Decrypting Android App Communication

Stefan Kunz
University of Passau
Passau, Germany
stefan.kunz@uni-passau.de

ABSTRACT

Many Android applications encrypt their produced network traffic by default. This prevents malware analysis, because for the differentiation of legitimate apps from malicious ones the decryption of their network traffic is necessary. Mostly TLS is used for network-based encryption. For this reason this paper presents the AndroidTLSInspector, which is an approach for the transparent decryption of an Android app's communication secured by TLS. This is done by network traffic analysis and extraction of the encryption key out of the app's memory.
The following paper presents the architecture of this approach and how this approach was implemented. At the end the results of the implementation are presented and further possible research steps are outlined.

Keywords

Android, Digital Forensics, Mobile Device Security, TLS, Networking, Network Analysis

1. INTRODUCTION

The distribution of malware for the mobile operating system Android is heavily increasing. In the U.S. for example in 2014 the malware encounter rates grew 75% in comparison to the preceding year[1]. Therefore the importance of mobile application analysis, which plays an important role in malware detection, is steadily increasing.
Additionally encrypted communication protocols have become ubiquitous in the Internet [8]. This fact makes it hard to differentiate between malicious apps, communicating with a command and control server, and legitimate client-server interactions. The determination of malicious apps from legitimate ones can be done by analyzing their network traffic. This is attempted to be prevented by network-based encryption. To overcome this issue an app's network traffic has to be decrypted without affecting the communication of the app, that it cannot realize that it is under observation and then maybe changes its behavior. To realize this the encryption key has to be obtained, to be able to read the produced network traffic of the app.
A possible solution how this problem could be solved directly on an Android device, without modifications of the target system, is introduced in this paper. To avoid, that observed apps detect their observation, our goal was to find a solution, which transparently decrypts the encrypted network traffic without manipulating it. We solved this problem by extracting the encryption key out of the memory of the devices.
The paper contains in section 2 related projects. Section 3 treats the necessary background information, followed by a presentation of our prototype's architecture in section 4. Afterwards the prototypical implementation of the described architecture will be explained in section 5 and the results are presented in section 6. Finally, we suggest in section 7 how our approach can be continued.

2. RELATED WORK

For decrypting Android app communication the encryption key securing it has to be obtained. For this purpose there are different approaches available.
Taubmann et al. [8] divided them into active and passive ones. Active approaches interfere actively in the communication between the endpoints by manipulating its content. For example **sslsniff**[2] acts as man-in-the-middle (MITM) between the client and the server. This approach works only if the server's certificate is not correctly verified by the client, because it is not fully transparent. The fake certificate used by the MITM could be recognized by the client. Also other active approaches fully removing the encryption of the communication channel by replacing the https-protocol with the http-protocol can be detected by the client application.
In contrast, passive approaches only capture the network traffic and try to decrypt it later with the knowledge gained about the key. **ssldump** for example is such an approach. ssldump is able to decrypt TLS packets if RSA keys were used to encrypt the data and the private key, used to encrypt the data, is known to it[3]. This key could for example

[1]Lookout: 2014 Mobile Threat Report, `https://www.lookout.com/img/images/Consumer_Threat_Report_Final_ENGLISH_1.14.pdf`, Accessed: 2016-01-10

[2]sslsniff, `http://www.thoughtcrime.org/software/sslsniff`, Accessed: 2016-01-10

[3]Using ssldump to Decode/Decrypt SSL/TLS Packets, `http://packetpushers.net/using-ssldump-decode-ssltls-packets`, Accessed: 2016-01-10

be extracted with full transparency from the main memory of the server on which ssldump is running. Additionally some browsers also support exporting the used private encryption key to a file, from which it can be accessed. Thereby their detection is more difficult. This fact is important for detecting malware, because malicious applications are not able to change their behavior under observation, if they are not aware of it. Another disadvantage of active approaches is that they reduce the monitored system's security by affecting its communication.
For this reason Taubmann et al. [8] chose a passive approach for the prototype of their **TLSInspector toolkit**, which decrypts TLS communication based on virtual machine introspection. They decrypt the network communication by taking a memory snapshot of the process belonging to the captured network communication. Out of this snapshot then the encryption key gets extracted enabling to decrypt the captured network traffic.

Specifically for Android devices such an approach to decrypt an app's communication is not yet available. Razaghpanah et al. [6] introduced the system **Haystack**, which is an active approach for client-side context-aware inspection of network communication in real-time. Their system manages capturing user traffic locally on the device without root privileges. This is realized by defining a VPN connection causing the Android-system to redirect all network packets to their system's process, which captures and forwards them to their destination. Also their system is capable of identifying contextual information for detecting privacy leaks. This includes for example the app generating the network traffic.
Additionally they are able to eavesdrop communication secured by TLS with a local TLS proxy actively interfering the TLS traffic. This proxy acts as MITM between the client and the servers forwarding the packets to the client-side with a self-signed certificate previously installed into the Android device's certificate store. But this approach works only, if the client app does not verify the certificate correctly and trusts the certificates provided by the Android certificate store. Otherwise their system can be detected.

3. BACKGROUND

For a better understanding of the chosen strategy to decrypt an app's communication, this section gives background information about the Android system in general and contains an overview about TLS, the encryption protocol used in this paper.

3.1 Android

The mobile operating system Android is based on a Linux kernel. Apps for it can be programmed in Java or native C/C++ code. When an app is started it runs always in its own application runtime environment. From Android L (API-Version 21) on the Android Runtime (ART) is used for this task, which is the predecessor of the Dalvik Virtual Machine (DVM). Android provides a runtime environment designed for the special requirements on mobile devices, like limited computational power, energy and storage resources. The ahead-of-time (AOT) compilation compiles at install time the apps to app executables for the target device. This improves the apps' performance, but it also extends the install time and increases the required storage space by the compiled apps' executables[4].
Running apps are isolated with a safety sandbox [12]. Each app is executed in its own process, which has a unique process id (pid). The required memory is assigned to the process by the operating system, which is responsible for resource management. This includes also stopping and killing processes, if the device's available memory recourses get low [1].

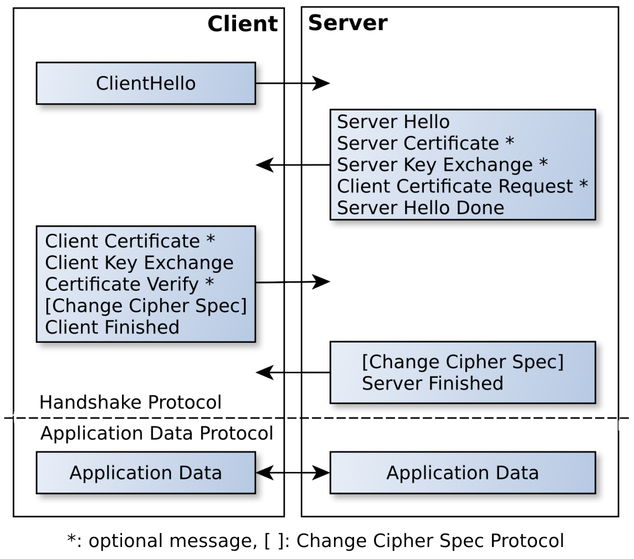

Figure 1: TLS Handshake

3.2 TLS

The Transport Layer Security (TLS) protocol is the successor of the Secure Sockets Layer protocol (SSL) and currently the most frequently used cryptographic protocol in the Internet [4].
The protocol is mainly based on an asymmetric cryptographic method and public-key cryptography, which is mainly used for server authentication. Client authentication is also possible.
The handshake protocol is responsible for session negotiation. Its sequence can be seen in figure 1. The client initiates a session with a message with the content type Client Hello (CH). This message contains a 28 byte client random, the client's latest supported TLS version and its supported compression methods and cipher suites. Out of these provided values the server selects the cipher suite consisting of an authentication method, a key exchange method, an encryption algorithm and a hash algorithm.
The server authenticates itself and tells these chosen methods to the client with the Server Hello (SH) message, which contains the given session id and the 28 byte server random, which was generated independently from the client random. Dependent on the chosen authentication method the server also transmits its certificate chain to the client to authenticate itself. These messages are followed by an optional server key exchange message, if the sent certificate message does not contain enough data, that the client can exchange a pre-master secret, which is the case for some cipher suites. If client authentication is required, the server also requests

[4]ART and Dalvik, `https://source.android.com/devices/tech/dalvik/`, Accessed: 2016-02-23

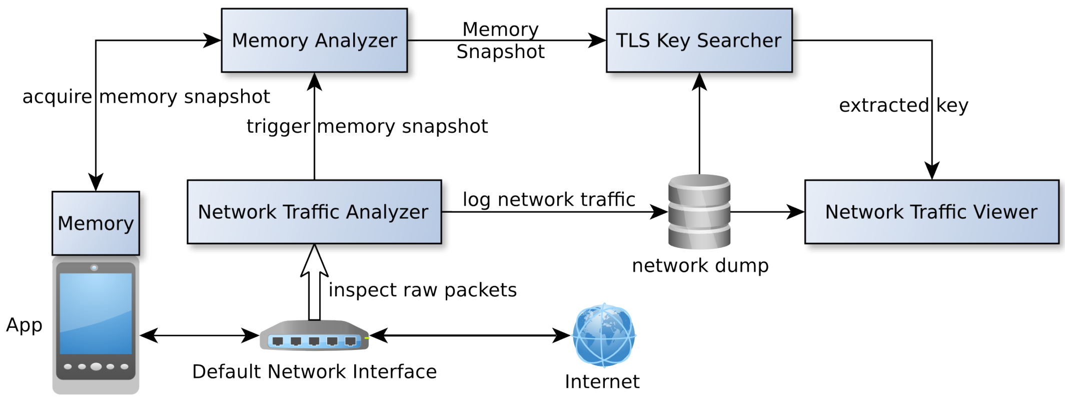

Figure 2: AndroidTLSInspector - Architecture

the client's certificate.
The client then provides its certificate, if requested and optionally also a Client Verify message for the verification of the client certificate. After the server and optionally also the client have been authenticated, they negotiate a pre-master secret using RSA encryption, Diffie-Hellman (DH) or elliptic curve Diffie-Hellman (ECDH) algorithm. From which DH or ECDH should be preferred, because they provide perfect forward secrecy (PFS). With the Client Key Exchange message the pre-master secret is set by transmitting it RSA-encrypted or its Diffie-Hellman parameters to compute it dependent on the previously chosen algorithm. The client's response is encrypted with the server's public key contained in its certificate to prevent active MITM attacks with forged certificates, provided that the private key of the server is unknown.
Out of this pre-master secret and the previously exchanged random values then both compute their common symmetric 48 byte session key (master secret) with a pseudo random function. This master secret, which was negotiated during session establishment, is used for securing the further communication with a symmetric encryption algorithm. After the master secret was computed or the cryptographic parameters change, each communication partner sends a Change Cipher Spec (CSP) message after it is finished with the computation of the new master secret. Thereby a communication partner indicates that the new master secret should from now on be used and all further messages from it will be encrypted with the new cryptographic parameters [2]. The Finished message is send directly after a CSP message, again using the handshake protocol, to enable the verification of the key exchange and authentication process. It is the first message encrypted with the new cipher specifications and it contains a hash of the preceding handshake messages to enable their verification.
If the cryptographic parameters were negotiated, application data packages can be exchanged using the application data protocol on the record layer. To prevent replay or modification attacks, they contain MACs computed among others out of the message's sequence number, length and content. For the computation of the required MAC secret, the master secret is hashed together with the client's and server's random value to enable the verification [11].
Particularly for TLS is that the messages of the TLS handshake protocol, used to establish a common master secret, are send using their record layer protocol, which contains for each message a particular type and only the payload is encrypted. The other parts of the message are transmitted in plain text. Due to this the content type of a message, which is indicated by its first byte, can be observed without decryption. Also multiple handshake messages can be combined into one single record layer message [10].

4. ARCHITECTURE

This section presents the architecture of the AndroidTLSInspector, which is an approach for the transparent decrypting of an Android app's communication secured by TLS.
This step is important for malware analysis, because it enables the inspection of an apps network communication. The AndroidTLSInspector's design is restricted to the cryptographic protocol TLS, because this protocol is currently the most frequently used in the Internet. Therefore we assume, that this protocol is used by the most malicious applications.

The approach for finding the key to decrypt the network communication has been divided into multiple steps. An overview can be seen in figure 2:

1. **Network Traffic Analyzer:** At first the network traffic of the analyzed app has to be permanently observed and captured. Later, when the encryption key is known, the network traffic can be decrypted. Also it is necessary to recognize when the moment has been reached, during which the encryption key is present in the memory. This part is explained in section 4.1.

2. **Memory Analyzer:** When the encryption key is likely to be in the app's memory, the memory snapshot needs to be triggered, which is handled by the Memory Analyzer described in 4.2.

3. **TLS Key Searcher:** If the memory snapshot is available, the encryption key can be extracted from it by the TLS Key Searcher, using the information gained from the captured network packets. This key extraction process is explained in 4.3.

4. **Network Traffic Viewer:** If the used encryption key

was extracted, it is possible to decrypt and view the app's communication. This step is described in 4.4.

4.1 Network Traffic Analyzer

The network traffic analysis should be performed with full transparency. Due to this, it is not possible to act as MITM and actively interfere into the communication by modifying the transmitted messages. We chose a passive approach for this task to prevent, that the analyzed app can detect its observation.
This includes capturing of all relevant packets and storing them for later analysis. If a message is captured, its type gets checked. When it is of the type CH, which indicates that the client wants to establish a new encrypted session, the contained secret has to be extracted to be able to extract later the master secret from the devices memory. The same applies to the subsequent SH message. From this also the selected cipher suite and compression algorithm has to be extracted.
When the client sends the CSP message, indicating that it is finished with the computation of the new master secret, a memory snapshot is triggered. Then the master secret, used for encrypting the following packets, should be available in the app's memory. Therefore it is necessary to identify the id of the process, which has opened the corresponding socket, that its pid can be forwarded to the memory analyzer to trigger a snapshot.
During this time the following application data packets get captured also to enable further forensic analysis and to be able to verify possible found master secrets using the MAC contained in the first sent application data package.

4.2 Memory Analyzer

The memory analyzer is responsible for memory acquisition. For this task different methods are available [1].

With the kernel module for **Linux Memory Extraction (LiME)**[5] it is possible to copy a device's full memory and store it on the SD card or transmit it over the network via a TCP socket. The kernel module requires the source code of the device's kernel and its appropriate drivers for compilation. Also root privileges are necessary for running the kernel module.
An advantage of this approach is that it leaves only a minimal footprint due to its small size and only a few kernel functions are required to acquire the memory snapshot [7]. However compiling the kernel module is problematic, because not all device manufacturers provide the source code for their devices. Also it should be more efficient to dump only the memory of a specific observed process instead of the whole device. For these reasons this approach is not suitable for our use case.

Another solution could be to dump the memory of a process by attaching to the process and sending the child process the kill command. But this has the drawback, that possible sensitive information in the memory of the process would be written to the disk and could be also analyzed by local attackers [13].

[5]LiME was formerly known as Droid Memory Dumpstr (DMD).

Additionally the **Android Debug Bridge (ADB)** provides the possibility to dump a device's volatile memory. With this tool it is possible to dump the heap of a process. The heap is preloaded with the app's classes and data, but does not contain all available pages of the processes in memory [3]. This solution has the same drawbacks, than the previous one. Most Android devices have adapted custom operating system versions, among which the memory dumps are differing.

Furthermore the process trace (**ptrace**) functionality provided by the kernel is capable of acquiring the memory of a process. This technique is often used for debugging, but is also useful for our use case in malware analysis [12].
The memory acquisition tool **memgrab**, introduced by Thing et al. [9], uses this functionality for live memory forensic analysis. It enables acquisition of the memory regions of a process specified by their physical address ranges [7].
To prevent dumping irrelevant memory regions they can be extracted from the memory maps. The memory regions consist of heap, stack, shared memory and the mappings to the system libraries, AOT executables and device drivers. The **ptrace** command traces a process to get access to its address space and controls its execution.
For performing the memory snapshot the execution of the process gets suspended and after the memory snapshot was completed the process gets resumed.

For this reason the most suitable solution for our use case is is the ptrace functionality. ptrace has the advantage, that it can dump the full memory of a specific process, which is much faster than dumping and analyzing a device's whole memory. Additionally we can filter the different memory regions to gain a memory snapshot of minimal size.

4.3 TLS Key Searcher

If the memory snapshot was acquired the key extraction process can start. This step searches the master secret in the memory snapshot of the app.
THe key extraction process can be realized by using a brute force approach, which tries to decrypt the first captured application data package with each possible 48 byte value contained in the snapshot, which is very slow. If the decryption was successful and the key was found, it can be validated by computing the MAC secret for the possible master secret. With this MAC secret the MAC for the decrypted TLS record can be computed and compared to the given one, already contained in the packet to protect its integrity. For the validation is important that the first application data package is used, because for the MAC computation the TLS record's sequence number is necessary, which is not contained in the packet and therefore only can be assumed for the first packet.

Another approach are pre-defined key structures. They have the advantage, that they are fast, but they can only be applied to known key structures. Brute forcing on the other hand can detect all key structures.

Therefore in our use case a hybrid approach, which combines pre-defined key structures with brute forcing, if the secret has an unknown structure [8, 11].

```
# currently opened sockets
root@generic:/ # cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   5: 0F02000A:96EF 3F70C2AD:01BB 01 00000000:00000000 00:00000000 00000000 10015        0 10769 1 0000000000000000 21 4 1 10 -1

# sockets opened by process with pid 2127
root@generic:/ # ls -l /proc/2127/fd
lrwx------ u0_a15   u0_a15            2016-02-25 13:25 126 -> socket:[10769]
```

Figure 3: Connection to pid translation

```
address                   perms offset   dev   inode pathname
7fbcf36c4000-7fbcf36c8000 rw-p  00000000 00:04 5635  /dev/ashmem/dalvik-indirect ref table (deleted)
7fbcf0c8c000-7fbcf0ca5000 r-xp  00000000 1f:00 1013  /system/lib64/libandroid.so
7fbcf0ca5000-7fbcf0ca6000 ---p  00000000 00:00 0
7fffcc554000-7fffccd53000 rw-p  00000000 00:00 0     [stack]
```

Figure 4: Memory Maps (/proc/<pid>/maps)

4.4 Network Traffic Viewer

After the key was found by the TLS Key Searcher it gets forwarded to the Network Traffic Viewer.
This can be a tool like Wireshark[6], which is able to decrypt and view the captured packets using the extracted master secret. It enables the inspection of the app's network communication for further analysis and malware detection until the master secret changes. Then the whole key extraction process needs to be restarted and the new encryption parameters have to be extracted.

5. IMPLEMENTATION

The prototypical implementation of the AndroidTLSInspector's architecture is explained in this section.
Design goals were, that the decryption of the TLS network traffic is performed passively. Additionally our prototype should run on Android devices of different manufacturers without the necessity of special adjustments. Therefore we don't use target system modifications and kernel modules.

5.1 Network Traffic Analyzer

The Network Traffic Analyzer observes and captures the raw packets flowing over the default network interface. The packets' inspection is realized passively without modifying the packets to provide full transparency and prevent, that the observed apps discover their observation. For this purpose we use **libpcap**[7], which provides the required functionality for network traffic capturing. If the establishment of a new TLS connection is recognized, the relevant packets get captured to be able to extract the required encryption parameters from them like explained in section 4.1.
Te CSP message indicates, that the app is finished with the computation of the master secret. Directly after this message the memory snapshot of the app can be triggered. Therefore the process, which has opened the corresponding socket, has to be identified.
For this reason the tool **procfs**[8] is used, which is an interface to the internal data structures in the kernel. It can be used to obtain system information and change certain kernel parameters during runtime. By matching the current opened sockets' inodes with the inodes of the sockets opened by each process, the pid of the corresponding process to the captured connection can be found like displayed in figure 3.

5.2 Memory Analyzer

The different methods for memory acquisition have already been discussed in section 4.2. We use the **ptrace** functionality provided by the kernel. It has the advantage, that it can dump the memory of a specific process, which is more efficient, than dumping a device's whole memory. Additionally we only dump writable memory regions to minimize the size of the memory snapshot, because the master secret gets computed during runtime. Therefore it has to be written into a writable memory region.
But the tool **ptrace** is inefficient for accessing large amounts of memory, because it can only read word sized blocks [5]. Therefore we only use this tool to get access to the process's address space and to control its execution. For performing the memory snapshot, we use ptrace to attach to the process and pause its execution to be able to access its memory.
The currently mapped memory regions of every process can be retrieved from **procfs** under /proc/<pid>/maps, like displayed in figure 4. This file contains information about each mapped memory region's starting and ending address in the process's address space. The permissions are used to filter out non-writable memory regions. Other important properties are offset and pathname of each region.

Using the information about the memory regions, gained by this file (figure 4), we can directly access the virtually mapped memory regions' pages of the process. This information is provided by **procfs** in the file /proc/<pid>/mem. After the memory snapshot was done, the execution of the observed process is resumed. This is done by detaching our process from the observed one using ptrace.

5.3 TLS Key Searcher

The TLS Key Searcher tests every byte sequence contained in the memory snapshot as possible key using a brute force approach. This process is realized by checking each possible key value's validity by computing its MAC and comparing the computed MAC with the MAC appended to the de-

[6]Wireshark, `https://www.wireshark.org/`, Accessed: 2016-02-24
[7]libpcap, `http://www.tcpdump.org/`, Accessed: 2016-02-25
[8]process information pseudo-filesystem (procfs), `http://man7.org/linux/man-pages/man5/proc.5.html`, Accessed: 2016-02-25

```
root@PC:$ android_tls_inspector eth0
start listening on eth0
Server: 192.168.43.13:52859 -> Client: 64.233.166.94:443
getting memory snapshot, PID: 9110
memory maps: 173 writable memory address space regions found
dumping memory, Page size: 4096
start | end | path name...
139889097031680|139889097060352|[stack:2995]
139889097125888|139889097134080|
139889097138176|139889097142272|/usr/lib/firefox/firefox
643002368 bytes dumped, 156983 pages read, 0 pages skipped due to 'I/0 error'

Server Random: 56 cb 3a e2 20 a6 b3 62 ea 79 09 3b 30 e9 69 c5 da 63 13 77 4a 5f 50 ab c1 8c 0d 03 f1 62 6f 4b
Client Random: fd c8 da 01 d8 c4 73 4c 28 d7 44 17 05 fl e4 d2 11 1e 93 d8 f6 11 54 20 79 5a f2 57 e5 51 5c eb
Cipher: c02f
First Record: 00 00 00 00 00 00 00 00
Searching key in a dump of size 156983
key found
0x000000: 81 a1 40 04 de a0 1d 61 df 9d 7c 12 df f0 28 44 ..@....a..|...(D
0x000010: 8f 76 dc 12 a8 c3 4b c6 bd b9 b8 74 1e df dd 98 .v....K....t....
0x000020: 35 0d 4f 02 c1 8d 12 30 a1 33 2e be 69 53 46 02 5.O....0.3..iSF.
CLIENT RANDOM FDC8DA01D8C4734C28D7441705F1E4D2111E93D8F6115420795AF257E5515CEB
              81A14004DEA01D61DF9D7C12DFF028448F76DC12A8C34BC6BDB9B8741EDFDD98350D4F02C18D1230A1332EBE69534602
```

Figure 5: AndroidTLSInspector - Test Run

crypted application data packet, like described in section 4.3. To lower the required computational effort and speed up the verification process, the possible keys get filtered before the MAC's computation is started and the key is tested to decrypt the communication. This filtering is realized by heuristics concerning the key structure and under the assumption that each 48 byte key is stored 4 byte aligned in memory. Additionally stochastic properties of the byte sequence are concerned using the **TLSkex framework** written by Taubmann et al. [8].

6. RESULTS

This section contains the results of the implementation described in the preceding section. It treats also the occurred implementation problems.

During the compilation of the AndroidTLSInspector some problems arose concerning the compilation of the required libraries for the TLSkex framework, which we use to extract the TLS key from the memory snapshot. Because of this reason, we were unable to test our approach as a whole on an Android device. Therefore we checked its operability on a desktop computer running Ubuntu Linux. The produced output of the AndroidTLSInspector's test run can be seen in figure 5.
For testing the AndroidTLSInspector we started it with root privileges and initialized it with the name of our default network interface. Then we established a TLS-encrypted connection to a website, using the operating system's built-in web browser (Forefox). Our system recognized the established package flow correct and triggered the memory snapshot in time. This can be verified by the successfully extracted master secret from the memory snapshot. The other extracted encryption parameters are also contained in the system's debug output shown in figure 5.

The test run shows, that our approach works properly. However, it is still more effort required to complete the compilation of all required libraries of the system for Android devices.
Single components of the system already work on Android. We verified this by testing different prototypes on an Android emulator with an armeabi-v7a Application Binary Interface (ABI) running Android L (API-Version 22). Our first prototype took the pid of an app as input to take a memory snapshot of 70 MB to the device's disk within 48 ms. We also verified successfully, that the information for the connection to pid translation can be gained from procfs with another prototype. This means the implementation is working partly, never the less we need more effort for completing the compilation of the TLSkex framework for Android.

7. FUTURE WORKS

Besides the compilation issue there are also other points, which could be improved.
Currently only the relevant packets for the key extraction are captured. For performing further malware analysis the whole communication flow needs to be captured, which can be decrypted with the extracted encryption parameters.
Also from the memory snapshot currently only non-writable pages are filtered out. If our system is compiled for an Android device a more precise filtering should be applied. This additional filtering only keeps anonymous mapped memory regions, heap and stack of a process to further minimize the snapshot's size and thereby speed up the key extraction process. This feature was not yet possible to implement, because we cannot very yet, if the encryption key is still contained in the snapshot for an Android device, after the more precise filtering was applied.
Another possible solution to speed up the key extraction would be to monitor the memory of a process during network connection establishment and key negotiation. Then only memory pages altered during this time span would have to be considered for the memory snapshot, which would also decrease its size [8].
Additionally the key extraction mechanism could be improved by extending the heuristics implemented by pre-defined key structures, which are used for filtering possible master secrets.

8. CONCLUSION

In this paper we introduced the **AndroidTLSInspector**, which is an approach for the transparent decrypting of TLS-encrypted network communication on Android devices to enable further malware analysis and the identification of malicious apps. This functionality was realized by observing the device's default network interface and extracting the master

secret from the observed app's memory, after the secret was negotiated and the relevant encryption parameters were extracted from the captured packets.
Although we already tested our approach successfully on a Ubuntu Linux machine, it was not yet possible to compile and test the AndroidTLSInspector on an Android device, because of the problems described in section 6. Nevertheless our approach looks very promising, because it enables the decryption of TLS-encrypted network traffic directly on the device without the necessity of system modifications. Thereby our approach enables the observation and analysis of apps to differentiate malicious apps from legitimate ones. For this task currently only active approaches are available acting as MITM using fake certificates and manipulating the communication, like the approach described in section 2. The active modification of the packet flow can be detected by observed apps, if they validate the packets correctly. Then they could change their behavior and behave differently under observation. Therefore our passive approach, which only captures the packets without active modification of them, looks very promising.

9. REFERENCES

[1] D. Apostolopoulos, G. Marinakis, C. Ntantogian, and C. Xenakis. Discovering Authentication Credentials in Volatile Memory of Android Mobile Devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*, volume 399 of *IFIP Advances in Information and Communication Technology*, pages 178–185. Springer Berlin Heidelberg, 2013.

[2] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Retrieved from `http://tools.ietf.org/html/rfc5246`, 2008. Accessed: 2015-12-07.

[3] A. P. Heriyanto. Procedures and tools for acquisition and analysis of volatile memory on android smartphones. 2013.

[4] H. Krawczyk, K. Paterson, and H. Wee. On the Security of the TLS Protocol: A Systematic Analysis. In R. Canetti and J. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer Berlin Heidelberg, 2013.

[5] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[6] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *ArXiv e-prints*, Oct. 2015.

[7] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3 - 4):175 – 184, 2012.

[8] B. Taubmann, D. Dusold, C. Fraedrich, and H. P. Reiser. Analysing malware attacks in the cloud: A use case for the TLSInspector toolkit. In *Proceedings of Workshop on Security in highly connected IT systems (SHCIS)*, 2015.

[9] V. L. Thing, K.-Y. Ng, and E.-C. Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7, Supplement:S74 – S82, 2010. The Proceedings of the Tenth Annual {DFRWS} Conference.

[10] S. A. Thomas. *SSL and TLS Essentials: Securing the Web with CD-ROM*. John Wiley &amp; Sons, Inc., New York, NY, USA, 2000.

[11] S. Vandeven. SSL/TLS: What's Under the Hood. *SANS Institute InfoSec Reading Room*, 2013.

[12] T. Vidas, C. Zhang, and N. Christin. Toward a general collection methodology for Android devices. *Digital Investigation*, 8, Supplement:S14 – S24, 2011. The Proceedings of the Eleventh Annual {DFRWS} Conference11th Annual Digital Forensics Research Conference.

[13] J. Viega and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. "O'Reilly Media, Inc.", 2003.