *PNP-block* as the basic component for job execution. The implementation of Serendipity uses Java reflection to execute tasks dynamically and achieve the ad hoc communication among devices by using *WifiManager's* hidden API. The methodologies of those works inspire Gnafuy on porting tasks to mobile devices.

Finally, it is possible that the involved computing nodes are ineffective, e.g., randomly replying results, or are contaminated, e.g., compromised by malware. In this case, the results from these end devices may not be trustful. To address this issue, various fault correction mechanisms have been proposed [15] to reduce the impact of problematic results from untrustful devices.

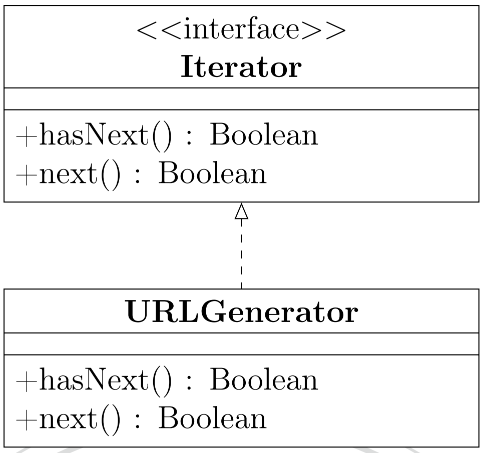
Some studies that propose reputation systems to determine the reduce the influence of unreliable agents [25, 26, 27]. Anis et al. [15] propose a learning automata approach that is able to distinguish between reliable and unreliable sensors without a knowledge of the ground truth. We adopt the similar idea for its applicability to uncertain mobile scenarios when performing fault correction.

3 Overview

This section outlines the operation mode of three Gnaguy framework components, *Job-Builder API, computing nodes and control center*, by demonstrating the classic word counting example. We perform a survey of software skill requirements from the employment agency website. Through splitting several job hiring requirements into single words and counting the occurrence of each word, a term-frequency table for each requirement comes into being, and then further merge all the respective table into one integrated table. Finally, the outcome represents the approximate popularity of the mentioned skills. For example, we can infer that Java is twice the popularity of Python in this period of hiring for the same reason that Java (295) is more than double the occurrence of Python (116).

To achieve the survey of the Gnafuy framework, we divide the algorithm into three parts for adapting the Gnafuy-provided APIs. Firstly, developers have to create an iterator which generates URLs to represent the initial data. The figure 1 demonstrates the interface of iterator and its implementation named *URLGenerator* in this case. Since the initial data set can be too large to fit into the memory, the iterator splits the whole set into relatively small parts as the abstraction of a memory-saving model. In general, there are two ways to implement the iterator. One is to put all the targeting URLs into a list

Figure 1: Iterator interface



data structure without considering the memory issue, and then convert the list to fit the form of iterator. The other is to hold a cursor to mark up the last visited link and identify another link following the location of current cursor when the *next()* method is called. In spite of occupying more memory, the first way performs better than the second when launching the initial data since the data already exists. However, the second way is more appropriate to adopt when the developer would like to generate initial data by reading a large text file line by line for saving memory.

Secondly, in addition to perform the algorithm of transforming contents of hiring requirements into separate term-frequency tables as stated in algorithm 1, developer also has to select an appropriate operation from the given API. In this case, *Mapper* interface is a proper target to implement since the basic form of the processing algorithm is $K \rightarrow V$ which means to transfer the type of one element into another.

Thirdly, in case that all the URLs have been converted into term-frequency tables which stand for the condensed information for each requirement, the developer has to design an algorithm resembling the algorithm refalg:wcr for the purpose of merging schismatic tables into a unified one. Meanwhile, we have to reiterate that *Reducer* interface is a perfect selection for this merging task due to the nature form of merging algorithm, $(K, K) \rightarrow K$, which means the two given elements would merge into one with the same data type through processing.

It is notable that both algorithm 1 and 2 focus on dealing with the minimum amount

Algorithm 1 Word Count Mapper

```
Input: The targeting URL with hiring requirement
Output: Term frequency table
 1: $content \leftarrow visitAndDownload(url)$
 2: $termFreqTable \leftarrow \phi$
 3: for each $word \in content$ do
 4:     if $\exists word \in getKeys(termFreqTable)$ then
 5:         $current \leftarrow termFreqTable[word]$
 6:         $termFreqTable[word] \leftarrow current + 1$
 7:     else
 8:         $termFreqTable[word] \leftarrow 1$
 9:     end if
10: end for
11: return $termFreqTable$
```

of elements since these algorithms are designed to deploy to smartphones as computing nodes for parallel computing. Therefore, the last thing is to combine the two algorithms

Algorithm 2 Word Count Reducer

```
Input: Two term frequency tables
Output: Term frequency table
 1: $termFreqTable \leftarrow \phi$
 2: $tables \leftarrow table1 \cup table2$
 3: for each $(key, count) \in tables$ do
 4:     if $\exists key \in getKeys(termFreqTable)$ then
 5:         $current \leftarrow termFreqTable[key]$
 6:         $termFreqTable[key] \leftarrow current + count$
 7:     else
 8:         $termFreqTable[word] \leftarrow count$
 9:     end if
10: end for
11: return $termFreqTable$
```

and specify the control flow. Listing 1 reveals how the control flow determined by provided API. *JobBuilder* possesses the ability to communicate with the control center, asking it to establish queues as a temporary data storage, organize the tasks, submit the encoded jar file, and activate the current job. As for queues, *QueueRef* is merely an abstract data storage in our framework, working as a specification to notify the control center to establish proper data container. In the flow, we create two queue references, named "urls" and "tables", to stand for data containers of initial links and term-frequency tables tasks respectively. With specifications of data containers and processing algorithms, developers could chain tasks by specifying the data flow.

The code segment *appendTask(urls, WordCountMapper.class, counts)* demonstrates the data flow that invokes an instance of WordCountMapper on every smartphone, ask them to process the data from *urls*, and then place the result into *counts* until all the elements of *urls* are exhausted. A more complex control flow might be constructed while the appendTask method is followed by appendTask many times.

Listing 1: Job Initialization

```
public class JobInitialization {
  public static void main(String[] args) throws Exception {
    Iterator<URLData> iter = new Generator();
    QueueRef urls = new QueueRef("urls");
    QueueRef tables = new QueueRef("tables");
    List<QueueRef> out = new ArrayList<>();
    outcome.add(tables);
    JobBuilder.createInstance("name.prefix","path/to/file.jar")
      .initialData(iter, urls)
      .appendTask(urls, WordCountMapper.class, out)
      .appendTask(tables, WordCountReducer.class, out).build();
  }
}
```

After all the configuration are settled down, the job would be submitted to designated host by calling *build()* method. Technically speaking, instead of performing a server to manage the computing nodes, JobBuilder would only validates the correctness of the task chain by checking each data types, encapsulates the information into a request, and then sends it to the designated host for constructing the required components when *build()* method being called. As long as the job is submitted, the control center will start deploying tasks to smartphones, collect and organize data till all the tasks end processing.

4 Programming Model

As a distributed computing framework, Gnafuy's APIs offer developers to specify the control flow of their application. Each job contains an iterator as the abstraction of the initial data set and a series of parallel tasks to compose the control flow which is known as embarrassingly parallel. Since the initial data set may not fit into the memory of developer's local device, we propose the iterator interface as the memory-saving model for developers to launch their application. Each task encapsulates an algorithm with a specific pattern which would deploy to smartphones and manages the data flow by defining the queue where it would take data from and the queues it would put data in. The type of task determines the programming pattern which could be considered as a function to be applied to each element from a queue. By definition, the general form of a function is a relation between a set of inputs and a set of outputs. There are some variant functions which derived from functional programming model were predefined in Gnafuy such as map, flatMap, foreach and reduce. The predefined functions give a more precise definition, especially for the data type of both inputs and outputs. For example, map function takes one element with type A from a queue and converts it to instance of type B; reduce function takes two elements with the same type and merge them together. The table 1 refers a brief view of these functions. In Gnafuy, all the input/output types

Table 1: Programming patterns

Function Name	Transformation		
map	(A)	$\rightarrow$	(B)
flatMap	(A)	$\rightarrow$	list(B)
reduce	(A, A)	$\rightarrow$	A
foreach	(A)	$\rightarrow$	$\emptyset$

have to implement an interface to make sure all types of data would be able to convert from string to object and vice versa. This limitation guarantees that we could perform the following action easily and correctly: 1. Save data into persistence layer (object to string) 2. Deliver message through network (object to string) 3. Retrieve object from a string (string to string) The ability of bidirectional converting is also known as serialization and deserialization.

Listing 2 illustrates the contract of mapper. The input K and output V have to follow the interface *ImmutableResource* to make sure the ability for serialization. The abstract method *map* is a template method for developer to implement their task to produce a element with type V by giving a element with type K. *QueueConsumer* is the ancestor of all Ganfuy task types listed in table 1. It injects the *Activity* provided by Android API also wrap some method to create *Intent* for developers to launch other existing apps like browser or google map.

When activating a job, JobBuilder introspects the configuration by validating the task chain then creates a task table indicating the control flow by predefined tasks and asks control center to prepare the components accordingly. On receiving the permission from control center, it starts to roll the iterator and asks control center to put the generated input data into the queue which is specified to consume it by the first task. The listing 1 in the previous section reveals the way to build a classic control flow of map-reduce pattern. Once the job is submitted, the computing nodes (smartphones) would be able

Listing 2: Mapper Abstract Class

```
public abstract class Mapper<K extends ImmutableResource, V extends
    ↪ ImmutableResource> extends QueueConsumer{
    public Mapper(String fromQueueName, List<String> toQueueName) {
        super(fromQueueName, toQueueName);
    }
    public abstract V map(K from) throws Exception;
}
```

to receive tasks and process data individually and simultaneously. In the end, the job terminates until all tasks are finished. Besides, Gnafuy's API also formulates the format of HTTP POST body for every request and response from the computing node. This approach provides loose coupling and less dependency between computing nodes and control server. In other words, developers could implement their own control center by hosting a service and complying the format of requests and responses since the computing nodes communicate with control center via HTTP protocol.

5 Control Center

Through the minimum requirement is to host an web service for handling requests send from computing nodes properly, there are still some practical suggestions that we strongly recommend developers user to take care. Since the computing nodes communicate with control center via HTTP protocol, the control center have to provide a corresponding service for handling a tons of requests. Therefore, it's necessary to take the scalability into consideration. Although the contemporary smartphone is good enough for calculation, it is still not stable compared to the workstations or even laptop computers due to the limitation of hardware or unknown network problems. The mechanism of message delivery is another issue we have to take care since it's possible that an unstable computing node asks data for processing and never replies. We promote a *Task Manager* for controlling task status and handling message delivery. Besides, the queue implementation should follow the competing consumers pattern and guarantees that a message is delivered at least once. The Advanced Message Queuing Protocol (AMQP) essentially fit our requirement. Meanwhile, fault correction of processed results is another important issue to be addressed in this paper since not all computing nodes are trustful and could be contaminated by other malwares. The following sections describe these components in more detail.

5.1 Facade

The component Facade is basically a service for handling requests from the computing nodes. In order to achieve scalability and maintainability, the recommended implementation of Facade is to build a RESTful service. With the REST architecture, we separate requests from different states through URI and maintain several caches for improving performance. For example, the available task information will be retrieved on the *Task Required* stage several times, it's wise to perform a cache for the task status without querying against database. Moreover, for handling lots of requests that send by numerous computing nodes on different states, our service is implemented with actor model [28] provided by spray framework [29] which gives the potential to scale up and scale out easily.

5.2 Task Manager

There are three responsibilities that task manager concentrates on in our implementation, one is job status tracking, another is message delivery, and the other is queue implementation.

For the job status tracking, it's important for task manager to keep which task is available for the current job and the remaining amount of elements of the working queue. Since there would be a tons of computing nodes to inquiry the next task information for each of them, the information should be kept persistent in memory instead of performing query against database on every request. Furthermore, it's possible to introduce server load balance mechanism by making read-only copies of cache and spread them to different servers. Since the cache is just a read-only copy for the real status, the timing of refreshing cache is another issue we have to addressed. By design, a task should terminate when the corresponding queue is exhausted and it's the proper timing to update job status and notify all listening servers to refresh their cache. However, in the period of notifying servers, computing nodes may inquiry the server which does not update to the latest task information. This scenario would lead some performance loss because some computing nodes are directed to an empty queue and back to *Task Required* state then inquiry control center again till they find a proper task.

Since the target computing nodes are mobile devices, which imply they are fragile and unreliable due to the network, power limitation and could be interrupted by any phone call. One duty of task manager is perform a mechanism to make sure the message deliver to the device correctly. Task manager makes copies of data with a delivery tag and sends the copy to multiple devices when popping a data from queue. Meanwhile, task manager maintains a list of delivery tags. Once the data being processed come back to control center with a delivery tag, task manager will acknowledge the processed data and remove the delivery tag from the outgoing data list. The mechanism of acknowledgment with delivery tag should place into a synchronize block as the critical section to prevent the duplicated data issue. This mechanism increase the successful probability of message delivery and gives the potential for verifying data correctness.

The minimum requirement of queue implementation should follow the competing consumers pattern and guarantees that a message is delivered at least once. It's nice to have other properties like high availability and fault tolerance. Our implementation adapts RabbitMQ [30] as the data provider since it follows AMQP model and features with non-blocking operations.

5.3 Fault Correction

The mechanism of spreading duplication to different devices increases the successful probability of message delivery and give the potential for fault correction especially on checking the corrupt data and crowdsourcing correction. Condorcet's jury theorem provide a simple solution that the majority group performs better than any selection of superior individual under the assumption that the probability of making the right judgment is greater than 0.5 for every member in jury.

Instead of dropping out the redundant result with the same delivery tag, task manager could collect them into a small group and check by the result of each other. This alternative approach could apply to task rely on human judgment or sniff out the malicious computing node by sampling.

6 Computing Node

In order to take leverage of computation power on mobile phone among the world, Gnafuy framework offers an app for users to install. As soon as the app starts running, it enters the initial state and communicates with specific control center to ask for the next instruction. Once the control center receives the request from computing node, it would start to diagnose the condition of computing node and assign tasks to the app accordingly. This approach makes smartphones to contribute its computing power as a cloud service provider while running the Gnafuy app. The Gnafuy app would maintain a state machine to imply the computing node what to do next. There are 4 states inside the state machine:

- Library Required
- Task Required
- Data Required
- Stop

Specifically speaking, when the state machine turns to a new state, computing node would check the current condition, send a HTTP request to control center after it finishes preparing the prerequisite of the next state, and then wait for the instruction of the next step. The name of the state implies which resource computing nodes desire for, and the flow reveals in algorithm 3 and figure 2. Algorithm 3 states the mechanism we sealed in the computing node is composed by a global dictionary and an infinite loop. For every iteration, the computing node takes an activity from the dictionary according to current state then sends a request via the retrieved activity. To be brief, the term dictionary here is a key/value hash table structure. The key is the activity status code and the value is the activity instance. These activities basically abide by the same interface which has the ability to send a request and create the activity for next state according to the response. The exact flow of each activity would be: 1. Collect required information. 2. Send request to control center. 3. Create an activity then put into global dictionary. The iteration continues until control center changes the state to stop which is the only way to stop this infinite loop. Another thing that needs to be mentioned here is that algorithm 3 must be placed into *AsyncTask* or *Service* provided by Android sdk since Android would raise *NetworkOnMainThreadException* if main thread program tries to send request or performs network communication. Android apps run by default on the main thread, also called the UI thread which handles all the user input as well as the output so it designs to avoid time-consuming operations on the main thread to prevent UI freeze and to have a better user experience. Gnafuy itself relies on lots of network communication to retrieve

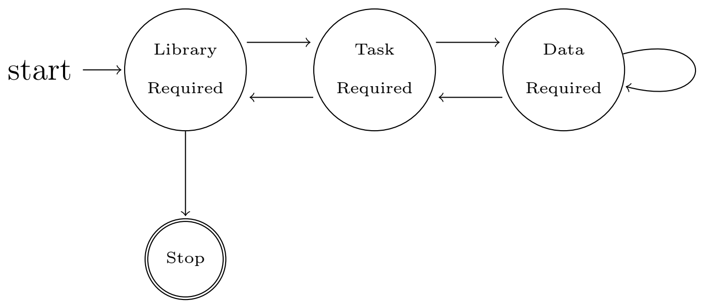


Figure 2: A brief view of state machine in the computing node

task or data to accomplish distributed computing so we create an *AsyncTask* then launch Ganfuy's state machine inside. Listing 1 demonstrates activity which requires library

Algorithm 3 State Machine of Ganfuy Computing Node

```
1: $activityTable \leftarrow anEmptyDictionary$
2: $currentState \leftarrow libraryRequired$
3: $activityTable[currentState] \leftarrow libraryRequiredActivity$
4: while $currentState \neq stop$ do
5:     $nextActivity \leftarrow activityTable[currentState]$
6:     $currentState \leftarrow nextActivity.execute()$
7: end while
```

information from control center. Each activity follows the contract of GnafuyStateActivity interface which implies the *execute* method must return an instance of the same interface for the next round.

Listing 3: class LibraryRequiredActivity

```
public class LibraryRequiredActivity implements GnafuyStateActivity {
    @Override
    public GnafuyClientState getGnafuyClientState() {
        return GnafuyClientState.LibraryRequired;
    }
    @Override
    public GnafuyStateActivity execute(){
        checkExistingLibs();
        LibarayMeta meta = electLibraryFromControlCenter();
        cleanUnnecessaryLibs();
        return GnafuyStateActivityFactory.createTask(meta);
    }
}
```

6.1 Job Loading

The goal we want to achieve here is to encapsulate developers' predefined library as a response and send back to mobile phone when the computing node required. With the library and *reflection mechanism* provided by the programming language of the platform, each computing node could invoke corresponding instances and execute methods accordingly. Though we could invoke arbitrary methods with proper hints, Gnafuy forces users to follow some programming pattern such as map and flatMap which is mentioned at the previous chapter in order to limit the domain of input/output types and keep the code structure clear.

As the computing node of Gnafuy framework, we aim to perform the implementation on mobile phones on the Android Platform. By narrowing down the target platform, the task deployment could be simplified as "How to load/reload classes at runtime in Java." since the Android SDK is written in Java-like language. Meanwhile, Java Reflection provides the following feature:

- Classes inspecting at runtime
- Instance invoking at runtime
- Dynamic Class Loading / Reloading

With the ability of dynamic class loading and instance invoking at runtime, we could load classes outside the JVM and invoke an instance without knowing its type at runtime. The following Java sample code reveals the idea of dynamic loading in Android.

Another issue we have to mention here is that Android runs on Dalvik VM or Android Runtime instead of JVM because the bytecode generated by Java is not compatible with Dalvik VM at all. Therefore, we have to convert developers' predefined Java bytecode to dex file before performing the runtime loading technique. Fortunately, Android-sdk provides some tools for converting Java bytecode to Dalvik bytecode that meets our requirement. Finally, for the purpose of loading some third party libraries from the bytecode imported by the developers, the developers have to make sure all the dependencies of pre-defined Java bytecode have been packaged into the .jar file correctly, which could be done by some project management tools like maven.

Though Gnafuy computing node's current implementation is based on Android, it's possible to take the same approach on other platforms which support reflection mechanism such as Swift, Objective-C and C#. Either it could be easier for task deployment if all the platforms run the same environment such as Node.js [31], and we could thus focusing on task deployment with only one programming language.

6.2 States of Computing Node

6.2.1 Library Required

In the stage of *Library Required*, computing node would create a candidate list of existing library files by looking into the job folder. The candidate list represents the remaining libraries downloaded by the computing node not long ago but got interrupted by some unknown reasons or just not clean yet. The computing node would ask control center to recommend at least one qualified library which contains all the definition of pending tasks from the list. If the control center could not elect a qualified library from the candidate list because they are all out of date, the control center will package one qualified library into the response for client to create. In addition to specifying at least one qualified library for applying dynamic loading technique in the next stage, control center have to suggest a list of file that computing node should eliminate in order to save the disk space since the job was no longer needed. In some cases, the state would become *Stop* since there is no job to run on the control center.

6.2.2 Task Required

Since a library contains a succession of tasks, computing node has to clarify which task to launch in this stage. The computing node would give the chosen name of library from the previous state to control center. With the *Task Required* request wrapped with library name, the control center will prepare the task name and the permitted queues for committing the result. With the task name and library, computing node could perform dynamic loading technique and invoke an instance for further computing. If all the tasks are done at some point, the control center would ask computing node to go back to the previous state for the other jobs. The state would go to *Data Required* if the client invoke the instance successfully.

6.2.3 Data Required

In this stage, computing node holds an instance as a computing unit and asks control center for data to process. The computing node would send a request with task information and the previous outcome if it exists. Once the control center gets the request from a computing node, it starts to prepare a data list for computing node to process. The amount of elements contained in the data list would be determined by task type in the task information. By definition, map, flatMap and foreach requires at least one element but reduce requires at least two. Meanwhile, if the request contains the previous outcome, the control center would push the outcome into the queue that specified by the task. The state will not change till the target queue running out of the data; otherwise the state would back to *Task Required* for invoking another instance of the task.

6.2.4 Stop

As the only accepting state, only a few conditions happens could arrive here. One is running out of the jobs to do, the other is the occurrence of unexpected error such like network error or device runs out of memory.

6.3 Permissions

For achieving the goal of porting tasks dynamically to clients, Gnafuy app requires lots of permission [32] from users. Gnafuy app requires permission of reading/writing file to SD card since it would create a temporary jar file in *Library Required* state for performing dynamic loading via reflection mechanism. Also we have to ask users for permission of accessing network state and permission of accessing power connected so we could detect that the device have Wi-Fi connected and the current charge state if users prefer to contribute their computing power when both Wi-Fi connected and battery is charging. In order to offer more information for developers, we also give permissions to access approximate/precise location and the string uniquely identifies the device (IMEI on GSM, MEID for CDMA). Table reveals the permissions that we acquire from users who install our app. By giving the permissions listed above, Ganfuy app is able to work correctly

Table 2: Android permissions that Gnafuy enabled

Android permissions	**Description**
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
READ_PHONE_STATE	Allows read only access to phone state.
ACCESS_NETWORK_STATE	Allows applications to access information about networks.
BATTERY_STATS	Allows an application to collect battery statistics.
INTERNET	Allows applications to open network sockets.
ACCESS_FINE_LOCATION	Allows an app to access precise location.
ACCESS_COARSE_LOCATION	Allows an app to access approximate location.

and shares computing power only when Wi-Fi connected and charging is charging. Also our developers would able to retrieve information from smartphones such as battery state, network state and precise location of computing nodes.

7 Experiment

7.1 Citation network

In this section we design an application to build a citation network of academic publishing. As the link of an initial paper from google scholar is entered, our application could find papers that had once cited the premier one by inspecting the *Cited by* hyperlinks on the google scholar page. Then, when the next round initiate, papers that had cited the original paper would become the next starting point thus performing the algorithm iteratively. Since the input is a paper and the output is a list of cited papers, the proper programming model of our fetching algorithm is described as *flatMap*, which takes one element of type K and produces a list of type T For the algorithm convergence, we only adopt the first n papers from the original paper. The value n is determined by the formula $n = \sqrt{c/r}$ where c stands for the amount of cited by papers and r stands the r round of current iteration. The algorithm stops to adopting papers on the condition when r is greater than n.

The next thing is to configure the control flow by Job Builder then activate build() method to send job with algorithm 4 to control center. Once control center received the job, it starts to federate smartphones to achieve this job. By far it's a simple application for experienced developers to implement as described below by mocking requests from the web browser to deceive the server. However, the response would redirect to the reCAPTCHA [33] page after some iterations which stops our robot fetching papers until our robot solve the problem. The problem would vary over time and the problem could be "Select all squares with street signs" or "Select all rivers and mountains". It's hard for us to figure out a general solution to answer the question like the example in figure 3 challenged by reCAPTCHA since it requires the knowledge of natural language processing to understand the question and image recognition to answer the question. Google offers reCAPTCHA as the protector of websites from spam and abuse with the slogan "Tough on bots, Easy on humans" which indeed keeps our application out.

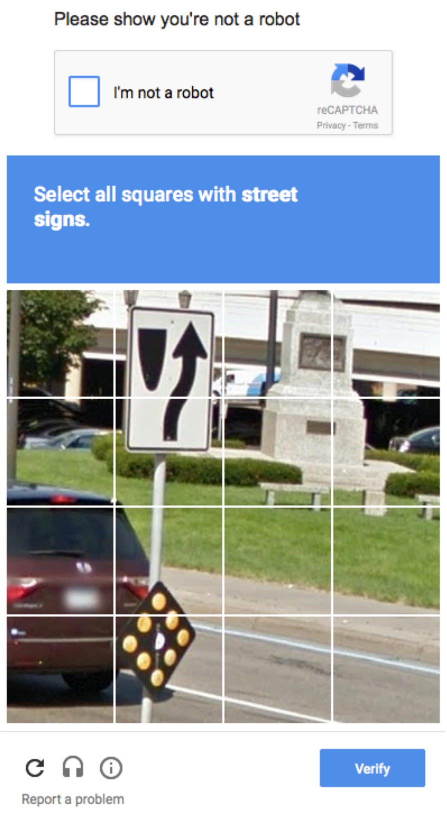

Please show you're not a robot
I'm not a robot
reCAPTCHA
Privacy - Terms
Select all squares with street signs.
Verify
Report a problem

Figure 3: reCAPTCHA example

Instead of figuring out a general solution for the puzzles, a more reasonable approach is to redesign our application by involving an user interface for retrieving user's suggestion on facing the reCAPTCHA problem. Though Ganfuy gains computation power by launching our app on mobile devices, the app itself does not display anything on the screen. Also we didn't provide API for developer to compose the user interface on the screen. It's seems not possible to achieve the goal of taking user input in this framework. Fortunately, Selenium, a well known utility for web browser automation, provides the capability to automates browsers and gives a slim chance of survival to solve the problem. By importing Selenium into the previous algorithm, we could manipulate the browsers on the mobile device as the user interface. Once our fetching robot get stuck, the algorithm could stop to wait for user's rescue to solve the puzzle. The function *waitForUserToSolveThePuzzle()* in algorithm 4 should be called on every time when we manipulate the browser.

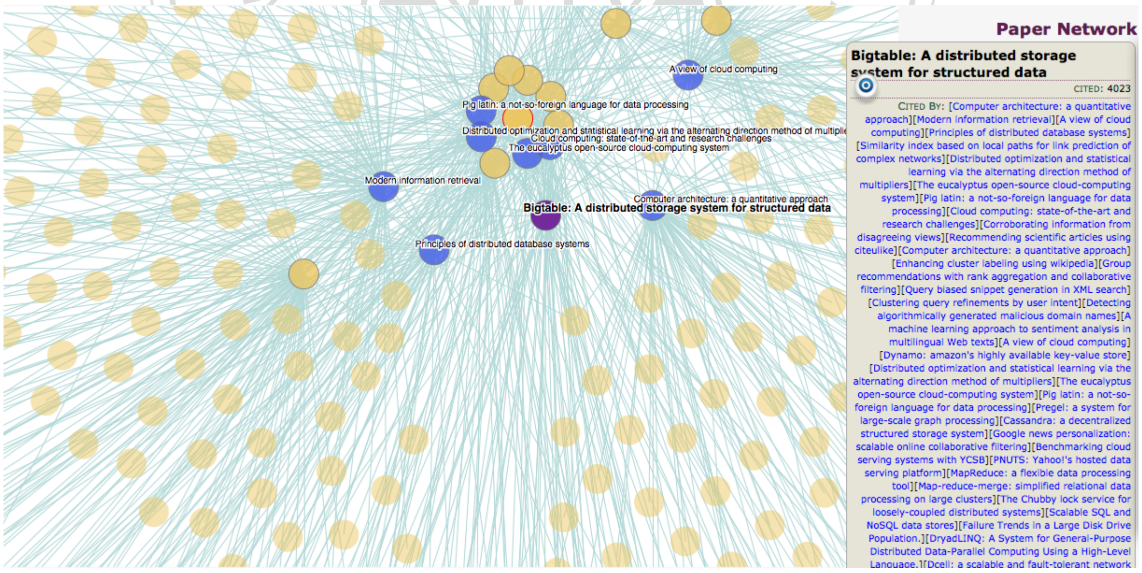
In this case, the *result* in algorithm 4 will send to the original queue where we get the link of an initial paper from and the initial link is updated with its descendants then send

Algorithm 4 Citation Network FlatMapper

Input: Link of an initial paper
Output: Papers that had once cited the initial paper

```
 1: $result \leftarrow \phi$
 2: $bound \leftarrow \sqrt{c/l}$
 3: $adoption \leftarrow 0$
 4: if $bound > l$ then
 5:     $browser \leftarrow openBrowser(initialLink)$
 6:     while $adoption < bound$ do
 7:         $waitForUserToSolveThePuzzle()$
 8:         $result \leftarrow result \cup fetchContent()$
 9:         $adoption \leftarrow adoption + 1$
10:         $browser.goToNextPage()$
11:     end while
12: end if
13: $initialLink.setChildren(result)$
14: $sendToQueue(initialLink, processed)$
15: return $result$
```

to the processed queue. The processed queue contains all the information we need for building a citation network since we have all papers with its descendants. By sorting out


A view of cloud computing
Pig latin: a not-so-foreign language for data processing
Distributed optimization and statistical learning via the alternating direction method of multiplie
Cloud computing: state-of-the-art and research challenges
The eucalyptus open-source cloud-computing system
Modern information retrieval
Computer architecture: a quantitative approach
Bigtable: A distributed storage system for structured data
Principles of distributed database systems

Paper Network

Bigtable: A distributed storage system for structured data

CITED: 4023

CITED BY: [Computer architecture: a quantitative approach][Modern information retrieval][A view of cloud computing][Principles of distributed database systems][Similarity index based on local paths for link prediction of complex networks][Distributed optimization and statistical learning via the alternating direction method of multipliers][The eucalyptus open-source cloud-computing system][Pig latin: a not-so-foreign language for data processing][Cloud computing: state-of-the-art and research challenges][Corroborating information from disagreeing views][Recommending scientific articles using citeulike][Computer architecture: a quantitative approach][Enhancing cluster labeling using wikipedia][Group recommendations with rank aggregation and collaborative filtering][Query biased snippet generation in XML search][Clustering query refinements by user intent][Detecting algorithmically generated malicious domain names][A machine learning approach to sentiment analysis in multilingual Web texts][A view of cloud computing][Dynamo: amazon's highly available key-value store][Distributed optimization and statistical learning via the alternating direction method of multipliers][The eucalyptus open-source cloud-computing system][Pig latin: a not-so-foreign language for data processing][Pregel: a system for large-scale graph processing][Cassandra: a decentralized structured storage system][Google news personalization: scalable online collaborative filtering][Benchmarking cloud serving systems with YCSB][PNUTS: Yahoo!'s hosted data serving platform][MapReduce: a flexible data processing tool][Map-reduce-merge: simplified relational data processing on large clusters][The Chubby lock service for loosely-coupled distributed systems][Scalable SQL and NoSQL data stores][Failure Trends in a Large Disk Drive Population.][DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.][Dcell: a scalable and fault-tolerant network

Figure 4: Citation Network

the outcome and utilizing some visualization tool, we build a graph as shown in figure 4 which provide a concise view for scholars to check the lineage of the paper they interest.

7.2 IP Location Finder

There are already some data provider such as IPInfoDB, IP2Location and MaxMind that offers API for users to find a geolocation of an IP address including latitude, longitude, city, region and country. All of them charges annual fees or buy-out payment for accessing their service since it's hard to collect this kind of data. With Gnafuy framework, we could easily retrieve the geolocation of IP addresses by only acquiring user's current location. In order not to send this task to the same device in a short period which would lead to many duplicate IP/geolocation pair in our result, Gnafuy API provides a hook for computing a unique key on each task which could prevent the redundant computation, this hook would be called when computing node asks for the next task. In this case, the implementation of this unique key could describe as the combination of device IP and geolocation. Once the computing node acquired the task and sends the result with unique key to control center, the control center would put the result into a queue as usual and persist the unique key into a bloom filter [34] for memorizing the executed computing node. After memorizing, the control center would ask computing node to switch to another task for contributing compute power. Meanwhile, the computing node would save a copy of current unique key for the executed task and would not apply the executed task until the unique key changes.

This approach helps us to prevent from retrieving redundant data and it's easy to launch by simply add a flag *distinct=true* as listing 4. The initial data varies from the amount of result we desired. We could simply create n rows in an iterator as the initial data if we aim to collect n geolocations of IP address. Since the input is just a dummy counter and the output is a pair of IP and geolocation, the proper programming model of this converting algorithm could be described as *Map*, which takes one element of type K and produces a element of type T. The task terminates when the initial queue get exhaust. Through the result of this job is a mapping table of IP and geolocation, it's still possible to export the bloom filter for the next time if we want to launch the some job but keep the redundant data out.

Listing 4: Job Builder of IP Location Finder

```
public class Geolocation {
  public static void main(String[] args) throws Exception {
    Iterator<URLData> nIter = new Generator();
    QueueRef initialData = new QueueRef("initialData");
    QueueRef result = new QueueRef("result");
    JobBuilder.createInstance("name.prefix","path/to/file.jar")
      .initialData(nIter, initialData)
      .appendTask(initialData, GetGeolocationMapper.class, result, true)
      .build();
  }
}
```

8 Future work

8.1 Security

Since our framework provides API for developers to port their program/algorithm to volunteers' smartphones dynamically which means developers could do whatever they want on volunteers' smartphones. This feature would raise lots of security issues and so far we categorized the malicious behavior into two parts:

1. Damage to volunteers' smartphones.
2. Damage to the whole network system.

From the aspect of the damaging to volunteers' smartphones, developer could simply write an infinite loop with some commands that drive the CPU crazy or create unnecessary files as a worm. Worst of all, we found that Android SDK provides some built in function that would "help" developer with evil intentions to invade volunteers' privacy by taking screenshots, viewing profiles generated by other apps or get volunteers' location by leveraging GPS module. For the damaging to the whole network system, the developer could simply write a program to harass targeting web site by invoking tons of request from numerous clients from this framework, it turns volunteers' smartphone into part of their zombie networks. For fixing these kind of problems we did some survey and found there are several ongoing researches including runtime memory introspection [35] and static program analysis [35] for discovering malicious behavior.

However, the hardest thing for us is to determine what's the real intention of the submitted program or which behavior is malicious. For example, the mapper of the word count example mentioned in the previous section asks every smartphones to visit different hiring requirements and split the content into a word-frequency table for further computing. Though the visited URLs are distinct but actually these URLs are in the same web site and it could burden the targeting web server when smartphones are crawling the same web site simultaneously from different location of the network. The intention of word count example is not to harass someone but it does.

8.2 Permission control

By far we gained permissions from users that enable Gnafuy app to receive data and perform tasks dynamically without upgrading or reinstalling the whole app. However, these enabled permissions increases the dangerousness of some smartphone users. Developers could retrieve information like International Mobile Equipment Identity(IMEI) and GPS location continuously within one task.

Unfortunately, the combination of IMEI and GPS location helps us to track specific user's daily routine and this would lead some invasion of privacy definitely. In order to prevent the abuse of crowd computing power, we could offer a configurable settings for smartphone users to limit the information they would like to provide. Meanwhile, we have to isolate Android SDK from Gnafuy API to prevent developers to invoke these native method directly. We could provide a series of functions in Gnafuy API that has the capability to retrieve these personal data only when user has acknowledged in configuration.

9 Conclusion

We present Gnafuy, a framework utilizing crowd-smartphones to fulfill ubiquitous distributed computation. Gnafuy provides flexible APIs for general purpose applications equipped with the ability of porting computation to mobile devices. Integrating the power of crowd sourcing with mobile applications, tasks that ask for human wisdom could advance in a fluent way.

Acknowledgment

We thank anonymous reviewers for their valuable comments. This work was supported in part by Taiwan Information Security Center (TWISC), Academia Sinica, and Ministry of Science and Technology, Taiwan, under the grant MOST 104-2218-E-001-002 and MOST-103-2221-E-004 -006 -MY3.

References

[1] "Apache Hadoop." `http://hadoop.apache.org/`. (Visited on 02/16/2016).

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets.,"

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive-a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 996–1005, IEEE, 2010.

[5] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.," in *OSDI*, vol. 8, pp. 1–14, 2008.

[6] "Dashboards." `http://developer.android.com/about/dashboards/index.html`, 2015. Online; Accessed 4 January 2015.

[7] "Compare iPhone models.." `http://www.apple.com/in/iphone/compare`. Online; Accessed 4 January 2015.

[8] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[9] G. Huerta-Canepa and D. Lee, "A virtual cloud computing provider for mobile devices," in *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, p. 6, ACM, 2010.

[10] J. H. Christensen, "Using restful web-services and cloud computing to create next generation mobile applications," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pp. 627–634, ACM, 2009.

[11] E. E. Marinelli, "Hyrax: cloud computing on mobile devices using MapReduce," tech. rep., DTIC Document, 2009.

[12] N. Palmer, R. Kemp, T. Kielmann, and H. Bal, "Ibis for mobility: solving challenges of mobile computing using grid techniques," in *Proceedings of the 10th workshop on Mobile Computing Systems and Applications*, p. 17, ACM, 2009.

[13] "Selenium - web browser automation." `http://www.seleniumhq.org/`. (Accessed on 02/22/2016).

[14] P. J. Boland, "Majority systems and the condorcet jury theorem," *The Statistician*, pp. 181–189, 1989.

[15] "On Distinguishing between Reliable and Unreliable Sensors Without a Knowledge of the Ground Truth (2015), author=Yazidi Anis, Oommen John and Goodwin Morten, year=2015,"

[16] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[17] "Apache phoenix." `https://phoenix.apache.org/`. (Visited on 06/13/2016).

[18] "The Scala Programming Language." `http://www.scala-lang.org/`. (Visited on 02/17/2016).

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 59–72, ACM, 2007.

[20] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, 2013.

[21] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pp. 4–10, IEEE, 2004.

[22] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[23] "Rosetta@home." `https://boinc.bakerlab.org/`. (Visited on 02/18/2016).

[24] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, pp. 145–154, ACM, 2012.

[25] S. Buchegger and J.-Y. Le Boudec, "A robust reputation system for peer-to-peer and mobile ad-hoc networks," in *P2PEcon 2004*, no. LCA-CONF-2004-009, 2004.

[26] C. Dellarocas, "Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior," in *Proceedings of the 2nd ACM conference on Electronic commerce*, pp. 150–157, ACM, 2000.

[27] S. Sen and N. Sajja, "Robustness of reputation-based trust: Boolean case," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pp. 288–293, ACM, 2002.

[28] "Akka." `http://akka.io/`. (Visited on 02/17/2016).

[29] "spray | rest/http for your akka/scala actors." `http://spray.io/`. (Visited on 02/17/2016).

[30] "RabbitMQ - Messaging that just works." `https://www.rabbitmq.com/`. (Accessed on 02/22/2016).

[31] "How to Run Node.js with Express on Mobile Devices." `http://www.sitepoint.com/how-to-run-node-js-with-express-on-mobile-devices/`. (Accessed on 02/22/2016).

[32] "Android Permission." `https://developer.android.com/reference/android/Manifest.permission.html`. Online; Accessed 4 May 2016.

[33] "Google reCAPTCHA." `https://www.google.com/recaptcha/intro/index.html`. (Accessed on 04/22/2016).

[34] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *Algorithms–ESA 2006*, pp. 456–467, Springer, 2006.

[35] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pp. 569–584, 2012.