A Fully Automated Deep Packet Inspection Verification System with Machine Learning

Uday Trivedi
Samsung R&D Institute, Bangalore (SRIB)
Bangalore, India
uday.trivedi@samsung.com

Munal Patel
Samsung R&D Institute, Bangalore (SRIB)
Bangalore, India
munal.patel@samsung.com

Abstract— **Deep Packet Inspection (DPI) technique has become very important for traffic detection and resource management in core networks. DPI systems use unique byte patterns as signatures to detect application traffic. Applications frequently update their version to add new features and/or to bypass firewall/DPI systems. Thus, an accurate DPI system needs to periodically verify existing signatures and update them if required. The manual task of application traffic generation and verification on multiple platforms is very tedious and error-prone. We propose a fully automated DPI verification system with machine learning techniques for periodic DPI signature verification and update. Automated mobile application traffic generation is achieved by open source tools GUITAR and Appium. Signature verification and new signature pattern suggestion from undetected flows are achieved by well-known and custom made machine learning algorithms, thus completing full signature verification and update cycle. Initial test results show that our solution saves lot of man hours and detects signature update in shortest possible time.**

Keywords-component; DPI, Deep Packet Inspection, GUI Automation Testing, Automated Signature Generation

I. Introduction

The advent of 4G LTE is changing the landscape of data usage. LTE offers the promise of a pure, all-IP converged core network with high data rate. With "all-IP" networks, lot of traffic and resource management issues like managing P2P application traffic load, providing high QoS for priority flows etc. can crop up. Network operators need to prioritize their resources and adjust resource allocation schedule based on network congestion and application traffic. Thus, they must find effective ways to identify application traffic and apply different policies based on various parameters like network status, application traffic, subscriber profile, time of day etc.

Deep packet inspection (DPI) is a network packet inspection system which examines packet payloads to identify traffic flows. DPI system uses application specific unique signatures to identify application traffic. The working DPI solution must identify applications in real time. Thus, the DPI solution must to be fast, accurate and easy to upgrade.

A. Need For Automation

The mobile application development world is a dynamic world. Developers frequently update their applications with new features, encryption etc. Therefore, it is essential to frequently analyze and update identification mechanism to detect flows of newer versions of applications. However, signature identification and verification process is very rigorous [1]. Manual approach for this requires capturing application traffic for multiple platforms, and analyzing hundreds of TCP/UDP flows and verifying them with detection result for false positive/negative cases. For newer application versions, same procedure must be repeated to detect and verify new signatures. Such a painstaking manual approach is error-prone and it will not scale if it has to be applied individually to the growing range and number of diverse applications [2]. With growing number of mobile apps coming with local versions and their agile development life cycle, automation testing is the only way to go. This paper proposes a method which automates the entire process of application traffic generation and signature verification.

B. Related Work

Mobile application test automation domain has seen lot of research. However, all of the solutions are mainly focused on finding mobile application GUI or implementation specific issues. As per our knowledge, this is the first time an attempt is made to automate entire DPI signature verification process using mobile application test automation. Since no existing tool for automated DPI test verification is found, we have referred related work for mobile app test automation and automatic signature generation.

Automation testing is widely used to perform effective and fast testing of software products that have a long maintenance life. Many proprietary and paid mobile test automation tools provide record and playback features that allow testers to interactively record user actions on mobile and replay them back while testing, comparing actual results to those expected. This method requires little or no test script development. However, this method may pose major reliability and maintainability problems. Relabeling a button or changing location of window on mobile app may require the test to be re-recorded. For testing of web sites, Headless browsers or solutions based on Selenium Web Driver are normally used.

Researchers [3] state that to cope with frequent upgrades of mobile devices and technologies, an elastic infrastructure to support large-scale test automation is needed. Research work [4] on integrated test automation frameworks has proposed research tools at the system level that support mobile testing on multiple heterogeneous platforms.

Automatic signature generation has been a thoroughly researched domain. Want et al [5] used k-common substrings and multiple sequence alignment algorithms to generate regular expressions with good accuracy. Researchers Wang et al., [6] have proposed to use supervised machine learning models automatically learned from the data sets as application signatures. We have devised very simple signature generation algorithm which uses K-means unsupervised machine learning algorithm with statistical analysis method.

II. DPI Test Automation and Verification

Following Fig. 1 shows complete architecture of DPI automated verification system. Signature database keeps existing application signatures. CLI is used to run configuration commands for DPI module. Once automation script is executed, flow details are sent to Intelligent Analysis module which identifies relevant and irrelevant flows using detection database and with the help of engines like HTTP, HTTPS, TCP and UDP. Based on its outcome, system suggests new signatures or false positive cases. We target following manual tasks to be automated:

- Generating mobile application traffic for Android and iOS platforms.
- Generating application website traffic on mobile browser.
- Verifying existing signatures and finding undetected application flows and wrongly marked application flows.
- Suggesting new signatures from undetected flows and verifying those signatures with application traffic.

Our contribution in this paper is to automate the task of generating application traffic using open source tools, devising algorithms for verifying existing DPI signatures and suggesting new signatures from undetected flows using machine learning algorithms. Following sub-sections discuss each of these tasks in detail.

A. Automated Application Traffic Generation

Test automation on mobile device poses many challenges. For example, multiple platforms like Android, iOS, Windows mobile, Tizen etc. No single tool supports all platforms.

Screen size and resolution (Depth per Inch): Each mobile device comes with different screen size and resolution. The automation script should be able to work with all of them.

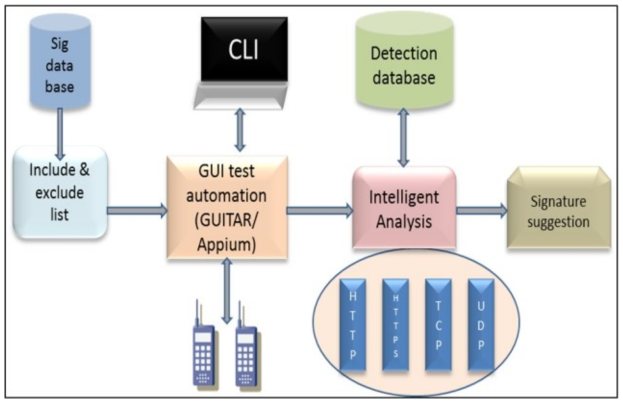

Figure 1: DPI Automated Verification System Architecture

Table 1: Comparison between mobile automation tools

Tool	Strengths	Limitations
Monkey Runner	Coordinate based automation. Easy to setup.	Supports only Android. Requires different scripts for different devices which can break with slight change in UI.
UI Auto-mator	Integrated with Android development IDE. Object based automation	Supports only Android. No support for web-view, long press.
Monkey Talk	Open source, eclipse based IDE. Supports iOS & Android apps.	Limited features in free version.
Appium [7]	Element based Navigation, Large Open source community	iOS Apps requires source/debug binaries, No image based navigation
GUITAR [8]	Open Source, Supports iOS & Android apps, Image based navigation	Small open source community, No Element based navigation, iOS requires rooting of device

Table 1 above shows comparison between popular mobile automation tools.

We had following criteria for selecting automation tool:

- Automation support for Android/iOS/Both
- Open source/proprietary/paid tool
- Coordinate /image/Object based automation

We analyzed multiple tools for their strength and limitations. Finally, we selected Appium and GUITAR for Android and iOS automation respectively.

1) Automated App Traffic Generation on Android

Appium is an open source test automation tool for mobile applications. It allows running the automated tests on actual devices, emulators and simulators. Appium tool does not modify or recompile the app on Android platform. Appium works on elements present on screen which can be accessed with APIs provided by respective OS. Once an element is selected, we can perform various gestures on it. Appium works as a server between test case script and application running on mobile device. Appium explores capabilities of Android to access view hierarchy represented with DOM (Document Object Model). While designing test cases, IDs from DOM can be used and operations can be performed on it. Appium supports all major operation on touch screen like touch, swipe, zoom, long press, etc.

Appium has few limitations as well. Appium does not support image comparison on mobile application GUI for performing gestures. Thus, if any element in given app gets changed, Appium script needs to be re-written. Appium requires app's developer image for iOS app automation. Since DPI task involves working on 3rd party apps, getting developer image is not possible. Thus, we selected GUITAR for iOS test automation.

2) Automated App Traffic Generation on iOS

GUITAR is a GUI driven mobile test automation framework based on Image search. GUITAR framework searches images on attached mirrored mobile device window and performs gestures on it. It supports all major gestures like touch, swipe, zoom, long press, text input, etc. GUITAR scripts are written with associated application specific images. Using OpenCV library, system compares images based on given tolerance value. The tolerance value helps in matching images having slight change in different versions.

As GUITAR works on application window on screen, we need to mirror iOS mobile device to test system with control capabilities. Since iOS does not provide any means to control device connected to a system, we need to root iOS device and use desktop sharing tools like VNC to control device. GUITAR supports multiple mobile devices at the same time so that we can run VoIP apps on both devices for VoIP call/chat.

B. *Generating Application WebTraffic on Mobile Browser*

Many services do not have mobile applications and require mobile browsers to operate. Considering this, it is necessary to include verification for mobile browser based traffic as well. As browser based web interface tends to change a lot, conventional image or element based navigation may not provide complete website coverage. To solve this problem, we came up with an approach where a script uses features of chrome browser to access all links from webpages till nth level depth. This way we can generate traffic with different scenarios.

AutoIt [9] is a scripting language designed for automating Windows GUI and general operations. It uses a combination of simulated keystrokes, mouse movement and window/ control manipulation. The AutoIT script sets chrome browser to emulate particular mobile device browser. After that, the script opens base URL on browser. After the page is loaded, a JavaScript fetches all links from that HTML page. The script hits few random URLs from this list. There is a configuration option to ignore/select certain URLs based on keywords. This process runs till nth level or for certain period of time to generate mobile browser based traffic.

C. *Automated Application Signature Verification*

Application signature verification task aims to find new signatures and identify wrongly marked signatures. This task is very complex task due to the fact that an application/web page can have large number of "application unrelated" flows going to 3rd party advertisement/statistics sites. Also, application can have 3rd party CDN flows from which it may fetch audio/video/images related to its service. For example, many sites have option of sharing video to Facebook, Twitter etc. Some webpages download scripts from other services for utility functions like fonts, CSS etc. The verification system needs to classify application specific relevant and irrelevant flows. If that task is not done properly, the verification system will check detection result for irrelevant flows and will generate many false "false positive" alarms.

We solve this problem by keeping "exclude list" and "include list". The script parses signature database and copies unique application specific patterns into "include list". "Exclude list" has patterns from unrelated 3rd party generic flows like CDN/Advertisement/statistics sites, etc.

Table 2: Sample configuration file showing Include/Exclude list for Times of India App

```
[HTTPParser]
ExcludeList=facebook,doubleclick,appsflyer,yahoo,.flurry,microsoft,amazon
IncludeList=timesofindia, indiatimes
[SSLParser]
ExcludeList=doubleclick,facebook,appsflyer,yahoo,.flurry,microsoft,amazon
IncludeList=timesofindia, indiatimes
[UDPParser]
ExcludeList=1812,1813,3527
IncludeList=
[Application]
ExcludeList= GOOGLE, YOUTUBE, TWITTER
IncludeList= TOI
```

Before running GUI test case, verification script starts capturing PCAP file using tshark. Tshark is command line extension of Wireshark network protocol analyzer. GUI script opens test application on mobile and generates application traffic by performing gestures on various app scenarios. After GUI run, the script extracts relevant flows from saved PCAP file using "include" and "exclude" lists. Based on pattern matching from these lists, HTTP, SSL, TCP and UDP flows are categorized as relevant or irrelevant flow. Flows with SSL renegotiation do not have any signature pattern in payload. This case is handled by retrieving SSL session id from all relevant SSL flows and marking SSL flows having same session id as relevant flow. For UDP flows, port number in "include" and "exclude" list is used to mark relevant flows. All remaining TCP flows are checked for flow size and marked as relevant flow if it is above threshold size.

The script then generates list of relevant flows. The verification system uses a utility which reads detection database and provides DPI detection result. Verification system also has application rule Include/Exclude list. If a given flow is identified as application X and X is added in application rule Exclude list, then such flow can be marked as irrelevant flow with this configuration. Table 2 shows sample Include/Exclude list which is prepared with manual analysis as well as with self-learning algorithm.

After completing the analysis, the verification system gives list of false negative cases if remaining relevant application flow is marked as HTTP/HTTPS/Undetected and not with the tested application id A. Similarly, verification system gives list of false positive cases if relevant application flow is marked as any another application id B, instead of tested application id A. Relevant flows which are not detected properly are forwarded to automated signature generation module. Auto signature suggestion module gives unique application pattern suggestion for HTTP, HTTPS, TCP and UDP flows. For HTTP and HTTPS flows, it checks standard headers like "Host", "Referer", "Client Hello server name" etc. For TCP and UDP flows, it finds unique byte pattern(s) by comparing multiple similar flows.

Each flow from the PCAP files is classified into four categories: HTTP, SSL, TCP (other than HTTP and SSL) and UDP. Each flow from these categories is processed separately to identify relevant flows.

1. HTTP: HTTP flows can have unique signatures in HTTP "Host", "Referer", "User-Agent" header and/or inside URI

string. The script checks “include list” and marks all flows with matching pattern from “include list” in HTTP request packet as relevant flows. The script checks “exclude list” as well and marks all flows with matching pattern from “exclude list” as irrelevant flows. From relevant flows, each server IP is selected and all HTTP flows having that server IP are also considered as relevant flows.

2. SSL: SSL flows can have unique signature in “Client Hello” and “Server Hello” Packets. The patterns in include and exclude list are used to mark flows as relevant and irrelevant flows like HTTP. SSL renegotiation case is handled by retrieving session id from all relevant flows and marking flows having same session id as relevant flow.

3. UDP: For UDP flows, port number is used in “include list” and “exclude list”. By default, port number <=1024 are excluded from relevant flows as they are used for standard protocol related flows. If any port <= 1024 is required for particular application, it can be added in “include list”. UDP flows above minimum size are considered as relevant flows.

4. TCP (Other than SSL and HTTP): All remaining TCP flows are checked for flow size and marked as relevant flow if it is above threshold size.

1) *Intelligent Analysis using Machine Learning Algorithms*

A machine learning algorithm tries to progressively learn from data to finally predict a desired outcome from a new, unknown set of data. Proposed verification system uses our custom made machine learning algorithm to automatically build irrelevant pattern database and learn more about possible weak signatures. Following techniques are employed:

a) Building irrelevant pattern database:

How does verification system know that a particular undetected flow is relevant flow or advertisement/ statistics/ Content Delivery Network (CDN) related flow? To correctly classify relevant and irrelevant flows, a robust irrelevant pattern database is required. However, one cannot build such database manually as there are millions of such links and applications keep on adding new statistical/advertisement/ CDN links in newer versions.

With our custom made algorithm, we build irrelevant pattern database from scratch by finding irrelevant flow with specific patterns getting detected in multiple application traffic. The system keeps track of relative frequency of such patterns from HTTP/HTTPS packets and keeps updating the entries to conclude about a given undetected flow. Table 3 shows sample irrelevant pattern database.

Table 3: Sample Irrelevant pattern database

Pattern	Flows detected	Apps detected	Prob. of irrelevant
doubleclick.com	30	3 (id x,y,z)	1
stats.com	6	1 (id w)	0.066

We compute probability of a pattern being part of irrelevant flow by following equation:

$$P = \frac{FlowsDetected \ * AppsDetected}{FlowThreshold * AppThreshold} \quad (1)$$

Where FlowsDetected is total flows detected with given pattern, AppsDetected is total apps detected with given pattern, FlowThreshold is flow threshold for marking the pattern as irrelevant pattern (around 30) and AppThreshold is total matching application threshold for marking the pattern as irrelevant pattern (around 3).

As noted, unique pattern(s) related to an application can be found from certain headers of HTTP/HTTPS packets. An irrelevant HTTP/HTTPS flow from application X will be marked as either HTTP or HTTPS if detection for application X is not supported by system. This is obvious as system does not have signature to detect flows of app X. If app X detection is supported, that irrelevant flow will be marked as application X. In both cases, verification system parses HTTP/HTTPS packet of such irrelevant flows and retrieves patterns from standard headers and adds them in irrelevant flow table. If that same pattern is repeated in multiple application traffic, the system concludes that it is generic flow and marks that pattern as irrelevant flow pattern.

Initially, only few manual entries are added in irrelevant flow pattern table. With more tests, this table is updated with more patterns. Once flow and app count reaches given threshold, given pattern is marked as irrelevant pattern and then onwards, flows having this pattern are marked as irrelevant flows. It should be noted that Google, Facebook, Twitter traffic is also found in majority of application traffic. However, they are already added in “exclude list” and thus, such traffic is not processed in this module.

For example, “doubleclick.com” pattern getting detected in multiple apps suggests that this pattern is part of a generic flow and should be part of “exclude list”. With each application test run, irrelevant pattern database grows and subsequent flow classification accuracy goes up.

b) Finding almost matching patterns

For certain application signature patterns, only one or two bytes might have changed in newer app version. Our algorithm finds such occurrences by finding almost matching pattern with longest subsequence matching algorithm. For example, we have initial signature pattern: “ABCDE” at offset X. During nth test, the system notices that pattern at offset X is “AFCDE” with change at X+1 offset (‘F’ instead of ‘B’). The system marks this offset as vulnerable offset and suggests regular expression “A[BF]CDE” as signature.

During subsequent tests for same app, if application flow is not detected, system checks bytes at vulnerable offset and finds longest matching subsequence as new regular expression pattern “A.CDE” where ‘.’ denotes any character. The system recognizes that last time there was a change in this pattern so there could be a further change at the same place.

D. *Automated Signature suggestion and Reverification*

Once AutoIT scripts find out relevant flows which are not detected by DPI system, such flow information is forwarded to next phase where automated signature generation module is running. This module examines payload of such flows and suggests unique byte pattern as possible signatures.

Auto signature suggestion module has following sub modules for various flow classes.

1. HTTP: The script gets "Host", "Referer" and "User-Agent" header values and removes patterns which are part of "exclude list". Remaining patterns are shown as possible signature patterns for given flows.
2. SSL/HTTPS: The script gets server name from SSL "Client Hello" packet and suggests that pattern as potential signature. If "Client Hello" packet has no server name and session id or session ticket value is non-zero, it is SSL renegotiation case. In SSL renegotiation case, SSL flow does not have SSL negotiation parameters like server name etc. as those parameters were exchanged in some linked flow earlier. The script finds the linked flow using session id/session ticket value, retrieves server name from that flow and suggests server name as potential signature.
3. TCP/UDP: For TCP/UDP flows on non-standard port, system needs to check multiple similar flows to find common bytes. For that, flows are categorized in first phase and then common patterns are found for each flow category in second phase. The system also gives useful information like multiple UDP flows having same port, multiple TCP/UDP flows having same destination IP and port etc. With multiple UDP flows with same local port, if initial flow is detected correctly, all subsequent flows can easily be detected by using simple 2-tuple (Destination IP + Port) table match.

3.1 Flow categorization

The target application may generate one or more types of flows for various services. For example, VoIP app can generate flows for login, User search/add, chat, voice and video calls among other services. We would need to analyze all such flows separately. If we try to compare a chat flow with a video call flow, it is obvious that we cannot find any potential signature as those two flows will be having totally different characteristics.

To overcome this obstacle, the flows are categorized based on flow characteristics which would result in formation of correct categories, wherein each category corresponds to one type of the flow. Then after, flows in each category can be analyzed separately for signatures. Effective categorization would ensure correct signature suggestion.

We have used statistical analysis for flow categorization. We calculate statistical vector for all application flows to put them into different categories. For statistical vector generation, flow properties like average packet size of whole flow/uplink/downlink, standard deviation of packet size of whole flow/uplink/downlink, Shannon entropy etc. are used.

K-means clustering algorithm is used for classifying the flow in required number of clusters. K-means clustering is an unsupervised machine learning algorithm for cluster analysis which aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. An analysis of the distances between flows and their cluster centers reveals that the distance distribution is Gaussian, so in each cluster most of the flows are placed close the cluster center [10]. This suggests that the flow's nearest neighbor is likely to fall in the same cluster with high probability, and only instances very far from the cluster center can have their nearest neighbor placed in another cluster. We have used k = 3 in our tests.

Once flows are categorized correctly, we examine multiple flows from similar category to find unique byte patterns among them. Fig. 2 shows statistical vector points for WhatsApp application flows. The statistical vector with 7 above-mentioned properties was calculated for each WhatsApp flow. The image shows n-dimensional statistical vectors in 2 dimension space. K-Mean algorithm has grouped WhatsApp control flows in cluster 1 (c1) and WhatsApp data flows in cluster 2 (c2). Some irrelevant flows are classified in cluster 3 (c3). Since c3 flows are not matched with c1 and c2 flows, we get accurate signature patterns for both control and data WhatsApp flows.

3.2 Finding unique byte pattern

Byte sequences which appear at a constant offset in packets across different flows are candidate for the signature set. To detect such signature, we line up flows of same category. Then we try to match bytes of 1st to ith packet of each flow. If there is a common byte pattern among the flows, that pattern will be selected as suggested pattern from start offset. For finding patterns on non-fixed offset, we use longest common substring algorithm. LCS algorithm can find matching patterns on different offsets of different packets.

Verification test is run for failed applications again with updated signatures to check detection result of previously undetected relevant flows. This process is repeated until we get all test cases getting passed.

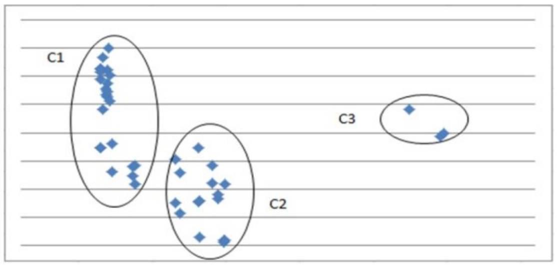

Figure 2: WhatsApp flows classified in clusters using K-means algorithm (k=3)

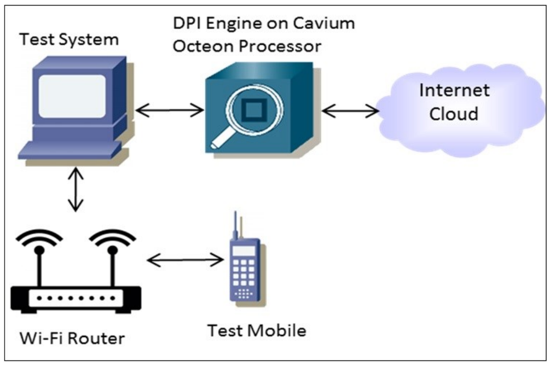

Figure 3: Test setup for DPI testing

III. TEST SETUP AND RESULT

This section discusses our experimental test setup and results for proposed method. Fig. 3 shows experimental test setup for DPI automation verification testing. We run DPI engine on Cavium Octeon 68XX Multi-Core MIPS64 processor which runs 32 cores on a single chip to accelerate packet processing. All application specific signatures are loaded on DPI engine. One Windows 7 system with 4 GB RAM and 2.0 GHz clock was used as test system. Android/iOS Test mobiles were connected to a Wi-Fi router which was connected to test system. All packets to/from test mobile pass through test system. The test system is connected to DPI engine which examines all packets and forwards the traffic to destination. If DPI engine detects any known signature, it saves flow information in detection table.

We have developed GUITAR test automation scripts for 300+ applications and Appium scripts for 20 apps. The verification system was tested for apps from categories like streaming, P2P, VoIP etc. Table 4 shows verification system performance for various parameters. As the results indicate, verification system found multiple false positive/negative detection issues in signature. The system also suggested new signatures for undetected flows. The accuracy to find relevant flow kept increasing as more generic app patterns were added in “Exclude list” by our algorithm.

One important analysis is cost analysis - cost of testing (man months) vs. cost of script development and running the test. With multiple apps, manual testing time grows very fast. However, automation testing requires only one time script development for those apps and then running the test multiple times. Fig. 4 shows number of apps vs. total time for both manual and automated testing. The results clearly indicate that automation testing saves lot of testing cost. Table 5 shows few signature suggestions that we got for various applications.

Table 4: Automated Verification System Performance

	Application Tested	Issues Found	Signature Suggested	Relevant flow Accuracy
Month 1	40	12	6	80%
Month 2	75	15	5	83%
Month 3	115	13	7	89%
Month 4	130	9	5	92%

Table 5: Auto Signature suggestions from our tool

HTTP	googlezip.net, .ngeo.com, .natgeotraveller.in, bmimages, thisisaim, truste, disqus
HTTPS	awsstatic.com, amazonwebservices., archive-it.org, dropboxusercontent , Mzstatic, shopify
TCP	“BitTorrent Protocol”, 0xd13a[ad]
UDP	0x000100242112a442 ,0x00060020

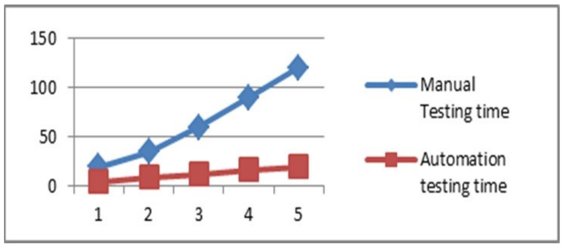

Figure 4: Cost for Manual Vs Automated test for multiple apps

IV. CONCLUSION

An average manual application signature test on all popular platforms requires lot of time and effort to generate mobile application traffic for various application features. The traffic can have hundreds of HTTP, HTTPS, TCP and UDP flows. Checking detection result for all these flows manually in detection database is very slow and error-prone. With automation, we save lot of time in generating traffic and verifying signatures. We find many false positive/negative cases which we tend to overlook in manual task. With every test run, our irrelevant pattern database grows and relevant flow detection accuracy increases.

Previous mobile test automation attempts were mostly confined to generating mobile application traffic and testing the app by generating multiple gestures and user actions. We extend that idea to verify DPI signatures and suggest new signatures for undetected flows. We use a self-learning algorithm which keeps on building irrelevant pattern database after each run and suggests signature for almost matching patterns. With that, the signature verification system can identify issues in existing signatures and update new signatures in short time after newer application version releases.

We successfully demonstrated that manual, error-prone DPI signature verification task can be fully automated. This verification system can identify signature update cases quite early and suggest changes. In future work, we would add more intelligence in machine learning algorithms, tighten detection of all false positive and false negative cases and increase the accuracy of automated verification system.

REFERENCES

[1] U. Trivedi, "A self-learning stateful application identification method for Deep Packet Inspection," Computing Technology and Information Management (ICCM), 2012 8th International Conference on, Seoul, 2012, pp. 416-421.

[2] ACAS: Automated Construction of Application Signatures - Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, Dongmei Wang SIGCOMM’05 Workshops, August 22–26, 2005, Philadelphia, PA, USA.

[3] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, Tadahiro Uehara, "Mobile Application Testing: A Tutorial", Computer, vol.47, no. 2, pp. 46-55, Feb. 2014, doi:10.1109/MC.2013.445

[4] H. Song et al., “An Integrated Test Automation Framework for Testing on Heterogeneous Mobile Platforms,” Proc. 1st ACIS Int’l Symp. Software and Network Eng., 2011, pp.141–145

[5] WangY,et al. Generating regular expression signatures for network traffic classification in trusted network management.JNetwork Comput Appl (2011), doi:10.1016/j. jnca.2011.03.017

[6] Wang Y, Xiang Y, Yu. SZ. An automatic application signature construction system for unknown traffic. Concurrency and Computation–Practice and Experience 2010;22(13):1927–44

[7] appium.io

[8] http://dev.naver.com/projects/guitar

[9] https://www.autoitscript.com/site/

[10] Crotti, M., Dusi, M., Gringoli, F., Salgarelli, L.: Traffic classification through simple statistical fingerprinting. Computer Commun. Review 37(1), 5–16 (2007)