Timeline analysis for Android-based systems

Yu Jin

Kongens Lyngby 2013
IMM-M.Sc.-2013-42

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-42

Summary (English)

The goal of the thesis is to design and implement a timeline analysis framework that can be used to analyse events within the Android operating system, with particular focus on their chronological relations. To achieve this, a study of Android system was taken place prior to the implementation of this framework. After that, a framework that consists of two functional components was implemented, the components are namely extraction component and visualisation component.

The extraction component consists of an Android application and several shell and python scripts, wherein the application and scripts can be used to extract and preserve evidential artefacts from mainly three places in the Android system, namely Logcat logging buffers, Android system files and SQLite database. Apart from that, the visualisation component, mainly a graphical timeline analysis tool was designed and implemented. The tool is powered by modern browsers with novel HTML technologies as well as a web server. This Browser/Server architecture chosen here has mainly three advantages. First, with proper access control on the web server, the evidence can be safely preserved. And which allows multiple investigators to work on the same case simultaneously without replication and distribution of the original evidence. Secondly, the use of the server in the backend makes the framework more scalable in case for bulky analysis demands. Last but not least, the use of browsers and HTML technologies make the analysis tool truly platform independent, the analysis task could even be done in mobile devices. In addition, the graphical presentations of evidence were implemented in the way that users can have full control over what should be displayed on the timeline and which period should be displayed. This feature is of utmost importance as patterns of activities can only be seen in relatively long periods. Whereas coherent activities need to be verified or correlated in manner of seconds.

Lastly, evaluations were carried out to verify the effectiveness of the framework. The results from the evaluations showed that the framework is theoretically effective to locate activity patterns as well as coherent events. However, considering the fact that the Android system evolves very quickly and the techniques used by malicious applications vary a lot, the framework should be considered as a proof of concept. In addition to that, from the experience gained in the development and evaluation processes, it is deemed that the technologies selected for this implementation were suitable for forensic timeline analysis. More importantly, the potential of these technologies can be further explored to make better analysis tools.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Telecommunication.

The thesis deals with the problem of the lacking available tools for conducting timeline analysis for Android-based system in forensic manner.

The thesis consists of a study of Google Android system in respect to forensics and a design and implementation of a framework. The framework is dedicated to provide evidence extraction and timeline analysis for Android-based systems.

Lyngby, 12-June-2013

Yu Jin

Yu Jin

Acknowledgements

I would like to firstly thank my supervisor, Doctor Robin Sharp, who inspired me a lot in this project and kept encouraging me to pursue a better development of this thesis. What's more, his spirit of adventure in respect to research and scholarship let me truely enjoy the thrill in this learning process. Secondly, I would like to give my gratitude to Android Malware Genome project hosted by Yajin Zhou and Xuxian Jiang at Department of Computer Science, North Carolina State University. With their malware samples, this project could be evaluated under empirical cases. Last but not least, I would also like to express my appreciation to my families and friends, who have supported me throughout the entire process by letting me be able to concentrate on this thesis and give my best to it.

Contents

Chapter 1

Introduction

Due to the massive deployment of Android devices (including smart phones, tablets, etc.), these devices have become a new battle field for digital criminals and investigators. This emerging situation then calls for a dedicated framework aiming at providing forensic timeline analysis for Android-based systems. In addition, to the best knowledge of the author, there is also a lack of timeline visualisation studies at present. However, it is deemed that the graphical presentation of a timeline immensely influences the efficiency of investigation tasks. Therefore, a carefully crafted timeline visualisation which is capable of displaying which coherent activities have happened within the target system is required.

1.1 Related works

So far, several forensic timeline analysis tools have been developed. However, none of those has visualised activity sequences. In fact, quite a few of those do not have a graphical representation of timeline.

1.1.1 Zeitline

Zeitline is a forensic timeline editor developed by *Florian Buchholz* and *Courtney Falk* [BF05]. Which is designed as an extensible tool that is capable of using various data source to generate system events. Zeitline treats events as either atomic events or complex events, wherein complex events consist of multiple atomic events. Based on these structures, the tool presents the events timeline just like a directory tree, where events are listed and indexed by the happened date. For complex events, there will be a sub-directory contains several nodes that represent the atomic events that form the corresponding complex events. Zeitline also supports filtering the results being displayed by using query mechanism. Nonetheless, the presentation of timeline in Zeitline is considered as text-based instead of graphics and it also lacks the ability to provide a high-level overview of the system which is deemed essential for users to better recognise coherence between multiple events.

1.1.2 TSK Autopsy

The Sleuth Kit is the most well known forensic toolset available today. Which contains a collection of Unix-like command-line tools that can work together to extract temporal artefacts from various file systems. Autopsy a.k.a Autopsy Forensic Browser [Car13a] is a web-based graphical interface to the TSK. Like Zeitline aforementioned, the timeline presentation is text-based also in Autopsy. And the timeline only contains file system activities. However, in the most recent beta version of Autopsy, single navigation tree is added and more types of activity artefacts other than file system activities are supported as well.

1.1.3 Ex-Tip

Ex-Tip is yet another extensible timeline analysis framework found by the author. The Ex-Tip is developed by *SANS Institute* [Clo08]. It supports a wide range of input sources from classic Mactime *body files* [Sle09] to various log files generated by anti-virus softwares. Particularly, for Windows system, the Registry keys can also be parsed by the Ex-Tip. Though Ex-Tip can support as many output format as possible by adding output modules to the framework, but it currently only supports Mactime output format, which is again a text-based timeline.

1.1.4 CyberForensic TimeLab

CyberForensic TimeLab is a timeline visualisation tool developed by *Jens Olsson* and *Martin Boldt* [OB09]. This tool can scan multiple source files or data structures to extract temporal artefacts from target system including JPEG files, file system itself and Windows events log just to name a few. The feature in this tool that differentiates from all the other tools mentioned above is that it expresses each evidence source that found in the target system as a histogram with Y axis stands for number of records while X axis represents the date. And these histograms are then vertically stacked together as the final view that is presented to the users. By showing that graphical timeline view, it makes it easier for users to find coherent events during inspections.

Summing up, though there are some forensic timeline generation tools exist, but the lack of graphical representation of timeline has not been filled. Moreover, these tools examined above mainly focus on conventional systems like Unix servers or Windows PCs and they only provide a single type of perspective of artefacts regarding temporal relations – indexing the artefacts by the date they happened. Indeed it is an intuitive view for timeline analysis but it may not be suitable for reflecting all the connections between events and it is vulnerable to date tampering.

1.2 Contributions

The contribution of this thesis is mainly twofold: *a*) built a framework which can be used to extract digital evidence from Android systems and process the evidential artefacts for timeline analysis; and *b*) explored a novel way for graphically presenting the timeline and also developed other forms of graphical presentations for serving the forensic purposes via Web application.

1.3 Thesis structure

The rest of this thesis is structured as following. Chapter 2 discussed design requirements and the architecture has been chosen to fulfil that requirements. Chapter 3 then described the presentations of artefacts along with corresponding data structures backing the presentations. After that, implementation details were given in Chapter 4 including evidence collection, Self-Organising Map and artefacts visualisation. Next, an evaluation of the prototype of this framework was conducted in Chapter 5. Lastly, conclusion including future work was given in Chapter 6.

Chapter 2

Framework design & architecture

Before the framework can be implemented, there were four fundamental aspects that had to be defined and answered. Based on the reasonable design, an architecture has been determined to as much as possible fulfil the requirements assumed and argued in design procedure. Details are discussed in following sections.

2.1 Design requirements

Four fundamental requirements of a framework, arguably can be formulated as: *a*) *Usability* that answers how and how well the users can use the framework; *b*) *Accessibility* that defines the convenience of the usability of the framework; *c*) *Extensibility* that enables the framework can be reused and expanded; and *d*) *Scalability* that tries to ensure the framework can be applied in different scale of incidents.

2.1.1 Usability

The first and foremost aspect of a framework is the ability to help users to solve their problems. For the sake of forensic timeline analysis and particularly in this thesis, the author believes that the procedures during both evidence collection phase and examination phase should be as automated and convenient as possible with very least unavoidable user interactions.

To be more specific, in the evidence collection phase, users should be able to use just a few scripts with simple parameters to achieve the data transformation that turns raw evidence data, that extracted from the target device, to the prepared dataset, which is then ready to be processed for timeline generation in the examination phase. On the other hand, during the examination phase, users should be able to filter the data being displayed on the timeline and be able to zoom and pan the timeline itself with simple and intuitive operations. On top of that, whatever changes that users want to apply to the timeline being inspected should be responsive, if possible, so that users' operations can be consistent without or with only a few interruptions or waiting time.

The three operations mentioned just then, filter, zoom and pan are of utmost importance for a timeline analysis, because the users may easily run into the cases that tons of records/events have been extracted from the target device and that makes a lot of "noise" on the timeline being presented. This situation, definitely, will make the inspection a lot more difficult and may bury the truth. To encounter this, filter operation can be applied to reduce the amount of data displayed on the timeline. But more exactly, users should not only have the ability to reduce unwanted events, but preferably be given the ability to specify which events should be displayed on the timeline. By doing so, users will get a better chance to locate coherent activities and find more concrete evidence to support their decision.

Meanwhile zoom and pan are the very intuitive operations that could be applied to a timeline. These operations can help the users to select the desired period and therefore also serve the purpose of reducing interferences. More importantly and specifically, the zoom capability enables users to not only have a general view of the whole period being examined to find events patterns but also can focus on a relatively shorter period in order to check small details that may not be clear enough in the boarder view.

It is also worthy to mention that the deployment of the framework should also be painless for users. In other words, the implementation should mainly rely on standard libraries as much as possible and only introduce third-party libraries when necessary and the third-party libraries that ever used in this framework must be easy to obtain and well maintained.

2.1.2 Accessibility

The accessibility should be carefully considered in this project for firstly it is pertinent to the deployment concern discussed in the previous section. The technology which carries the timeline inspection tool should be independent of any specific hardware or operating systems, so the tool is easy to install or even without installation on users' devices. Besides, due to the fact that there are lots of technologies which can be utilised to make the graphical user interface for this tool, and these GUI bearers are not always available in the full span of systems or devices. Therefore, the most portable technology that is capable of backing the timeline visualisation should have the priority in this situation as such a technology will most likely make this tool easy to access and require less or even no additional knowledge or training for the users.

2.1.3 Extensibility

Since the Android system is changing rapidly over time, the complexity of existing Android systems ever increases. In addition to that, manufactures tend to have their own custom editions of Android system. Therefore, a modularised implementation is required so that when specific demand arises, new input parser modules or interpreter modules can be loaded into the framework or the old ones can be extended in order to meet the new scenarios. To achieve this, extraction modules and format modules should be separated clearly with well defined data exchange interfaces, so modifying an existing extraction module or adding new extraction module would be painless. What's more important, the parser modules should be independent from the core of the framework as they are the ones that most probably changing over time. On top of that, the backing storage technology should be flexible to store various data types and meanwhile it should be able to keep the data structure unchanged as it is in the original dataset.

2.1.4 Scalability

Though the amount of artefacts in Android devices nowadays is quite limited due to Android system's constraints, but the investigation might apply upon a massive set of devices, or the artefacts are collected from a single device with a fairly long period. So the amount of data can be large under certain circumstances. Therefore, the framework should be capable of scaling with the amount of artefacts available to the users.

2.2 Framework architecture

To meet the requirements discussed in previous section. The architecture of the framework was designed as demonstrated below. The core of this framework is a web server which is hosting the collection of evidential artefacts extracted from mobile devices and the generation of classification of activities that have been observed in the devices being inspected.

Apart from the server itself, there are other three parts constitute the whole framework:

- Server side scripts for extracting and parsing evidence from physically connected devices;
- An Android app being installed in mobile devices for extracting and uploading evidence; and
- finally Browser site technology for rendering the graphical presentations of evidence and user interface.

The Figure 2.1 better demonstrates the architecture of this framework.

2.2.1 Web server

This centralised web server makes it possible to allow multiple agents to work on the same case (dataset) at the same time without risking forensic principles as all evidence are processed and saved on the server by a single authority. And then from client's perspective, they have no means to modify the original dataset resides on the remote but can only fetch the data into local memory and proceed with analysis.

The functionality of this web server is twofold: *a*) to collect uploaded evidence from a remote device and use the accumulated data to generate a Self-organising map [Mas13] of activities of the Android device; and *b*) to respond to queries from browsers with corresponding evidence stored in database.

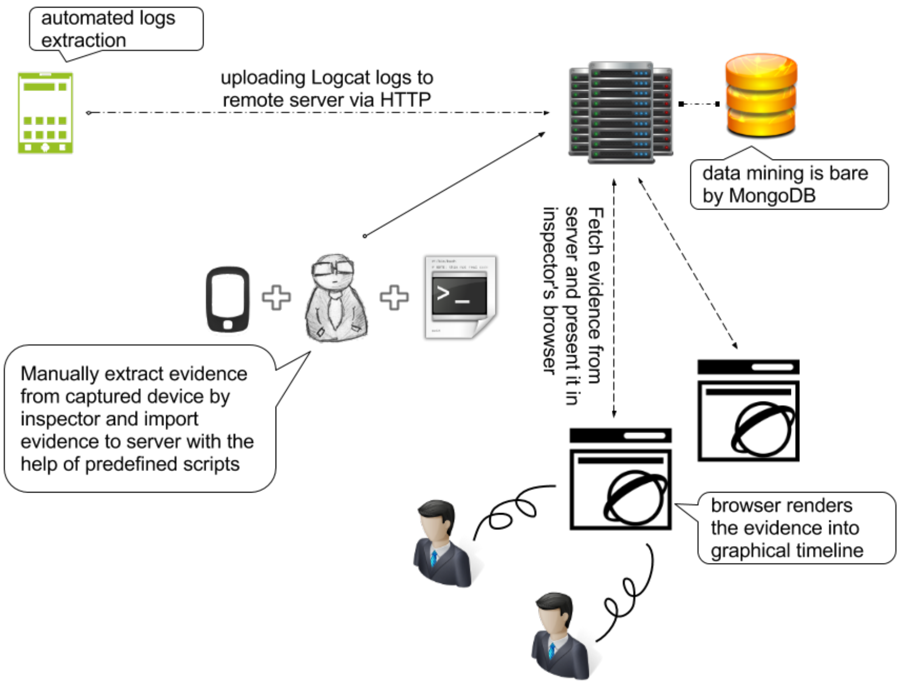

automated logs extraction
uploading Logcat logs to remote server via HTTP
data mining is bare by MongoDB
Fetch evidence from server and presence it in inspector's browser
Manually extract evidence from captured device by inspector and import evidence to server with the help of predefined scripts
browser renders the evidence into graphical timeline

Figure 2.1: Framework Architecture

2.2.2 Server side scripts

A set of scripts that can be combined together to serve as the evidence extraction tool. These scripts further can be categorised by whether user interference is needed or not. For instance, as shown in Figure 2.1, the upload of logs will trigger the automated scripts to parse the log content and the generation of Self-organising map on the server side. Whereas, upon a suspicious device is captured, the experts can use some other scripts to extract more artefacts from that device. The intermediate results may be temporarily stored as plain-text files in the server, and the final format of all evidential artefacts are converted to JSON before being written to database.

2.2.3 Android App

This prototyping app serves in two scenarios. Investigators may install the app on devices over which they have full control, then there is a background service started by the app to collect logs and send them to the server over time. Thus, these logs from devices controlled by legitimate users can be used to set up a classification of activities that would happen during normal usage. And this classification later can be used to detect malicious or unusual usage cases. In addition to that, this app can also be installed to a captured suspicious devices to manually extract more evidence for analysis.

2.2.4 Browser side rendering

d3.js is an open-source Javascript library that helps developers to create various graphics on web page based on HTML SVG techniques. There are quite a lot technical specifications or documentations about *d3.js* available online or in some dedicated books but the details are out of the scope of this thesis. However *d3.js* is the key enabler for rendering the evidential data into graphical timeline and other relevant graphical representations. This procedure happens in the users' browser and users can interact with the graphics to filter and adjust the display of data. All these operations are driven by Javascript and the libraries used in this framework are supported by most modern browsers if not all.

2.3 Summary

In a nutshell, the four abilities described above together decide which bundle of technologies can best fit the requirements of this framework. Thus, an architecture combines *Browser/Server* model and *Client/Server* model was selected to bear the timeline visualisation. Wherein the browser provides the high level usability and accessibility to users while by putting heavy computations on the server side gives the answer to scalability. Besides, the data format in all transmissions are unified in JSON. This unification gives the dignity of reducing complexity when defining data exchange interfaces in all parts of this framework.

Chapter 3

Presentation of artefacts

In this work, four types of presentation of artefacts have been implemented, namely Timeline View 3.3, Aggregation View 3.4, Deltatime View 3.5 and Classification View 3.6. Each presentation provides a different perspective towards artefacts and thus serves for different investigation goals.

3.1 Goals of presentations

Specifically, As demonstrated in Figure 3.2, Timeline View presents the entire history that extracted from the target device in a timeline to give an overview of all activities. This presentation actually deals with the event reconstruction for the target device by the means of “time-lining” [BT07]. What’s more, by vertically piling all the series of events on the timeline, this presentation helps inspectors to locate coherent activities between different Apps or between Apps and Android system. This sort of correlation helps the timeline analysis in two perspectives. First of all, finding coherent activities itself is a very important task in digital forensics as it can lead to the conclusion that one activity is the cause of another activity. This relationship between activities should be carefully examined, particularly when the two activities seem not having such a casual relation. Secondly, two coherent activities can be also used to correlate each other’s timestamps, that is, by fitting one series of events to the coherent series to see if the timestamps could match. Then the chronological order of two series can be verified basing the matching result [SMC06].

In addition, Aggregation View (shown in Figure 3.5) that presents events on a timeline just like in the Timeline View. But herein, the events are aggregated by system calls or PIDs. And thus this view focuses on activities that related to a particular system call or process. In essence, the intent of this view is to give a closer look at the behaviours or possible patterns of a particular process or a system call throughout the time span. And this can possibly yield coherent activities and/or behavioural patterns of subject being inspected. What's more, this view is also complementary to the Timeline View under the scenario that when some system calls can not be bound to any series of events due to the logging format but they do belong to certain applications or are triggered by some applications, then this Aggregation View can be used to examine those system calls that will not be shown from Timeline View. In addition to that, since this view shows a timeline of a certaion system call and the occurences are distinguished by their trigger processes, then this view also provides a means of comparing how different processes (applications) make use of system calls. And this comparison may lead inspectors to abnormalies if some applications execessively use more particular system calls than any other applications.

Moreover, the Deltatime View illustrated in Figure 3.8 focuses on the time intervals between consecutive related events from an App. This presentation is intended to assist in locating regular repeated events and thus a steady behaviour pattern. It is reasonable to assume that normal Apps will have a rather random distribution of time intervals among events as they rarely do tasks periodicaly but just respond to users' actions. Whereas malicious services may present a special temporal pattern on particular events as these services will be, for example, fetching device's GPS location or connecting to a C&C server repeatedly. Note that, though this time interval analysis would be helpful to detect abnormal behaviour, but the intervals of which related events should be examined and which pair of events should be considered as related are difficult to determine. Under this circumstance, it has been decided to grant the ability to define related events of an application to the users. Though this approach will then require more efforts and experience from the users but which can best eliminate the substantial drawback of either black list or white list approach. To be more specific, if the black list or white list are not well formed, the investigation will fail if the malware can somehow circuvment the list mechanism. Seemingly, this is not acceptable for a forensic investigation. And if the list covers too many events, then the users may suffer a lot from the noise caused by the trivial ones. Summing up, the goal of digital investigation, finding abnormalies, can not be compromised. Therefore, this sort of gray list approach was selected for conducting time interval analysis.

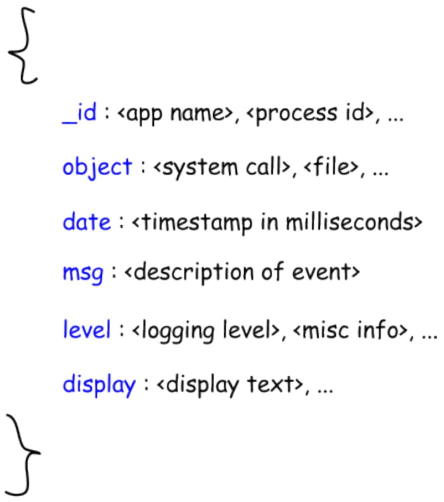
Lastly, an experimental classification graphical presentation was also implemented to support abnormality detection. Though there are quite a few available classification techniques can be used for this purpose, but the Self-Organising Map (a.k.a SOM) was chosen based on a blend of considerations that artefacts extracted from Android system are ambiguous and the resulting classifications should be presented on a two-dimensional canvas nicely.

To be more specific, as part of a graphical analysis tool, the visualisation of the neural network comes first. The Self-Organising Map is capable of mapping multi-dimensional (more than three dimensions) input vectors onto a two-dimensional map [Mas13]. This unique capability gives a clear view of how various Apps fall into different classifications and their differences in the multi-dimensional space can also be preserved in the two-dimensional map. On the contrast, for example, the K-means cluster algorithm, though which is capable of both supervised and unsupervised learning but the result clusters can only be expressed in that multi-dimensional space. In addition, taken into account that there is no convincing methods at present to reduce the dimensions of that result, so the SOM outperforms other clustering algorithms, with the better presentation capability of its clustering result. Moreover, this type of neural network can be trained without supervision, this is especially appealing given that the large variety of Android-based systems makes supervised training almost impractical. Therefore, to the best knowledge of author, Self-Organising Map was chosen to best fulfil the requirements in this work.

It is worthy mentioning that all the graphical presentations of evidential dataset are driven by the corresponding data structures backing them. Presumably, these data structures themselves illustrate well how the visualisations will be done and how the relations of interest among artefacts can be revealed.

3.2 Generic data structure

Due to various data sources from Android system have quite different formats, a generic data structure that can unify these ambiguous data types is defined. Considering the forensic context, information pertaining to "who did what at what time" must be included in the generic data structure. Thus, the data structure is defined as shown in Figure 3.1. All data being used to generate graphical presentations will be firstly converted to generic data structure, then the unified data can be grouped and/or ordered in respect to the type of presentations they belong to.



```
{
    _id : <app name>, <process id>, ...
    object : <system call>, <file>, ...
    date : <timestamp in milliseconds>
    msg : <description of event>
    level : <logging level>, <misc info>, ...
    display : <display text>, ...
}
```

Figure 3.1: Generic Data Structure

3.3 Timeline view

Timeline View is the view that gives all series of events in multiple rows with a different color for each row on the timeline. Each row represents an application that owns the series of events on that row. All the events in the series and all series are ordered chronologically. Figure 3.2 shows this Timeline View and Figure 3.3 illustrates the corresponding data structure. From the data structure, it can be seen that there are four attributes for each event. The *_id* attribute contains the name of the application to which this event belongs. Seemingly, which color will be used for each event is determined by this attribute at the time the events are being rendered on the timeline. In addition, *date* attribute is the time at which the event occurred, this attribute has two kinds of format, one is the Unix epoch format while the other is then a human readable date string. This attribute is the only reference when deciding the event's chronological position in a series. Apart from that, *display name* is just as the name indicated, a string representation of the event. However the *content* attribute has a rather more complicated structure than any other attributes. Specifically, this attribute contains a dictionary structure to store all the system calls invoked in the event, wherein the key attribute of this dictionary is the system call object and the value attribute is the log messages produced by that call. Each system call can have more than one log message because the system call can be invoked at the same times multiples times and for different purposes in Android system. What's more for these messages is that a process ID is appended to

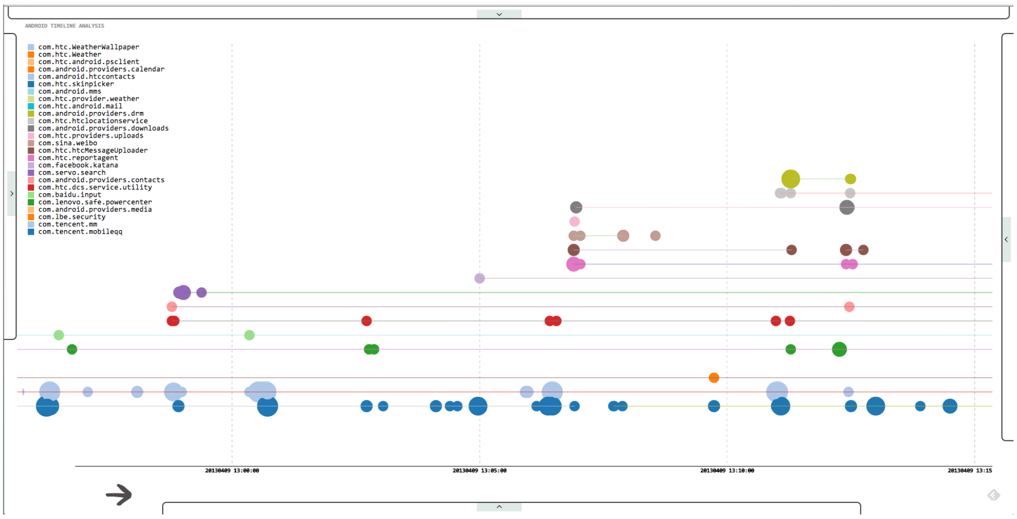
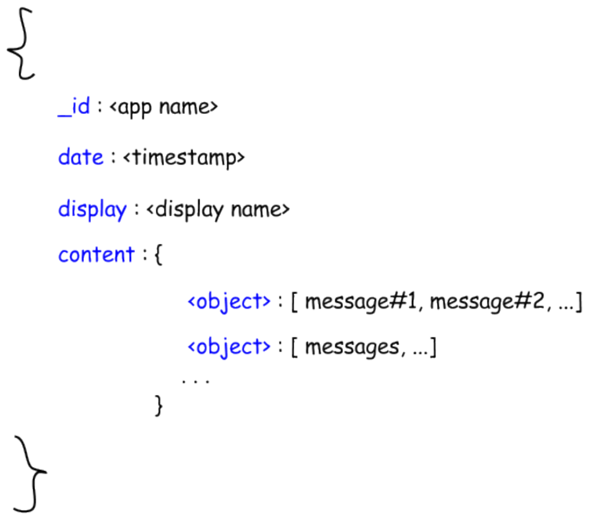

ANDROID TIMELINE ANALYSIS
🟦 com.htc.WeatherWallpaper
🟧 com.htc.Weather
🟧 com.htc.android.psclient
🟧 com.android.providers.calendar
🟦 com.android.htccontacts
🟦 com.htc.skinpicker
🟦 com.android.mms
🟨 com.htc.provider.weather
🟦 com.htc.android.mail
🟨 com.android.providers.drm
⬜ com.htc.htclocationservice
⬛ com.android.providers.downloads
🟪 com.htc.providers.uploads
🟫 com.sina.weibo
🟫 com.htc.htcMessageUploader
🟪 com.htc.reportagent
🟪 com.facebook.katana
🟪 com.servo.search
🟥 com.android.providers.contacts
🟥 com.htc.dcs.service.utility
🟩 com.baidu.input
🟩 com.lenovo.safe.powercenter
🟧 com.android.providers.media
🟧 com.lbe.security
🟦 com.tencent.mm
🟦 com.tencent.mobileqq
20130409 13:00:00
20130409 13:05:00
20130409 13:10:00
20130409 13:15

Figure 3.2: Timeline View

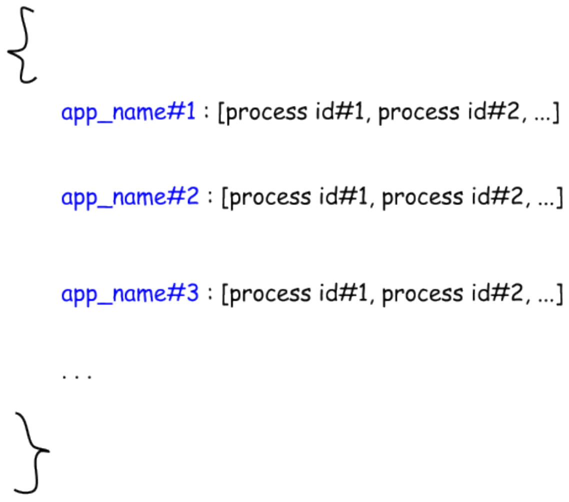


```
{
    _id : <app name>
    date : <timestamp>
    display : <display name>
    content : {
                <object> : [ message#1, message#2, ...]
                <object> : [ messages, ...]
                . . .
              }
}
```

Figure 3.3: Timeline Event Data Structure

the end of each message so later all the events related to the same process can be connected as shown in the figure. This modification of log messages also explains the data structure demonstrated in Figure 3.4, which is used for generating paths for different processes of an App. This is done by iterating the

PID array of each App and then for each process ID, the events with messages that contain the same ID will be connected with a path line. These lines are not trivial for investigation because each process should be examined separately and there should be a way for distinguishing different processes. Note that each



```
{
    app_name#1 : [process id#1, process id#2, ...]

    app_name#2 : [process id#1, process id#2, ...]

    app_name#3 : [process id#1, process id#2, ...]

    . . .
}
```

Figure 3.4: Timeline Event Series Data Structure

process stands for an active period of an application and each application may have several active periods in Android systems. Within an active period, a process of the application is started and destroyed with various time intervals in between. Further, in these durations the process may do some operations or simply be idle, but either way it will be the case that two or more events can be grouped by their process ID and therefore form a series of events that actually expresses an active period of an application, and then such a series of events is defined as an activity of the application. This activity notion will be referred to in the following text whenever the word “activity” is used.

The reason for introducing the activity concept for applications instead of showing single events is twofold: firstly, displaying all events without any grouping will make the investigation tedious and hard to find a start point. However, if the events are grouped together and displayed separately on the timeline presentation, it yields a clearer view of the relations between events. Second of all, by finding the start event and end event of each activity, it automatically produces the time bounds [GP05] or the time frame for all the events that related to or within an activity. Such a time bounding is critical for timeline analysis as it establishes relations between timestamps among events and therefore which can further correlate the chronological order of events.

3.4 Aggregation view

Aggregation View shows another picture of occurrences of system calls or processes in the temporal manner. Before this presentation is made, a system call object or a process ID will be specified by the user, then all the artefacts related to that system call object or the ones contain that specified process ID will be aggregated. Note that if it is a system call is specified for the aggregation, then the result is grouped by process ID, otherwise, the aggregation result is grouped by system call. This grouping is to achieve that when the result is presented on the timeline, events that come from different system calls or processes will be assigned a different Y index on the Y axis, and the ones with same system call or process will be plotted on the same row. Finally the aggregated data is grouped again by their occurrence date when presented to the user so that events of same system call or process that happened at the same time will be presented together and the number of those events decides the size of the circle in the timeline. The result timeline is shown in Figure 3.5. Seemingly, the

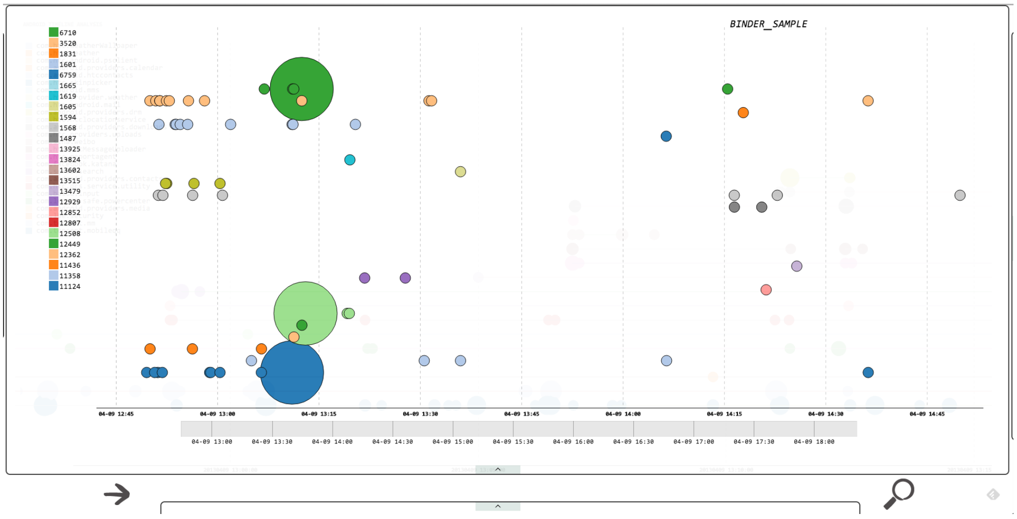
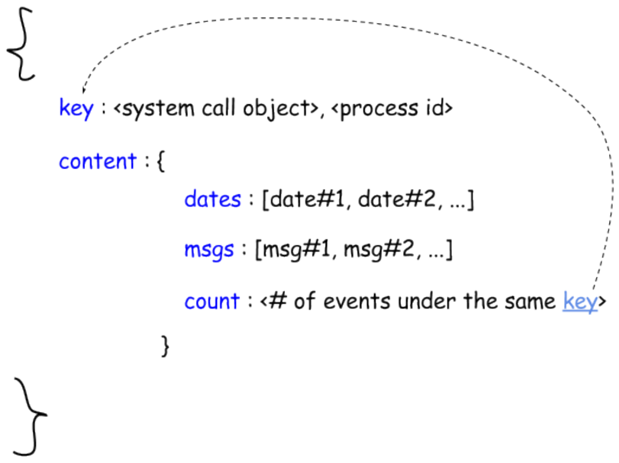


Figure 3.5: Aggregation View

data structure (shown in Figure 3.6) behind this expresses the same meaning as the graphical representation, wherein data is firstly aggregated by the *key* field (either a system call or a process ID), associated with the *key*, there are



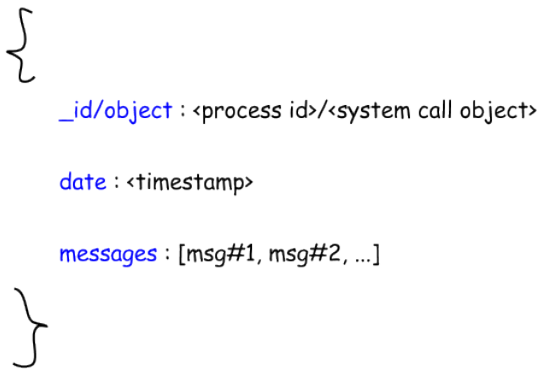
```
{
    key : <system call object>, <process id>
    content : {
              dates : [date#1, date#2, ...]
              msgs : [msg#1, msg#2, ...]
              count : <# of events under the same key>
           }
}
```

Figure 3.6: Aggregation Data Structure 1

two arrays carrying all the occurrence dates and corresponding event messages. Noteworthy that values with the same index in these two arrays are from the same event. In other words, the original events can be reformed by taking a date value from *dates* array and then using the same index to fetch the corresponding log message from the *msgs* array. In addition, there is a *count* attribute stores the total occurrence times of the system call or the process specified by the *key* field. Before the aggregated data being presented to the user, the original events are reformed as described above and then further grouped again by occurrence date. Therefore, the data structure transforms into another form as given in Figure 3.7.

3.5 Deltatime view

DeltaTime View is a quite different temporal perspective than the ones above, which focuses on showing the variance of time intervals – delta time of all related event pairs of a particular App. By related, it means the event pairs have some particular meanings than other event paris, for example the start event of a process and the end event of a process may form an event pair, and which expresses the life circle of a process, therefore the interval between these two events are the lasting time of that process. Alternatively, this event pair can also be a event indicates the start of a service and the other event is of a certain operation. Then this pair will reveal if there is a causality between the two



```
{
    _id/object : <process id>/<system call object>

    date : <timestamp>

    messages : [msg#1, msg#2, ...]
}
```

Figure 3.7: Aggregation Data Structure 2

events. More over, if the event pair is of two identical operations, then it can show whether the frequency of the operation and also tell if the operation is periodically occurred. The view is illustrated in Figure 3.8. Wherein the X axis expresses the delta time of those event pairs, the color differentiates types of event pairs and lastly the length of those bars indicate the number of event pairs. In order to hold this expression, the data structure is defined as shown in Figure 3.9. Noteworthy that a complementary structure named *signature* is defined to distinguish various event pairs, this structure is demonstrated in Figure 3.10. The event pair of interest is defined by users and upon an event pair is set, a signature will be produced to represent the chosen event pair. Then this signature is later used to filter event pairs in the dataset. When a matching pair is found in the dataset, the data structure in Figure 3.9 is produced to contain the content of the matching event pair. As can be seen in this data structure that a count is associated with each event pair of a given delta time. The count of each event pair with this delta time is critical to the analysis as for excessively periodically repeated events should have significant difference in the timeline presentation in this view, and the difference, the height of the rectangle, is directly related to the count of the event pair.

3.6 Classification view

There are two types of data structure drive the presentation of classification of activities in Android system. The first one is used to generate the Self-organising map which holds the map nodes data in training process. However, when pre-

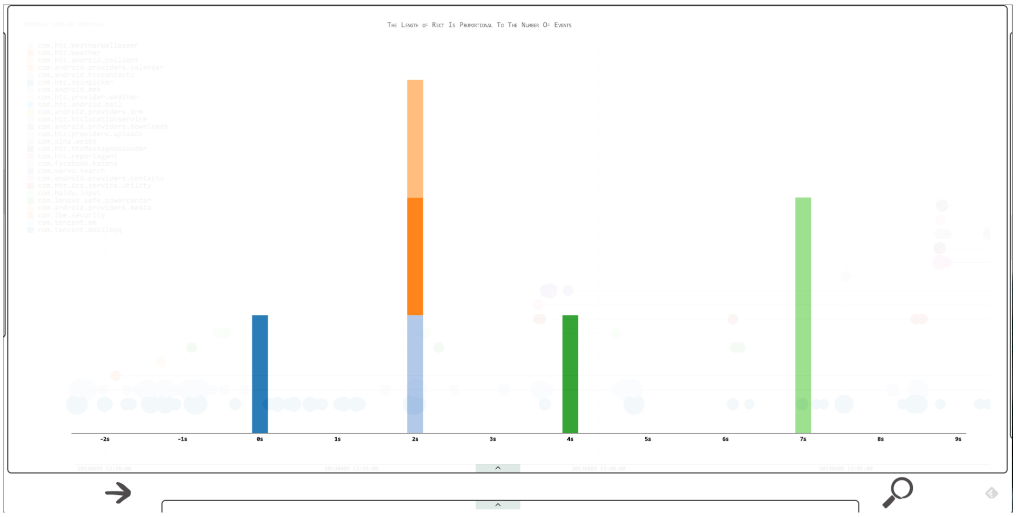
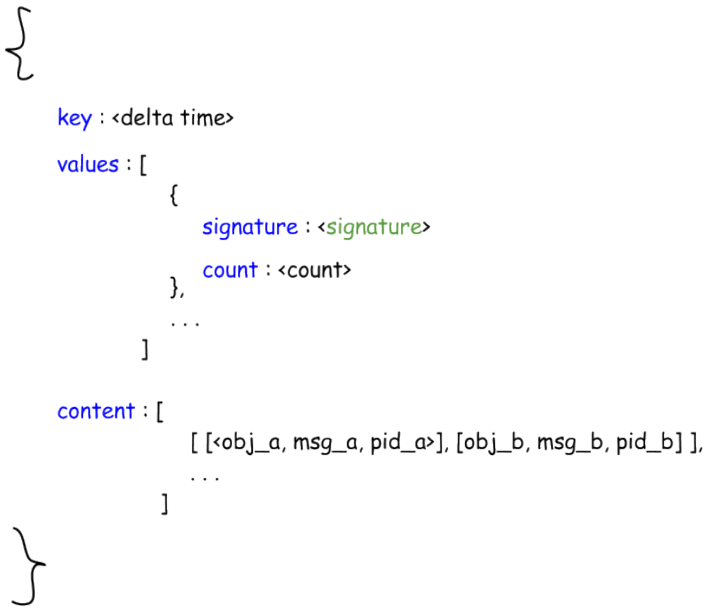
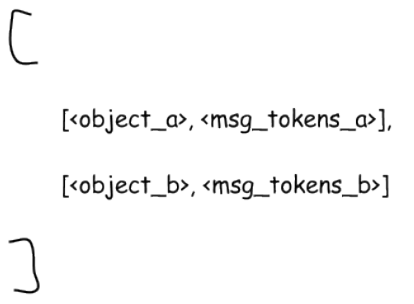


Figure 3.8: DeltaTime View



```
{
    key : <delta time>
    values : [
                {
                    signature : <signature>
                    count : <count>
                },
                . . .
             ]
    content : [
                [ [<obj_a, msg_a, pid_a>], [obj_b, msg_b, pid_b] ],
                . . .
              ]
}
```

Figure 3.9: DeltaTime View Data Structure



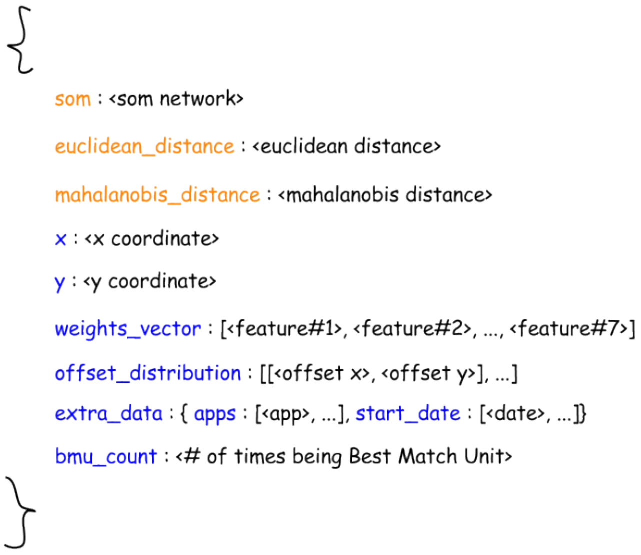
```
[
    [<object_a>, <msg_tokens_a>],
    [<object_b>, <msg_tokens_b>]
]
```

Figure 3.10: DeltaTime Event Pair Signature Structure

senting the map to users, the data used for training becomes meaningless. Thus, these data are cut off after map training and therefore the second data structure comes into play. To be more specific, the transition of the two data structures is that *Euclidean Distance* field, *SOM* field and *Mahalanobis Distance* field are removed from the original data structure. The tree fields are marked in a different color in the figure and the data structure for training the map is given in Figure 3.11. Seemingly, the *x* and *y* attributes denote the position of the map node in SOM and the *weights_ vector* is kept in the structure because it would be helpful to see the weights from the SOM presentation in order to compare to other clusters or appended applications. In addition, the *offset_ distribution* describes how these apps that of this cluster are distributed in a spatial relation to the central point. This distribution is used later to measure whether an application is close enough to the central point of this cluster so that it can be categorised in this cluster. In addition, *extra_ data* and *bmu_ count* are kept for rendering the map node, where node's radius depends on the *bmu_ count* and related apps are drawn depends on the *extra_ data*. A view of the generated Self-organising map is shown in Figure 3.12. Note that each App can have several quite different activities which then lead to the situation that an App may relate to more than one classification in the map.

3.7 Summary

In essence, four kinds of graphical presentation described in previous sections provide sort of basic and necessary views and interpretations of evidence. The presentations are intended to help investigators to find implicit connections/-patterns among massive dataset. Whereas under the skins, the data structure is the backbone of any types of graphical representation. No doubtably, there



```
{
    som : <som network>
    euclidean_distance : <euclidean distance>
    mahalanobis_distance : <mahalanobis distance>
    x : <x coordinate>
    y : <y coordinate>
    weights_vector : [<feature#1>, <feature#2>, ..., <feature#7>]
    offset_distribution : [[<offset x>, <offset y>], ...]
    extra_data : { apps : [<app>, ...], start_date : [<date>, ...]}
    bmu_count : <# of times being Best Match Unit>
}
```

Figure 3.11: Self-organising Map Training Node

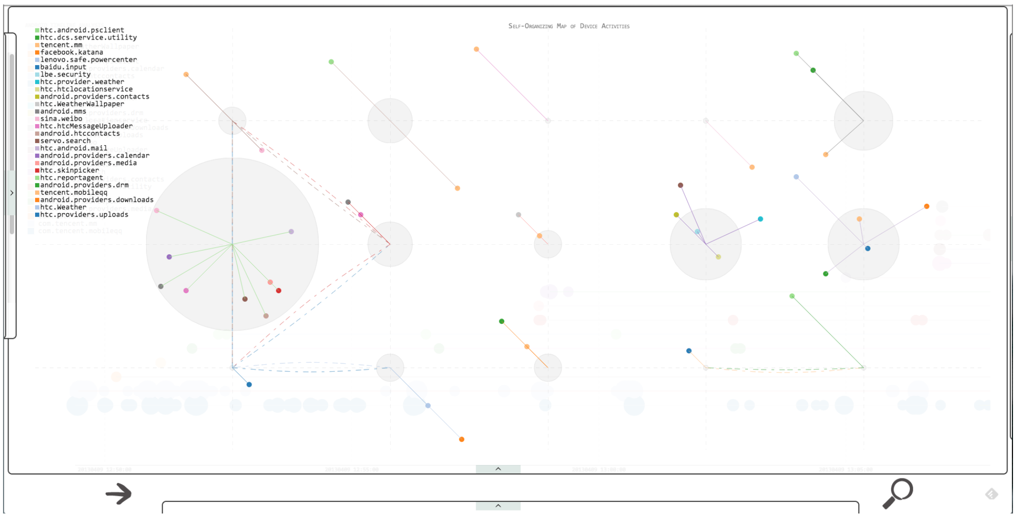


Figure 3.12: Self-organising Map View

are other tons of ways of organising or interpreting the raw artefacts. But in whichever way, a data structure is created to interpret the relations among the artefacts. And then such a structure is visualised, on the other hand, to help improve the explicitness of the relations or even revealing unknown connections.

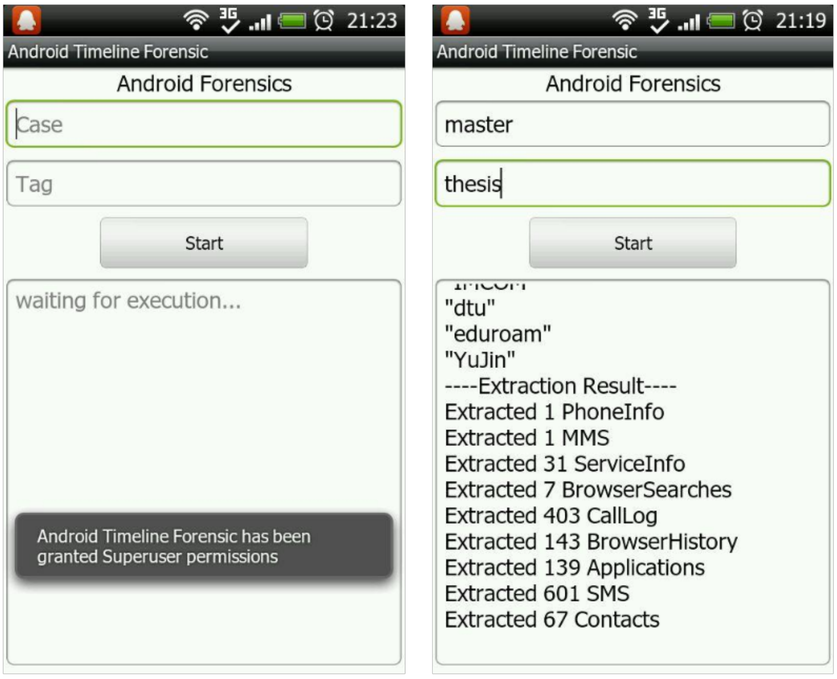
CHAPTER 4

Implementation

The implementation of this framework are demostrated as three parts *Evidence collection*, *Self-organising map* and *Visualisation of artefacts*. Wherein the implementation of evidence collection involving an Andorid App and some other complementary scripts are given in the first section. The second section illustrates the feature selection and training process of the self-organising map. Last but not least, visualisation techniques are demonstrated in the last section of this chapter.

4.1 Evidence collection

This framework relies on evidence collected from two kinds of procedure. The first way of getting evidential artefacts is through an Android app as shown in Figure 4.1. The second procedure consists of several scripts that pull out evidence from a physically connected device via USB connection. The app is in charge of extracting evidence from *Logcat* logs as well as Android build-in *ContentProvider*. Whereas, the scripts are used also for retrieving evidence from *Logcat* logs, SQLite database files and disk image of either internal or external (SD card) storage of the device. In addition, part of the scripts are also in charge of parsing the ambiguous raw artefacts and saving them into JSON format.



```
waiting for execution...

Android Timeline Forensic has been
granted Superuser permissions
```

```
"dtu"
"eduroam"
"YuJin"
----Extraction Result----
Extracted 1 PhoneInfo
Extracted 1 MMS
Extracted 31 ServiceInfo
Extracted 7 BrowserSearches
Extracted 403 CallLog
Extracted 143 BrowserHistory
Extracted 139 Applications
Extracted 601 SMS
Extracted 67 Contacts
```

Figure 4.1: Application Screenshots

4.1.1 Android Content Providers

In a forensic context, Android Content Providers have a rich meaning in respect to how and when the device is used or modified. Android system provides the access to various records through Content Providers including but not limited to Contacts, Call Logs, SMS, Installed Applications etc.. In this work, these informations are extracted in an Android App, wherein an *Interface Class* namely *Extractor* is defined for extensibility. Then, due to Android system stores aforementioned records more or less in the same way in SQLite database, so an *Abstract Class* – *GenericExtractor* implements the *Extractor* interface is defined for providing generic extraction function for most content providers. Lastly, several independent extractor *Class*es that extend the *GenericExtractor* are implemented to complete all the other functions and/or particular extraction functions for extracting records of interest. A class diagram is given in Figure 4.2 to illustrate the relations between these components. The extraction is de facto done by means of database query. Where the table being queried

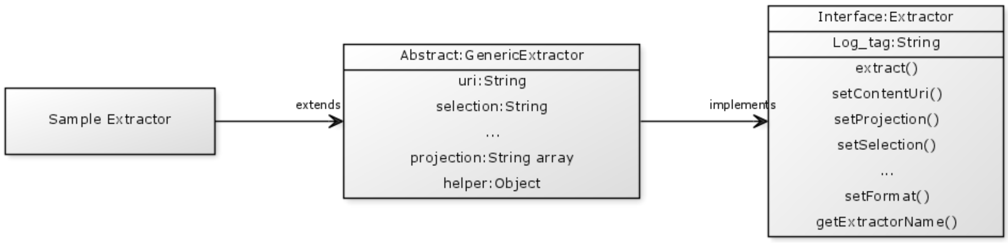


Figure 4.2: Extractor Class Diagram

is selected by a URI and the fields interested are specified by selection, output results format is defined by projection, there are other parameters when execute a query but those parameters are non-sense in this context. Then these parameters are passed to an instance of *ContentResolver* to finish the query. Code snippets are attached in Appendix A.

4.1.2 Logcat logs

Logcat logs are the merely main logging source can be found in Android system. There are four logging buffers managed by Logcat. Namely *main*, *system*, *events* and *radio*. Wherein *main* is the log buffer to which applications write their logs by default, while *events* buffer stands on the system side to record system event information. Besides, *system* buffer and *radio* buffer logs low-level system activities and cellular network information respectively.

These buffers are presented as binary files in directory */dev/log*. The parent directory of these log buffers is in fact a file system named *tmpfs* that the file system is backed in RAM as a piece of virtual memory [Hoo11], which means the content under this directory is volatile and can only be accessed when the device is running. The buffers are of different sizes, where *events* has 256KB space and the other three have 64KB for each. These limited buffer sizes are presumably obstacles for forensic purposes as important evidence will be easily overwritten due to the sizes are small. In general, if the extraction is done in a post-incident manner, the period that can be reflected in those logs are around couple of hours to the best.

4.1.3 Filesystem

In this framework, two command line tools *fls* and *ils* were compiled from the latest source code of *The Sleuth Kit* [Car13b]. The *fls* is to extract temporal artefacts from disk images. The image can be the external SD card or the internal storage of the device. Notably, for the sake of internal storage *root* privilege is required to use *dd* command in Android shell environment. The output of *fls* is the *Bodyfile* of MAC times of all files reside in the storage being examined [Car05]. Whereas, *ils* aims at extracting temporal evidence from *inode*, the metadata structure in Unix like systems [Nar07]. Both output from *fls* and *ils* are passed into complementary scripts to convert them into JSON format. The purpose for extracting temporal artefacts from filesystem is to correlate them with the evidence in logs so that the timeline analysis is more resistent to timestamp tampering. Scripts for parsing output of those tools are appended in Appendix B.

4.1.4 Misc.

There are yet other sources of evidence in the device. One of them is *SQLite* database file that stored in the device storages. In Android system, Apps save usage statistics and user credentials just to name a few using SQLite so these database files are the actual holders of aforementioned information. However, one cannot access to these databases programmatically since the internal representation of these databases and tables are unknown to inspectors. However, if with *root* privilege one can pull out the files and access them directly outside the device via various script languages, here *Python* is used. The database files can be loaded into memory and where queries can be done to fetch content of the database.

Besides, there are some system files that contain precious information regarding the time and installed application. Firstly, there is a file *packages.list* in directory "/data/system/" which provides all Apps' name along with their *uid*, *gid* and installation location. Then this information is used to connect Apps with files in the device storage where *uid* is the bridge. Secondly, in directory "/proc/", a file *stat* stores the date that the device is lastly booted as well as the system up time. Seemingly, putting these two timestamps together, a non-trivial bounding of all timestamps found in the artefacts can be formed [GP05].

4.2 Self-Organising Map

To speak of Self-Organising Map, the foremost task is to define the feature set. In contrast to the feature selection in scenarios of network intrusion detection, where artefacts can be easily vectorised. For instance the port numbers, connection duration and payload length and so on. However, the artefacts, mainly log entries, from an Android system are more linguistic, complex and ambiguous so that they can not be vectorised directly. Therefore, text searching technique has to be introduced to turn those artefacts into vectorial dataset. That is to say, for each log entry, it will be firstly tokenised. And then from the resulting tokens, keywords can be detected. Later on, by analysing those keywords, types of the event can be distinguished. Moreover, subjects and objects involved in those logs can be carved as well.

Presumably, to classify every single event in the logs would not tell any behavioural characteristics of Apps and also they are too verbose to be classified. So the concept of series of events which has been described in Section 3.3 was brought up again in the process of vectorization. Therefore a feature set with a cardinality of seven can be defined out of a series of events. The features are:

- *duration of the series*
- *total number of events*
- *number of different system calls*
- *tokens signature index*
- *number of database operations*
- *number of ContentProvider operations*
- *number of network operations*

For most of them, their name speaks the content of the feature. However, *tokens signature index* requires a bit more explanation. Recall that all the events will be tokenised, thus a series of events will have set of tokens. These tokens contain the information about ever appeared subjects (system calls) and objects (database, files etc.) of operations occurred in that series of events, perhaps also some operation details like query selections. However, the tokens themselves can not be used in the feature vector, so in order to involve this information, these tokens have to be indexed somehow to represent them as a numeric value. The method applied in this implementation is that, all token sets are compared with each other by calculating the *Jaccard distance* between two arbitrary token sets. The distance is defined as:

$$distance = (|A \cup B| - |A \cap B|)/|A \cap B|, |A \cap B| \notin \phi$$

$$distance = max(|A|, |B|), otherwise$$

Then by randomly choosing a token set as index zero, the whole set of token sets is ordered as each following token set has the minimum distance to its previous token set. The resulting set gives the situation that if two indices are close to each other, then the content of two token sets are relatively similar. Due to time constrains of this work, this is only a rough and almost trivial similarity estimation which should be redefined if given sufficient amount of time.

These features were selected because they can well express the behaviour of Android Apps. For example a contact App will relatively have more Content-Provider operations and very few database or network operations. On the other hand, an social networking App will have its own database for friends, news feeds and so on. So it can be expected that it will have higher number of database operations. Another example for explaining this could be that a weather forecast App will have relatively shorter duration and less number of events as users just open the App, check the whether and exit it. However, say a game App, then it is very likely that lots of events will happen during the execution because of complex user interactions. For the sake of malware, it can be arguably expected that they will have extremely high values on some particular features and meanwhile have extremely low values for the rest. This is because that the malware have the trend to either do something repeatedly (communicating with C&C server) or do as little as they can in order to hide themselves.

Upon the feature vector was defined, the distance algorithm that used in the training process of the neural network was selected, the *Mahalanobis distance*. Wherein the distance algorithm considers the mutual influence of features in the vector. This is more accurate and reasonable than *Euclidean distance* in the way that the features defined in this implementation are coherent. For instance the duration and number of events of a series. Also, it is arguably that given a total number of events within the series then the number of database operations, content provider operations and network operations are mutually exclusive.

4.3 Visualisation of artefacts

The visualisation of artefacts in different perspectives are powered by Javascript, or to be more general, browser side rendering techniques. The visualisation techniques on browser side has been used in business or biological fields for a while but it is novel to forensic context. Conventional approaches have two main shortages that the tools are out compatible to all platforms and the procedure of processing evidence can only rely on one sigle workstation which leads to performance issues when the amount of evidence is sufficiently large. Nevertheless, these two are no long problems as the visualisation can be done in a browser. Because the heavy computations can be carried in remote servers while HTML and Javascript are presumably most compatible techniques throughout all platforms. If ignoring the fact that small screen size downgrades the display effects, this visualisation can also be accessed from mobile devices or tablets, even the target device itself.

For the sake of actual implementation, *d3.js* [Bos12] is the key enabler which is a Javascript library using HTML, SVG and CSS techniques to visualise various data in web pages. Combining with AJAX queries backed by *Zepto.js* [Fuc13], the JSON formatted evidential artefacts stored in database are transmitted to the front-end and further rendered and plotted on web pages by *d3* instances. Besides, a tool-tip like popup dialog that shows more details of an event that being currently focused by users is implemented via *OpenTip* [Men13] and *jQuery* [jF13]. The whole visualisation process can be further divided into four sequential parts, fetching dataset, initiating graph axes, appending dataset and finally associating extra information to the data plotted on the graph, the whole procedure is demonstrated in Figure 4.3. And code snippets are enclosed in Appendix C. In data retrieval part, there is only one type of interface existing between front-end and back-end, AJAX. However, when speaking of fetching data from the database, there are two different approaches since MongoDB supports 3rd party drivers as well as native shell commands and scripting. When using a database driver, like the one used in this work *mongoosejs* [Lea11], data models represent corresponding items in MongoDB have to be defined before any queries can be done. This mechanism provides relatively strong regulations on how the database can be manipulated programmatically. But it becomes a tedious work when writing data models for dynamic or complex data. On the other hand, by using MongoDB scripting technique, no data model is needed. Thus it is a light-weight way for saving or reading data with MongoDB. However, there is a limitation of the output buffer size in MongoDB shell, which means, a database driver is required for massive data exchange.

On data retrieval complete, the boundaries of the dataset can be determined. For instance, the earliest timestamp and latest timestamp. The two timestamps

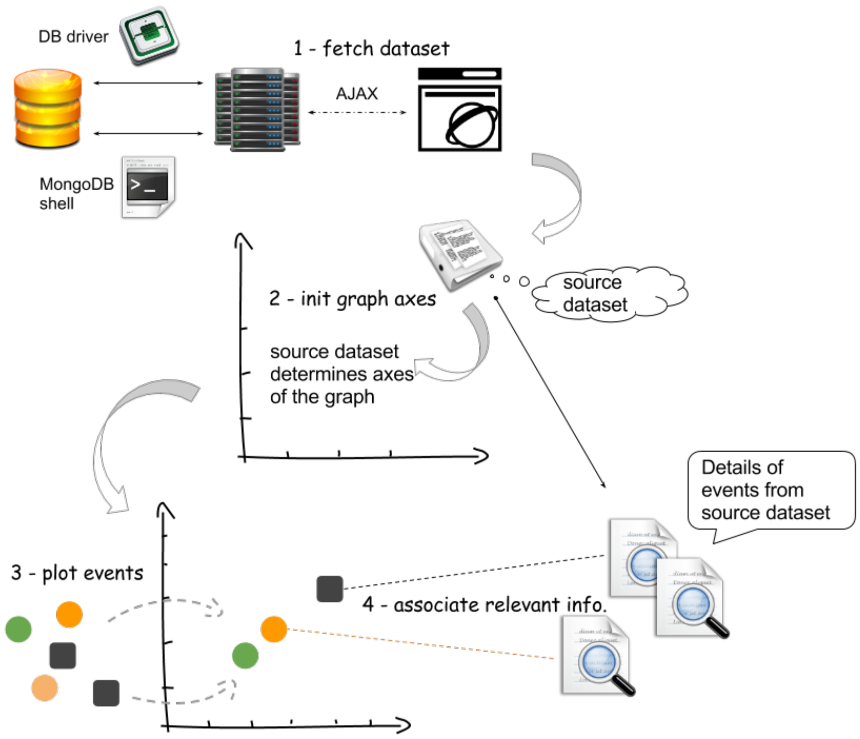

DB driver
1 - fetch dataset
AJAX
MongoDB shell
2 - init graph axes
source dataset
source dataset determines axes of the graph
Details of events from source dataset
3 - plot events
4 - associate relevant info.

Figure 4.3: Visualisation Procedure

are used for setting up the horizontal axis of the graph. Besides, basing on the type of the graph, Apps' names or system call objects are needed to initialise the vertical axis of the graph. Upon the boundaries of axes are set, corresponding intervals on each axis will be calculated so that the graph can present an understandable and clear view. That is to say, a proper interval is chosen so that events and description text will not overlap.

Once the axes of the graph is ready, events can be plotted onto the graph depending on their properties. Intuitively, the time property of an event is used to determine the horizontal position of the event. Whereas, the index property which can be the App name, process ID etc. is used to determine the vertical position. Therefore, every event will have a set of coordinates in the scale of the graph. In addition, to distinguish different events, the color, shape and size of different events are calculated in respect to events properties like event type, number of events at the given time, system call object and so on. Recall that the actual graph shows not only events but rather the series of events. Wherein the series of events are formed using three source logs *events*, *main* and *system* and a complementary file *packages.list*.

Firstly, all installed App names are extracted from *packages.list*, then each of the App names is used to form an initial selection statement to find processes that of the App specified by the name. Due to the events that indicate the start or end of a process are logged in *events* buffer and those events contain App names as well as process IDs that are needed as reference for further queries, so the *events* collection are queried firstly with previously formed selection statement. The result of the query has two parts which are a list of process IDs related to the App specified in the query and another list that contains ever occurred system calls in respect to the same App. After that, with the detected process IDs, more events pertinent to specified App can be located in all the source logs. Thus, a refined selection statement is formed to include detected process IDs found in previous query and then applied to all source logs. Note that because *events* collection is queried twice and to avoid duplicate results, the selection statement with App names is removed in the second query to *events* collection.

After the queries finished, each App has a set of system calls that are ever used by the App. Then, for each App, all these system calls are further divided into three parts: system calls that indicate the start of a process, the ones represent the end of a process and the rest that happened during each process's life circle. Recall that the process ID can be found in objects that represent the start of a process, which means all system calls in the duration part can be assigned to a corresponding start system call and an end system call. Seemingly, the result of this assignment is the series of events pertaining to each App and each execution period. Figure 4.4 demonstrates the grouping procedure described above. Lastly, relevant information of each event that being plotted on the graph is associated with the graphical representation of that event. This is achieved by assigning the corresponding event id to the element in the generated DOM tree. In this way, each element in the HTML DOM that stands for an event then can be linked to its source data in the dataset, where all information about the event can be fetched.

4.4 Features & user interface

It is of utmost importance to allow users be able to choose what should be displayed on the screen in a timeline analysis tool. Therefore, the zoom and pan functionalities were implemented on the graphical presentations to enable users to have a closer/wider view of the timeline or pick a particular display period from the timeline. These functionalities are accomplished by a blend

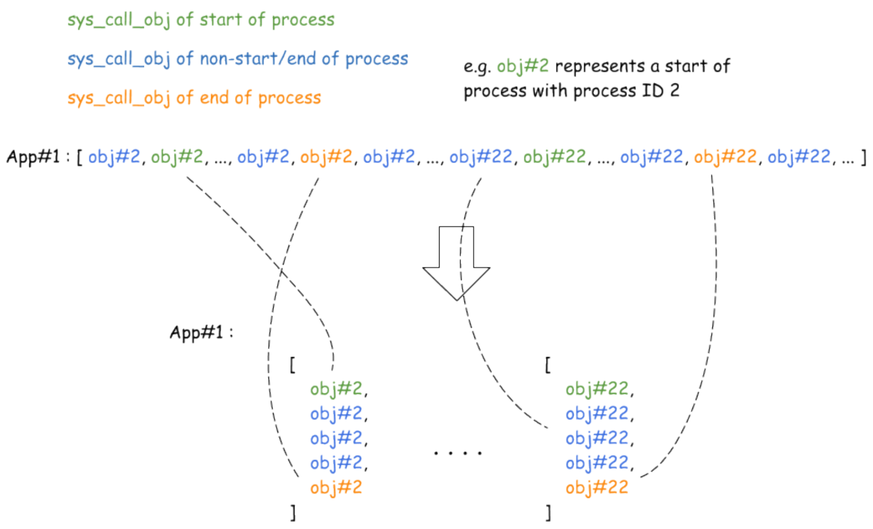
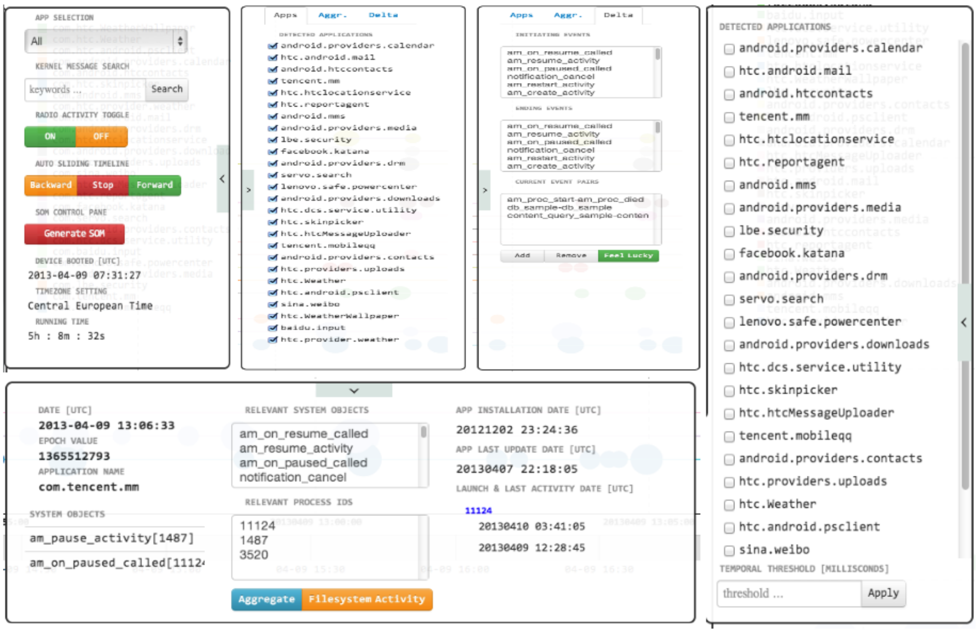

sys_call_obj of start of process
sys_call_obj of non-start/end of process
sys_call_obj of end of process
e.g. obj#2 represents a start of process with process ID 2
App#1 : [obj#2, obj#2, ..., obj#2, obj#2, obj#2, ..., obj#22, obj#22, ..., obj#22, obj#22, obj#22, ...]
App#1 :
[obj#2, obj#2, obj#2, obj#2, obj#2]
. . . .
[obj#22, obj#22, obj#22, obj#22, obj#22]

Figure 4.4: Form Of Event Series

of mouse events response and a time brush feature. Wherein the graphical presentations can respond to mouse actions applied on the graphics to zoom or pan the timeline. And the time brush provides a similar feature but in a quicker way as users can select a period for display and navigate to that period at once. Besides, an auto scrolling (both directions) of timeline was added to free users' hands in the case that the whole timeline has to be examined thoroughly. However, with zoom and pan are just not sufficient for a timeline analysis tool. There is yet another crucial feature has to be implemented, that is, the ability to append or remove specified events on the timeline. By appending new events on the timeline, it not only enables users to correlate different types of evidence to either find inconsistent evidence or coherent activities but also grants users the ability to decide what should be displayed on the timeline. On the other hand, by removing events from the timeline helps user to eliminate interference from trivial evidence and thus concentrate on important ones. So that there are five control panes embedded in the window. These panes provide interfaces where users can add or remove certain evidence on or from the timeline being presented. A screenshot is given in Figure 4.5.

In order to save window space so that timeline presentations could be as large as possible, these panes are hidden in the edges by default and control buttons are provided to control their display. All the operations from the panes are responsive. Thus the changes are applied to the timeline immediately without refreshing the page upon users' input. This ensures that users have the



DEVICE BOOTED [UTC]
2013-04-09 07:31:27
TIMEZONE SETTING
Central European Time
RUNNING TIME
5h : 8m : 32s

DETECTED APPLICATIONS
- android.providers.calendar
- htc.android.mail
- android.htccontacts
- tencent.mm
- htc.htclocationservice
- htc.reportagent
- android.mms
- android.providers.media
- lbe.security
- facebook.katana
- android.providers.drm
- servo.search
- lenovo.safe.powercenter
- android.providers.downloads
- htc.dcs.service.utility
- htc.skinpicker
- htc.htcMessageUploader
- tencent.mobileqq
- android.providers.contacts
- htc.providers.uploads
- htc.Weather
- htc.android.psclient
- sina.weibo
- htc.WeatherWallpaper
- baidu.input
- htc.provider.weather

INITIATING EVENTS
am_on_resume_called
am_resume_activity
am_on_paused_called
notification_cancel
am_restart_activity
am_create_activity

ENDING EVENTS
am_on_resume_called
am_resume_activity
am_on_paused_called
notification_cancel
am_restart_activity
am_create_activity

CURRENT EVENT PAIRS
am_proc_start-am_proc_died
db_sample-db_sample
content_query_sample-conten

DATE [UTC]
2013-04-09 13:06:33
EPOCH VALUE
1365512793
APPLICATION NAME
com.tencent.mm

SYSTEM OBJECTS
am_pause_activity[1487]
am_on_paused_called[1112

RELEVANT SYSTEM OBJECTS
am_on_resume_called
am_resume_activity
am_on_paused_called
notification_cancel

RELEVANT PROCESS IDS
11124
1487
3520

APP INSTALLATION DATE [UTC]
20121202 23:24:36
APP LAST UPDATE DATE [UTC]
20130407 22:18:05
LAUNCH & LAST ACTIVITY DATE [UTC]
11124
20130410 03:41:05
20130409 12:28:45

DETECTED APPLICATIONS
- android.providers.calendar
- htc.android.mail
- android.htccontacts
- tencent.mm
- htc.htclocationservice
- htc.reportagent
- android.mms
- android.providers.media
- lbe.security
- facebook.katana
- android.providers.drm
- servo.search
- lenovo.safe.powercenter
- android.providers.downloads
- htc.dcs.service.utility
- htc.skinpicker
- htc.htcMessageUploader
- tencent.mobileqq
- android.providers.contacts
- htc.providers.uploads
- htc.Weather
- htc.android.psclient
- sina.weibo

TEMPORAL THRESHOLD [MILLISCONDS]

Figure 4.5: User Interface of Control Panes

minimum waiting time and therefore their thoughts can be consecutive. These operations include append radio network activities on the timeline to correlated activities from Apps and cellular network events; append file system activities to verify consistence of file modification and App operations; append or remove Apps/processes on/from the timeline; and especially for the Deltatime View, event pairs of interest can be fully defined by users so that users are able to to analysis delta times between various event combinations case by case. Though this approach requires extra efforts from users but it can avoid possible misses caused by either white list or black list filtering.

4.5 Summary

In this chapter, the key techniques used in this implementation were discussed along with the reasons why these techniques had been chosen. In essence, the characteristics of Android system is the key factor in consideration of which techniques should be applied to best eliminate negative impacts caused by that system and to leverage the useful aspects of it for forensic purposes. The visualisation techniques were chosen mainly due to its high flexibility, reasonable responsiveness and huge potential in providing better graphical representations in foreseeable future.

Chapter 5

Evaluation

This framework has been evaluated under two empirical malware infection scenarios for its effectiveness. Wherein the first case is of a malware that sends IMSI to a remote server and communicates with sort of C&C server for further operations. And in the second case the malware stealthily registers the victim mobile subscription to some services that will cause significant financial loss to the user. The registration is done by SMS (cellular network) without user's admission. Noteworthy that to avoid financial loss as well as to have potential threats in control, all evaluations were done in an emulator created via Android Development Kit. The emulator runs Android 2.3.3 system which is theoretically "old" but the dominated system version at present. Between two evaluations, the emulator was swapped thoroughly to avoid interference. And the malware samples were obtained from Android Malware Genome Project [ZJ12].

5.1 LoveTrap

The setup of first evaluation is demonstrated as below. Firstly the emulator aforementioned was started freshly so only system Apps were existed. Then, those Apps were used randomly to fill up the log buffers. After that *LoveTrap* was installed to the emulator via *ADB* shell command and executed upon installation. The *LoveTrap* is pretended to be simple game between lovers, but once it gets running it starts sending device info to a remote server. Besides, it has a Client/Server mechanism so that the App keeps communicating with a remote server. However, the server had been swapped out so the content of their communication left unknown in this evaluation and only error messages recorded in the log. After left the *LoveTrap* running for some time, scripts from this framework were used to extract logs from the emulator and then save them into MongoDB. At this point, the setup of the evaluation was done and so the timeline analysis began.

Figure 5.1 shows the series of events related to *LoveTrap* (dark blue circles) on Timeline View. And the first isolated blue circle represents the installation of *LoveTrap* and the actual execution starts by the second blue circle. Wherein the radius of circles indicates the number of events happened at the given time. The details of installation will be shown when users put mouse over a circle as demonstrated in Figure 5.2. As a comparison, series of events from system Apps are given in Figure 5.3. Roughly, activities in Figure 5.3 followed similar patterns as the device user opened an App, made some operations and quitted it. With normal human operations, the intervals between consecutive activities are rather random, and the number of events within a given activity also varies. On the other hand, as shown in Figure 5.1, there are quite a few activities have exactly same radius (number of events) and the intervals are mostly identical. Therefore, the Deltatime View can be utilised to have a concentrated view over those intervals. Besides, in Deltatime View similar events will be group and further distinguished by colors. Figure 5.4 and Figure 5.5 illustrate two deltatime

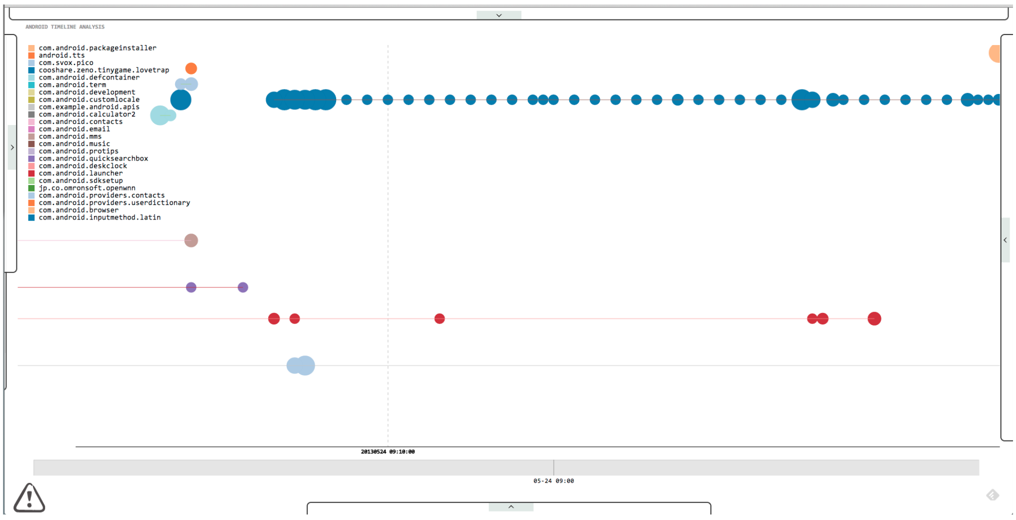
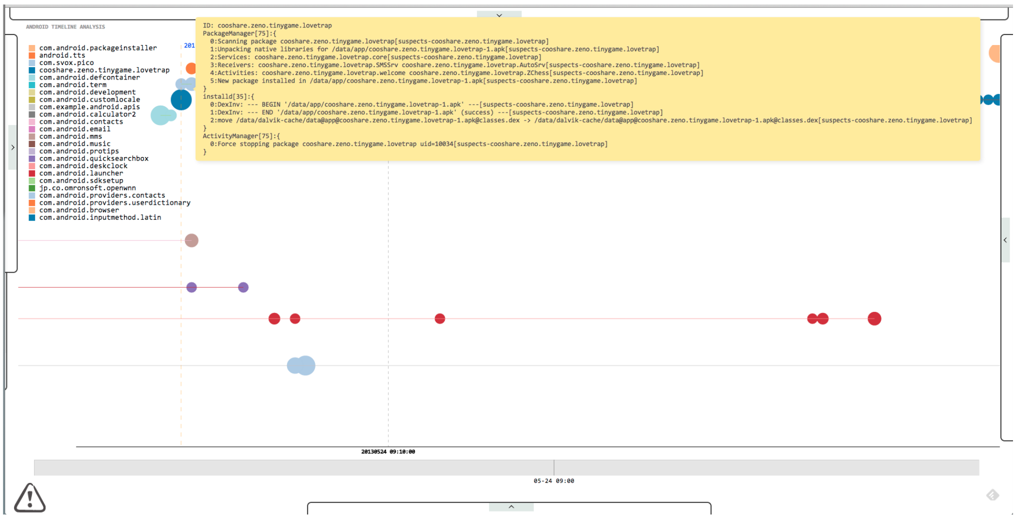


Figure 5.1: LoveTrap Timeline



```
ID: cooshare.zeno.tinygame.lovetrap
PackageManager[75]:{
  0:Scanning package cooshare.zeno.tinygame.lovetrap[suspects-cooshare.zeno.tinygame.lovetrap]
  1:Unpacking native libraries for /data/app/cooshare.zeno.tinygame.lovetrap-1.apk[suspects-cooshare.zeno.tinygame.lovetrap]
  2:Services: cooshare.zeno.tinygame.lovetrap.core[suspects-cooshare.zeno.tinygame.lovetrap]
  3:Receivers: cooshare.zeno.tinygame.lovetrap.SMSSrv cooshare.zeno.tinygame.lovetrap.AutoSrv[suspects-cooshare.zeno.tinygame.lovetrap]
  4:Activities: cooshare.zeno.tinygame.lovetrap.welcome cooshare.zeno.tinygame.lovetrap.ZChess[suspects-cooshare.zeno.tinygame.lovetrap]
  5:New package installed in /data/app/cooshare.zeno.tinygame.lovetrap-1.apk[suspects-cooshare.zeno.tinygame.lovetrap]
}
installd[35]:{
  0:DexInv: --- BEGIN '/data/app/cooshare.zeno.tinygame.lovetrap-1.apk' ---[suspects-cooshare.zeno.tinygame.lovetrap]
  1:DexInv: --- END '/data/app/cooshare.zeno.tinygame.lovetrap-1.apk' (success) ---[suspects-cooshare.zeno.tinygame.lovetrap]
  2:move /data/dalvik-cache/data@app@cooshare.zeno.tinygame.lovetrap-1.apk@classes.dex -> /data/dalvik-cache/data@app@cooshare.zeno.tinygame.lovetrap-1.apk@classes.dex[suspects-cooshare.zeno.tinygame.lovetrap]
}
ActivityManager[75]:{
  0:Force stopping package cooshare.zeno.tinygame.lovetrap uid=10034[suspects-cooshare.zeno.tinygame.lovetrap]
}
```

Figure 5.2: LoveTrap Activity Details

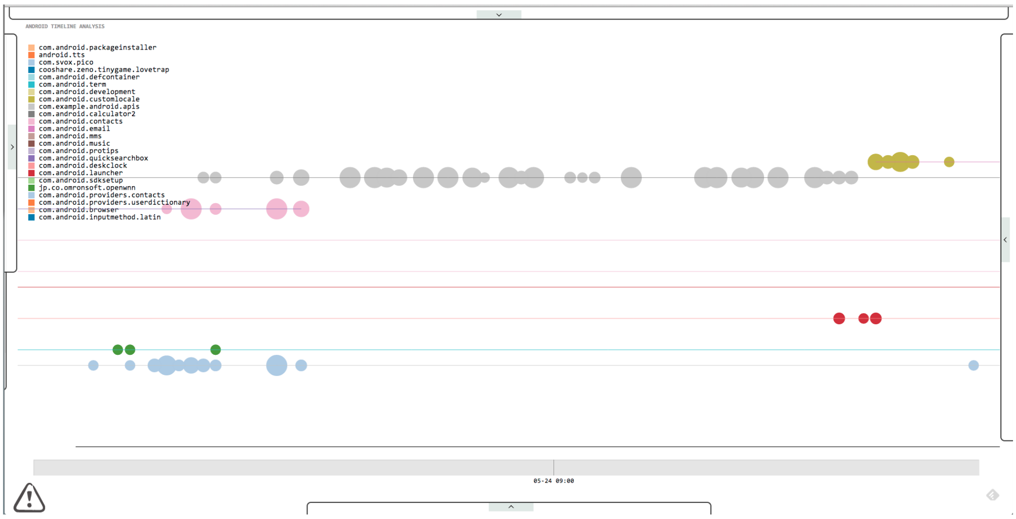


Figure 5.3: System App Activity Timeline

views where the first one is from a system App with human interactions and the latter is from the *LoveTrap*. Significant difference can be located based on the two differed graphs. Wherein for system App, though lots of activities happened at the same time but the details are different as indicated by their colors. Also there are a few activities had serveral different intervals and the number of same events are well balanced (indicated by the length of each rectangle). To the contrast, as can be seen in Figure 5.5, abundance of identical events happened with the same interval. And this is the signal that those events are very likely not triggered by human operations. Then the details of those events can be inspected as demonstrated in Figure 5.6. From the details one can then tell there should be a process running in background started by the App *LoveTrap* and that process was trying to connect to a server repeatedly at a given rate with no sucess. The point of this analysis is not to give an answer that what this malicious App does but to rise the awareness of this App does something abnormal. The further examination or analysis of the malicious App is out of scope of this work.

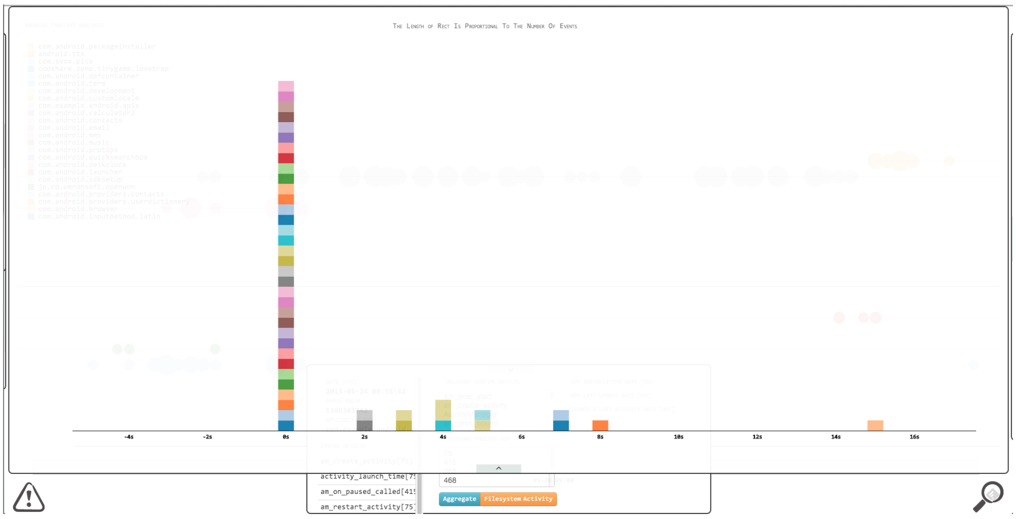
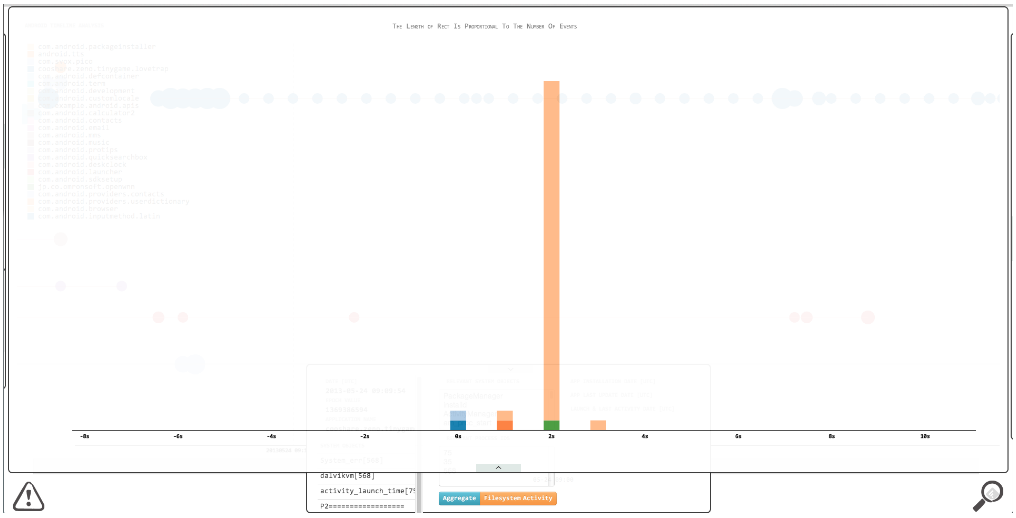


Figure 5.4: System App Deltatime Distribution

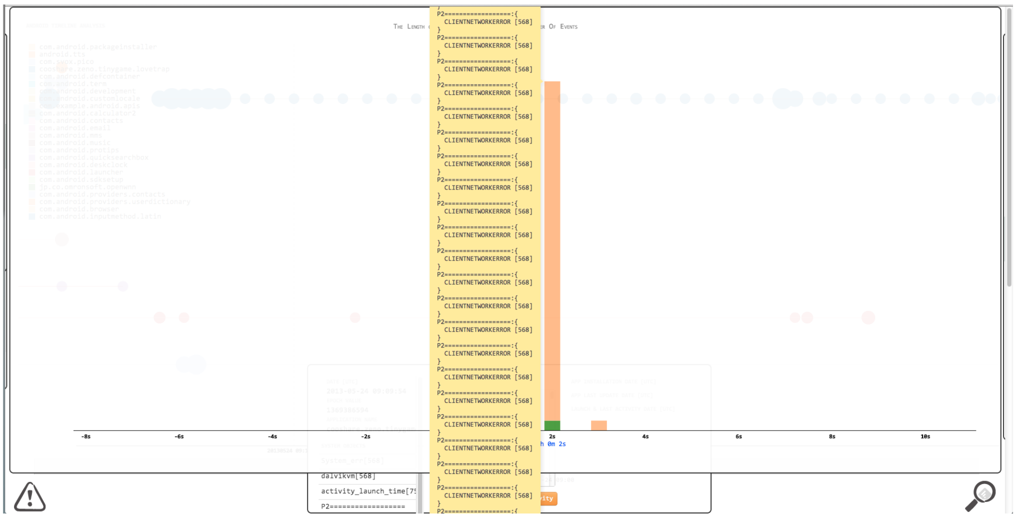
5.2 FakePlayer

The setup of the second evaluation is mostly identical to the first one, but the emulator was swapped and rebooted and the malware for this evaluation is called *FakePlayer* which is an App that claims it is a media player in the App store. Nonetheless, after the installation the App names itself “PornPlayer” to cheat users to open it. Once users clicked the launcher the App starts without any windows or user interfaces. But, two SMSes will be sent out to some services for registering premium account. And that premium membership will charge



```
dalvikvm[568]
activity_launch_time[75
P2=================
```

Figure 5.5: LoveTrap Deltatime Distribution



```
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
  CLIENTNETWORKERROR [568]
}
P2==================:{
```

```
dalvikvm[568]
activity_launch_time[75
P2=================
```

Figure 5.6: LoveTrap Deltatime Distribution

a great amount of money to the subscribers. In this case the launch of the malware FakePlayer is coherent to a radio network activity – sending SMS. And the use of this framework as demonstrated below can help investigators to find such coherent activities.

As shown in Figure 5.7, cellular network activities particularly SMS events can be appended to the Timeline View as the black squares in the graph. From

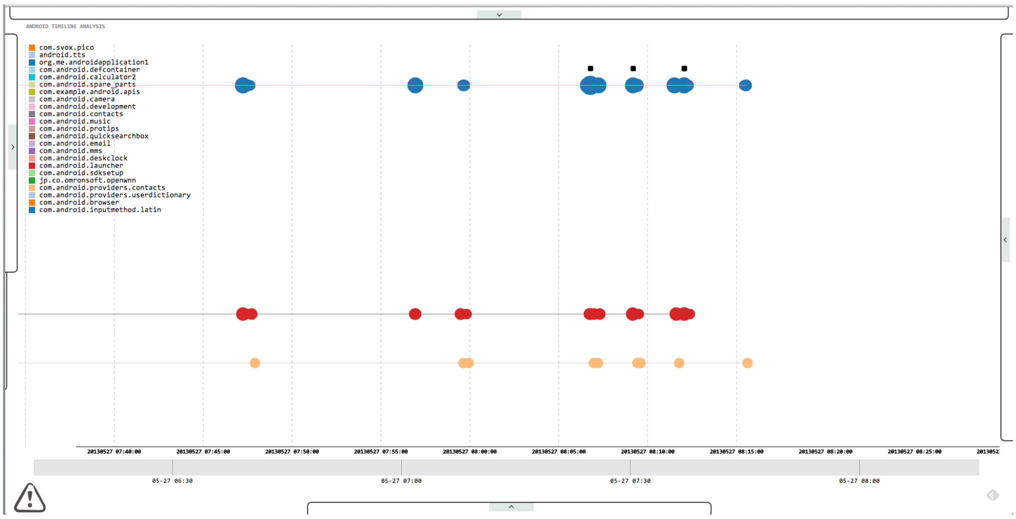
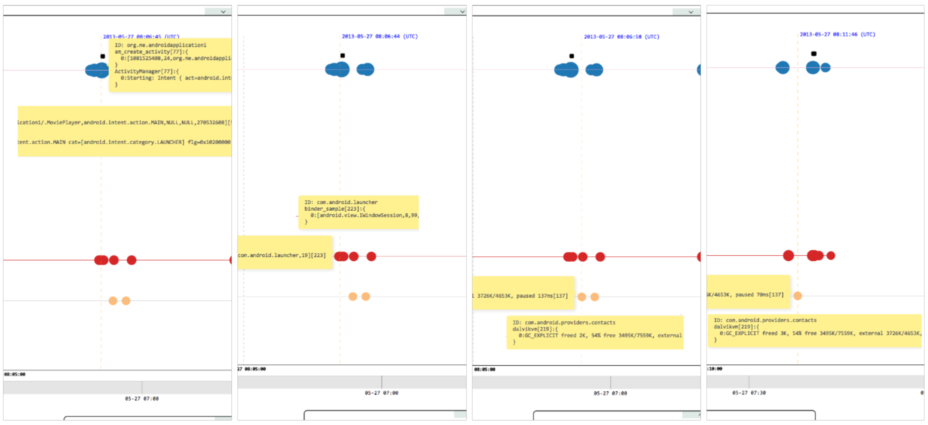


Figure 5.7: FakePlayer Timeline Overview

the first sight, it is deemed that the SMS activities have potential relations to three Apps: FakePlayer represented by the blue circles, Android.Launcher represented by red ones and lastly Android.Contacts represented by the yellow circles. Then a closer view for each SMS event can be achieved by zooming in on those events. And the view therefore yielded the following graphs as demonstrated in Figure 5.8. Note that irrelevant parts of those graphs were cut off for better demonstration here. As can be seen in these closer view, Android.Contacts can be firstly concluded having no connections with those SMS events because the occurrence sequence of SMS events and Android.Contacts activities do not match causality restrictions. And in the only exception captured in the graph (first graph from right), the activity from Android.Contacts is garbage collection which would not cause any SMS related events. Secondly, apart from the SMS events, the activities from FakePlayer and the ones from Android.Launcher are actually coherent. This is due to that the Android.Launcher records launches of Apps in the system. Thus, there is only one App left in this analysis and which should be further examined for sound evidence. In order to accomplish the task mentioned in previous step, the view was furthered zoomed in and then an occurrence pattern appeared. In Figure 5.9, graph *A-1* and *A-2* illustrate the pattern that an SMS will be sent after the launch of FakePlayer, then the App does nothing else but waits for termination. Besides, it is clear that graph *B-1* and *B-2* have a nice compliance with the pattern induced previously.


2013-05-27 08:06:45 (UTC)
2013-05-27 08:06:44 (UTC)
2013-05-27 08:06:58 (UTC)
2013-05-27 08:11:46 (UTC)
08:05:00
05-27 07:00
05-27 07:30
:10:00

```
ID: org.me.androidapplication1
am_create_activity[77]:{
  0:[1081525408,24,org.me.androidappli
}
ActivityManager[77]:{
  0:Starting: Intent { act=android.int
}
```

```
lcation1/.MoviePlayer,android.intent.action.MAIN,NULL,NULL,270532608][
tent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000
```

```
ID: com.android.launcher
binder_sample[223]:{
  0:[android.view.IWindowSession,8,99,
}
```

```
com.android.launcher,19][223]
```

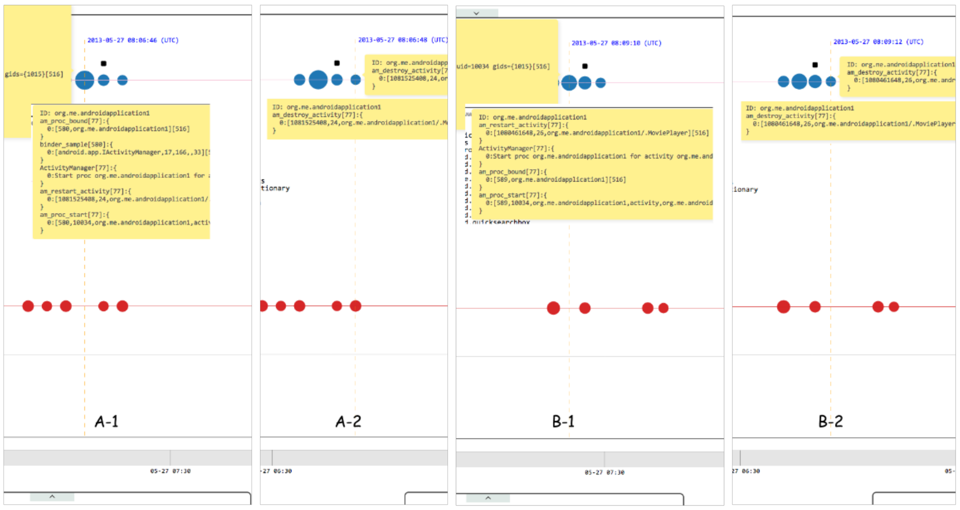
```
l 3726K/4653K, paused 137ms[137]
```

```
ID: com.android.providers.contacts
dalvikvm[219]:{
  0:GC_EXPLICIT freed 2K, 54% free 3495K/7559K, external
}
```

```
K/4653K, paused 70ms[137]
```

```
ID: com.android.providers.contacts
dalvikvm[219]:{
  0:GC_EXPLICIT freed 3K, 54% free 3495K/7559K, external 3726K/4653K,
}
```

Figure 5.8: Coherence Comparison Between Candidates


2013-05-27 08:06:46 (UTC)
2013-05-27 08:06:48 (UTC)
2013-05-27 08:09:10 (UTC)
2013-05-27 08:09:12 (UTC)
A-1
A-2
B-1
B-2
05-27 07:30
-27 06:30
05-27 07:30
27 06:30
05-

```
gids={1015}[516]
```

```
ID: org.me.androidapplication1
am_proc_bound[77]:{
  0:[580,org.me.androidapplication1][516]
}
binder_sample[580]:{
  0:[android.app.IActivityManager,17,166,,33][
}
ActivityManager[77]:{
  0:Start proc org.me.androidapplication1 for
}
am_restart_activity[77]:{
  0:[1081525408,24,org.me.androidapplication1/
}
am_proc_start[77]:{
  0:[580,10034,org.me.androidapplication1,acti
}
```

```
ID: org.me.androidapp
am_destroy_activity[7
  0:[1081525408,24,or
}
```

```
ID: org.me.androidapplication1
am_destroy_activity[77]:{
  0:[1081525408,24,org.me.androidapplication1/.M
}
```

```
s
tionary
```

```
uid=10034 gids={1015}[516]
```

```
ID: org.me.androidapplication1
am_restart_activity[77]:{
  0:[1080461648,26,org.me.androidapplication1/.MoviePlayer][516]
}
ActivityManager[77]:{
  0:Start proc org.me.androidapplication1 for activity org.me.and
}
am_proc_bound[77]:{
  0:[589,org.me.androidapplication1][516]
}
am_proc_start[77]:{
  0:[589,10034,org.me.androidapplication1,activity,org.me.android
}
```

```
d.quicksearchbox
```

```
ID: org.me.androidapplication1
am_destroy_activity[77]:{
  0:[1080461648,26,org.me.andr
}
```

```
ID: org.me.androidapplication1
am_destroy_activity[77]:{
  0:[1080461648,26,org.me.androidapplication1/.MoviePlayer
}
```

```
ionary
```

Figure 5.9: Events Occurrence Pattern

This evaluation demonstrated that the framework is capable of showing coherent activities between running Apps and cellular network events. This is an important feature for inspectors as they can quickly stick out irrelevant Apps and thus put more focus on suspicious ones.

5.3 Reliability discussion

Though the framework showed promising results in the above evaluations, but these results are not enough to eliminate the concerns of the reliability for this framework. In fact, there are scenarios that can have negative impacts on the effectiveness of the analysis.

For instance, if there are quite a lot applications are running at the same time, then this kind of runtime environment may obfuscate the delta time analysis or the attempts to find coherent activities (as demonstrated in the second evaluation). This is because that the Android system allows multiple processes to run in the background at the same time but as for mobile devices, the computing resources are quite constraint, like memory space or CPU time span etc.. Therefore, the more running processes in the system, the more frequent the Garbage Collection (a.k.a. GC) will be conducted by the system. And this means that processes or services running in the background will be terminated and started again later at some point. Seemingly this phenomenon, if severe enough, will cause the sort of time skew in the intervals, and thus the delta time analysis will suffer as the regularly repeated events may disappear. And it will be harder to locate coherent activities with the time skew as well because the skew may lead to the situation that more applications may seem to be coherent with the event of interest chronologically.

Besides, as already noticed by the author, the logging format in Android system is rather ambiguous so the series of events approach could fail if not enough information can be obtained from the log. This leaves the risk that important system calls that invoked by the malware may not be bound to that malware during analysis and so the malware may look "innocent" in this case. The countermeasure for this scenario is to conduct the Aggregation View on all "innocent" suspects to have a complete view of system calls. What's more, basing on the experience gained from the evaluations, it is worthy to mention that if the authors of malware do not leave log messages in the log buffers, there will be hardly no means to find the traces of the malware and thus the analysis on that malware can not be conducted. But if this situation ever occurs, it means the application may be deliberately hiding something, so other forensic methods can be used to inspect that application.

Apart from these, due to the current implementation of Deltatime View, the delta time analysis would fail if a pattern can not be seen by checking the intervals between every two consecutive events in the sequence, but rather it has to be detected in a more sophisticated manner. For example, given a sequence of events with indices, if there is a pattern only exists in the events with odd index, then the current implementation will not be able to detect such a pattern because the events with even index may well crumble the intervals between every two consecutive events to somewhat random state. Though there is still a chance to locate such patterns through Timeline View or Aggregation View with very intense inspection but it certainly requires an awful lot of efforts from the users and there is no guarantee that the pattern can be detected. In a nutshell, this kind of situations will immensely compromise the effectiveness of this tool and thus the reliability of the analysis results.

5.4 Summary

Summing up, based on the results from evaluations conducted above, it is deemed that this framework can reveal useful insights about the stochastic evidential dataset. Wherein particular patterns can be seen from the graphical presentation and coherent activities can be discovered. However, some important pieces of information might be overlooked because some data of unknown formats can not be parsed at present. Also, it is noteworthy that these evaluations were conducted in a way that the existence of malicious Apps is known. So to some extent that the analysis process in the evaluation can not be considered as it is supposed to be in a real investigation scenario.

Chapter 6

Conclusion

In this thesis, a framework for conducting timeline analysis targeting Android systems was implemented. The framework provides means to *a*) extract evidential artefacts in post-incident context; *b*) automatically gather logs and use them to nurture a neural network to generate activity classifications; and *c*) visualise extracted artefacts in four different perspectives for timeline analysis. One of the two main features that differentiates this framework from other timeline analysis approches is that the timeline of the target system as well as other forms of evidence presentation are indeed visualised and the graphics are rendered in the browser via HTML technology family. In addition, the second feature is the use of a centralised web server and thus the Browser/Server architecture which breaks the limitation of computational resources in conventional analysis tool, where the data processing can only be done in a single computer. What's more, this architecture also makes the collaborative investigation much easier as there is no need to replicate and distribute original evidence to each investigator.

Basing the results of evaluations, the graphical presentations can reveal nice impressions on behaviour patterns or coherent activities in the system, therefore presumably help inspectors to locate suspicious behaviours in an efficient way. However, due to the complexity of Android system and the variety of Android Apps, the implementation and evaluations had been done in this thesis are very primitive. From the experience ever gained in the development of this framework, especially the part that deals with parsing log contents, a concrete conclusion can be made that to develop an effective timeline analysis tool targeting Android system requires not only tremendous amount of time and efforts but also expertise in Android system. In addition to that, as the Android system is evolving or being customised rapidly over time and the amount of Apps available in the market is ever growing, an once and for all development for such analysis tools is nearly impractical. Moreover, it is worthy to mention that though the implementation of this work had tried to avoid using *root* privilege in Android system as having *root* privilege on all devices are not guaranteed. But the final implementation is not a compliance of that requirement because reading log contents from an Android App has to be done with *root* privilege. Lastly, recall the fact that log buffers are quite small so the evidence can therefore be easily overwritten if not preserved in somewhere else. This intrinsic setting of Android systems leaves high uncertainty in evidence collection procedure and thus may have huge negative impacts on analysis results.

Due to time and hardware constraints, not all evidence extracted from Android system are visualised or even used in analysis and some data formats that appeared in newly released Android systems may not be supported in this work. Thus, if given more time on this project, more experiments should be conducted to gain better understanding of Android system and the contents of logs. For instance, the kernel log of the system which can surely provide valuable information for correlation. In addition, due to the ambiguous logging formats used by various Apps, more advanced text searching methods, expectedly with linguistic intelligence, should be developed to further interpret the contents of logs. This is of utter importance for forensic analysis as valuable information may be neglected due to the lack of understanding of log messages. On top of that, there are spaces to try out more graphical presentations like a coherent occurrence matrix, which can be used to establish connections between high-level events (from Apps) and low-level events (system or kernel level events). By the time this thesis was being written, it had been reported that some malicious Apps can only be activated under certain physical environments like being exposed under particular radio frequencies or brightness of the environment. For the sake of this emerging scenario, the occurrence matrix would help the inspectors to find the "coincidence" that substantially triggers the abnormal behaviours.

In essence, this work, to its current development state, should only be considered as a proof of concept implementation. However, it indeed proves that the timeline analysis on Android systems is feasible to achieve and the techniques and architecture used by this work are capable for the given tasks. More importantly, this work signifies that the effectiveness of such a graphical timeline analysis tool can be promising in the filed. What's more, this work also shows that a post hoc analysis on Android systems may not be accurate as important evidence may be overwritten due to the small capacity of log buffers. Thus, a real time approach like the App implemented in this work that monitors the devices of interest and periodically collects log messages to preserve the evidence for later analysis is recommended for Android systems. Last but not least, this work denotes that a framework which can be expanded to support more types of evidence source is more appropriate than a single tool because the former can best fit the large variety and rapid development of Android life circle.

Appendix A

Android App Code Snippet

```
package imcom.forensics.extractors;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import android.content.ContentResolver;
import android.content.Context;
import android.database.Cursor;
import android.net.Uri;
import android.util.Log;
import imcom.forensics.EscapeWrapper;
import imcom.forensics.Extractor;
import imcom.forensics.FormatHelper;

public abstract class GenericExtractor implements
    Extractor {
  protected final String extractor_name;

  protected Uri uri;
  protected String selection;
  protected String[] selection_args;
protected String sort_order;
protected String[] projection;
protected FormatHelper helper;

public GenericExtractor(String extractor_name) {
  this.extractor_name = extractor_name;

  this.uri = null;
  this.selection = null;
  this.selection_args = null;
  this.sort_order = null;
  this.projection = null;
}

public int extract(ContentResolver resolver, Context
    context, File dst_dir) {
  Log.d(LOG_TAG, extractor_name + "␣launches");
  Cursor cursor = null;
  BufferedWriter record_writer = null;

  if (helper != null && uri != null) {
    cursor = resolver.query(uri, projection, selection,
        selection_args, sort_order);
    Log.d(LOG_TAG, extractor_name + "'s␣done␣query,␣" +
        cursor.getCount() + "␣items");
  } else {
    Log.d(LOG_TAG, extractor_name + "␣-␣URI␣or␣Format␣
      not␣set");
    return -1;
  }

  if (cursor != null && cursor.moveToFirst()) {
    try {
      //SDWriter sd_writer = new SDWriter(context);
      record_writer = new BufferedWriter(
        new FileWriter(
          new File(
            dst_dir,
            //sd_writer.getStorageDirectory(
                investigation, tag),
            extractor_name + ".pjson"
          )
        )
      );
```

```
        String[] col_names = cursor.getColumnNames();
        int records_num = cursor.getCount();
        int col_num = cursor.getColumnCount();

        do {
          StringBuilder extracted_item = new
              StringBuilder();
          for (int index = 0; index < col_num; ++index) {
            String field_value = EscapeWrapper.nomarlize(
                cursor.getString(index));
            String formatted_field = helper.formatString(
                col_names[index], field_value);
            extracted_item.append(formatted_field);
            extracted_item.append("␣");
          }
          record_writer.write(extracted_item.toString().
              trim());
          if (!cursor.isLast()) record_writer.newLine();

        } while (cursor.moveToNext());

        if (record_writer != null) record_writer.close();

        Log.d(LOG_TAG, extractor_name + "␣−␣fetches␣" +
            records_num + "␣records");
        return records_num;

        /*} catch (TimelineException e) {
        // TODO Auto−generated catch block
        e.printStackTrace();
        Log.e(LOG_TAG, e.getMessage());
        return −1;*/
      } catch (IOException e) {
        // TODO Auto−generated catch block
        e.printStackTrace();
        Log.e(LOG_TAG, e.getMessage());
        return −1;
      } finally {
        if (cursor != null) cursor.close();
      }
    } else {
      Log.d(LOG_TAG, extractor_name + "␣−␣Empty␣query...␣
          Query␣uri:␣" + uri);
```

```
            return -1;
        }
    }
}
```

Listing A.1: GenericExtractor Class

Appendix B

Scripts Snippets

```
#!/bin/bash

## shell script for parsing body file and inode
    activities

if [ $# -lt 4 ]
then
    echo 'Usage: '$0' body_file disk_image output_dir
        start_time(yyyy-mm-dd)'
    exit -1
fi

# using [0-9] is more generic
echo $4 | egrep "[0-9]{4}-[0-9]{2}-[0-9]{2}" 1>/dev/null
   2>&1
if [ ! $? -eq 0 ]
then
    echo 'Invalid start time, date should be like `yyyy-
        mm-dd`'
    exit -1
fi

BODY_FILE=$1
```

```
DD_IMAGE=$2
OUTPUT=$3
START_TIME=$4

INODE_TIMELINE=$OUTPUT"/inode.tmp"
FS_TIMELINE=$OUTPUT"/fs.tmp"

# wait until the device is connected
echo -ne 'waiting for device connecting...\t'
adb wait-for-device
echo 'ok'

if [ ! -d $OUTPUT ]
then
    echo "creating directory... "$PWD"/"$OUTPUT
    mkdir -p $PWD"/"$OUTPUT
else
    echo $OUTPUT" exists!"
fi

# get timezone from the device
TIME_ZONE=`adb shell date +%Z | awk '{sub("\r$", ""); 
    printf "%s", $0}'`
echo 'device Timezone: '$TIME_ZONE

echo -ne 'start processing inode times...\t'
# remove the description on first line
mactime -b $BODY_FILE -d -y -z $TIME_ZONE $START_TIME |
    sed '1 d' > $FS_TIMELINE

inodes=`cat $BODY_FILE | grep -v 'Orphan' | awk -F '|' '{
    print $3}' | sort -n | uniq | awk '{printf "%s ", $0}'
    `

# create a clean inode timeline file
> $INODE_TIMELINE

for inode in $inodes
do
    # when set timezone, all time is set to UTC
    istat=`istat -z $TIME_ZONE -f ext $DD_IMAGE $inode`
    if [ $? -eq 0 ]
    then
        inode_time=`echo $istat |
            sed 's/␣('$TIME_ZONE')//g' |
            awk '{printf␣"%s,",␣$0}' |
            sed 's/Direct.*$//' |
            sed 's/Group.*uid/uid/' |
            sed 's/num.*Accessed/Accessed/'`
        echo $inode_time | grep 'Not' 1>/dev/null 2>&1
        if [ ! $? -eq 0 ]
        then
            line=`echo $inode_time | sed 's/Allocated/
                Allocated:␣1/'`
        else
            line=`echo $inode_time | sed 's/Not␣Allocated
                /Allocated:␣0/'`
        fi
        echo $line |
            awk '{$6␣=␣":"$8",";$8␣=␣"";$9␣=␣"";␣print␣$0
                }' |
            sed 's/Allocated/,allocated/' |
            sed 's/uid/,uid/' |
            sed 's/mode/,mode/' |
            sed 's/size/,size/' |
            sed 's/Accessed/,accessed/' |
            sed 's/File␣Modified/,file_modified/' |
            sed 's/Inode␣Modified/,inode_modified/' |
            sed 's/Deleted/,deleted/' |
            grep -v '0000-00-00' >> $INODE_TIMELINE #
                filter out meaningless records
    fi
done
echo 'ok'

echo -ne 'parsing␣and␣formatting␣timestamps...\t'

python fs_times.py $FS_TIMELINE $OUTPUT"/fs_time.json"
    1>&2

python inode_times.py $INODE_TIMELINE $OUTPUT"/inode_time
    .json" 1>&2

echo 'done!'
```

Listing B.1: File System Extractor

```
#!/usr/bin/python
```

```
'''
    python script for parsing body file generated via TSK
        fls tool

    Author: Yu Jin (imcom)
'''

import sys
import json
import codecs
import iso8601
import time

col_names = ['date', 'size', 'activity', 'perms', 'uid',
    'gid', 'inode', 'file']

mactime_file = open(sys.argv[1])
output_file = codecs.open(sys.argv[2], 'w', encoding = "
    utf-8")

content = mactime_file.read()

records = list(map(lambda x:x.split(','), content.strip()
    .split('\n')))

for record in records:
    json_dict = dict()
    for index, value in enumerate(record):
        if index == 0:
            date = iso8601.parse_datetime(value)
            timestamp = time.mktime(date.timetuple())
            json_dict[col_names[index]] = int(timestamp)
        elif index == 1: # convert the size from string
            to int for query selection purpose
            json_dict[col_names[index]] = int(value)
        else:
            json_dict[col_names[index]] = value
    json.dump(json_dict, output_file)
    output_file.write('\n')

mactime_file.close()
output_file.close()
```

Listing B.2: Bodyfile Parser

```
#!/usr/bin/python

'''
    python script for parsing inode activities and
        converting result to json
    source file is generated by TSK ils tool

    Author: Yu Jin (imcom)
'''

import sys
import json
from datetime import datetime
import time
import re

inode_file = open(sys.argv[1])
output_file = open(sys.argv[2], 'w')

date_mask = re.compile("^(?P<date>\d+-\d+-\d+ \d+:\d+:\d
    +)\s?.*$")

content = inode_file.read()

records = list(line.split(',') for line in content.strip
    ().split('\n'))

for record in records:
    json_dict = dict(map(lambda x:x.split(':', 1), record
        ))
    for key in json_dict:
        value = json_dict.pop(key).strip() # remove the
            old key, since in some cases the key has extra
             spaces
        date = date_mask.match(value) # get rid of
            tailing timezone e.g. (CEST)
        if date:
            value = int(time.mktime(datetime.strptime(
                date.groupdict()['date'], "%Y-%m-%d %H:%M
                :%S").timetuple()))
        if key.find('size') is not -1:
            value = int(value); # convert size from
                string to int
        json_dict[key.strip()] = value # remove extra
            spaces in the key string
    json.dump(json_dict, output_file)
    output_file.write("\n")

inode_file.close()
output_file.close()
```

Listing B.3: iNode Parser

Appendix C

Visualisation Code Snippet

```
/*
 *
 * Timeline class for generating timeline
 * Author: Yu Jin
 * Date: 2013-03-06
 *
 */

/*
 * Parameter:
 *       name -- specify the div to bear SVG element
 *
 */
function Timeline(name) {
    // static constant values
    this.name = name;
    //this.timeline_height = 850;
    var height_margin = 100;
    this.timeline_height = window.innerHeight -
        height_margin;
    //this.width = 1850;
    this.width = window.innerWidth - 50;
    this.color_scale = d3.scale.category20();
    // chronological path generator
    this.time_path = d3.svg.line()
        .x(function(d) { return d.x; })
        .y(function(d) { return d.y; })
        .interpolate("linear");
   /*Disable global name $ from jQuery and reload it into
        Zepto*/
    jQuery.noConflict();
    $ = Zepto;

    // OpenTip config
    Opentip.styles.tooltip_style = {
        stem: true,
        hideDelay: 0.2,
        delay: 0.3,
        tipJoint: "right",
        target: true,
        borderWidth: 0
    };
// init timeline SVG and properties
Timeline.prototype.initTimeline = function() {

    this.timeline = d3.select(this.name)
        .append("svg")
        .attr("width", this.width)
        .attr("height", this.timeline_height)
        .attr("class", "timeline-graph")
        .append("g")
        .attr("transform", "translate(10,␣10)"); //
            timeline margins

    this.timeline.append("svg:clipPath")
        .attr("id", "timeline-clip")
        .append("svg:rect")
        .attr("x", 0)
        .attr("y", 0)
        .attr("width", this.width)
        .attr("height", this.timeline_height);
}
// prepare raw dataset for display
Timeline.prototype.setDataset = function(dataset,
    path_dataset) {
    // only check for suspects for android logs
    var self = this;
```

```
this.path_dataset = path_dataset;
var _dataset = {};
// group data by 1st date, 2nd _id (application), 3rd
    object (system call)
dataset.forEach(function(data) {
    // these properties has nothing to do with date
    var _id = data._id;
    var object = data.object;
    var message = data.msg + "[" + data.level + "]";
    if (!_dataset.hasOwnProperty(data.date)) {
        _dataset[data.date] = {};
    }
    var date_group = _dataset[data.date];
    if (!date_group.hasOwnProperty(_id)) {
        date_group[_id] = {};
        date_group[_id].display = data.display;
        //date_group[_id].level = data.level;
        date_group[_id].content = {};
    }
    var id_group = date_group[_id];
    if (!id_group.content.hasOwnProperty(object)) {
        id_group.content[object] = [];
    }
    id_group.content[object].push(message);
});

// output data sample:
// data {
//      date: <timestamp>,
//      _id: <app name>,
//      display: <display_name>,
//      content: {<object> : [messages,...], <object>
    : [messages,...], ...}
// }
for (var timestamp in _dataset) {
    if (timestamp != 'undefined') {
        for (var record_id in _dataset[timestamp]) {
            if (record_id != 'undefined') {
                this.updateYDomain(record_id); //
                    form an app name array for Y-axis
                    domain
                var display_name = _dataset[timestamp
                    ][record_id].display;
```

```
                        //var level = normal_dataset[
                            timestamp][record_id].level;
                        this.dataset.push({
                            date: Number(timestamp),
                            _id: record_id,
                            display: display_name,
                            //level: level,
                            content: _dataset[timestamp][
                                record_id].content
                        });
                    }
                }
            }
        }

    // sort all events by date
    this.dataset.sort(function(x, y) {
        if (x.date <= y.date) return -1;
        if (x.date > y.date) return 1;
    });
   this.onDataReady();
}
// draw timeline on SVG
Timeline.prototype.onDataReady = function() {

    var self = this;
    // timeline coords, color and radius functions
    function x(d) {
        return d.date;
    }

    function y(d) {
        return d._id;
    }

    function color(d) {
        return d._id;
    }

    function radius(d) {
        return self.getMessageNumber(d);
    }
```

```
// convert epoch timestamp to date for d3 time scale
    and init display dataset
var display_dataset = [];
this.dataset.forEach(function(data) {
//this.dataset.slice(this.start_index, this.end_index
    ).forEach(function(data) {
     var display_data = {};
     var date = new Date(data.date * 1000); // convert
          to milliseconds
     display_data.date = date;
     display_data._id = data._id;
     display_data.content = data.content;
     display_data.display = data.display;
     display_dataset.push(display_data);
});
// get the entire time period
var start_date = display_dataset[0].date;
var end_date = display_dataset[display_dataset.length
     - 1].date;
var total_period = (Number(end_date) - Number(
    start_date)) / 1000;

// define circle radius scale
var radius_range = [10, 20];
var radius_scale = d3.scale.pow()
     .exponent(2)
     .domain(this.initRadiusDomain())
     .range(radius_range)
     .clamp(true);

// define Y axis scale
this.y_scale = d3.scale.ordinal()
     .domain(this.y_domain)
     .rangePoints(this.y_range, 1.0);

// define X axis scale
this.x_range = this.initXRange();
this.x_scale = d3.time.scale.utc()
     .domain([start_date, end_date])
     .range(this.x_range);

// init tick interval on X axis
this.initTickInterval();
// define X axis
```

```
var x_axis = d3.svg.axis()
    .scale(this.x_scale)
    .orient("bottom")
    .ticks(this.tick_unit, this.tick_step)
    //.ticks(d3.time.minutes.utc, 5) // static test
        config
    .tickPadding(this.tick_padding)
    .tickSize(0);

// define X axis label format
x_axis.tickFormat(function(date) {
    formatter = d3.time.format.utc("%Y%m%d␣%H:%M:%S")
        ;
    return formatter(date);
});

// append X axis on Timeline
this.timeline.append("g")
    .attr("class", "time-axis")
    .attr("id", "timeline_main")
    .attr("transform", "translate(0,␣" + (this.
        timeline_height - 100) + ")")
    .call(x_axis);

// append gird on the timeline
var grid = this.timeline.selectAll("line.grid-main")
    .data(this.x_scale.ticks(this.tick_unit, this.
        tick_step))
    .enter()
    .append("g")
    .attr("clip-path", "url(#timeline-clip)")
    .attr("class", "grid-main");

// append lines on the grid
grid.append("line")
    .attr("class", "grid-line-main")
    .attr("x1", this.x_scale)
    .attr("x2", this.x_scale)
    .attr("y1", 0)
    .attr("y2", this.timeline_height - 100);
// ... ///
// append a layer for File / Radio activities
this.extra_arena = this.timeline.append('g')
    .attr("id", "extra-arena")
```

```
        .attr("clip-path", "url(#timeline-clip)");

    // append events on timeline
    this.timeline.append('g')
        .attr("id", "events-arena")
        .attr("clip-path", "url(#timeline-clip)")
        .selectAll("circle")
        .data(display_dataset)
        .enter()
        .append("circle")
        .attr("class", "timeline-event")
        .attr("id", function(data, index){
            return "dataset" + "-" + index;
        })
        .attr("cx", function(data) {
            return self.x_scale(x(data));
        })
        .attr("cy", function(data) {
            return self.y_scale(y(data));
        })
        .attr("r", function(data) {
            return radius_scale(radius(data));
        })
        .attr("fill", function(d) {
            return self.color_scale(color(d));
        })
        .sort(function(x, y) {return radius(y) - radius(x
           )});

    // draw chronological paths on timeline for
        application traces
    if (this.path_dataset !== null) {
        for (var app in path_dataset) {
            if (app === undefined) continue;
            path_dataset[app].forEach(function(path_group
                ) {
                 self.fillPathData(path_group);
                 self.drawPath();
            });
        }
    }

    // append graph legend (application names)
    var text_padding = 30;
    var legend = this.timeline.selectAll(".legend")
        .data(this.y_scale.domain().reverse())
        .enter().append("g")
        .attr("class", "legend")
        .attr("transform", function(d, i) { return "
           translate(0," + i * 14 + ")"; });

    legend.append("rect")
        .attr("x", 10)
        .attr("width", 12)
        .attr("height", 12)
        .style("fill", this.color_scale);

    legend.append("text")
        .attr("x", text_padding)
        .attr("y", 5)
        .attr("dy", ".35em")
        .style("text-anchor", "start")
        .text(function(d) { return d; });
} // function onDataReady()
```

Listing C.1: Timeline Class

Bibliography

[BF05] Florian Buchholz and Courtney Falk. Design and implementation of zeitline: a forensic timeline editor. In *Digital forensic research workshop*, 2005.

[Bos12] Michael Bostock. D3.js - data-driven documents. `http://d3js.org/` [Visited 31 May 2013], 2012.

[BT07] Florian Buchholz and Brett Tjaden. A brief study of time. *digital investigation*, 4:31–42, 2007.

[Car05] Brian Carrier. *File system forensic analysis*, volume 3. Addison-Wesley Boston, 2005.

[Car13a] Brian Carrier. The sleuth kit and the autopsy forensic browser. `http://www.sleuthkit.org/autopsy/` [Visited 31 May 2013], 2013.

[Car13b] Brian Carrier. sleuthkit/sleuthkit. `https://github.com/sleuthkit/sleuthkit` [Visited 31 May 2013], 2013.

[Clo08] Michael Cloppert. Ex-tip: an extensible timeline analysis framework in perl. *Bethesda, MD: SANS Institute*, 2008.

[Fuc13] Thomas Fuchs. Zepto.js: the aerogel-weight jquery-compatible javascript library. `http://zeptojs.com/` [Visited 31 May 2013], 2013.

[GP05] Pavel Gladyshev and Ahmed Patel. Formalising event time bounding in digital investigations. *International Journal of Digital Evidence*, 4(2):1–14, 2005.

[Hoo11] Andrew Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Syngress, 2011.

[jF13] The jQuery Foundation. jquery. `http://jquery.com/` [Visited 31 May 2013], 2013.

[Lea11] LearnBoost. Mongoose odm v3.6.11. `http://mongoosejs.com/` [Visited 31 May 2013], 2011.

[Mas13] Giulia Massini. Visualization and clustering of self-organizing maps. In *Intelligent Data Mining in Law Enforcement Analytics*, pages 177–192. Springer, 2013.

[Men13] Matias Meno. Opentip | the free tooltip. `http://www.opentip.org/` [Visited 31 May 2013], 2013.

[Nar07] Gregorio Narváez. Taking advantage of ext3 journaling file system in a forensic investigation. *SANS Institute Reading Room*, 2007.

[OB09] Jens Olsson and Martin Boldt. Computer forensic timeline visualization tool. *digital investigation*, 6:S78–S87, 2009.

[Sle09] SleuthKitWiki. Body file - sleuthkitwiki. `http://wiki.sleuthkit.org/index.php?title=Body_file` [Visited 31 May 2013], 2009.

[SMC06] Bradley Schatz, George Mohay, and Andrew Clark. A correlation method for establishing provenance of timestamps in digital evidence. *digital investigation*, 3:98–107, 2006.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.