Forensic Science International: Digital Investigation 49 (2024) 301766

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

TLS key material identification and extraction in memory: Current state and future challenges

Daniel Baier[a,*], Alexander Basse[b], Jan-Niclas Hilgert[a], Martin Lambertz[a]

[a] Fraunhofer FKIE, Germany
[b] Institute of Computer Science, University of Bonn, Germany

A R T I C L E I N F O

Keywords:
TLS
Transport layer security
Memory forensics
Network forensics
Live forensics
Malware analysis

A B S T R A C T

Memory forensics is a crucial part of digital forensics as it can be used to extract valuable information such as running processes, network connections, and encryption keys from memory. The last is especially important when considering the widely used Transport Layer Security (TLS) protocol used to secure internet communication, thus hampering network traffic analysis. Particularly in the context of cybercrime investigations (such as malware analysis), it is therefore paramount for investigators to decrypt TLS traffic. This can provide vital insights into the methods and strategies employed by attackers. For this purpose, it is first and foremost necessary to identify and extract the corresponding TLS key material in memory.

In this paper, we systematize and evaluate the current state of techniques, tools, and methodologies for identifying and extracting TLS key material in memory. We consider solutions from academia but also identify innovative and promising approaches used “in the wild” that are not considered by the academic literature. Furthermore, we identify the open research challenges and opportunities for future research in this domain. Our work provides a profound foundation for future research in this crucial area.

1. Introduction

With the advent of digital communication, the encryption of internet traffic has significantly increased. Around 85 % of internet traffic was encrypted in 2020, a significant increase from 55 % in 2017. This trend shows a substantial increase in data privacy and security. However, the same encryption that safeguards data from malicious actors also poses significant challenges in the realm of digital forensics (Gigamon, 2023; Pric, 2013).

The importance of TLS decryption in digital forensics cannot be understated. With the increasing sophistication of cyber threats, forensic investigators frequently encounter encrypted network data in their analyses (cf. (Papadogiannaki and Ioannidis, 2021)). This makes it necessary for investigators to be able to decrypt the encrypted network traffic in such scenarios.

In recent years, obtaining decrypted network traffic for forensic purposes and analyses has become more and more challenging for forensic researchers and law enforcement agencies. In the context of live forensics, one of the most prominent techniques for TLS decryption in the past was a man-in-the-middle attack using a network proxy. However, this approach poses problems, such as certificate invalidation and detectability (Jarmoc and Unit, 2012). In the realm of malware analysis, this may not pose a significant issue; nevertheless, it is crucial to avoid detection by malicious actors during investigations.

Despite the increasing prevalence of TLS 1.3 (cf. (Warburton and Vinberg, 2021)), research has predominantly focused on TLS 1.2 in memory forensics (cf. Chapter 3). This leads to a gap in the identification of key material in the memory of systems using the latest TLS standards. While current research is concentrated on live forensics and clustering TLS traffic (cf. (Chen et al., 2022; Kim et al., 2022; Xavier de Carné de Carnavalet and van Oorschot, 2023)), the fundamental challenges of digital forensic investigations in the field of network forensics remain underexplored.

This paper evaluates and systematizes advanced techniques, tools, and methods for identifying and extracting TLS key material in system memory. We bridge the gap between academic research and practical, field-tested approaches, considering post-mortem and live forensic scenarios. Additionally, we identify open research challenges and opportunities for future exploration in this domain. By doing so, our study lays a comprehensive groundwork while also highlighting the challenges

* Corresponding author.
E-mail addresses: daniel.baier@fkie.fraunhofer.de (D. Baier), s6albass@uni-bonn.de (A. Basse), jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), martin.lambertz@fkie.fraunhofer.de (M. Lambertz).

https://doi.org/10.1016/j.fsidi.2024.301766

Available online 5 July 2024

involved in effectively identifying TLS keys in memory. In summary, this paper makes the following contributions.

- A comprehensive overview of techniques, tools, and methodologies for identifying and extracting TLS key material in memory.
- Insights into the challenges of decrypting TLS 1.2 and TLS 1.3 traffic.
- Highlights areas where limited or no existing research has been conducted in TLS key material identification and extraction, outlining potential future research directions.

2. TLS fundamentals

The TLS protocol is the successor of SSL (Secure Sockets Layer). It is designed to secure the communication between two parties on the network by providing privacy and integrity (Allen and Dierks, 1999). The protocol consists of two separate protocols: the TLS Record Protocol and the TLS Handshake Protocol.

One of the most prominent usages of the TLS Record Protocol is HTTPS (HTTP Over TLS (Eric Rescorla, 2000)), but the TLS Record Protocol can be used to encapsulate any other higher-level protocol. It provides privacy through encryption of the data with a symmetric cipher (like AES or 3DES) and integrity by appending a message authentication code (MAC) using a secure hash function (for example, SHA or MD5).

The TLS Handshake Protocol, on the other hand, is used to initiate a connection and negotiate which cipher and hashing function should be used (cf. Section 2.1). A combination of those algorithms is called a cipher suite.

Nowadays, almost all websites support TLS version 1.2, and 66 % already support the latest version 1.3 (SSL Pulse). Moreover, according to a 2021 TLS Telemetry Report by F5 Labs, TLS 1.3 had become the preferred encryption protocol for 63 % of the top one million web servers (Warburton and Vinberg, 2021).

However, since TLS 1.2 is still the most supported version (cf (SSL Pulse)) and all directly related methods explained in Chapter 3 are also based on this version, the following Sections 2.1-2.3 describe TLS based on version 1.2 according to RFC 5246 (Rescorla and Dierks, 2008).

In the remainder of this paper, we will use TLS to refer to all versions from SSL 1.0 to TLS 1.3.

2.1. Handshake

A general overview of the handshake and the corresponding client/

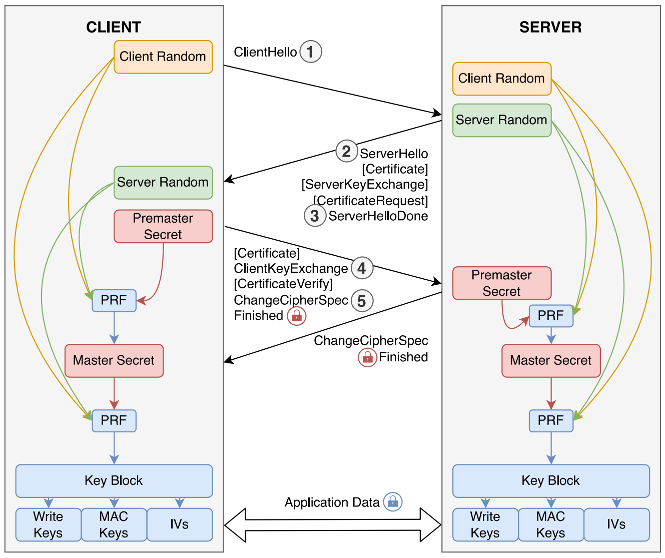

Fig. 1. Overview of the full TLS handshake (based on (Lee and Wallach, 2018)).

server key derivation can be seen in Fig. 1. The client initiates the connection with a `ClientHello` message (1 in Fig. 1).

This message contains the highest version of the protocol, all cipher suites, and all compression methods supported by the client. Additionally, it can include a session ID if the client wants to resume a previous session (cf. Section 2.3). Most importantly, this message also contains the client random, which is later used for the key derivation. The server now decides which algorithms to use and responds with the `ServerHello` message (2). It contains the highest version of the protocol supported by both parties and the choice of cipher suite and compression algorithm. Beyond that, this message also includes the server random.

If the server does not find a set of algorithms supported by both parties, it aborts the handshake with a handshake failure alert.

The server will send its certificate if the selected key exchange method is RSA. Otherwise, it will send its public Diffie-Hellman values with the `ServerKeyExchange` message.

The server can also request a certificate for client authentication with a `CertificateRequest` message. In the end, the server will finish the initialization phase with the `ServerHelloDone` message (3). Afterward, the client responds with its certificate if requested.

Then, the client sends the `ClientKeyExchange` message (4). This message contains either the premaster secret (PMS) encrypted with the server certificate or, in the case of Diffie-Hellman, the public client values. If the client has transmitted a digitally signed certificate, it sends a `CertificateVerify` message to verify its certificate.

In the end, both parties will send the `ChangeCipherSpec` message (5) followed by the `Finished` message. The `Finished` message is the first one encrypted with the just negotiated algorithms, keys, and secrets to verify that the handshake process was not tampered with.

This indicates that both parties are ready for encrypted communication, and the handshake is complete. All subsequent traffic is also encrypted, and both parties can start transferring the application data.

2.2. Key derivation

The TLS key derivation is based on a pseudorandom function (PRF) that generates outputs of arbitrary length. This PRF comprises the hashing function utilized in generating the Hash-based message authentication code (HMAC), characterized by three key parameters: secret, label, and seed. Internally, the label and seed are concatenated and used together as a seed for the hashing function to compute a hash over the secret.

The general key derivation process can be seen in Fig. 2. The master secret is computed using the PRF with the PMS and the client and server random. If RSA was chosen for the key exchange, the PMS is a 48-byte value generated by the client; otherwise, it is the negotiated key from the Diffie-Hellman key exchange and can be of different lengths.

The PMS is used as the secret for the PRF, the “master secret” as the label, and the concatenation of client and server random as the seed.

The resulting master secret is always 48 bytes long. At this point, the PMS is no longer needed and can be discarded.

The master secret is then used to derive the symmetric keys. This is done using the PRF to expand the master secret into a sequence of secure bytes called key block. Afterward, the key block is split into the following parts, where the initialization vector (IV) is only needed for implicit nonce ciphers: Client/Server write key, IV, MAC.

In this paper, the term *session keys* will be employed as a general descriptor to collectively refer to all keys utilized within a TLS session, as detailed above.

2.3. Session resumption

TLS supports session resumption by reusing negotiated secrets, thereby shortening the handshake.

The client can request resumption of a previous session by including the session ID in the `ClientHello` message. This is only possible if the

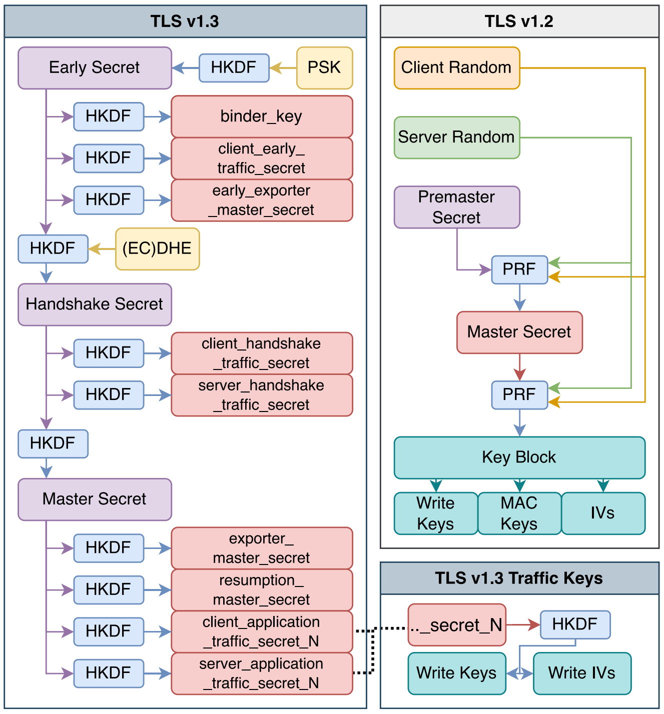

Fig. 2. TLS key derivation in version 1.2 and 1.3 (simplified) as outlined in RFC 5246 and RFC 8446 (Rescorla and Dierks, 2008; Rescorla, 2018).

selected cipher suite parameters and compression method match the old session.

If the server still stores the corresponding session, it responds with the same session ID in the `ServerHello` message. Otherwise, this field is set to zero, indicating to the client that a new connection should be established.

Resuming an older session leads to a reuse of the master secret. Therefore, the key exchange in the handshake can be skipped. However, the client and server random are taken from the new handshake. Thus, the session keys differ in a resumed session and still need to be derived.

2.4. *Differences in version 1.3*

The latest TLS version 1.3, specified in RFC 8446 (Rescorla, 2018), introduces some major changes; the most important ones for analysis are briefly described below.

Firstly, the handshake protocol was revised and shortened from the 4-way handshake to a 3-way handshake. All messages in the handshake after the `ServerHello` message are now encrypted. Moreover, all supported cipher suites must now provide perfect forward secrecy, resulting in the removal of all RSA and static Diffie-Hellman cipher suites.

As of TLS 1.3, the protocol has moved exclusively to using AEAD (Authenticated Encryption with Associated Data) ciphers. In AEAD ciphers, a single key is used for both encryption and authentication. When a message is encrypted, the AEAD algorithm also generates a MAC as part of the encryption process.

Furthermore, the key derivation process has been revised: the previously explained PRF has been replaced by an HMAC-based Extract-and-Expand Key Derivation Function (HKDF). As a result, it is no longer sufficient to retrieve the master secret for traffic decryption. Many other secrets have been added and are necessary to reveal the entire communication: `client/server_handshake_traffic_secret` and `client/server_traffic_secret_N`.

In addition to this, the capability to update traffic encryption keys during an ongoing communication has now been realized.

Finally, the session resumption mechanism has been replaced with a pre-shared key (PSK) exchange which is required for the Zero Round Trip Time (0-RTT) feature of TLS 1.3. In 0-RTT, the client can send data to the server in its first message, and this data is protected using keys derived from the `client_early_traffic_secret`. This secret is derived from an early secret which, in turn, is based on a PSK established in a previous TLS session.

Further differences can be found in RFC 8446.

2.5. *Traffic decryption keys*

In TLS 1.2, as explained in Section 2.2, the master secret, along with the client random and server random, is used to derive all the necessary key material for encryption and decryption (refer to the right part of Fig. 2). This means that by identifying the master secret, one can gain access to the information needed to decrypt the recorded TLS traffic.

In TLS 1.3, however, it is necessary to identify either the application traffic keys or the client/server handshake secrets in addition to the master secret to decrypt the application traffic. The application traffic keys enable decryption exclusively for the specific data stream where these keys are applied. In contrast, possessing the client/server handshake secrets allows for the decryption of the handshake messages, followed by the computation of necessary keys with the master secret to decrypt all subsequent traffic.

Fig. 2 (on the left side) provides an overview of how the individual keys are derived. The process begins by using the PSK in the HKDF to extract the early secret. This early secret is used in the HKDF to derive the `binder_key`, `client_early_traffic_secret`, and `early_exporter_master_secret`. For the first two secrets, the HKDF requires the ClientHello message and the early secret for key derivation. This initial step is only performed if a PSK is present; otherwise, the key derivation begins with the next step.

If present, the early secret is combined with the exchanged secret from (elliptic curve) Diffie-Hellman in the HKDF. If not, only the exchanged secret is used in the HKDF to extract the handshake secret.

This handshake secret is then used in the HKDF to derive the `client_handshake_traffic_secret` and `server_handshake_traffic_secret`. The HKDF requires both the `ClientHello` and `ServerHello` messages for this step.

Additionally, the master secret is derived from the handshake secret in the HKDF. Finally, the master secret is used in the HKDF to derive the final set of secrets, as shown in Fig. 2, bottom left.

In this last step, the HKDF utilizes all handshake messages from the `ClientHello` to `ServerFinished`. These derived secrets are then employed in the HKDF to obtain the write key, IV, and other necessary components for encryption and decryption.

Therefore, to decrypt the traffic, it is essential to identify not only the master secret but also the appropriate handshake secrets. Alternatively, the application data can at least be decrypted using the application traffic keys.

3. TLS key identification in memory dumps

Whenever a memory dump and its associated encrypted network traffic are available, dead forensic methodologies can be employed to extract TLS key material from the dump. The key advantage of this approach is the ability to conduct analysis independent of time constraints.

This section provides an overview of the various approaches that have been proposed to identify and extract TLS key material in memory dumps so far.

3.1. *Brute force search*

In cases where there is no prior knowledge of an application's memory structure and its implementation, brute force searching is a viable method for extracting key material from memory dumps. Taubmann et al. introduced TLSkex (Taubmann et al., 2016), a brute force approach to extract the master key to TLS connections at runtime from virtual machine memory, thus enabling the decryption of associated application data.

For this approach, the only requirement is a recording of the TLS connection that should be decrypted. The idea is to move a 48-byte sliding window over the memory dump and test every sequence as a master secret.

The drawback of this approach lies in its computationally intensive nature, particularly regarding decryption and the required key derivation. Therefore, Taubmann et al. present three different heuristics to decrease the search space as much as possible without eliminating too many potential keys. The first assumption is that secrets are 4-byte aligned, which already quarters the total search space. Furthermore, all secrets are generated by a PRF and should contain roughly the same amount of ones and zeroes. This allows to check if the amount of one bits in a byte sequence is within a certain range. The last heuristic implemented in TLSkex is to check whether an eight-byte sequence contains either only one bits or only zero bits. Combining all these heuristics should, in theory, be satisfied for 87.7 % of all master secrets (cf. (Taubmann et al., 2016)).

If the heuristics result in a secret missing, they can be disabled. Real-world experiments on memory dumps from Apache2, curl, Wget, and s_client from OpenSSL show that these heuristics can already reduce the search space to about 1.5–3.5 % of the total memory (cf. (Taubmann et al., 2016)). In their research, Sentanoe et al. (Stewart et al., 2022) developed another heuristic-based brute force approach to efficiently reduce the size of the heap being analyzed. This involves filtering the heap to remove irrelevant eight-byte segments based on simple criteria and then applying an entropy-based threshold to further refine the data. This results in a filtered heap that is more manageable for locating session keys.

The aspect of key omission by these heuristics was not addressed in their study.

Considering the focus on master secrets, these methods are limited to TLS versions up to 1.2 without further adjustments, as TLS 1.3 adopts a different key management approach.

3.2. *Pattern matching in unknown memory*

To find anything in large amounts of data, it is often helpful to identify characteristics of the searched-for data and locate data blocks that satisfy these characteristics.

One of the first theoretical approaches to locate cryptographic key material by searching for high-entropy regions was proposed by Shamir et al. (Shamir and van Someren, 1999). Their idea is based on the fact that cryptographic keys are chosen randomly, while most code and data is not. Random data generally has higher entropy than structured information. Therefore, it should be possible to distinguish cryptographic keys from regular data and code by selecting regions with unusually high entropy values.

Their experiments indicate that entropy, measured by counting unique byte values in data blocks, can effectively identify RSA secrets. It should be noted, however, that the success of this statistical method is highly dependent on the type of program and data analyzed.

Klein (2006) presents an approach to find and extract RSA private keys based on the ASN.1 syntax. The basic idea is that some types of cryptographic keys are stored in a standardized format referred as storage format. These storage formats can be used to create a signature and perform a simple but efficient pattern matching method to locate them in large amounts of data such as memory dumps.

Given that the employed signature comprises only seven bytes, its strength is limited, leading to a high incidence of false positives. Consequently, it is essential to verify the accuracy of the extracted keys in the final stage.

As only RSA keys are taken into account, these approaches can only work up to and including TLS 1.2.

3.3. *Leveraging unique structure identifiers*

When recovering cryptographic keys from applications or libraries with known data structures, targeted searches for these structures can be effective. This method extends to TLS libraries, which typically store the master secret in predictable data structures, as shown by the research of Anderson et al. (2019).

Listing 1: OpenSSL session structure with master secret (Anderson et al., 2019).

```
struct ssl_session_st {
    int ssl_version;
    unsigned int key_arg_length;
    unsigned char key_arg[8];
    int master_key_length; // 48
    unsigned char master_key[48];
    unsigned int session_id_length; // 32
    unsigned char session_id[32];
    ...
}
```

For the example of OpenSSL, the `ssl_session_st` structure depicted in Listing 1 is used to store the master secret.

It contains known values like the `master_key_length`, which is always 48, and the `session_id_length`, which is always 32. Additionally, the `ssl_version` used for the corresponding connection is present. This SSL/TLS version can be extracted directly from network traffic, as its value is transferred in plain text during the TLS handshake (cf. Chapter 2.1). In memory, the `ssl_session_st` is represented as shown in Fig. 3. The first two bytes 0x0303 indicate TLS version 1.2. Following, on the second line, the byte 0x30 defines the master secret length of 48. Afterward, an example of a master secret is highlighted and directly followed by a byte 0x20 representing the session ID length of 32.

This combined information can be synthesized into a regular expression serving as a searchable pattern within memory. This enables the extraction of all OpenSSL master secrets from the entire memory in mere seconds, making it an extremely efficient technique for retrieving TLS key material from memory dumps. Additionally, Anderson et al. delineate analogous expressions for the TLS implementations in BoringSSL, NSS, and Schannel.

Similarly, Kambic's research (Jacob, 2016) targets the Schannel TLS implementation in Windows. He identified specific 'magic values' within the Local Security Authority Subsystem Service (LSASS), which is essentially responsible for the TLS handshake process in Schannel. These values help to identify a structure containing session keys and another structure for the master secret. Within these structures, the corresponding keys are located at a specific offset. This insight led to the development of plugins for the Volatility and Rekall frameworks, enabling the extraction of these keys from memory dumps.

```
03 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 00 44 0E 70 5C 1C 22 45 07 6C 1C ED 0D
E3 74 DF E2 C9 71 AF 41 2C 0B E6 AF 70 32 6E C3
A3 2C A0 E6 3A 7A FF 0E F3 70 A2 8A 88 52 B2 2D
D1 B3 F6 F2 20 00 00 00 CD 31 58 BF DF 97 B0 F8
C0 86 BA 48 47 93 B0 A5 BA C1 5B 4B 35 37 7F 98
```

Fig. 3. Memory representation of an OpenSSL session (Anderson et al., 2019).

Based on this knowledge, Anderson et al. show in their patent (Anderson et al., 2020) that in the TLS libraries OpenSSL, BoringSSL, Schannel, wolfSSL, CoreTLS, and GnuTLS, the master secret consistently resides within a predictable data structure at a specific offset.

In their study, McLaren et al. (2019) adopted a methodology akin to Kambic's, centering their search on the ASCII strings `KSSM` and `3lss` within memory, for the TLS implementation Schannel. To allow for potential data structure changes, as may result from operating system upgrades, an entropy-based search is then carried out in a defined area of a few kilobytes to identify the master secret.

The fact that these approaches solely rely on particular characteristics that need to be present in memory could be disadvantageous. Missing these characteristics due to compiler optimization, etc., these expressions would not find any secrets. Further, these methods are limited to master secrets. Hence, they are only applicable up to TLS version 1.2 without adjustments.

3.4. Machine learning

Sentanoe et al. (Stewart et al., 2022) propose a machine learning approach to predict TLS 1.2 session keys based on patterns and features extracted from memory. To train their model, they utilize Virtual Machine Introspection (VMI) to extract the heap of a process using the TLS protocol. Additionally, they employ network monitoring tools to extract the necessary data.

To reduce the search space in the heap, the authors leverage the high entropy property of encryption keys by removing low-entropy data segments. This is achieved by reshaping the heap data into an 8-column matrix, as described in detail in (Stewart et al., 2022).

During the testing phase, the authors predict memory slices containing encryption keys. Furthermore, they predict the offsets within these slices to precisely locate the keys. This method enables the prediction of probable memory slices containing encryption keys and the corresponding offsets. However, it should be noted that the machine learning method can only determine if a key exists within a slice of data. Therefore, a brute force method (cf. Section 3.1) is employed to find the offsets within the probable slices.

Finally, the authors rank the slices based on the probability of the predictions and execute the most probable memory slices first. For the evaluation, the programs lynx and curl were employed. Their methodology successfully identified 99 % of all keys in memory.

4. TLS key identification in live sessions

A significant drawback of dead forensic approaches is that their results depend on the lifetime of the corresponding key material. When memory dumps are not readily practical (e.g., because the key material does not persist in memory) during a forensic investigation, live forensics approaches come into play.

Although live forensic approaches are more invasive than dead forensic approaches, we must remember that even a full memory dump is invasive to some extent due to the memory acquisition process.

This chapter overviews current methods for identifying and extracting TLS key material in live sessions.

4.1. SSLKEYLOGFILE

The simplest way is the `SSLKEYLOGFILE` supported by widely used cryptographic libraries such as OpenSSL and NSS (NSS Key Log FormatN). If enabled, the libraries write the TLS master secrets and the corresponding client random to the file specified in an environment variable. This approach works independently of the TLS version used. Although this approach is easy to deploy, nowadays, the default compilation process of some libraries deactivates the resulting callback functions (cf. (Mozilla Inc, 2023)). Furthermore, even if the used library supports this feature, it ultimately depends on the application whether it is passed to the library or not.

4.2. Debugging

When the target TLS library is compiled with debugging symbols, it is possible to parse the TLS structures with the debugger. The basic idea is to enable breakpoints in all TLS-related functions, and whenever a breakpoint is reached, the TLS object of these functions is parsed to retrieve the key material (cf. (Wu, 2023)). The lack of debug information in production builds significantly hampers the feasibility of these approaches, rendering them impractical in real-world scenarios.

4.3. Commencement-based structure traversal

Although popular libraries are often used to handle TLS connections, the concrete data structure and alignments in memory can still vary based on the compiler and its settings. An effective strategy to overcome these variations involves tracing paths from a known starting point directly to the precise location of the secret (e.g., master secret, application traffic secret, etc.).

Taubmann et al. (2018) developed DroidKex, a method tailored for the Android OS that leverages this principle to extract TLS keys.

In a training phase, a memory snapshot is created for a single application, and a path for extracting its master secret is then calculated.

The effectiveness of this approach hinges on a consistent memory layout for each memory snapshot. To achieve this consistency, hooking techniques are employed to intercept specific function calls (hooking-based approaches are further described in Section 4.5). These serve as triggers, utilizing pointers on the stack linked to function arguments as the foundation for path calculations.

Building on this, the method operates under the premise that cryptographic libraries invariably invoke certain network-related system functions for communication. Thus, functions are targeted for hooking, employing the stack as an initial reference point.

After intercepting the listed function calls, the path calculation works as follows. If a function from a cryptographic library like `SSL_read` calls the network function `read`, its stack frame must be somewhere above on the stack. Therefore, the first step is to find this stack frame and store the offset from which the path can be calculated. This defines the starting point.

The path calculation is iterative, involving a depth search from identified start points to the master secret, and uses a heuristic approach for path selection. An illustration of this is shown in Fig. 4.

Experiments conducted on 86 applications show that once all unique paths are identified for an application, they can be used to extract master secrets during run-time if one of the networking functions is intercepted without interrupting the execution of the application itself for more than 1 s.

Following the same principle, Pan et al. introduce a method called hyper TLS traffic analysis (HTTA) (Pan et al., 2019) evaluated on the Windows OS. Instead of using the stack content as a starting point, they choose a global variable.

For this, their approach leverages a feature in modern web browsers, where session structures are linked and cached in a specific memory region during session resumption. These linked structures are then referenced by global variables, a characteristic shared by various browsers. Consequently, if the addresses of the target global variables can be located, the session information can be extracted by traversing the hierarchical structures. This session information includes the key and parameters necessary for decrypting the traffic, which are stored in the memory space of the target process and referred as TLS session information (TSI) in (Pan et al., 2019).

Owing to the observed pattern, specifically that various structures are associated with a particular global variable in memory, the applicability of their approach is primarily confined to browsers and similar software applications.

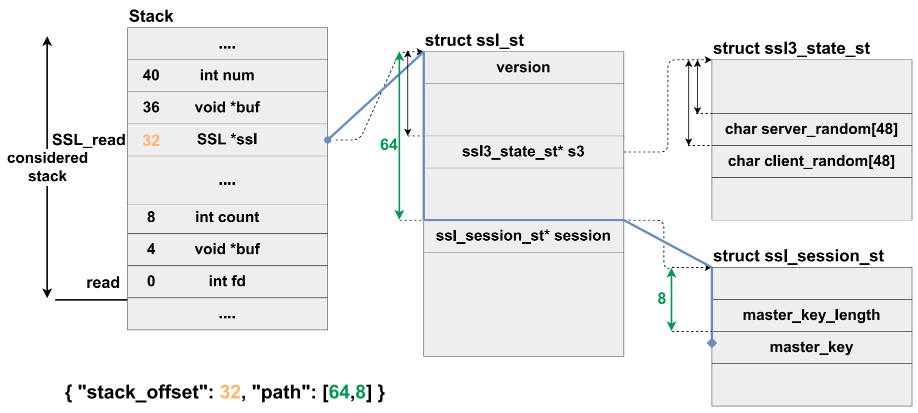

Fig. 4. Stack content when `read` is invoked and the corresponding path from the stack to the master secret located in the `ssl_session_st` struct (based on (Taubmann et al., 2018)).

4.4. *Memory diffing techniques*

The following methods leverage the TLS handshake to time memory dumps, specifically capturing only writable (dirty) pages. This reduces the memory dump size to pages modified during the handshake.

Caragea (2016) only targets TLS 1.2, beginning dumps at the `ServerHello` receipt and stopping after the client's `Finished` message. His method relies on live virtual machine migration to track dirty pages, making it suitable only for virtual environments. Nevertheless, it is OS and crypto-library agnostic. In order to identify the master secret, he utilized specialized known-plaintext attacks for current ciphers (cf. (Caragea, 2016)).

Similarly, Moriconi et al.'s X-Ray-TLS (Moriconi et al., 2024) targets TLS 1.2 and 1.3. It employs eBPF to monitor TLS connections and takes memory snapshots between the `ClientHello` and the first `ApplicationData` message. Utilizing a Linux kernel feature introduced into kernel v3.9 (cf. (Moriconi et al., 2024)), they track dirty pages and analyze snapshot differences to extract the secret keys. Then, they use a brute-force search for the TLS 1.3 handshake/application traffic secrets and the TLS 1.2 master secrets. This method is crypto-library agnostic but relies on Linux's memory tracking.

4.5. *Hooking*

Hooking is a technique used to intercept function calls of applications or libraries (Lopez et al., 2017). It allows injecting custom code to be executed before or after a certain function is called (Brubacher, 1999). Although this technique is not directly related to memory forensics, it can still be used as part of live forensics to extract TLS key material by hooking the functions used to generate the key material.

Various methods, such as Frida and LD_PRELOAD, are available for intercepting and manipulating software processes. This paper shifts focus from the hooking methods to the strategic considerations of what elements within a system to hook. This aspect is crucial as the effectiveness of function hooking for key extraction is not solely dependent on the how but significantly on the what.

4.5.1. *Hooking of key generation and deriving functions*

As described in Section 2.2, the PRF is used to generate the master secret and later derive the symmetric session keys. Curran et al. (Curran and van Bockhaven, 2016) present an approach of hooking the PRF `tls_handshake_internal_prf` in CoreTLS to extract TLS session keys from iOS devices. Through this process, the master secret and the client/server random are extracted by evaluating the arguments provided.

As the PRF has been replaced by an HKDF in TLS 1.3 (cf. Chapter 2.4), this approach is only applicable for TLS version 1.2. Furthermore, their approach was only developed for CoreTLS; therefore, it will not work without adjustments on other TLS implementations.

Choi and Lee (2016) retrieve the master secret by hooking the key derivation function of LSASS. Their method focuses solely on extracting the master secret, limiting its applicability to TLS 1.2. Additionally, the approach is specifically designed for Windows 32-bit systems.

The work from Noseevich follows a similar approach but also targets TLS 1.3 in hooking the different key generation functions of Schannel (George, 2022). As Schannel is used only in Windows, this approach only works for this platform.

4.5.2. *SSL_read and SSL_write hooking*

Each TLS library implements functions for the decryption/encryption of the payload and calls the corresponding `read` and `write` functions for this purpose. In OpenSSL, for example, such functions are called `SSL_read` and `SSL_write`. FriTap (Baier and Egner, 2022) implements hooks for these functions to extract plain text communication from applications using common TLS libraries. These functions generally have the same calling parameters, with the pointer to a TLS object as the first argument.

As shown in Fig. 4, this TLS object also contains the master secret in substructures. This should theoretically allow TLS secret extraction based on `SSL_read` and `SSL_write` function hooking by taking memory dumps of the TLS object and all further substructures. FriTap, for example, uses parsing of these structures to extract the TLS-key material from NSS (cf. (Baier, 2023)).

4.5.3. *Installing keylog callback functions*

The underlying mechanism of the `SSLKEYLOGFILE` (cf. Section 4.1) essentially operates through keylog callback functions. Thus, an alternative method involves setting up these keylog callback functions that are available in most TLS libraries. By doing so, it becomes possible to extract key material (cf. (Baier and Egner, 2022; Tunius, 2023; Valadon, 2022)).

The procedure involves hooking the function designated for initiating new TLS/SSL connections. Subsequently, the pertinent keylog callback functions are installed through this established hook. All these hooking approaches work for both TLS 1.2 and TLS 1.3, but in many cases, they are implemented only for TLS 1.2 (cf. Table 1).

5. Discussion and future challenges

Many methods exist to identify and extract TLS key material from memory dumps (cf. Chapter 3). What all methods have in common is that they consider only selected TLS implementations. Furthermore, the procedures are carried out exclusively up to TLS 1.2.

A brute force method (see Section 3.1), effective for data discovery

Table 1
Overview of supported TLS libraries and its support for TLS 1.3 in different hooking approaches.

SSL libraries	OpenSSL	BoringSSL	NSS	GnuTLS	wolfSSL	CoreTLS	Schannel	other
Hooking approach								
eBPF based hooking (Valadon, 2022)	◐	◐	○	○	○	○	○	○
friTap (Baier and Egner, 2022; Baier, 2023)	●	●	●	◐	◐	◐	●	◐
PRF Hooking (Curran and van Bockhaven, 2016)	○	○	○	○	○	◐	○	○
TLS keylogger (Tunius, 2023)	●	●	○	○	○	○	○	○
Choi and Lee (Choi and Lee, 2016)	○	○	○	○	○	○	◐	○
lsasslkeylog-easy (George, 2022)	○	○	○	○	○	○	●	○

● = supported TLS 1.2 and TLS 1.3; ◐ = supported only TLS 1.2; ○ = library not supported.

without prior knowledge, methodically examines every byte sequence in memory for potential matches. While this ensures the location of the targeted data, such as TLS 1.2 master secrets, it is highly inefficient due to the need to test every possible 48-byte sequence in a memory dump. The complexity escalates notably for TLS 1.3, as it necessitates identifying multiple secrets (handshake and traffic secrets for both server and client), increasing the search challenge tremendously due to unknown memory order and offsets. Hence, this technique is better reserved for situations without viable alternative methods.

The machine learning approach to identify TLS session keys from memory has a high success rate in identifying keys (cf. Section 3.4). A limitation is its inability to pinpoint exact key locations within data slices, requiring an additional brute force method for precise key localization.

Pattern matching, as described in Section 3.2, looks for specific patterns in memory, such as high entropy or standardized storage formats. Since all TLS secrets are generated randomly, entropy might be a good measure to narrow down the search to certain regions. An additional method, like a brute force search, remains necessary to locate the secrets within these regions. Integrating this with the initial approach can significantly reduce the complexity of the brute force search. However, it should be noted that searching for storage formats is not suitable for TLS keys when they are not generated via RSA, as they do not follow a strict, standardized storage definition.

Memory pattern search methods show potential, yet their effectiveness for identifying TLS key material in non-RSA contexts still needs to be investigated. One advantage of this approach is that the key material should be independent of the TLS implementation.

Finally, there is the search for program structures (cf. Section 3.3), which requires prior knowledge of the structures and certain known values or patterns within those structures to locate them. This approach is similar to a brute force search, with the advantage that matches must be validated only if the entire pattern matches the data. Depending on the quality of the patterns, there may be only a single match, making validation irrelevant since that single match should be the correct one.

Live session approaches, as discussed in Chapter 4, typically support a broader spectrum of TLS libraries and have started integrating TLS 1.3, marking a significant advancement. Nevertheless, the number of supported libraries and the support for TLS 1.3 still needs to be increased. Additionally, most live session methods focus on a single operating system, limiting their applicability across diverse environments.

Approaches like debugging or using the `SSLKEYLOGFILE` are often not feasible due to challenges like missing debug symbols or the inability to specify an `SSLKEYLOGFILE`. The HTTA method (cf. Section 4.3) requires additional research for identifying global variables in non-browser applications. This also applies to using the stack content as a starting pointer to traverse structures to identify the appropriate secrets.

On the other hand, hooking techniques, particularly those that install keylog callback functions, appear promising in various TLS implementations (cf. Section 4.5.3). The same applies to memory diffing techniques (cf. Section 4.4), enabling TLS library agnostic secret key extraction. The only disadvantage could be the applicability in mobile environments as they are more difficult to virtualize, and their distinct architecture and operating constraints can impede efficient memory analysis.

This results in the following unresolved challenges.

5.1. Challenge: limited TLS library support

Although the principles of previous work discussed in Chapter 3 should generally be applicable across different TLS libraries, the evaluation of these approaches against various TLS implementations is still necessary, as most of the prior work has been evaluated against specific TLS libraries only. Table 2 lists various approaches, indicating the specific TLS libraries used in the evaluation. It clarifies whether these approaches are tested for TLS 1.2 exclusively or if their applicability extends to TLS 1.2 and TLS 1.3. The table excludes work concentrating on pattern matching for RSA key material identification in memory dumps in light of the removal of RSA in TLS 1.3.

As shown in Table 2, none of the listed approaches was evaluated with all of the most common TLS implementations. An evaluation of the effectiveness of these methods is most informative when applied to prevalent TLS implementations.

The y-axis of Table 2 provides an overview of the common TLS libraries based on Wikipedia (Wikipedia contributors, 2024). While not exhaustive, this overview focuses on widely used, actively developed, and publicly available TLS implementations. It is extended by language-specific libraries like Rustls.

5.2. Challenge: TLS 1.3 key identification and extraction

Identifying TLS keys in memory dumps presents different challenges between TLS 1.2 and TLS 1.3, primarily due to the differences in the keys required for decrypting the traffic as described in Section 2.5.

To our knowledge, no research has been conducted on identifying TLS key material for TLS 1.3 in the context of dead forensics. Therefore, future research should investigate whether and how the presented methods can be used for TLS 1.3 and its generation of distinct keys and different memory structures.

In live forensics, there are already methods for identifying and extracting key material, which can also be applied to TLS 1.3. However, there is a notable gap in the comprehensive evaluation of these

Table 2
Overview of evaluated TLS libraries from previous work.

	Procedures						
	Brute force search		Leveraging unique structure identifiers				Machine learning (Stewart et al., 2022)
	TLSkex (Taubmann et al., 2016)	Entropy-based BF (Stewart et al., 2022)	Anderson research (Anderson et al., 2019)	Anderson patent (Anderson et al., 2020)	Kambic (Jacob, 2016)	McLaren (McLaren et al., 2019)	
TLS libraries							
Botan (Botan SSL, 2023)	○	○	○	○	○	○	○
BoringSSL (BoringSSL. Computer software, 2023)	○	○	◐	◐	○	○	○
Bouncy Castle (Bouncy castle, 2023)	○	○	○	○	○	○	○
Secure Transport(CoreTLS) (Apple Inc, 2024a, 2024b)	○	○	○	◐	○	○	○
GnuTLS (GnuTLS. Computer software, 2023)	◐	◐	○	◐	○	○	◐
Golang crypto/tls (The Go Authors, 2024)	○	○	○	○	○	○	○
Java Secure Socket Extension (Java Secure Socket Extension, 2023)	○	○	○	○	○	○	○
LibreSSL (LibreSSL. Computer software, 2023)	○	○	○	○	○	○	○
MatrixSSL (now Rambus TLS) (MatrixSSL, 2023)	○	○	○	○	○	○	○
Mbed TLS (Mbed TLS, 2023)	◐	◐	○	○	○	○	◐
Network Security Services (NSS) (NSS. Computer software, 2023)	◐	◐	◐	○	○	○	◐
OpenSSL (OpenSSL. Computer software, 2023)	◐	◐	◐	◐	○	◐	◐
Rustls (Birr-Pixton et al., 2024)	○	○	○	○	○	○	○
s2n (s2n-TLS. Computer software, 2023)	○	○	○	○	○	○	○
Schannel (Schannel SSP, 2023)	◐	◐	◐	◐	◐	◐	◐
wolfSSL (wolfSSL. Computer software, 2023)	◐	◐	○	◐	○	○	◐

● = evaluated TLS 1.2 and TLS 1.3; ◐ = evaluated only TLS 1.2; ○ = not evaluated.

techniques across all TLS libraries, as shown in Table 2. This highlights a critical area for further research, emphasizing the need for a more exhaustive analysis encompassing a broader range of TLS implementations.

Another aspect is to adopt procedures specific to TLS 1.2 to the changed properties of TLS 1.3. As discussed in Section 4.5.1, the PRF of the TLS library is hooked to retrieve the master secret. As TLS 1.3 replaces the PRF with the HKDF, it is necessary to analyze whether this approach is still applicable to TLS 1.3.

5.3. *Challenge: decrypting TLS 1.3 application traffic*

Finally, currently available tools such as Wireshark only support the decryption of TLS 1.3 traffic when all key material is available. Therefore, there is a need for an implementation that can decrypt these parts of the traffic for which the secrets are available.

5.4. *Challenge: key lifespan*

In TLS key material identification, the timing and duration of secrets in memory are crucial. Fig. 5 illustrates the minimal lifespan of these secrets in TLS 1.2.

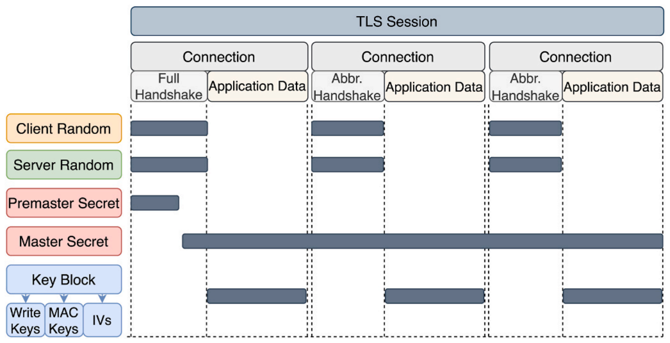

Fig. 5. Theoretical lifespan of TLS secrets (based on (Lee and Wallach, 2018)).

Essential base values, such as client random, server random, and the PMS, are needed only during the handshake, with the PMS being required only in the early phase to compute the master secret. Session keys, generated at the handshake's conclusion, remain in memory for the duration of the connection as they encrypt/decrypt application data.

The master secret, potentially used for multiple sessions through session resumption, has the longest lifespan. Following RFC 5246 (Rescorla and Dierks, 2008), session IDs, and thus master secrets, should not be stored beyond 24 h.

Therefore, the different lifetimes of the respective keys in the memory must be considered in future research for TLS 1.2 and TLS 1.3 with its corresponding TLS implementations.

6. Conclusion

Our study on identifying and extracting TLS key material in memory has revealed several critical insights and directions for future research. Firstly, there is a notable void in research on TLS 1.3 within the realm of dead forensics. This gap is significant, as TLS 1.3 handles and derives key material differently than TLS 1.2. Because of this, approaches working for TLS 1.2 will likely only work for TLS 1.3 with further adjustments.

Furthermore, our study highlights the substantial influence of TLS implementations on the storage and management of key material in memory. We compiled a list of common TLS implementations that should be prioritized in future research endeavors. This focus is crucial, given the diversity in how different TLS implementations handle key material within memory structures and patterns.

Another critical aspect of our findings relates to the lifespan of TLS keys in memory in the context of dead forensics. As our study reveals, there has been minimal exploration into the duration of these keys remaining accessible in memory after their initial use, especially for various TLS implementations. Investigating the duration that these keys persist in memory for the various TLS implementations, especially for both TLS 1.2 and TLS 1.3, thus emerges as a pivotal area for future studies.

Lastly, our research presents a structured classification of existing TLS key material identification and extraction methodologies. This categorization not only aids in understanding the current landscape but also serves as a foundational framework for future research, highlighting areas ripe for exploration and improvement.

In essence, this study not only elucidates the current state of TLS key material identification and extraction in memory but also underscores the emerging challenges and opportunities in this dynamic field.

References

Allen, Christopher, Dierks, Tim, 1999. The TLS Protocol Version 1.0. RFC 2246.

Anderson, Blake, Chi, Andrew, Scott, Dunlop, McGrew, David, 2019. Limitless HTTP in an HTTPS world: inferring the semantics of the HTTPS protocol without decryption. In: *Proceedings Of the Ninth ACM Conference On Data And Application Security And Privacy*, CODASPY '19. Association for Computing Machinery, New York, NY, USA, pp. 267–278.

Anderson, Blake Harrell, Chi, Andrew, McGrew, David, Dunlop, Scott William, 2020. Passive Decryption on Encrypted Traffic to Generate More Accurate Machine Learning Training Data. US Patent 10,536,268.

Apple Inc. coretls. https://github.com/apple-oss-distributions/coreTLS/tags, 2024, 2024-02-01.

Apple Inc, 2024b. Secure transport. https://developer.apple.com/documentation/security/secure_transport/, 2024-02-01.

Baier, Daniel, 2023. friTap - parsing structures to extract key material. https://github.com/fkie-cad/friTap/blob/main/agent/ssl_lib/nss.ts#L585, 2023-09-11.

Baier, Daniel, Egner, Francois, 2022. friTap - Decrypting TLS on the Fly. https://lolcads.github.io/posts/2022/08/fritap/, 2023-11-25.

Birr-Pixton, Joe, Ochtman, Dirkjan, McCarney, Daniel, Aas, Josh, 2024. Rustls - a modern tls library in rust. https://github.com/rustls/rustls, 2024-02-02.

2023-09-18 BoringSSL, 2023. Computer software.

2023-09-18 Botan SSL, 2023. Computer software.

2023-09-18 Bouncy Castle, 2023. Computer software.

Brubacher, Doug, 1999. Detours: binary interception of Win32 functions. In: Windows NT 3rd Symposium (Windows NT 3rd Symposium).

Caragea, Radu, 2016. Telescope-real-time Peering into the Depths of Tls Traffic from the Hypervisor. Bitdefender Labs.

Chen, Jianxi, Huang, Jiahao, Lu, Xinghua, 2022. Convolutional neural network-based identification of malicious traffic for TLS encryption. In: 2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP), pp. 1544–1549.

Choi, Hyoung-kee, Lee, Hyoseok, 2016. Extraction of TLS master secret key in Windows. In: 2016 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, pp. 667–671.

Curran, Tom, van Bockhaven, Cedric, 2016. TLS Session Key Extraction from Memory on iOS Devices. University of Amsterdam.

Eric Rescorla, 2000. HTTP over TLS. RFC 2818.

George, Noseevich, 2022. Decrypting Schannel TLS Traffic. Part 1. Getting Secrets from Lsass. https://b.poc.fun/decrypting-schannel-tls-part-1/#6-obtaining-tls13-keys, 2024-01-15.

Gigamon, 2023. The Importance of TLS/SSL Decryption for Network Security. https://blog.gigamon.com/2023/10/06/the-importance-of-tls-ssl-decryption-for-network-security/, 2024-01-15.

2023-09-18 GnuTLS, 2023. Computer software.

Jacob, M Kambic, 2016. Extracting Cng Tls/ssl Artifacts from Lsass Memory.

Jarmoc, Jeff, Unit, D.S.C.T., 2012. SSL/TLS interception proxies and transitive trust. Black Hat Europe.

2023-09-18 Java Secure Socket Extension (JSSE), 2023. Computer software.

Kim, Hyundo, Kim, Minsu, Ha, Joon-Soo, Roh, Heejun, 2022. Revisiting TLS-encrypted traffic fingerprinting methods for malware family classification. In: 2022 13th International Conference on Information and Communication Technology Convergence (ICTC), pp. 1273–1278.

Klein, Tobias, 2006. All Your Private Keys Are Belong to Us - Extracting RSA Private Keys and Certificates from Process Memory. https://www.trapkit.de/articles/all-your-private-keys-are-belong-to-us/, 2023-11-25.

Lee, Jaeho, Wallach, Dan S., 2018. Removing secrets from android's TLS. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society.

2023-09-18 LibreSSL, 2023. Computer software.

Lopez, Juan, Babun, Leonardo, Aksu, Hidayet, Uluagac, Selcuk, 2017. A survey on function and system call hooking approaches. Journal of Hardware and Systems Security.

2023-09-18 MatrixSSL - Now Rambus TLS Toolkit, 2023. Computer software.

2023-09-18 Mbed TLS, 2023. Computer software.

McLaren, Peter, Buchanan, William J., Russell, Gordon, Tan, Zhiyuan, 2019. Discovering Encrypted Bot and Ransomware Payloads through Memory Inspection without a Priori Knowledge arXiv preprint arXiv:1907.11954.

Moriconi, Florent, Levillain, Olivier, Francillon, Aurélien, Troncy, Raphaël, 2024. X-ray-tls: transparent decryption of tls sessions by extracting session keys from memory. In: ACM (Ed.), ASIACCS 2024, 19th ACM ASIA Conference on Computer and Communications Security, 1-5 July 2024, Singapore, Singapore, Singapore.

Mozilla Inc, 2023. NSS Makefile. https://github.com/mozilla/gecko-dev/blob/80432ae524a5360af40bb9c8b8e381008e9a001b/security/nss/lib/ssl/Makefile#L42C3-L42C69, 2023-09-11.

2023-09-18 NSS, 2023. Computer software.

NSS Key Log Format. https://firefox-source-docs.mozilla.org/security/nss/legacy/key_log_format/index.html. Accessed: 2023-11-27.

2023-09-18 OpenSSL, 2023. Computer software.

Pan, Jiaye, Zhuang, Yi, Sun, Binglin, 2019. Efficient and transparent method for large-scale tls traffic analysis of browsers and analogous programs. Secur. Commun. Network. 1–22 (10), 2019.

Papadogiannaki, Eva, Ioannidis, Sotiris, 2021. A survey on encrypted network traffic analysis applications, techniques, and countermeasures. ACM Comput. Surv. 54 (6), 1–35.

Pric, J.W., 2013. Significant SSL performance loss leaves much room for improvement. https://www.nsslabs.com/reports/ssl-performance-problems, 2024-01-15.

Rescorla, Eric, 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446.

Rescorla, Eric, Dierks, Tim, 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246.

2023-09-18 s2n-TLS, 2023. Computer software.

2023-09-18 Schannel SSP, 2023. Computer software.

Shamir, Adi, van Someren, Nicko, 1999. Playing "hide and seek" with stored keys. In: Proceedings of the Third International Conference on Financial Cryptography, FC '99. Springer-Verlag, Berlin, Heidelberg, pp. 118–124.

SSL Pulse - Best Protocol Support. https://www.ssllabs.com/ssl-pulse/. Accessed: 2023-12-15.

Stewart, Sentanoe, Fellicious, Christofer, Reiser, Hans P., Granitzer, Michael, 2022. "the need for speed": extracting session keys from the main memory using brute-force and machine learning. In: 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, pp. 1028–1035.

Taubmann, Benjamin, Frädrich, Christoph, Dusold, Dominik, Reiser, Hans P., 2016. TLSkex: harnessing virtual machine introspection for decrypting TLS communication. Digit. Invest. 16, S114–S123. DFRWS 2016 Europe.

Taubmann, Benjamin, Omar, Alabduljaleel, Reiser, Hans P., 2018. DroidKex: fast extraction of ephemeral TLS keys from the memory of Android apps. Digit. Invest. 26, S67–S76.

The Go Authors, 2024. Tls package - crypto/tls - go packages. https://pkg.go.dev/crypto/tls, 2024-02-02.

Tunius, Hugo, 2023. TLS Keylogger. https://codeshare.frida.re/@k0nserv/tls-keylogger/, 2023-09-11.

Valadon, Guillaume, 2022. When eBPF Meets TLS! *CanSecWest*.

Warburton, D., Vinberg, S., 2021. The 2021 TLS telemetry report. F5 Labs. WWW-dokumentti. Saatavissa. https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report [viitattu 11.2. 2023].

Wikipedia contributors, 2024. Comparison of tls implementations. https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations, 2024-02-01.

2023-09-18 wolfSSL, 2023. Computer software.

Wu, Peter, 2023. GDB Keylogger. https://git.lekensteyn.nl/peter/wireshark-notes/tree/src/sslkeylog.py, 2023-09-11.

Xavier de Carné de Carnavalet, van Oorschot, P.V., 2023. A survey and analysis of tls interception mechanisms and motivations. ACM Comput. Surv.