DFRWS
DIGITAL FORENSIC RESEARCH CONFERENCE

DroidKex - Fast Extraction of Ephemeral TLS Keys from the Memory of Android Apps

By

Benjamin Taubmann, Omar Alabduljaleel, Hans P. Reiser

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2018 USA

Providence, RI (July 15th - 18th)

Digital Investigation 26 (2018) S67–S76

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps

Benjamin Taubmann*, Omar Alabduljaleel, Hans P. Reiser

University of Passau, Germany

Keywords:
Memory forensics
Semantic gap
TLS
Android

A B S T R A C T

Fast extraction of ephemeral data from the memory of a running process without affecting the performance of the analyzed program is a problem when the location and data structure layout of the information is not known. In this paper, we introduce DroidKex, an approach for partially reconstructing the semantics of data structures in order to minimize the overhead required for extracting information from the memory of applications. We demonstrate the practicability of our approach by applying it to 86Android applications in order to extract the cryptographic key material of TLS connections.

1. Introduction

Live forensics on mobile devices becomes more and more important in the day-to-day jobs of forensic investigators (Casey and Turnbull, 2011). One common use case is the decryption of encrypted communication channels such as TLS, which is the most frequently used protocol for that purpose in the Internet (Krawczyk et al., 2013).

In this paper, we discuss the problems that arise when ephemeral TLS session keys of an application should be extracted at runtime when the exact layout of the data structures and their position in memory is unknown. The most important questions we address are: Where is the data located, how can we extract it with minimal overhead and when is it available in memory?

The concrete use case of this paper is the extraction of the cryptographic key material required during a TLS connection from Android applications. One important aspect of our discussion is that we address the problem of extracting the cryptographic key material as efficient and generic as possible so that the performance and usability of an application should not be affected.

There are different approaches to address the problem of decrypting TLS connections. The most important approaches are: using a *man-in-the-middle approach*, *manipulation of the control flow* and *extraction of the key from main memory*. All three approaches have advantages and disadvantages.

The man-in-the-middle approach requires that the client application do not implement certificate pinning, i.e., that it communicates with the server even if the server does not present a valid certificate. However, many Android applications already use this technique (Fahl et al., 2012). To bypass certificate pinning, the certificate of the proxy can be installed in the application key store. With Android Nougat it gets even harder to use this approach since applications do not trust user or admin-added CAs unless the applications allow it (Brubaker, 2016). The application of these tools can lower the security of the inspected connections (Durumeric et al., 2017; US Cert, 2017; Carnavalet et al., 2016).

The control flow of an application can be intercepted to extract information or manipulated to accept any certificate. To extract the raw communication, the data can be accessed by intercepting encrypt and decrypt functions of the crypto library. Alternatively, the control flow can be patched statically (manipulation of the binary) or dynamically (interception of function calls) so that the crypto library accepts any certificate even if it is not valid for a certain domain so that the application can be used with a proxy (Cipolloni, 2017). These approaches manipulate the normal routine of an application and make it vulnerable to attacks (Durumeric et al., 2017; US Cert, 2017; Carnavalet et al., 2016). Only extracting the encrypted communication data without being able to decrypt the network stream might lower the forensic soundness of the data.

Another method is to extract the cryptographic keys from the main memory of a process (Maartmann-Moe et al., 2009). One possible approach is to test every byte sequence in the memory of an application as a potential key to decrypt the first TLS message of

* Corresponding author.
E-mail addresses: bt@sec.uni-passau.de (B. Taubmann), alabdu01@gw.uni-passau.de (O. Alabduljaleel), hr@sec.uni-passau.de (H.P. Reiser).

https://doi.org/10.1016/j.diin.2018.04.013
1742-2876/

a connection (Caragea, 2016; Taubmann et al., 2016). In our evaluation, we measured an average time of 1.35 s for taking a snapshot of an Android application on the mobile device and 15.08 s to locate the key in it with that approach (see Section 6). When the data acquisition process requires pausing the application while the snapshot is taken, this approach does not work on common Android applications that use multiple TLS connections simultaneously. Each snapshot would disrupt the usability of an application even when the key extraction process is executed on a PC or in the cloud since transferring the snapshot to an external entity can be costly and time intensive.

The contribution of the paper is twofold. First, we present an approach for locating information quickly in a memory snapshot by partially reconstructing the semantics of data structures in memory instead of scanning the full memory. Thus, this approach narrows the *semantic gap*, i.e., the difference between high-level information and its low-level representation in memory. In order to achieve that, we analyze a memory snapshot of an application in the *training phase* and try to find a path of pointers that link data structures which allows the extraction of information with minimal memory access. Afterwards, in the *normal mode*, we use that knowledge to extract information by following the pointers in corresponding data structures. Second, we provide a proof-of-concept implementation – DroidKex – that uses this approach for fast key extraction of TLS connections at run-time of Android applications and we evaluate it on Android 6 with 86different applications.

The structure of this paper is as follows: Section 2 provides brief background knowledge about SSL/TLS and the Android crypto libraries. Section 3 describes the DroidKex architecture and the interaction of its components. Section 4 describes the process of extracting the key from applications by intercepting the control flow and Section 5 describes the algorithm to find a path to the data structures that hold the information. Section 6 measures the performance of DroidKex and Section 7 compares our approach to related work on decrypting TLS communication. Finally, Section 8 concludes the paper.

2. Transport layer security

The Transport Layer Security (TLS) protocol is the successor of the Secure Sockets Layer (SSL) protocol and one of the most frequently used cryptographic protocols on the Internet (Krawczyk et al., 2013). In this paper, we will use the abbreviation TLS for both TLS and SSL. TLS provides a standardized way for exchanging cryptographic key material for establishing symmetric encrypted communication.

To initiate a new TLS connection, the client and the server exchange “Hello” messages that contain client random (CR) and server random (SR) byte sequences. After the “Hello” messages, further details may be exchanged including digital certificates depending on the selected cipher suite. Furthermore, both parties generate a pre-master secret (PS) using the key exchange algorithm (such as Diffie Hellmann) specified in the cipher suite. PS, CR, and SR are used to compute the master secret (MS). MS, CR, and SR are then used with a pseudo-random function (PRF) to derive the symmetric keys that are used to encrypt the communication and to verify the integrity of a TLS. After having the key material exchanged, both parties send a change cipher spec (CSP) TLS message to notify each other to start encrypting messages. The byte sequences MS, CR, and SR are the cryptographic key material that is extracted by DroidKex.

In order to increase the speed of negotiating new sessions with the same server, TLS implements the concept of session resumption. Therefore, the negotiated parameters, i.e., the session state, are stored either by the client (session ticket) or the server (session ID) (Salowey et al., 2008). By using one of these methods the overhead for exchanging new cryptographic key material can be reduced. DroidKexis able to handle all connections of these three cases, i.e., negotiation of new material, usage of session tickets and session IDs.

2.1. Android and SSL

Android uses at least three different layers which are important for the use of SSL/TLS functionality in applications. The first layer on top is the application itself. Each Android application is written in Java and executed by the Dalvik/ART run-time environment. Applications can either come with their own crypto routines, e.g., in a separate native library or they can use the libraries provided by Android.

The core concept behind Android's crypto system is the *Java cryptography architecture (JCA)*, an interface used by different crypto libraries and implementations. A *cryptographic service provider* implements the interface of the JCA and provides the implementation of cryptographic routines. The most common cryptography providers for Android are Bouncy Castle and AndroidOpenSSL (Elenkov, 2014).

Bouncy Castle is a pure Java implementation of cryptographic algorithms and protocols whereas AndroidOpenSSL uses Java native interface (JNI) calls to access the OpenSSL library on the native level. In Android 6, Google replaced OpenSSL with BoringSSL, which is a fork of OpenSSL with Android specific patches. The default crypto provider of Android is the “AndroidOpenSSL” provider.[1]

3. The DroidKex architecture

This section discusses the general approach of the DroidKex architecture, the assumptions under which it works and the components that are required for the proof-of-concept implementation and the requirements on the target device.

3.1. Approach and goals

The goal of the DroidKex architecture is to extract the ephemeral information required for decrypting a TLS connection of an Android application from its main memory, namely the MS which is required to derive symmetric session keys. The key extraction is executed synchronous to the control flow of the application, i.e., the MS is extracted during a TLS session. Otherwise, there is no guarantee that the MS is still in main memory because an application might free or overwrite it directly after the connection terminates. Thus, we intercept all network related *send* and *receive* functions of an application. If they are handling a TLS connection (the interception framework resolves the remote address of a file descriptor), we extract the corresponding MS by dereferencing pointers that point to a structure holding the cryptographic key material that are passed to the functions of the crypto library. Even though the layout of the data structures is known since the implementation of OpenSSL and BoringSSL is open source, the exact layout of the data structures is unknown. This is caused by the fact that it changes based on the used compiler and compiler settings as well as the version of the library.

To address this problem, we follow a precomputed path starting with pointers on the calling stack to a data structure holding the

[1] https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html.

MS. Such a path is computed during the *training phase* in which we run an application until we have found a path for every TLS connection it is initiating. To bootstrap this approach, we locate the MS in a snapshot using the brute force testing method of TLSKex (Taubmann et al., 2016). The computation of a path is discussed in more detail in Section 5.

3.2. Assumptions

The first assumption we make is that all crypto libraries directly call network related *sending* and *receiving* functions (`read, write, receive, recvmsg, sendmsg, sendto, recvfrom`) and a starting point leading to the data structure holding the cryptographic key material is still on the stack.

The second assumption we make is that the crypto libraries do not use complex data structures such as linked lists, trees or hashmaps to store the MS. We assume that we can find one path from the start to the MS that always passes or visits the same data structures in the same order. Finding approaches that regenerate the layout of complex data structures to efficiently extract information should be addressed in the future.

Finally, we assume that each application can use different (versions of) libraries and different cryptographic parameters (e.g., the SSL/TLS version) which results in the fact that different paths for the same application can be required. We try to learn these paths in the training phase. If we did not learn a path, e.g., because the library was not used in the training phase, we will not be able to extract the corresponding MS.

3.3. Components

The network monitor monitors all TCP connections of the application ① and captures traffic for later analysis ② (see Fig. 1). It parses all packets to detect messages that are used to set up a new TLS connection. During the negotiation of a TLS connection, it extracts all parameters that are required to validate a master secret belonging to a connection by decrypting the first record. The parameters are: CR and SR, the cipher and the first data record. Additionally, it extracts the source and destination IP address/port of a connection which later allows the key managerto link a network packet with an intercepted function call. Afterwards, it sends this information to the key manager ③.

The interception framework is a combination of Python and JavaScript tools which use the Frida toolkit (Ravnås, 2018). The purpose of it is to extract the MS of each connection by intercepting network connection related function calls of an application. Every time a function call is intercepted, it follows the pre-computed

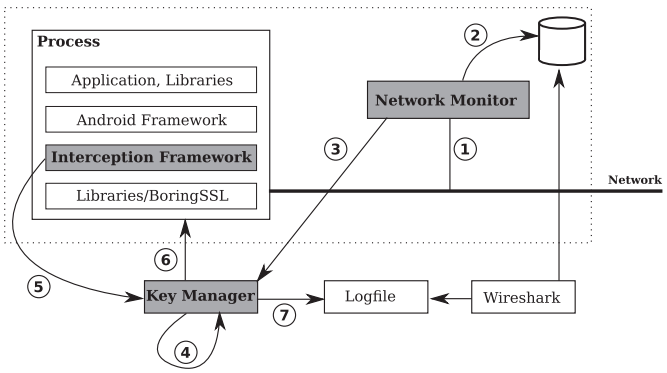

Fig. 1. The components of DroidKex architecture and their interactions.

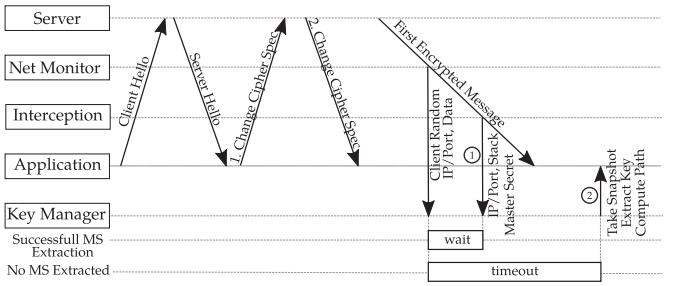

Fig. 2. The timing of messages during a TLS connection with all components in two distinct cases. In the first case the control flow of receiving a data message is intercepted and the key is sent to the key manager. In the second case the master secret was not extracted successfully (message 1 is not delivered) and after a timeout the key managertakes a snapshot of the application.

paths with the goal to locate a MS. Whenever a potential MS is found, it is sent to the key manager ⑤. Additionally, it sends for each function call all values on the stack within a certain range above the stack pointer. During run-time, it polls the list of connections with a valid key so that function calls of these connections are not intercepted.

The key manager validates and stores the extracted keys of TLS connections in a file which can be used by Wireshark for analysis ⑦. Therefore, the key managercombines the information about TLS negotiations with the data received from the interception framework in order to validate keys coming from the interception framework. To validate a key, it attempts to decrypt the first data record of the TLS connection with the extracted MS. Additionally, the key managerstores the session ID and the corresponding master secret of connections to improve the key extraction process in case of session resumption. If a following connection uses the same session ID, it is more likely that the same master key will be used. Consequently, there is no need to go through the key extraction process since the MS is already known and can be looked up.

Furthermore, the key managertakes a snapshot of an application ⑥ in the training phase, which is required to compute paths. Therefore, it waits a defined time interval after the network monitor reports that the application negotiated a new key ④ (see Fig. 2). If it does not receive a key belonging to the corresponding connection, the key managertakes a snapshot.[2] The time difference between receiving the information from the network monitor and the interception framework depends on many factors. The first one is caused by the workload of the key managerwhich processes the messages from the other components sequentially. Thus, if one message takes longer to process (for example the validation of several extracted keys), the next message is delayed, i.e., the time difference between receiving a key and verifying it is increasing. Additionally, the communication overhead affects the timing. In our proof-of-concept implementation, all components send messages via TCP sockets. Due to the architecture of Frida, one part of the interception framework is running on a separate analysis PC which increases the latency of the communication as well. Another reason for late reception of keys is the fact that some applications do not directly communicate after the key exchange and the key is not extracted when no network function call is executed. This can be a problem when the data structures of the crypto library are not fully initialized after the key exchange. To address this problem, the key manager waits for a certain time until it takes a snapshot.

[2] During a TLS connection, the parties can renegotiate the parameters. Since we never saw this during our testing, we do not consider this for our approach.

The exact timing for a snapshot is a trade-off between generating more overhead or losing the MS of a connection. The snapshot can be taken either directly after the network monitor sees the second CSP message, which indicates that both parties computed the MS, or a bit later. In the first case it can happen that the interception framework did not yet send the MS to the key manager, e.g., because it is not copied to the final data structure of a TLS connection. Then, the key manager would take a snapshot and interrupt the application even though we would receive the MS later. However, if the snapshot would be taken too late, the MS might not be in the memory anymore.

Consequently, in the *training phase*, we choose a short timeout for taking a snapshot after seeing the second CSP message since finding a MS is more important than maintaining the usability of an application. In the *normal mode*, we only intercept the control flow to extract the MS and do not take snapshots. If there has been a network connection of which we were not able to extract the MS, we need to compute the missing path. In that case, we manually switch back the DroidKexprototype to training phase and try to compute the missing path.

To evaluate the impact of taking snapshots in the training phase on the time of the message processing, we measured the time difference between the key negotiation and the time when the corresponding MS is validated for two cases. In the first case a snapshot is taken when a key is received late (0.5–3.5 s after the negotiation). In the second case, we do not take a snapshot. In both cases we measured the time after computing the paths for each application. In average we measured a time of 0.7s to receive the MS in normal mode from the interception framework after the network monitor detected the second CSP message (case 2). By taking a snapshot 0.5–3.5 s after the negotiation, we measured a time of 1.11s (case 1). Thus, the process of taking snapshots earlier delays the validation process by about 0.41s. The impact on the timing is bigger when more snapshots are taken, e.g., when several snapshots are taken in the case of a missing path.

3.4. Requirements

The only requirement for deploying DroidKexon a target device is to have root permissions and to be able to disable SELinux which is required by Frida. It does not require any additional modifications to the Android runtime system, which simplifies the deployment on devices for which the full device-specific source code is not available.

4. Data extraction

There are two different ways to locate the MS in main memory: *scan the memory address space sequentially* or *follow pointers* in data structures that lead to it. The first option can be either signature-based or by testing all byte sequences as potential keys to decrypt one TLS record (Klein, 2006). Unfortunately, the MS of TLS connections does not have a defined format such as PKCS12 which could be easily identified. Thus, the signature-based approach does not work. However, the brute force based approach works when the MS is stored as a byte sequence in the main memory of an application. On the other hand, the derivation of session keys and the decryption of a message is computationally intensive and slows down the whole approach, which makes it infeasible for runtime key extraction.

The second option interprets the data structures in main memory and follows pointers to the data structure holding the MS. It requires: (1) a start point from where on we can follow pointers (2) knowledge about where the pointers are stored in each data structure (how to get the knowledge is discussed in the next section). A start point can be defined in two ways. The first one is a global symbol or variable in a binary. Such a variable (if it exists) is usually not exported in a binary file. Especially, crypto libraries try to encapsulate and hide internal functions and data structures to prevent data leak. The second option for a start point are the pointers to the arguments passed to a function. They can be extracted from the stack by intercepting the function call. This requires that the function symbols are exported and the knowledge about the utilized functions.

DroidKex implements the second approach of control flow interception since the runtime cost for the key extraction does not depend on the size of the main memory of the application. Instead, it depends on the amount of memory access operations for following a path and the overhead added to intercept a function call. Therefore, DroidKexintercepts all network related function calls for *sending* and *receiving* data and checks based on the file descriptor whether they belong to a TCP connection on port 443. We assume that an entry point leading to the MS is in a certain range above the stack pointer of the intercepted function. We could also directly intercept crypto library functions, but we aim at finding a generic approach that also works if the function names are not known or not available, e.g., when the symbols are stripped from a binary.

To improve the performance, we reduce the number of considered elements on the stack by saving the position of a good start point in relation to the stack pointer. Additionally, we

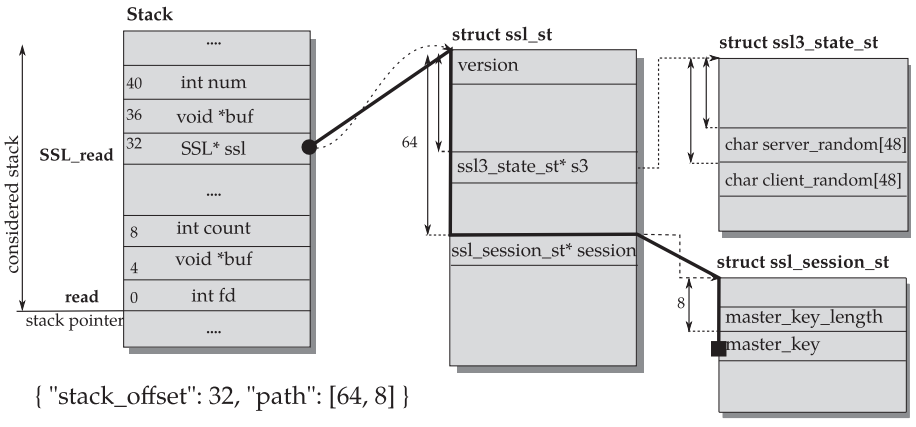

Fig. 3. The contents on the stack when the read function is called by the `SSL_read` of OpenSSL function. The path from the starting point – the SSL pointer (black dot) to the MS (black square) – is marked bold and the corresponding. The computed path and the offset on the stack are noted on the bottom left side.

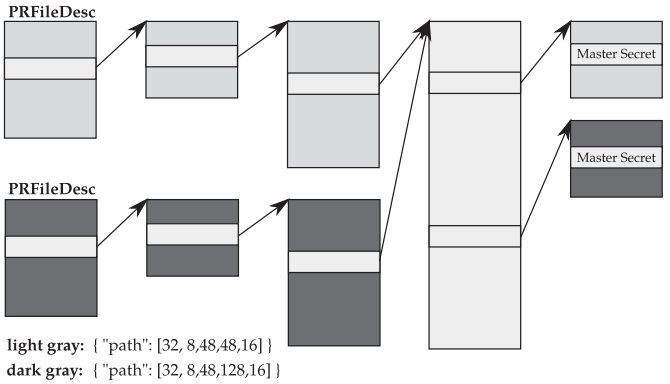

Fig. 4. The path from the PRFileDesc struct to the MS in the memory of the Firefox application that uses the libnspr library.

minimize the extraction attempts by intercepting only I/O calls of connections of which we do not yet have a valid MS. In Fig. 3, we depict the stack layout at the point in time when the `SSL_read` function of OpenSSL/BoringSSL calls the `read` function. To extract the MS, we need to know the position of the pointer to the SSL struct in relation to the stack pointer and the path from the SSL struct to the MS.

5. Path computation

For computing a path in the *training phase*, we use a data-centric approach, which means that we only consider the contents in main memory and not the instructions of a program (Dolan-Gavitt et al., 2011). We define a path as a list of integers. Each integer defines the offset in a data structure where the pointer to the next data structure is stored. The last element in the list stores the offset to MS. (for examples see Figs. 3 and 4). The most important steps to compute a path are: taking a snapshot of the process holding the information, identifying the position of the MS, computing the path based on the snapshot and finally validate the data path. These steps are discussed in more detail in this section.

5.1. Prerequisite

In order to compute a path to a MS, three components are required. The first one is a *snapshot of the application* while the MS is in the memory. The second one is a *starting point* for a path. Fur this purpose DroidKexuses the values the on the stack of each *send* and *receive* functions handling the corresponding networking connection. The third one is the *location of the MS (endpoint)*. To find a MS in a snapshot, we use the brute-force approach of TlsKex (Taubmann et al., 2016).

5.2. Data structure layout

DroidKexaims at extracting the MS of applications using either the OpenSSL/BoringSSL or libnspr crypto library. In both cases, a pointer to a data structure containing the cryptographic key parameters of a connection must be passed by the application to the functions that handle the encryption of payloads. For OpenSSL/BoringSSL the struct is called `SSL` and for libnspr `PRFileDesc` (see Figs. 3 and 4). The path from the `SSL` struct to the MS is always the same. The path to the MS from a PRFileDesc struct is not the same since it uses a global array where all the cryptographic key material is stored. Thus, one element in the path is always different but within a defined range (the size of the array). Since these libraries are the most common options for Android applications, we only consider these two different types of paths for DroidKex.

5.3. Linking information

While an application is running, it can start several TLS connections. Consequently, we need to link a snapshot with the stack of the corresponding function call and the information from the network monitor. Additionally, the content on the stack of multiple function calls belonging to the same connection must be grouped. To do so, we use the file descriptor from the argument of the monitored network functions to resolve the corresponding TCP connection by looking up the source and destination IP address & port. By having this information, we can combine the information extracted by the network monitor (first data record, CR, etc.) with the information of the function call (content on the stack) by matching source and destination IP address & port. If we would intercept functions that do not have a file descriptor (e.g., the `SSL_read` function of BoringSSL), the task of linking a function call to a network connection has to be established differently.

5.4. Path computation

After having a snapshot together with the start and end point of a path in the memory, we can start computing paths in it. In order to calculate possible paths, we perform a depth search from start pointers (contents on the stack) to the MS. To do so, we dereference the first n bytes above the address where the pointer pointed to and test whether each pointer points m bytes before the MS, i.e., to a struct containing it. If it does not, we recursively continue searching by dereferencing again the first n bytes of the next structure and test if one of them points before the master secret. We continue that search until the path is longer than a defined value i (1) or the path reaches an address that has been visited before (2). If we found a path that ends m bytes before the master secret, we store that path and continue searching.

The values n, m and i have to be adapted to the assumed size of the data structures. By choosing them too small, the correct path cannot be found. By choosing them too big, the depth search takes longer and many false positives are found, i.e., when we search across the borders of a data structure and continue searching in the data structure mapped above the dereferenced one. The worst case complexity of the approach is $\mathcal{O}(stack\_size * n^i)$. However, most of the data values on the stack and in the data structures cannot be dereferenced which removes possible paths and lowers the search time.

To improve the speed, we store for each address in memory that we visit the computed paths. Thus, if an address is reached again, we do not need to compute the path again and can simply take it from the cache. This approach also helps mitigating getting stuck in loops, i.e., caused by pointer to the same data structure or lists.

5.5. Data path selection

The outcome of the depth search is that we usually have several paths for each snapshot and we have to decide which of them to choose.[3] The selection is implemented as a heuristic based iterative approach: first, we run the application with a set of paths (in the initial phase the set is empty which means that we take a snapshot for each TLS connection). If we did not extract the MS of all TLS

[3] We cannot take all paths and remove not working paths because too many possible paths slow down the extraction of the key and the normal operation of an application.

```
[8,256,96,16,40] → [-1,256,96,16,40] [8,-1,96,16,40] [8,256,-1,16,40] [8,256,96,-1,40] [8,256,96,16,-1]
[8,256,96,12,40] → [-1,256,96,12,40] [8,-1,96,12,40] [8,256,-1,12,40] [8,256,96,-1,40] [8,256,96,12,-1]
[8,256,96,36,40] → [-1,256,96,36,40] [8,-1,96,36,40] [8,256,-1,36,40] [8,256,96,-1,40] [8,256,96,36,-1]
[8,256,96,48,40] → [-1,256,96,48,40] [8,-1,96,48,40] [8,256,-1,48,40] [8,256,96,-1,40] [8,256,96,48,-1]
[8,296,96,48,40] → [-1,296,96,48,40] [8,-1,96,48,40] [8,296,-1,48,40] [8,296,96,-1,40] [8,296,96,48,-1]
[256,96,12,40]   → [-1,96,12,40]     [256,-1,12,40]  [256,96,-1,40]   [256,96,12,-1]
```

Fig. 5. Path expansion: the left side shows sample paths extracted from Mozilla Firefox; the right side shows the expanded paths; the rectangle highlights the selected path.

connections, we compute the paths out of the corresponding snapshots and select two paths that work for most of the snapshots we took. If two paths have the same coverage, we take the shorter path first. If both paths have the same size, we select one arbitrarily. Then, we repeat step one to check if the selected paths are enough to decrypt all connections of an application within a time frame or if more paths are required or if the path does not provide working MS. We iteratively repeat the two steps in the training phase until we can decrypt all connections of an application within a time frame. If we did not see any TLS connections during the run-time, we start the application again.

If a path did not extract any MS within a certain amount of rounds, we remove it from the set of paths. In each iteration, we remove paths that did not successfully extract a MS in the last four rounds and add it to a blacklist so that it is not selected in the future. Not working path can be caused for example when data structures are mapped closely together which produces paths that are only valid in that special case or when pointers on the stack point to ephemeral data structures that are used only during initialization. We progress with that approach (repeat step 1 and 2) until we can find a MS for each connection of an application and all paths were able to extract at least one MS.

Mozilla Firefox. The approach of finding a path by counting exact occurrences only works in the case of OpenSSL and BoringSSL paths, i.e., when the offsets in the data structures are always the same. In the case of Mozilla Firefox the paths are different since they contain one offset that is different for each path (the array). Thus, this approach of finding the exact same path does not work directly. In order to use it, we need to adapt it by searching a path that is the same except for one element. For example, the outcome of the path computation of four snapshots are the paths shown in Fig. 5 on the left side.

In order to find a generic path that extracts all secrets of Firefox, we have to define a path that can handle the array (see Fig. 4). To achieve that, we have to solve two problems: First, we have to find a group of paths that have only one different entry at the same position, i.e., the position of the array. Second, we have to identify at which position in the array the key of a certain connection is stored to extract from main memory.

To address the first problem, we expand all computed paths by inserting a variable (`-1`) at each position (see Fig. 5). Afterwards, we use the approach from above and choose the path which is the same in all four cases (here: [`8, 256, 96, -1,40`]). To address the second problem, we implement a simplistic approach that iterates over all elements in the array until the last element stored in the array is 0 or the index grows beyond a defined size. Then, the key managerhas to identify the correct MS out of the extracted values.

6. Evaluation and discussion

In this section, we measure the overhead and effectiveness of the DroidKex prototype, and discuss the limitations of our approach. To do so, we measure the time required for taking a snapshot and extracting the MS with the approach of intercepting the control flow and follow paths to it. In Section 6.1, we measure the size of common application snapshots, the time required to take it and the time to find the MS of a TLS connection in it. In Section 6.2, we measure the time required for computing the paths. In Section 6.3, we measure the time that is added to a network function call that is required whenever the control is intercepted to extract the MS.

All measurements in this section are executed on a Nexus 5x with Android 6.0.1. For testing, we chose a set of 86applications from the top hundred free applications in the Google Playstore that initiated TLS connections after starting them without any interaction. In each round, we execute the application for 90 s and take a snapshot 0.5 s after the key managerprocesses the message that the TLS session is negotiated. We do not take a snapshot 3.5 s after the negotiation to maintain usability of the application in the case when multiple snapshots are triggered.

6.1. Extraction from snapshot

DroidKex takes a snapshot only of memory areas that are likely to contain the MS, e.g., the heap and stack. To provide an estimation of the time for taking a snapshot and extracting the MS, we run the steps of the training phase with 86 applications. To access the process memory, we stop and resume the process execution using ptrace and access the memory using the mapped process memory file in the proc file system. In Fig. 6 we depict the histogram of the size of the snapshots for each application. Fig. 7 shows the relation between the size of the snapshot and the time required to take it, each dot represents one snapshot. Despite of the outliers which can be caused by background activities, a linear correlation can be asserted.

Fig. 8 depicts the time for finding the MS of a TLS connection in a snapshot. The time for taking a snapshot and extracting the MS depends mainly on the size of the address space and the position of the MS. The big variance in time for searching a MS is caused by two facts: First, the MS can be located in the beginning or in the end of a snapshot and we stop when we have found it. Second, the brute-force approach tests the entropy to enhance speed before actually decrypting the first record with it since the key derivation and decryption is computational intensive (Taubmann et al., 2016). Thus, the search is slower on snapshots with high entropies, e.g., when they contain compressed images.

Based on these measurements, we can conclude that the approach of taking a snapshot and extract the MS is not feasible for permanently monitoring applications that have multiple TLS connection at the same time, which is the case for most of the modern Android applications. The time required for taking a

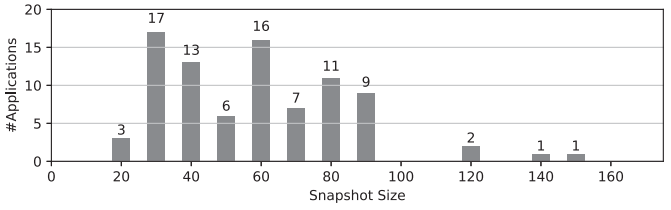

Fig. 6. Number of applications with the same memory size.

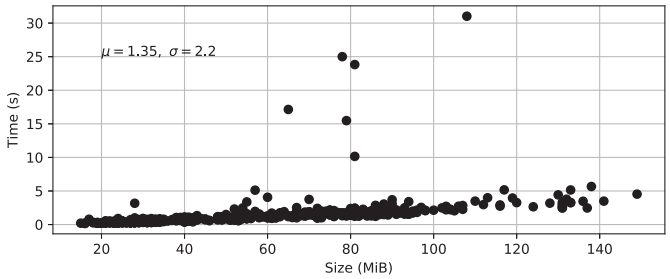

Fig. 7. Time to take the snapshot of an application in relation to its size. Every dot represents one application.

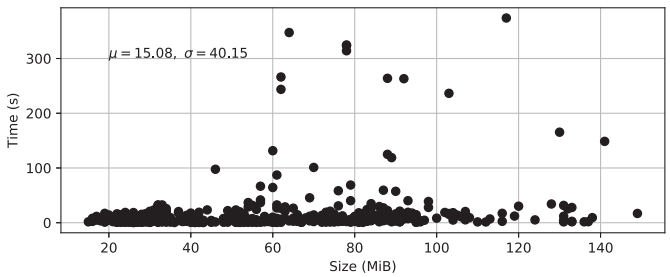

Fig. 8. Time to extract the key material of a snapshot. Every dot represents an application snapshot.

snapshot and extracting the MS is so high that application can not be used normally anymore if multiple connections are initiated at the same time. Thus, an approach with less overhead is required to extract the MS of the main memory of applications.

6.2. Path computation

To measure the performance of the path computation algorithm described in Section 5, we compute the paths for the 86 applications and measure the number of iterations and the required time until the algorithm converges. For these measurements, we use a struct size of $n = 700$ and use a limit of $i = 3$ iterations for the depth search.

Fig. 9 depicts the time that is required to compute the path of a snapshot compared to the number of pointers from the stack collected from the network function calls of a connection to the MS. The figure shows that the computation time depends on the number of possible start points to the path computing algorithm.

Fig. 10 shows the number of iterations that are required to compute the path for an application. The number of iterations defines the number of times an application was started and during the runtime a connection was found where the control flow

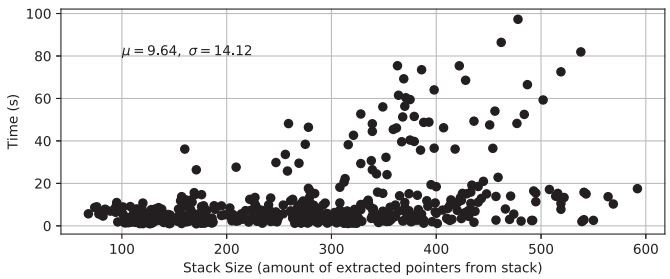

Fig. 9. Time to compute the data path compared to the size of distinct dereferenceable pointer on the stack collected from I/O calls.

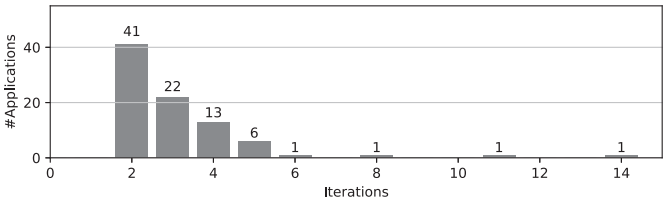

Fig. 10. Number of iterations required to compute paths.

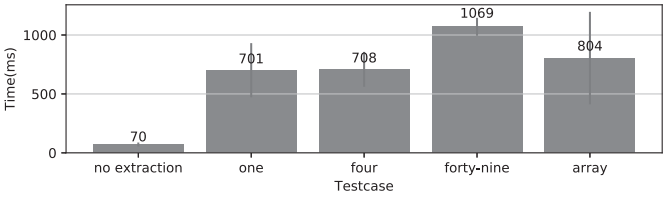

Fig. 11. Time for a single TLS https GET request with a different number of extraction paths.

interception did not provide a valid MS. In most of the cases, we were able to find valid offsets with one to four iterations.

During our evaluation, we extracted 12 distinct paths. Since about 95% of the applications only use one or two paths they can be used for most applications on the same device without recomputing them.

6.3. Control flow interception

The overhead added by intercepting the control flow is caused by two components: the overhead of the Frida interception mechanism and the time required to extract the MS. To measure the impact of Frida we call one native C++ function from Java which is similar to calling BoringSSL functions. Without Frida intercepting the control flow, the average time for doing 10,000 JNI calls 100 times is 31.58 ms (std. deviation: 1.94 ms). After activating the tracing of the same function, it takes 90.56 ms (std. deviation: 2.04 ms) which is an overhead of 5,9µs per function call.

In order to measure the time for the interception overhead added to normal TLS connections, we implemented a sample application that initiates 100 TLS connections to the same server with and without intercepting the network function calls. Without extracting the MS, the average time for a https GET request is 70.9 ms (std. deviation: 17.0 ms).

To measure the impact of different paths on the extraction process, we determine the optimal path of our sample application and modify it so that one, four and forty-nine offsets on the stack are used as a starting point.[4] Additionally, we tested one path that includes an array with an assumed size of 128 entries. For each possibility, we measured the average time for a https GET request (see Fig. 11).

The reasons for the longer run-time of a connection are: intercepting too many function calls that are not required, performance of the extraction routine and probing of none-required values on the stack and in arrays. Intercepting all network function calls of an application causes a high overhead since these functions are also used for other I/O operations such as file access. If we would only intercept one crypto library function, the overhead would be less since we would not need to search for an entry point on the stack

[4] The path computation found forty-nine possible entries on the stack that could lead to the MS. However, only one was working for data extraction.

and just take the argument of the corresponding function, e.g., SSL_read. However, with this approach, we would also lose the link to a certain network connection since we could not link a function call to a network connection based on its IP address. Additionally, this approach is less generic and requires knowing all TLS related function calls. Nevertheless, the impact of the extraction routine decreases for long living connections, since after successfully acquiring the MS we stop the extraction of the corresponding connection.

We also assume that it is possible to improve the overhead by optimizing the proof-of-concept extraction routine, e.g., by implementing the extraction routine in a low-level language and not in JavaScript which is required by Frida.

The performance of using a path that includes an array can be improved by implementing an approach that finds a path to the index value so that not all values of the array have to be tested.

6.4. Discussion and limitations

By using Frida for control flow interception, we are aware that we can only analyze applications that work when Frida is attached, e.g., applications that do not prevent the use of debuggers. Because of that, we can only trace applications that have one process since Frida has problems with tracing multiple processes.

The approach of DroidKex would be infeasible when the access to memory holding the key material is not granted to the key managersuch as using a secure keystore instead of the main memory. Such key store can be implemented for example using the TrustZone of the ARM architecture. However, to the best of our knowledge, there is no TrustZone application that provides such a service. Another approach is that parts of the TLS protocol implementation are integrated into the Linux kernel to improve the performance (Edge, 2015). In that case, the kernel has more control over the keys and could protect them from malicious user-space applications. But then the kernel could also easily log all SSL keys without the help of an application.

7. Related work

This section discusses different approaches and related work that addresses the problem of extracting cryptographic key material, decrypting network communication and recomputing data structures. There are three approaches for decrypting network communication, which are: the usage of a man-in-the-middle proxy that modifies the communication traffic, a passive approach of logging the network connection then decrypting by having access to the session key, and the interception of the application control flow in order to access the key material. Table 1 shows a comparison between implementations of the three approaches. The comparison is done based on performance overhead at run-time, in which layer the key extraction occurs, the ability to overcome certificate pinning, and the impact on the security of the encrypted connections.

7.1. Man-in-the-middle proxy

The usage of man-in-the-middle proxies is the most common approach to decrypt TLS communication. An example for this approach is (Mitmproxy, 2018) which operates on network level. Thus, it does not require access to or patching any of the communicating parties. It modifies network traffic to use self-generated certificates allowing it to de-/encrypt communication between the two entities. Haystack (Razaghpanah et al., 2016) is an application which can be installed on a regular Android device. It redirects all traffic of applications using a VPN tunnel to a man-in-the-middle proxy in order to decrypt their TLS traffic. For these approaches, it is necessary, that the application installs a self-signed certificate. If an application uses key pinning, i.e., check the server certificate, this approach is not feasible.

By actively modifying the crypto parameters of a TLS connection, man-in-the-middle approaches lower the security of the communication. (Carnavalet et al., (2016)) summarize the weaknesses of widely used TLS Proxy implementations. US-cert published an alert concerning the usage of these approaches (US Cert, 2017). Durumeric et al. (2017) also investigated the impact of proxy solutions to the security of TLS connections.

7.2. Knowledge Based

Another way to decrypt TLS communication is by knowing the cryptographic key material that is used for the symmetric encryption during a session. (Wireshark (2015)) and SSLdump (Iveson, 2014) are passive approaches that monitor the network traffic and decrypt TLS connections when appropriate key material is provided.

There are two ways to negotiate session keys with TLS, either using asymmetric encryption, or by using the key exchange algorithms such as Diffie–Hellman (DH) and Elliptic curve Diffie–Hellman (ECDH) protocol. In the first case, it is sufficient to know the corresponding private key of the server in order to extract and decrypt the session keys used in encrypted sessions following the TLS handshake. The private key can be obtained for example by extracting it from the hard disk of the server that uses the key. Saxon et al. (2015) describe a way how to efficiently extract RSA keys from virtual machines. If the position of a key in a memory dump is not known, heuristics help to identify potential keys. (Shamir et al., (1999)) describe theoretical approaches to find RSA cryptographic keys efficiently using stochastic information. Klein (2006) uses an alternative approach by searching for sequences of the ASN.1 encoding which is commonly used to store such keys.

When Diffie-Hellman (DH) or Elliptic curve Diffie-Hellman (ECDH) is used to negotiate the symmetric key, it is not possible

Table 1
Comparison of different TLS decryption solutions.

Type	Approach	Performance	Access	Cert. Pinning	Security Level
Proxy	mitmproxy (Project, 2018)	middle	Network	no	low
	Haystack (Razaghpanah et al., 2016)	middle	Android App	no	low
Decryption	ssldump (Iveson, 2014)		Private Key/Log	yes	good
	Wireshark (Wireshark contributors, 2015)		Private Key/Log	yes	good
Key Extraction	Tlskex (Taubmann et al., 2016)	low	Virtual Machine	yes	good
	TeLeScope (Caragea, 2016)	low	Virtual Machine	yes	good
Control Flow	Cuckoo (Bremer, 2015)	high	Windows Process	yes	good
	jsslkeylog (Schierl, 2017)	high	Java VM	yes	good
	Frida Pinning Bypass (Cipolloni, 2017)	middle	Android Process	yes	low
	DroidKex	**middle**	**Android Process**	**yes**	**good**

to obtain the symmetric session keys used in TLS sessions even if the private key of any peer is known. In such a case, session keys can be acquired if the application logs them or by extracting the MS from application's memory. The latter approach is followed by TLSKex (Taubmann et al., 2016) and Telescope (Caragea, 2016). These tools extract the cryptographic key material from the process memory in virtual machines. The main problem with these approaches is that they are comparatively slow and they do not scale for many connections.

7.3. Control flow interception

The Cuckoo sandbox (Oktavianto and Muhardianto, 2013) instruments the PRF function of the local security authority subsystem service (LSASS) process of Windows in order to extract SR and CR in addition to the MS (Bremer, 2015). However, this method does not work when an application comes with its own TLS implementation and hence does not use crypto services of the operating system.

Key material can also be extracted by intercepting the function calls of crypto libraries. For example, SSLKeyLog (Schierl, 2017) intercepts SSL function calls using Java agents mechanism. Since Dalvik/ART environments do not support Java agents, SSLKeyLog cannot be deployed in Android. Wu (2015) intercepts OpenSSL function calls in order to extract the MS from the memory of applications and relies on the fact that the exact layout of the data structures is known. He uses the GNU debugger (GDB) to intercept the control flow of applications.

Cipolloni (2017) describes an approach with the Frida framework that manipulates the control flow of Android applications so that it accepts the self-signed certificate of a man-in-the-middle proxy in order to bypass the problem of certificate pinning. As described before, this method can lower the security of the encrypted communication.

7.4. Narrowing the semantic gap

There are different approaches on how the semantic gap can be bridged in order to locate data in main memory. Sigpath (Urbina et al., 2014) implements a similar approach as DroidKexand uses a data centric approach to compute a path from an entry point (e.g., a symbol) to the required information by following the pointers in data structures. In the corresponding paper, Urbina et al. applied the approach to computer games in virtual machines. Dolan-Gavitt et al. (2011) narrow the semantic gap by monitoring the instructions of a process and isolate only those that are used to access the required information. Another approach to narrow the semantic gap is to analyze the source code and extract the layout of the data structures from it. This is for example done by volatility (The Volatility Foundation, 2018) or (Rekall (2015)) which store the information in profiles. Xu et al. (2017) provide a general overview of different strategies to bridge the semantic gap.

8. Conclusion

In this paper, we presented DroidKex, an approach for efficiently extracting data from main memory of applications by narrowing the semantic gap. We have shown the effectiveness of our approach, by extracting the TLS master secret of a connection from the memory of Android applications. DroidKexis a valuable tool for different use cases, e.g., for malware analysis on real devices or for live forensics and reverse engineering especially in cases when applications use certificate pinning. Since the approach of DroidKexdoes not require patching an application or installing a man-in-the-middle proxy that alters the traffic, the forensic soundness of the gained data can be increased. Another advantage of DroidKexis that the network traffic can be captured somewhere in the infrastructure, e.g., by the Internet service provider, and not on the device. Only the session keys must be transferred from the device to the forensic investigator. We evaluated our approach on Android but it can be adapted to different use cases on other platforms, e.g., for virtual machine introspection in cloud environments.

Acknowledgment

This research was supported by the Federal Ministry of Education and Research, Germany, as part of the BMBF DINGfest project 16KIS0503(https://dingfest.ur.de). Additionally, we would like to thank Stefan Kunz for his implementations that supported this project.

References

Bremer, Jurriaan, 2015. Transparent MITM with Cuckoo Sandbox. http://jbremer.org/mitm/ (Accessed: 10 February 2015).

Brubaker, Chad, 2016. Changes to Trusted Certificate Authorities in Android Nougat. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html (Accessed: 26 December 2017).

Caragea, R., 2016. TeLeScope - real-time peering into the depths of TLS traffic from the hypervisor. In: HITBSECCONF2016 - THE 7TH ANNUAL HITB SECURITY CONFERENCE.

Casey, E., Turnbull, B., 2011. Digital Evidence on Mobile Devices. Eoghan Casey, Digital Evidence and Computer Crime. Forensic Science, Computers, and the Internet, third ed. Academic Pres.

Cipolloni, Piergiovanni, 2017. Universal Android SSL Pinning Bypass with Frida. https://techblog.mediaservice.net/2017/07/universal-android-ssl-pinning-bypass-with-frida/ (Accessed: 26 December 2017).

de Carné de Carnavalet, Xavier, Mannan, Mohammad, 2016. Killed by proxy: analyzing client-end TLS interception software. In: Network and Distributed Systems Symposium (NDSS'16).

Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W., 2011. Virtuoso: narrowing the semantic gap in virtual machine introspection. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11. IEEE Computer Society, Washington, DC, USA, pp. 297–312.

Durumeric, Z., Ma, Z., Springall, D., Barnes, R., Sullivan, N., Bursztein, E., Bailey, M., Halderman, J.A., Paxson, V., 2017. The security impact of https interception. In: Network and Distributed Systems Symposium (NDSS'17).

Edge, J., 2015. TLS in the Kernel. https://lwn.net/Articles/666509/ (Accessed: 26 December 2017).

Elenkov, N., 2014. Android Security Internals: an In-depth Guide to Android's Security Architecture. No Starch Press.

Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M., 2012. Why eve and mallory love android: an analysis of android ssl (in)security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12. ACM, New York, NY, USA, pp. 50–61.

Iveson, S., 2014. Using Ssldump to Decode/decrypt SSL/TLS Packets. http://packetpushers.net/using-ssldump-decode-ssltls-packets/ (Accessed: 19 July 2015).

Klein, T., 2006. All Your Private Keys Are Belong to Us – Extracting RSA Private Keys and Certificates from Process Memory. http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf (Accessed: 19 July 2015).

Krawczyk, H., Paterson, K.G., Wee, H., 2013. On the Security of the TLS Protocol: a Systematic Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 429–448.

Maartmann-Moe, C., Thorkildsen, S.E., Årnes, A., Sept. 2009. The persistence of memory: forensic identification and extraction of cryptographic keys. Digit. Investig. 6, S132–S140.

Oktavianto, D., Muhardianto, I., 2013. Cuckoo malware analysis. Packt Publishing.

Mitmproxy, Project, 2018. Mitmproxy. https://mitmproxy.org/ (Accessed: 26 January 2017).

Ravnås, O., 2018. Frida - a World-class Dynamic Instrumentation Framework. https://www.frida.re/ (Accessed: 16 March 2018).

Razaghpanah, A., Vallina-Rodriguez, N., Sundaresan, S., Kreibich, C., Gill, P., Allman, M., Paxson, V., 2016. Haystack: a Multi-purpose Mobile Vantage Point in User Space. http://arxiv.org/abs/1510.01419v3.

Rekall, January 15 2015. Memory Forensics Analysis Framework. http://www.rekall-forensic.com.

Salowey, J., Zhou, H., Eronen, P., Tschofenig, H., 2008. Transport Layer Security (TLS) Session Resumption without Server-side State. https://tools.ietf.org/html/rfc5077.

Saxon, J.T., Bordbar, B., Harrison, K., March 2015. Efficient retrieval of key material for inspecting potentially malicious traffic in the cloud. In: 2015 IEEE International Conference on Cloud Engineering, pp. 155–164.

Schierl, Michael, 2017. jSSLKeyLog - Java Agent Library to Log SSL Session Keys to a File for Wireshark. http://jsslkeylog.sourceforge.net/ (Accessed: 26 January 2017).

Shamir, A., van Someren, Nicko, 1999. Playing "hide and seek" with stored keys. In: Proc. of the 3rd Int. Conf. on Financial Cryptography, FC '99. Springer-Verlag, London, UK, UK, pp. 118–124.

Taubmann, B., Frädrich, C., Dusold, D., Reiser, H.P., 2016. TLSkex: harnessing virtual machine introspection for decrypting TLS communication. Digit. Invest. 16, S114–S123. DFRWS 2016 Europe.

The Volatility Foundation, 2018. Volatility Framework. https://github.com/volatilityfoundation (Last accessed: 31 June 2017).

Urbina, D., Gu, Y., Caballero, J., Lin, Z., 2014. SigPath: a Memory Graph Based Approach for Program Data Introspection and Modification. Springer International Publishing, Cham, pp. 237–256.

US Cert, 2017. Alert (TA17-075A) HTTPS Interception Weakens TLS Security. https://www.us-cert.gov/ncas/alerts/TA17-075A (Accessed: 08 July 2017).

Wireshark contributors, 2015. Wireshark Wiki about Secure Socket Layer (SSL). https://wiki.wireshark.org/SSL (Accessed: 15 March 2018).

Wu, Peter, 2015. Sslkeylog. https://git.lekensteyn.nl/peter/wireshark-notes/tree/src/. Accessed: 26 December 2017.

Xu, X., Zhao, B., Wang, X., Zhao, R., Dec 2017. Research on semantic gap problem of virtual machine. Wireless Pers. Commun. 97 (4), 5983–6004.