



DFRWS APAC 2025 - Selected Papers from the 5th Annual Digital Forensics Research Conference APAC

All your TLS keys are belong to Us: A novel approach to live memory forensic key extraction

Daniel Baier^{*}, Martin Lambertz

Fraunhofer FKIE, Zanderstr. 5, 53177, Bonn, Germany



ARTICLE INFO

Keywords:

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

ABSTRACT

Extracting TLS key material remains a critical challenge in live memory forensics, particularly for forensic investigators and law enforcement seeking to decrypt network traffic for investigative purposes. Existing methods focus on TLS 1.2 and rely on manual processes limited to specific implementations, leaving gaps in scalability and support for TLS 1.3.

This research introduces a novel approach that automates key aspects of identifying and extracting TLS key material across all major TLS implementations. Our approach leverages unique strings defined by TLS standards to identify key derivation functions, eliminating the need for manual identification and ensuring adaptability to evolving libraries.

We validate our methodology using a ground truth dataset of major TLS libraries and real-world applications, dynamically intercepting the identified functions to extract session keys. While initially implemented on Linux, the underlying concept of our approach is platform-agnostic and broadly applicable.

This work bridges a critical gap in live memory forensics by introducing a scalable framework that automatically locates TLS key derivation functions and uses this information in library-specific hooks, enabling efficient decryption of secure communications. These findings offer significant advancements for forensic practitioners, law enforcement, and cybersecurity professionals.

1. Introduction

Recent studies show that a vast majority of Internet traffic is now encrypted (Gigamon, 2023; Google Transparency Report, 2024). While encryption enhances privacy and secures data, it also poses substantial challenges for forensic investigators and law enforcement agencies, particularly in live memory forensics where the extraction of cryptographic key material is critical for decrypting network traffic (Lindenmeier et al., 2024).

Memory forensics is commonly associated with acquiring full memory dumps for later analysis. However, this approach is inadequate when decryption of network traffic is required, as key material may be transient. In such cases, key extraction-based lawful interception—focusing solely on recovering cryptographic keys from a monitored device—offers a more effective solution (Lindenmeier et al., 2024; Stoykova, 2023).

The significance of Transport Layer Security (TLS) decryption in

digital forensics cannot be overstated. As cyber threats become increasingly sophisticated, forensic analysts frequently encounter encrypted network data during their investigations (Papadogiannaki and Ioannidis, 2021). Thus, the ability to decrypt this traffic is essential for conducting thorough and accurate forensic examinations.

The evolution from TLS 1.2 to TLS 1.3 has fundamentally altered the protocol's key schedule (cf. Section 2.2), leaving most established forensic methods, which primarily target TLS 1.2 and focus on extracting only the master secret, unable to accommodate the restructured derivation process of TLS 1.3 (Baier et al., 2024). Existing approaches illustrate these constraints (cf. Section 3): Neither of these approaches achieves universal cross-platform coverage for TLS 1.3 sessions. As TLS 1.3 becomes the default for major web services (Warburton and Vinberg, 2021), forensic investigators find themselves ill-equipped to handle the altered secret generation model of TLS 1.3 (Baier et al., 2024), creating a widening gap in analyzing encrypted network traffic.

This paper proposes an approach that extends previous work to

^{*} Corresponding author.

E-mail addresses: daniel.baier@fkie.fraunhofer.de (D. Baier), martin.lambertz@fkie.fraunhofer.de (M. Lambertz).

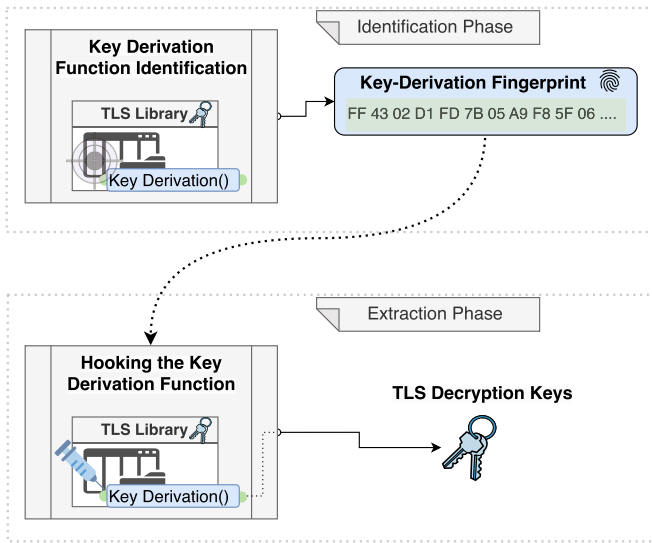


Fig. 1. TLS key extraction via intercepted key derivation.

extracting the key material required to decrypt TLS traffic consisting of two complementary phases (cf. Fig. 1). In the first phase (detailed in Section 5), we use automated static analysis to identify the TLS key derivation functions and extract their fingerprint signatures. Our approach considers specifically the Pseudo-Random Function (PRF) in TLS 1.2 and the HMAC-based Key Derivation Function (HKDF) in TLS 1.3. Once we have these fingerprints, we use them in the second phase (cf. Section 6), where live forensic analysis is applied to perform the actual TLS key extraction. To the best of our knowledge, no prior research has systematically addressed the identification of these TLS primitives to facilitate key material extraction by intercepting them during runtime.

In summary, this paper makes the following contributions:

- We present TLSKeyHunter, a framework that is able to identify the PRF and HKDF in an automated manner and extract signatures that can be used for hooking them during live forensics.
- We provide a ground truth dataset for all major libraries described in Baier et al. (2024), establishing standardized baselines for future research on TLS 1.2/1.3 decryption.
- We showcase how to utilize the hooking of the HKDF to identify and extract TLS key material.
- We provide Frida scripts for each TLS library to facilitate the TLS key extraction for TLS 1.2 and TLS 1.3 during live forensics.

The implementation of our approach and the ground truth dataset of TLS libraries are publicly available.¹

The remainder of this paper is structured as follows: Section 2 recaps TLS fundamentals followed by a related work section; Section 4 formalizes our static analysis; Section 5 details the implementation; Section 6 explains the live extraction; Section 7 introduces our dataset; Section 8 presents our evaluation; Section 9 concludes.

2. TLS fundamentals

TLS is a widely adopted protocol designed to secure communications between clients and servers over untrusted networks such as the Internet. TLS guarantees data confidentiality, integrity, and authentication (with optional client authentication) for exchanged information. It underpins a broad range of applications including HTTPS (Rescorla,

2000), QUIC (Thomson et al., 2021), HTTP/3 (Bishop, 2022), virtual private networks (e.g. OpenVPN (OpenVPN, Inc., 2017)), and industrial control systems (e.g., Modbus TCP Security (Modbus Organization, 2018)).

TLS has evolved considerably since its introduction. TLS 1.0, introduced in 1999 as an enhancement to SSLv3, was followed by TLS 1.1 (2006), TLS 1.2 (2008), and most recently TLS 1.3 (2018). Current best practices strongly recommend the use of TLS 1.2 and TLS 1.3, as earlier versions are deprecated and unsupported by modern web browsers (cf. SSL.com (2023); Internet Engineering Task Force (2021)).

A typical TLS session begins with a handshake that establishes a shared secret between the communicating parties. During the handshake, the client issues a ClientHello message, which advertises supported TLS versions, supplies a random nonce (the *client random*), and lists supported cipher suites. The server responds with a ServerHello message that indicates the negotiated TLS version, provides its own random nonce (the *server random*), and selects the cipher suite. Subsequent handshake messages are protected by keys derived—often via an Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) exchange—from these nonces. Following certificate-based authentication of the server, the handshake is concluded by the client with a Finished message, and application data is exchanged using keys derived during the handshake.

The remainder of this section outlines the keying material used to secure TLS 1.2 and TLS 1.3 traffic, including an in-depth description of the key schedule—an essential foundation for the discussion of our developed framework to automatically identify the key derivation function (cf. Section 5).

2.1. Traffic decryption keys

In TLS 1.2, all keys necessary for encrypting and decrypting traffic are derived from a single master secret.

This secret is computed from the PreMasterSecret (PMS) in conjunction with the client and server random values. Once established, the master secret is expanded into a key block that is segmented into the client write key, server write key, MAC keys, and—if applicable—initialization vectors (IVs). Hence, knowledge of the master secret yields the complete keying material needed to decrypt TLS 1.2 traffic (see Fig. 2a).

In contrast, TLS 1.3 relies on a multi-layered key derivation process in which decrypting recorded application data demands the recovery of specific secrets beyond the master secret (see Fig. 2b). The decryption of application messages requires the client and server application traffic secrets, whereas the decryption of handshake messages requires the client and server handshake traffic secrets. These handshake secrets also facilitate the derivation of subsequent application secrets. Consequently, obtaining only the master secret is insufficient for decrypting all TLS 1.3 traffic. Instead, each stage of the TLS 1.3 handshake generates context specific secrets that must be collected to enable full decryption.

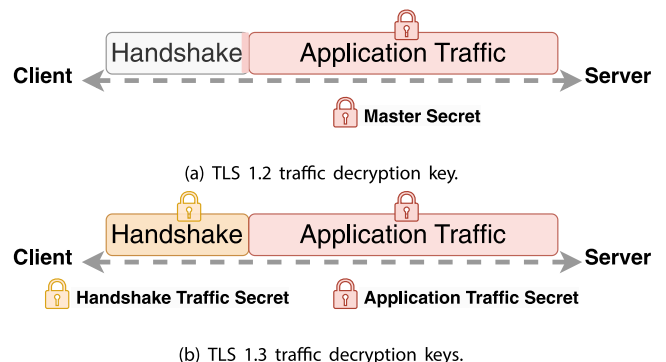


Fig. 2. TLS decryption keys in versions 1.2 and 1.3.

¹ <https://github.com/monkeywave/TLSKeyHunter>.

2.2. Key schedule

The TLS key schedule defines a sequence of operations that generate cryptographic keys from initial input secrets. This design is central to ensuring that every session produces unique keying material, even when certain inputs, such as a pre-shared key, are reused.

2.2.1. TLS 1.2 key schedule

TLS 1.2 employs a two-stage key derivation process based on a PRF, as specified in RFC 5246 (Rescorla and Dierks, 2008a). In the first stage, the master secret is derived from the PMS together with a seed. The PMS, which may be a fixed 48-byte value (e.g., in RSA key exchange) or an (EC)DHE-derived secret of variable length, is combined with the label “master secret” and the concatenation of the client random and server random values. A label is a fixed, context-specific string to ensure that each derived key is uniquely bound to its intended purpose (cf. Section 4). This two-stage key derivation process is illustrated in Listing 1.

```

master_secret = PRF(
    PreMasterSecret, "master secret",
    ClientHello.random + ServerHello.random)
[0..47];
    
```

Listing 1. Master Secret derivation using the PRF in TLS ≤1.2 (Rescorla and Dierks, 2008a).

When the extended master secret mechanism is employed (cf. RFC 7627 (Bhargavan et al., 2015)), the seed for the PRF is taken from a hash of the complete handshake transcript, rather than from the client and server random values.

Once the master secret is computed, the PMS is discarded. In all key exchange methods, the master secret has a fixed length of 48 bytes, as stated in Section 8.1 of RFC 5246 (Rescorla and Dierks, 2008b).

After the master secret is generated, a second PRF invocation is used for key expansion. This step uses the label “key expansion” and a seed composed of the random values. The resulting key block is then partitioned into the client and server write keys, MAC keys (for non-AEAD ciphers), and any required initialization vectors. Fig. 3 provides a visual summary of this two-stage process.

Internally, the PRF in TLS 1.2 is implemented through an iterative construct known as P_<hash>, which is based on the HMAC algorithm and employs a hash function determined by the cipher suite.

Through this mechanism, the PRF furnishes both the master secret and the expanded key block, ensuring that the final keying material is cryptographically bound to the handshake parameters. In this way, the PRF acts as the key derivation function in TLS 1.2.

2.2.2. TLS 1.3 key schedule

TLS 1.3 fundamentally redefines key derivation by replacing TLS 1.2’s PRF with a layered HKDF-based model as specified in RFC 8446

(Rescorla, 2018). While both protocols employ key derivation functions, their mechanisms diverge critically in structure. In TLS 1.2, a single PRF generates all secrets through repeated invocations with varying labels. Conversely, TLS 1.3 decouples key derivation into two distinct HKDF phases: HKDF-Extract for entropy concentration and HKDF-Expand for context-specific key generation, as formalized in RFC 5869 (Krawczyk and Eronen, 2010). Fig. 4 depicts the overall TLS 1.3 key schedule, showing how secrets are computed.

The TLS 1.3 key schedule initiates with the *Early Secret*, derived via HKDF-Extract using either a pre-shared key (PSK) or a zero-length input keying material (IKM) if no PSK is available. This secret enables 0-RTT data encryption in PSK-based sessions but is computed in all handshakes to maintain protocol uniformity.

Subsequent secrets are chained cryptographically: the *Handshake Secret* is computed by applying HKDF-Extract to a derived salt (from the Early Secret) and the ephemeral Diffie-Hellman shared secret. From this, the handshake traffic secrets are generated using the Derive-Secret function which is a thin wrapper around HKDF-Expand-Label, ensuring encryption of handshake messages. Finally, the application secrets (client/server traffic secrets, resumption master secret) are derived similarly, with each step cryptographically isolated through unique labels and handshake transcript hashes.

```

Derive-Secret(Secret, Label, Messages) =
    HKDF-Expand-Label(Secret, Label,
        Transcript-Hash(Messages), Hash.length)
    
```

Listing 2. Definition of Derive-Secret in TLS 1.3 (Rescorla, 2018).

Listing 2 formalizes the Derive-Secret mechanism. The Transcript-Hash parameter, a hash of all preceding handshake messages, cryptographically binds derived secrets to the specific protocol execution. This ensures keys are invalidated if messages are reordered, modified, or replayed, mitigating downgrade and replay attacks. In contrast, the Label parameter guarantees role separation by algorithmically distinguishing secrets used for different purposes, even when derived from the same input. For example, the server handshake traffic secret is derived with the label “s hs traffic”, while the client application traffic secret uses “c ap traffic”.

Note that the output length of Derive-Secret is determined by the underlying hash function (e.g., 32 bytes for SHA-256, 48 bytes for SHA-384).

In summary, TLS 1.2 relies on a single PRF (invoked twice) to generate two distinct secrets, incorporating the client and server random values as inputs. TLS 1.3 shifts the derivation of keys into multiple steps, where each cryptographic stage is derived from a previously computed secret.

3. Related work

Numerous prior efforts have aimed at extracting TLS key material through various hooking or memory analysis techniques. Most of these approaches focus exclusively on TLS 1.2, particularly targeting the master secret via the PRF. For instance, Curran and van Bockhaven (2016) hook the `tls_handshake_internal_prf` function in CoreTLS on iOS devices to extract session secrets. Similarly, Choi and Lee (2016) retrieve the master secret by hooking the PRF of LSASS on Windows, but their solution is limited to TLS 1.2 as well. Caragea (2016), Taubmann et al. (2018), and Pan et al. (2019) present related approaches also restricted to TLS 1.2, showing the general trend of targeting the master secret and not adapting to the fundamental changes introduced in TLS 1.3.

Noseevich (2022)’s technique partially addresses TLS 1.3 by hooking Schannel’s internal key derivation logic. However, the solution is platform-specific and remains confined to Windows environments. In contrast, the framework proposed by Moriconi et al. (2024) introduces a TLS library and version agnostic approach based on memory tracking

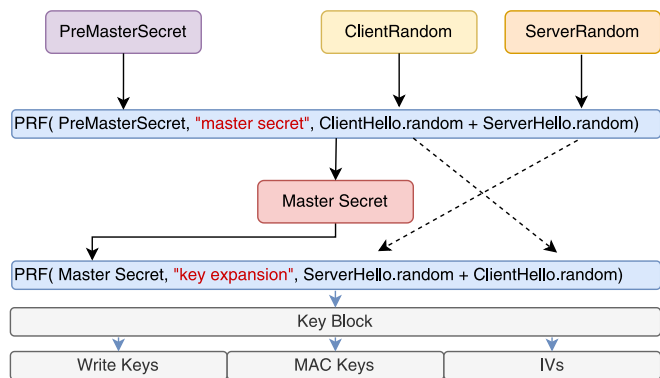


Fig. 3. TLS 1.2 key schedule (Rescorla and Dierks, 2008a).

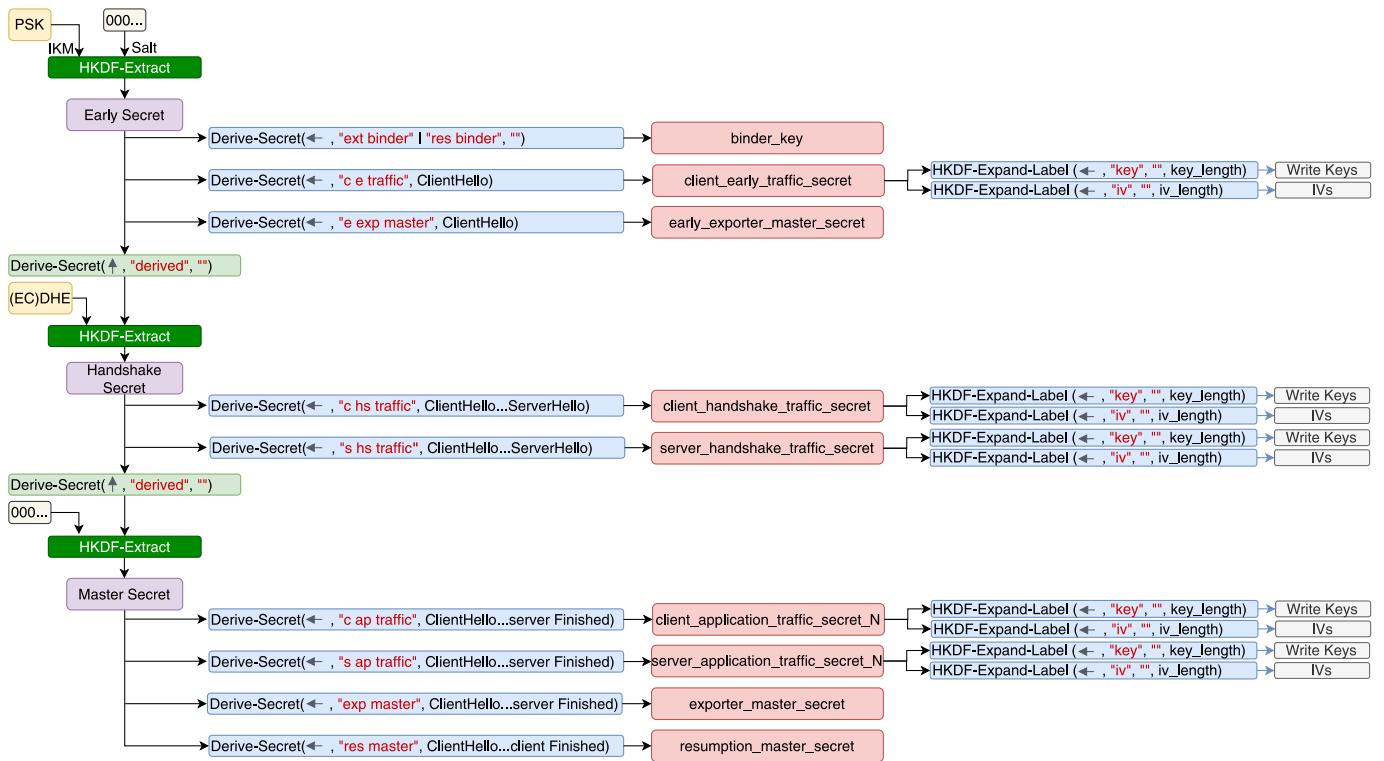


Fig. 4. TLS 1.3 key schedule as outlined in RFC 8446 (Rescorla, 2018). Secrets are derived hierarchically using HKDF. Arrows indicate the flow of secret derivation, with labels specifying the derivation context.

but is exclusively applicable to Linux systems.

Another class of solutions relies on installing keylog callback functions available in libraries like OpenSSL or NSS exposed via an SSLKEYLOGFILE. This is effective across TLS versions but often fails in practice due to compilation flags disabling the feature (Mozilla Inc., 2023).

Tools like friTap (Baier et al., 2020), FridaTLSKeylogger (Tunius, 2023), and eBPF-based hooks (Valadon, 2022) extend this idea by injecting or activating these callback functions via hooking mechanisms. However, they often depend on debug symbols or exported functions being available. Furthermore, none of these tools attempt to locate key derivation functions in an automated way; all rely on manually identified function signatures and are tuned to specific libraries.

In summary, current research suffers from several limitations: a focus on TLS 1.2 and the master secret, library- and platform-specific instrumentation, reliance on available symbols, and the absence of automation in locating key generation functions.

4. Key derivation identification

Identifying key derivation functions in TLS implementations presents a challenge due to the diversity of cryptographic libraries and the absence of symbolic information in compiled binaries. However, the TLS specifications mandate fixed derivation labels that serve as key separation identifiers for role-specific key derivation. These labels are not arbitrary but are an integral part of the key schedule, ensuring that specific secrets are derived correctly and in their appropriate context. Their primary objective is to bind the derived key material to context-specific information, ensuring for instance that secrets designated for handshake encryption cannot be misused for application data or session resumption.

In TLS 1.2, the PRF relies on the labels “master secret” and “key expansion” to define key derivation operations (cf. Section 2.2.1). TLS 1.3 replaces the PRF with a HKDF, using labels such as “c hs traffic” and

“s hs traffic” for handshake secrets and “c ap traffic” and “s ap traffic” for application secrets (cf. Section 2.2.2). Throughout this paper, we will refer to these labels as TLS labels. RFC-compliant TLS implementations require these standardized TLS labels; their absence prevents interoperability, as the derivation of shared secrets would not be compatible with each other.

Since these TLS labels must be passed as input to the key derivation functions, they provide a reliable starting point for identifying these functions across different TLS implementations, irrespective of their internal design.

Fig. 5 illustrates our approach in identifying the key derivation function on a conceptual level. At its core, the approach is built around finding functions in TLS libraries that take the aforementioned TLS labels as arguments.

The first step in the key derivation identification involves locating the TLS labels in the binary. Depending on the compiler and implementation, labels may be stored in various formats, including ASCII, UTF-16, wide character encoding, or hex representations. Compiler optimizations introduce further complexities. Some labels are placed in the .rodata section but referenced indirectly, while others appear as substrings within larger structures due to string interning or constant folding in languages such as Java or C++. We detect and cross-reference such variations to ensure accurate identification.

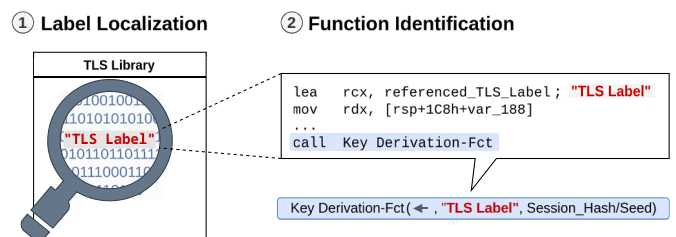


Fig. 5. Conceptual overview of the key derivation identification.

Once a TLS label is located, the next step is to analyze its data flow to determine how it is used. Simply identifying the presence of a TLS label is insufficient; it is essential to track its propagation through the binary to verify whether it is being passed as an argument into a function. The way in which a TLS label is transferred from its initial location to its eventual use as a function argument provides strong evidence for pinpointing a cryptographic key derivation function.

The following section describes TLSKeyHunter, which implements this idea by leveraging static analysis and forward data flow analysis (Alfred et al., 2007) to extract unique signatures of these key derivation functions from TLS libraries.

5. TLSKeyHunter

This section introduces TLSKeyHunter, our platform-agnostic static analysis framework designed to identify the TLS key derivation function through forward data flow analysis. By tracking the TLS labels and analyzing their propagation through the binary, TLSKeyHunter detects the key derivation function and extracts relevant patterns, offsets, and function labels. This extracted information enables live forensic key extraction, facilitating the recovery of cryptographic material from TLS sessions.

Although TLSKeyHunter is conceptually instruction set-agnostic, our current implementation targets ARM, ARM64, x86, and x86-64. These architectures dominate modern embedded, mobile, and desktop/server platforms, providing broad practical coverage. TLSKeyHunter is implemented using Ghidra (version 11.1.2), a widely used open-source disassembler. While Ghidra was used in our prototype, the underlying concept of TLSKeyHunter remains disassembler agnostic and adaptable to other tools.

5.1. Architecture

As shown in Fig. 6, TLSKeyHunter accepts a binary that either statically links a TLS library or is the TLS library itself. A preprocessing stage then ensures retention of critical information—such as strings and cross-references. Following this, the framework performs the TLS label localization as detailed in Section 4. Using forward data flow analysis, TLSKeyHunter tracks the propagation of an identified TLS label until it is passed as an argument to a function. This function is then assumed to be the key derivation function and its initial bytes are extracted to form a signature called Key-Derivation Fingerprint. This fingerprint is subsequently used to hook the function during live forensic analysis.

5.2. Key derivation identification

Identification of a key derivation function within a TLS library requires a structured approach capable of handling variations in implementations across different architectures and compilation strategies. The algorithm used for this purpose consists of three primary steps: identifying TLS labels, performing forward data flow analysis, and

validating the identified function.

To locate the key derivation function, we first search for predefined TLS labels in different representations, as discussed in Section 4. The detection process must account for compiler- and language-specific transformations that may obscure direct string references.

After identifying a relevant string reference, we perform forward data flow analysis to track its propagation through the binary (cf. Allen and Cocke (1976)). This analysis is conducted using an intermediate representation (IR) of the disassembled binary. In our case, we utilize P-Code, Ghidra’s IR, which abstracts low-level assembly instructions into a representation that is architecture-independent.

When a string reference is passed as a function argument, the function is flagged as a candidate key derivation function. This candidate must then be validated, as it may be a wrapper rather than the actual key derivation function. In such cases, the function processes the TLS label into an internal representation, encapsulates it in a data structure, and returns it for use in the true key derivation function. To distinguish between a wrapper and the actual key derivation function, we analyze the operations performed within the function. In particular, we focus on detecting string manipulation routines as a key indicator of wrapper behavior. For example, the presence of standard string operations—such as calls to `strcpy`, `strlen`, or custom loops that iterate over character arrays—suggests that the function is primarily processing an input label.

Another distinguishing characteristic is the function’s return value. Most genuine key derivation functions return only a success indicator, as their primary role is to generate keying material. In contrast, wrapper functions tend to return a constructed object or a pointer to a structure that is later processed by the actual key derivation function.

By incorporating these heuristics, we ensure that our identification process reliably isolates the actual key derivation function rather than an intermediate wrapper.

Although it is theoretically possible to detect cryptographic operations (such as HMAC calculations) within a function, in our observations, key derivation routines often do not execute these operations directly. Instead, they delegate the core cryptographic tasks to lower-level functions, obscuring the overall operation in the disassembly. Hence, we did not implement this heuristic in our prototype.

Finally, when identifying the PRF for TLS 1.0–1.2, we must distinguish between a single “unified” implementation (which dynamically selects the internal hash) and separate version-specific routines (e.g., TLS 1.0–1.1 with MD5/SHA1 vs. TLS 1.2’s P_Hash; cf. Section 2.2.1). Throughout this paper, we refer to the all-in-one approach as the “unified PRF”. If TLSKeyHunter detects multiple cross-references to the master secret label, it infers that the library provides separate version-specific implementations, and both are flagged as key derivation functions.

5.3. Key-Derivation Fingerprint

As shown in Fig. 6, once the key derivation function is identified, TLSKeyHunter extracts a pattern—the Key-Derivation

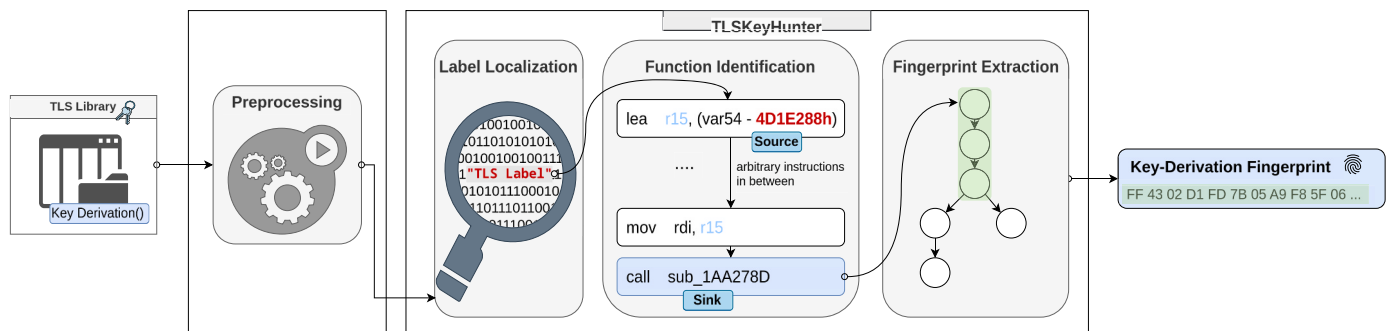


Fig. 6. Conceptual overview of TLSKeyHunter.

Fingerprint—from that function. This fingerprinting phase is modular by design, allowing future integration of more advanced techniques such as Germini (Xu et al., 2017), MCRIT (Plohmann et al., 2023), or N_Match (Xia et al., 2023). However, due to practical constraints of the hooking framework (cf. Section 6), we use a classical pattern approach relying on raw byte sequences.

In practice, the extraction process begins at the function prologue and continues until the first non-call branching instruction (e.g., a jump or a conditional branch). If this signature remains shorter than 32 bytes, the extraction is extended until a subsequent branch is encountered. This procedure ensures sufficient uniqueness for reliable identification, guided by prior research on function fingerprinting, including FLIRT (Hex-Rays, 2004), which suggests that a signature of roughly 16–32 bytes is typically adequate. The resulting byte pattern is paired with its offset in the target library, enabling precise identification of the key derivation function at runtime.

This final step complements TLSKeyHunter’s static analysis, which employs forward data flow analysis to locate the key derivation routine within the binary. Once identified, the extracted Key-Derivation Fingerprint allows to pinpoint and intercept the key derivation function during runtime.

6. TLS key extraction

To extract the necessary cryptographic key material for TLS traffic decryption, we hook into the key derivation function of the TLS library used within a target application. This process relies on the Key-Derivation Fingerprint generated by TLSKeyHunter, which provides a unique byte sequence associated with the key derivation function. By leveraging this fingerprint, we dynamically instrument the function responsible for key derivation and intercept the generated key material at runtime.

For the implementation, we utilize Frida (version 16.5.2), a dynamic instrumentation toolkit that supports multiple instruction set architectures (ABIs) and operating systems. The choice of Frida ensures broad compatibility, though the approach remains agnostic to the specific instrumentation tool. Alternatives such as DynamoRIO or Intel Pin could be employed for similar results.

We developed a custom hooking script that uses the precomputed Key-Derivation Fingerprints to locate and intercept corresponding function calls in real time. The implementation and signature of the key derivation function varies depending on the library. Notably, how the key material is returned is highly library-specific. We have to determine if the material is returned directly or if it is written into an output argument. If it is an output argument, we have to identify which argument exactly. Moreover, some libraries encapsulate the key material in complex data structures. Our hooking script is tailored to each TLS implementation we considered, ensuring the correct parameters are extracted from the function calls. We experimented with various heuristics to automatically determine the key material, such as entropy, but found them too error-prone. Moreover, our preliminary evaluations showed that the way a key-derivation function returns the key material is relatively stable. However, these aspects need further investigation in the future.

In TLS 1.2, the intercepted key material corresponds to the master secret (cf. Section 2.2.1), which is derived during the handshake and subsequently used to generate encryption and authentication keys. In TLS 1.3, the extracted key material comprises the handshake traffic secrets and the application traffic secrets, addressing the hierarchical key derivation structure that requires multiple secrets (cf. Section 2.2.2).

Alongside the key material, we extract the corresponding client random, which uniquely identifies the associated TLS session. This random value, exchanged during the handshake and stored in the session state, is used by tools like Wireshark to correlate key material with encrypted TLS streams for decryption. By integrating client random extraction into our method, we ensure accurate association of keys with

their respective TLS sessions.

In practice, the hooking is performed by executing the target application (including its TLS library) on a system where dynamic instrumentation is applied with administrative privileges. These privileges are required to attach to the target process and to intercept the key derivation function at runtime. Upon invocation of the key derivation function, our instrumentation layer intercepts the execution flow and extracts the computed secrets and the associated client random. These keys can subsequently be used to decrypt TLS-protected communication streams, facilitating traffic analysis.

Fig. 7 illustrates how the Key-Derivation Fingerprint is used to identify the function, which is then hooked. Once the function is intercepted, the key material is extracted and made available for further analysis. This methodology enables platform-agnostic key extraction by separating the function’s identification from the runtime context and relying solely on the Key-Derivation Fingerprint with its TLS library-specific hooking script. However, the hooking implementation requires library-specific adaptation. While the Key-Derivation Fingerprint automates identification of the target function across TLS libraries, the analyst must manually determine which function argument contains the secret material or client random and encode it according to the NSS key protocol specification. Depending on the analyst’s familiarity with Frida and the TLS stack, the required effort may range from minutes to hours and 10–50 lines of Frida code.

7. Ground truth dataset

At the time of writing, there was no comprehensive ground truth dataset of TLS clients covering both TLS 1.2 and TLS 1.3 across major TLS libraries. This absence poses a significant challenge for analyzing key derivation mechanisms and their respective cryptographic implementations. To address this gap, we systematically developed dedicated TLS clients for each major TLS library, enabling controlled testing and evaluation of their PRF- and HKDF-function.

The TLS clients developed in this paper can be found in our GitHub repository, with library versions documented in the `version.md` file. The selection of TLS libraries is based on the dataset presented by Baier et al. (2024), which categorizes widely used implementations and indicates that the TLS implementation influences the management of cryptographic key material in memory. For each library, we developed two distinct clients per TLS version: one performing a standard handshake and another that additionally prints the derived key material to the terminal. Consequently, each implementation includes four TLS clients—two for TLS 1.2 and two for TLS 1.3—allowing for both functional verification and forensic analysis.

The only exception to this selection is CoreTLS, which was excluded due to its deprecation (Apple Inc, 2025a). CoreTLS supports only TLS 1.2 and is part of Apple’s deprecated SecureTransport framework (Apple Inc, 2025b; Curran and Nigmatullin, 2015). Table 1 provides an over-view of the ground truth for each TLS library.

By establishing a well-defined set of TLS clients on x86-64, we provide a reproducible and structured approach for analyzing key derivation behavior across different implementations. This dataset forms the

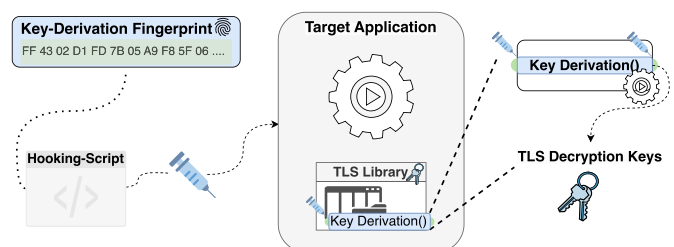


Fig. 7. TLS key extraction during live forensics using the TLS Key-Derivation Fingerprint.

Table 1

Evaluation of TLSKeyHunter's key derivation function identification across TLS libraries using the ground truth dataset.

Library Name	Unified PRF	PRFFingerprint	HKDFFingerprint
Botan SSL	Yes	✓	✓
BoringSSL	Yes	✓	✓
Bouncy Castle	Yes	✓	✓
Secure Transport	N/A	-	-
GnuTLS	Yes	✓	✓
Golang crypto/tls	Yes	✓	✓
JSSE	Yes	✓	✓
LibreSSL	Yes	✓	✓
MatrixSSL	No	✓	✓
Mbed TLS	Yes	✓	✓
NSS	No	✓	✓
OpenSSL	Yes	✓	✓
Rustls	No	×	×
s2n-TLS	Yes	✓	✓
Schannel SSP	Yes	✓	✓
wolfSSL	Yes	✓	✓

foundation for further research into automated key derivation identification and forensic analysis of TLS key material.

8. Evaluation

Our evaluation is centred on Linux x86-64, a decision motivated by practical and methodological considerations. The majority of TLS libraries analyzed here are distributed in a form already compiled for x86-64 by their developers, making it a comprehensive baseline for testing. To cover Windows-specific deployments, we additionally evaluated Schannel on Windows x86-64, as it is the only major TLS stack exclusive to that platform. We evaluate two main aspects of TLSKeyHunter's effectiveness. First, we measure its accuracy in locating the key derivation function within a given TLS library. Second, we verify whether the corresponding TLS key material and client random can be extracted for TLS 1.2 and TLS 1.3.

To validate correctness, we manually confirm that TLS-KeyHunter's identified function is indeed the key derivation function, and we hook it at runtime to extract the derived secrets. To validate the extracted secrets, we import them in NSS key-log format into Wireshark and decrypt captured TLS traffic, confirming the correctness of the recovered keys. This validation strategy, is applied to our ground truth dataset as well as a diverse set of real-world applications that utilize various TLS implementations.

8.1. Ground truth

Table 1 shows TLSKeyHunter's ability to identify the key derivation function on our ground truth dataset. A check mark (✓) indicates the successful identification of the key derivation function by extracting the Key-Derivation Fingerprint, while a cross (×) denotes that TLSKeyHunter was unable to identify the key derivation function.

Our approach successfully extracts the key derivation function patterns from all evaluated libraries. The only exception is Rustls, for which the extraction failed due to limited Rust disassembly support in Ghidra.

Table 2 summarizes the key material extraction results. The table indicates whether derived secrets and the corresponding client randoms were successfully extracted for the PRF and the HKDF. The extraction of the secret was successful across all libraries, both for the PRF and the HKDF. Rustls is the only exception again as we were not able to create Key-Derivation Fingerprints.

However, the extraction of the client random value exhibited divergence between protocols: it succeeded universally for TLS 1.2 PRF interceptions but failed for Botan SSL, LibreSSL, Mbed TLS, Rustls, Schannel, and s2n-TLS in TLS 1.3 HKDF contexts.

This discrepancy stems from fundamental protocol differences: in

Table 2

TLS Key Extraction results for key derivation components.

Library Name	PRF Extraction		HKDF Extraction	
	ClientRandom	Secret	ClientRandom	Secret
Botan SSL	✓	✓	×	✓
BoringSSL	✓	✓	✓	✓
Bouncy Castle	✓	✓	✓	✓
GnuTLS	✓	✓	✓	✓
Golang crypto/tls	✓	✓	✓	✓
JSSE	✓	✓	✓	✓
LibreSSL	✓	✓	×	✓
MatrixSSL	✓	✓	✓	✓
Mbed TLS	✓	✓	×	✓
NSS	✓	✓	✓	✓
OpenSSL	✓	✓	✓	✓
Rustls	×	×	×	×
s2n-TLS	✓	✓	×	✓
Schannel SSP	✓	✓	×	✓
wolfSSL	✓	✓	✓	✓

TLS 1.2, the client random is an explicit input to the PRF, whereas TLS 1.3's HKDF omits it from the key derivation logic. Nevertheless, some TLS 1.3 implementations propagate the client random via session state structures, which our approach leverages when passed as function arguments. Thus, extraction feasibility does not depend on cryptographic necessity, but on library-specific implementation patterns, a constraint our implementation addresses via its tailored hooking for each TLS library.

In cases where HKDF instrumentation provides derived secrets without an associated client random, the missing random value can be gathered from ClientHello messages in the recorded TLS traffic. Each derived secret is systematically paired with the recovered client random values to determine valid session key combinations.

We leverage the temporal correlation between client random values and derived secrets to streamline the pairing process. The sequential nature of TLS handshakes allows us to restrict candidate pairs to a sliding temporal window, reducing the practical complexity.

Client vs. Server Context: Since the code path for TLS secret derivation is identical for client and server roles, TLSKeyHunter treats them equivalently. To verify this empirically, we extended our ground truth dataset with a minimal TLS server using OpenSSL, BoringSSL, and GnuTLS. Applying the client-side fingerprints and hooks without modification yielded valid secrets in all cases, indicating that the method generalizes to server deployments.

8.2. Real world applications

In order to validate the approach with more complex programs we selected a diverse set of real-world applications. These applications span several widely deployed TLS implementations and provide a broad range of usage scenarios. The selected applications and their associated TLS libraries are summarized in Table 3.

These applications were chosen because they are either available directly as packages on Ubuntu (x86-64) or can be easily built from source. This selection ensures that TLSKeyHunter is evaluated on a

Table 3

TLS secret extraction success in real-world apps.

Client (Library Name)	PRF Extraction		HKDF Extraction	
	Client_Random	Secret	Client_Random	Secret
Chrome (BoringSSL)	✓	✓	✓	✓
Firefox (NSS)	✓	✓	✓	✓
Docker (Go lang crypto/tls)	✓	✓	✓	✓
curl (GnuTLS)	✓	✓	✓	✓
lighttpd (Mbed TLS)	✓	✓	×	✓
Powershell (Schannel)	✓	✓	×	✓

representative mix of TLS implementations in real-world environments. Detailed version information are provided in our repository.

The evaluation results align with those obtained from the ground truth evaluation, despite the programs using different versions of the TLS library. This consistency highlights that TLSKeyHunter operates independently of implementation-specific details such as memory layouts.

These results underscore the robustness of our approach. The generated Key-Derivation Fingerprints are sufficiently distinctive to identify the corresponding key derivation functions even in complex software. Moreover, our hooking technique also worked for different versions without modifications, indicating that the way the relevant structures are used in the libraries tend to be relatively stable.

Finally, no false positives were observed in any evaluation; all identified functions were confirmed as key derivation routines. While false positives are theoretically possible, their absence—even in large applications like Chrome—suggests the heuristics are sufficiently precise. This precision may partly be rooted in characteristics of the dataset, which includes minimal test cases and benign real-world applications. Future work should explore conditions that may lead to false positives, including the use of deliberately obfuscated samples, and develop mitigation strategies.

8.3. Cross-platform applicability

To demonstrate the platform-agnostic nature of our approach, we conducted preliminary evaluations beyond Linux x86-64. Specifically, we evaluated BoringSSL on Android across multiple architectures including ARM, ARM64, x86, and x86-64. Additionally, we assessed the Windows-specific Schannel library on both x86-64 and ARM64 architectures. In all evaluated configurations, TLSKeyHunter successfully identified the target key derivation functions and extracted the corresponding TLS secrets.

These results provide initial evidence for the cross-platform applicability of our method. The successful identification of key derivation functions across different operating systems (Linux, Android, Windows) and processor architectures (x86, x86-64, ARM, ARM64) suggests that our approach's fundamental principles are indeed platform-agnostic. A comprehensive evaluation across all target platforms remains important future work to fully validate our technique's robustness across diverse environments.

8.4. Identification runtime

To assess TLSKeyHunter's performance, we measured the time required to identify key derivation functions in three representative binaries from our dataset: the smallest (libmbedtls), the median-sized (libgnutls), and the largest (Chrome). Ghidra's preprocessing dominates the end to end runtime, and its share grows with binary size: it represents $\approx 47\%$ of the time for the libmbedtls library, 77% for libgnutls, and virtually all the time ($> 99\%$) for the Chrome image (cf. Table 4).

In contrast, TLSKeyHunter exhibits only modest sensitivity to input size. Moreover, key derivation identification is performed once and does not affect subsequent hooking.

A in-depth performance evaluation is left for future work, which should quantify runtime overhead—including hook latency, extraction time, and throughput under load—given the known impact of dynamic instrumentation frameworks like Frida (cf. sec. 6.3 in Taubmann et al. (2018)).

8.5. Limitations and future work

A limitation of TLSKeyHunter is its reliance on manual identification of cryptographic parameters during HKDF or PRF hooking. While our evaluation showed robustness across TLS implementations, the current

Table 4
TLSKeyHunter performance.

Target	Size	Ghidra	TLSKeyHunter
libmbedtls	464 KB	10 s	11.35 s
libgnutls	8.7 MB	33 s	10.06 s
Chrome 124.0	254 MB	73 144 s	111.10 s

approach requires knowledge of which parameter in a key derivation function corresponds to the target secret. Automating this identification should be a priority for future work.

Additionally, TLSKeyHunter currently depends on manual parsing of library-specific SSL structures to extract the client random. Automating this process—for example, by developing heuristics to detect and interpret memory layouts of TLS session objects—would eliminate implementation specific assumptions and broaden compatibility with obscure or proprietary libraries. Furthermore, while our evaluation focused on x86-64 architectures, TLSKeyHunter's platform-agnostic design warrants validation on alternative platforms, particularly in environments where memory layout or endianness may differ.

TLSKeyHunter relies on the assumption that standardized label strings defined in RFC 5246 and RFC 8446 appear in plaintext within the target binary. If a vendor encrypts or compresses these literals—decrypting them only at runtime before HKDF/PRF execution—the static fingerprinting phase would fail to locate the key derivation function, rendering the live hook ineffective. While technically feasible, such heavy-handed obfuscation is uncommon in production-grade TLS libraries as it adds measurable start-up latency, complicates string interning, and hampers debugging; we found no instance of it in any mainstream TLS libraries. On Windows systems, the Schannel TLS implementation operates within the standardized Local Security Authority Subsystem Service (LSASS), further mitigating obfuscation opportunities. Malware likewise tends to use standard OS-provided TLS libraries (e.g., OpenSSL on Linux, Schannel on Windows) to maximize compatibility, leaving TLS labels unmodified and detectable. However, sophisticated attackers may adapt by deliberately obfuscating or customizing TLS implementations, potentially reducing the reliability of label-based detection. Currently, such adaptations are uncommon due to complexity and reduced compatibility, but they remain a noteworthy limitation. Future enhancements to our approach could incorporate heuristic-based methods to address these potential challenges, strengthening resilience against evolving obfuscation techniques.

9. Conclusion

In this paper, we have demonstrated that hooking the HKDF is sufficient to extract all secret keys necessary to decrypt TLS 1.3 traffic. Furthermore, we showed that hooking the PRF in TLS 1.2 not only enables the extraction of the master secret but also the client random, which is essential for associating the extracted keys with specific TLS sessions, regardless of the TLS library used.

We applied our approach to a ground truth dataset comprising major publicly available TLS libraries, achieving successful key material extraction for both TLS 1.2 and TLS 1.3 in the context of live forensics. Additionally, we tested our method on real-world applications, further demonstrating its practical applicability.

A key contribution of this paper is a two-phase workflow to live TLS key extraction. The identification phase is fully automated: static analysis derives fingerprints for the PRF (TLS 1.2) and HKDF (TLS 1.3) without any prior knowledge of the target library. In the subsequent extraction phase, these fingerprints steer lightweight Frida hooks that capture the secrets at run time; the only manual input is a small, library-specific mapping. This largely automated solution not only streamlines the key extraction process but also helps adaptability to new libraries or substantial updates to existing ones. By providing the ground truth dataset and a reproducible methodology, our work offers a scalable and

flexible framework for TLS key extraction, bridging critical gaps in live memory forensics.

Acknowledgments

We would like to thank Julian Lengersdorff who provided invaluable assistance in building the majority of our ground truth dataset. His efforts were instrumental in ensuring the breadth and reliability of our test environment.

References

- Alfred, V.A., Monica, S.L., Jeffrey, D.U., 2007. *Compilers Principles, Techniques & Tools*. Pearson Education.
- Allen, F.E., Cocke, J., 1976. A program data flow analysis procedure. *Commun. ACM* 19, 137.
- Apple Inc, 2025a. coreTLS - TLS security. <https://support.apple.com/en-ca/guide/security/sec100a75d12/web>. (Accessed 1 February 2025).
- Apple Inc, 2025b. Secure transport. <https://developer.apple.com/documentation/security/secure-transport>. (Accessed 1 February 2025).
- Baier, D., Basse, A., Hilgert, J.N., Lambertz, M., 2024. Tls key material identification and extraction in memory: current state and future challenges. *Forensic Sci. Int.: Digit. Invest.* 49, 301766.
- Baier, D., Lengersdorff, J., Ufer, M.J., 2020. friTap: real-time key extraction and traffic decryption for security research. <https://github.com/fkie-cad/friTap>.
- Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., Ray, M., 2015. Transport layer security (TLS) session hash and extended master secret extension. RFC 7627. <https://www.rfc-editor.org/info/rfc7627>.
- Bishop, M.E., 2022. HTTP/3. RFC 9114. <https://datatracker.ietf.org/doc/html/rfc9114>.
- Caragea, R., 2016. Telescope-real-time peering into the depths of tls traffic from the hypervisor. Bitdefender Labs.
- Choi, H.k., Lee, H., 2016. Extraction of TLS master secret key in windows. In: 2016 International Conference on Information and Communication Technology Convergence (ICTC). IEEE.
- Curran, T., van Bockhaven, C., 2016. TLS Session Key Extraction from Memory on iOS Devices. University of Amsterdam.
- Curran, T., Nigmatullin, M., 2015. Tls session key extraction from memory on ios devices. <https://rp.os3.nl/2015-2016/p52/presentation.pdf>. (Accessed 1 February 2025).
- 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/>. (Accessed 15 January 2025).
- Google Transparency Report, 2024. Https encryption in transit. <https://transparencyreport.google.com/>. (Accessed 11 February 2025).
- Hex-Rays, 2004. Flirt: fast library identification and recognition technology. <https://docs.hex-rays.com/user-guide/signatures/flirt/ida-f.l.i.r.t.-technology-in-depth>. (Accessed 18 February 2025).
- Internet Engineering Task Force, 2021. Deprecating TLS 1.0 and TLS 1.1. Technical Report. IETF. <https://datatracker.ietf.org/doc/rfc8996/>. RFC8996.
- Krawczyk, H., Eronen, P., 2010. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869. <https://datatracker.ietf.org/doc/html/rfc5869>.
- Lindenmeier, C., Hammer, A., Gruber, J., Röckl, J., Freiling, F., 2024. Key extraction-based lawful access to encrypted data: taxonomy and survey. *Forensic Sci. Int.: Digit. Invest.* 50, 301796.
- Modbus Organization, 2018. Modbus TCP security, version 21. https://modbus.org/docs/MB-TCP-Security-v21_2018-07-24.pdf. (Accessed 20 May 2025).
- Moriconi, F., Levillain, O., Francillon, A., Troncy, R., 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>. (Accessed 11 May 2025).
- Nosevich, G., 2022. Decrypting schannel TLS traffic. Part 1. Getting secrets from lsass. <https://b.poc.fun/decrypting-schannel-tls-part-1/#6-obtaining-tls13-keys>. (Accessed 15 January 2025).
- OpenVPN, Inc, 2017. openvpn/README.mbedtls at master. <https://github.com/OpenVPN/openvpn/blob/master/README.mbedtls>. (Accessed 20 May 2025).
- Pan, J., Zhuang, Y., Sun, B., 2019. Efficient and transparent method for large-scale tls traffic analysis of browsers and analogous programs. *Secur. Commun. Network.* 2019, 1–22. <https://doi.org/10.1155/2019/8467081>.
- Papadogiannaki, E., Ioannidis, S., 2021. A survey on encrypted network traffic analysis applications, techniques, and countermeasures. *ACM Comput. Surv.* 54, 1–35.
- Plohmann, D., Blatt, M., Enders, D., 2023. Mcrit: the minhash-based code relationship & investigation toolkit. *The Journal on Cybercrime and Digital Investigations* 8, 7–18.
- Rescorla, E., 2000. HTTP over TLS. RFC 2818. <https://www.rfc-editor.org/info/rfc2818>.
- Rescorla, E., 2018. The transport layer security (TLS) protocol version 1.3. RFC 8446. <https://www.rfc-editor.org/info/rfc8446>.
- Rescorla, E., Dierks, T., 2008a. The transport layer security (TLS) protocol version 1.2. RFC 5246. <https://www.rfc-editor.org/info/rfc5246>.
- Rescorla, E., Dierks, T., 2008b. The transport layer security (TLS) protocol version 1.2. RFC 5246. <https://datatracker.ietf.org/doc/html/rfc5246#section-8.1>.
- SSL.com, 2023. SSL/TLS best practices. <https://www.ssl.com/guide/ssl-best-practices/OnlineguideforSSL/TLSconfigurationandsecuritypractices>. (Accessed 15 May 2025).
- Stoykova, R., 2023. Encrochat: the hacker with a warrant and fair trials? *Forensic Sci. Int.: Digit. Invest.* 46, 301602.
- Taubmann, B., Alabduljaleel, O., Reiser, H.P., 2018. DroidKex: fast extraction of ephemeral TLS keys from the memory of android apps. *Digit. Invest.* 26, S67–S76.
- Thomson, M., Turner, S., Weston, S., 2021. Using TLS to secure QUIC. RFC 9001. <https://datatracker.ietf.org/doc/html/rfc9001>.
- Tunius, H., 2023. TLS keylogger. [https://codeshare.frida.re/@k0nserve/tls-keylogger/](https://codeshare Frida.re/@k0nserve/tls-keylogger/). (Accessed 11 May 2025).
- Valadon, G., 2022. When eBPF meets TLS! presentation slides. <https://github.com/qualyslabs/conf-presentation/blob/master/Confs/CanSecWest-2022/WhenPresentedatCanSecWest2022>.
- Warburton, D., Vinberg, S., 2021. The 2021 TLS telemetry report. <https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report>. (Accessed 29 April 2025).
- Xia, B., Pang, J., Zhou, X., Shan, Z., Wang, J., Yue, F., 2023. Binary code similarity analysis based on naming function and common vector space. *Sci. Rep.* 13, 15676.
- Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D., 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363–376.

Further reading

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