electronics

MDPI

Article

Encrypted Traffic Decryption Tools: Comparative Performance Analysis and Improvement Guidelines

Minwoo Jo, Hayong Jeong, Binwon Song and Heeseung Jo *

Department of Computer Science, Chungbuk National University, Cheongju 28644, Republic of Korea; chminoo@g.cbnu.ac.kr (M.J.); gkdyd7167@cbnu.ac.kr (H.J.); qlsdnjs236@cbnu.ac.kr (B.S.)
* Correspondence: heesn@cbnu.ac.kr

Abstract: With the exponential growth of encrypted communication over the internet, research into systems capable of analyzing large volumes of encrypted traffic is essential. This study focuses on evaluating the performance of two prominent tools, ssldump and tshark, in decrypting and inspecting encrypted network traffic, assuming an environment where decryption keys are available. The performance of ssldump and tshark was assessed using various metrics, including execution time, and the ability to handle different file sizes and session counts. The results showed that tshark exhibited faster processing speeds for smaller file sizes and a higher number of sessions, while ssldump demonstrated better performance for larger file sizes and fewer sessions. However, notable performance differences were not observed based solely on the type of cipher suite or encryption method used. To enhance performance, the study proposes the session-based split and conquer (SSC) technique for automating parallelization using a multi-process approach. SSC shows up to a 39× improvement in performance, depending on system capabilities and workload.

Keywords: decryption; TLS/SSL; packet; sniffing; network; performance

check for updates

Citation: Jo, M.; Jeong, H.; Song, B.; Jo, H. Encrypted Traffic Decryption Tools: Comparative Performance Analysis and Improvement Guidelines. *Electronics* **2024**, *13*, 2876. https://doi.org/10.3390/electronics13142876

Academic Editor: Hung-Yu Chien

Received: 5 June 2024
Revised: 15 July 2024
Accepted: 20 July 2024
Published: 22 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The worldwide advancement in AI computing trends and the evolution of big data have elevated data security as a critical issue. Encryption of data is a key means of keeping information secure. However, as encryption technology progresses, the importance of decrypting encrypted traffic is also becoming more pronounced. While encrypted communication over networks is a fundamental and essential method of data protection, they present significant challenges for system administrators responsible for security management. Encrypted data, being indecipherable in its encrypted state, poses considerable difficulties for administrators in addressing issues such as data leaks and virus detection over the network.

There are several methods available for encrypting data, and the choice of method depends on factors such as the type of data, security requirements, and available technologies. One of the most common methods is symmetric key encryption, where the same key is used for both encrypting and decrypting data. This method offers high encryption efficiency and fast processing speeds. Representative algorithms include the Advanced Encryption Standard (AES) [1], Data Encryption Standard (DES) [2], Blowfish [3], and Twofish [4]. However, because the same key is used for both encryption and decryption in symmetric key encryption, ensuring secure key distribution is crucial.

Asymmetric Key Encryption, also known as Public Key Encryption, utilizes a pair of keys, a public key and a private key, to encrypt and decrypt data. Encryption is performed using the public key, while decryption is carried out using the private key. Representative algorithms include Rivest–Shamir–Adleman (RSA) [5], Elliptic Curve Cryptography (ECC) [6], and Diffie–Hellman Key Exchange (DH) [7]. The advantages of asymmetric key encryption include relatively simple key management and the ability to address key

exchange challenges inherent in symmetric key encryption. However, compared to symmetric key encryption, asymmetric key encryption tends to be slower and requires more computational resources for encryption/decryption operations.

A hash function is an algorithm that transforms data into a fixed-size hash value. Hash functions are one-way functions, meaning that the original data cannot be recovered from the hash value. A well-known hash function is SHA-256. Hash functions are commonly used to verify data integrity or store important information, like passwords, securely.

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are cryptographic protocols used to provide security in network communications. TLS is the successor to SSL and, currently, SSL is discouraged from use due to security vulnerabilities. These encryption protocols encrypt and protect communication between clients and servers, thereby safeguarding against Man-In-The-Middle (MITM) attacks, data theft, and surveillance. TLS/SSL is widely used in various applications, including the HTTPS protocol between web browsers and web servers, making it the primary focus for data decryption in this study.

There are several methods available for capturing network traffic in real-time, including using network protocol analysis tools, switch port mirroring, network hub monitoring, network tap replication, or installing proxy servers. These methods enable the capture of network traffic, which is typically saved in data file formats for subsequent analysis. The pcap library [8] is commonly used for this purpose. In this study, we aim to capture network traffic using network protocol analysis tools and save it in the pcap format for decryption. There are various tools available for decrypting encrypted traffic, with popular choices including Wireshark, ssldump, and Microsoft Network Monitor.

Upon analyzing the decryption process, we observed that these programs spend more time on packet disassembly and reassembly than on actual decryption. To address this issue, we employed the Session-based Split and Conquer (SSC) method to divide the pcap file into segments and utilized multiprocessing techniques to decrypt each segmented pcap file. Furthermore, to increase the disk I/O speed of a huge amount of pcap files, we devised an additional in-memory version using asynchronous processing with tmpfs. When evaluating the execution time performed by SSC for tshark4 with packet sizes of 4096k with 10 sessions, the following time reductions were observed compared to standard execution: 40.2% reduction in the native environment, 50.3% reduction in the asynchronous environment, and 68.2% reduction in the in-memory environment. For packet sizes of 4096k with 1000 sessions, tshark4 demonstrated a time reduction of 93.5%, while ssldump showed a reduction of 97.4%.

The structure of this paper is as follows. This chapter describes the motivation for the research and summarizes this paper. Section 2 explains the issues concerning the differences between related research and this study. Section 3 briefly summarizes and explains the background knowledge used in this study. Section 4 presents a detailed analysis of the decryption performance of TLS encrypted traffic data for the two selected decryption tools, tshark and ssldump. In Section 5, we propose an SSC technique to improve performance based on the analysis. We evaluate the performance when a pcap file is divided and executed through multiprocessing. Section 6 provides a discussion, and Section 7 concludes the paper.

2. Related Work

This study focuses on analyzing and improving performance in the process of decrypting encrypted traffic to inspect the plaintext. Related research can be classified into the following categories based on decryption methods and performance aspects.

Research on decryption methods: Research on decrypting encrypted traffic has been ongoing for a long time, dating back to the inception of encrypted communications. Blake et al. [9] conducted a study to enhance the efficiency of malware detection by increasing the understanding of HTTPS confidentiality limitations. Florian et al. [10] presented a collaborative approach where endpoints selectively send TLS keys to the NMS for decrypting

TLS connections. They implemented a prototype based on Zeek NMS that can decrypt TLS connections and analyze plaintext. Husak et al. [11] proposed a mechanism for real-time identification of HTTPS clients based on network monitoring and SSL/TLS fingerprinting. They developed a list of SSL/TLS encryption suites and an HTTP user agent dictionary, assigning user agents to observed SSL/TLS connections to identify communication clients. Moriconi et al. [12] implemented X-ray-TLS, which operates in major TLS libraries by using a memory change reconstruction algorithm to extract TLS secrets from process memory. It has the advantage of an operation based on existing kernel functions, thus not requiring modifications to hypervisors or existing programs. However, most research on decryption focuses on the methods, not paying much attention to execution time and performance, which differentiates our study.

Research focused on performance: Markus et al. [13] investigated the performance impact of Post-Quantum cryptography on TLS 1.3, examining various signature algorithms and key agreements through black-box and white-box measurements to present their performance. Yildiz [14] proposed a combination of quantitative and qualitative methods by scanning the HTTPS configurations of domains and analyzing case studies and literature reviews. This approach has been known to yield good results in improving security and privacy for public websites and promoting the adoption of best practices. Junghan et al. [15] based their research on the observation that traffic monitoring middleboxes on the client side are often read-only. Given that generalized architectures, like mbTLS [16] or maTLS [17], are very inefficient due to the complexity of key sharing and additional computations, they suggested improvements. These proposals are either for performance enhancement in decryption or mechanisms operating under specific conditions, thus having limitations in universality. Our research deals with performance analysis and enhancement by achieving complete decryption to view plaintext.

3. Background Knowledge

This section presents the results of the evaluation of the performance of leading tools for decrypting existing network traffic data. The analysis offers detailed insights into performance variations based on data size, number of sessions, protocols, and cipher suites.

3.1. SSL/TLS Handshake

The SSL/TLS protocol is designed as an encryption protocol to provide secure communication over computer networks, primarily used for protecting data transmission via the internet, such as websites, email, VPNs, VoIP, and instant messaging applications. The SSL protocol ceased updates due to significant security issues after its 3.0 version in 1996. Currently, most communications utilize versions 1.2 and 1.3 of TLS. The process of SSL/TLS handshake proceeds as follows.

1. Client hello: Client sends a client hello message to the server detailing the client's configuration settings. It passes the SSL/TLS version, a list of available cipher suites, a session ID, and a random byte.
2. Server hello: SSL/TLS server receives the client hello packet and sends the client a cipher suite, SSL protocol version, public key, digital certificate, session identifier, and server random byte.
3. Certificated: The server sends its SSL certificate to the client. Inside the certificate is the public key issued by the server. The client decrypts the SSL certificate using the CA's public key, which is known to everyone. Successful decryption results in authentication and successful verification.
4. Key exchange: The symmetric key is generated by the client, encrypted with the server's public key extracted from inside the SSL certificate, and passed to the server. The key encrypts the data that the client and server want to exchange.
5. Finished: Finally, after all the information to be exchanged has been exchanged, the handshake ends by sending a 'Finished' packet to indicate that they are ready to communicate.

3.2. Cipher Suite

A cipher suite encompasses key exchange algorithms, server authentication algorithms, symmetric cipher algorithms, block cipher modes of operation, and message authentication hash algorithms. During the handshake phase, the client lists the cipher suites it can use in a message to the server (client hello). The server then selects one from the list and communicates its choice back to the client (server hello), and encryption proceeds using the selected cipher suite.

The structure of a cipher suite is in the format of 'SSL/TLS_A_B_with_C_D_E', where A through E sequentially represent the key exchange algorithm, authentication algorithm, symmetric encryption algorithm, block cipher mode of operation, and message authentication hash algorithm, respectively. For instance, TLS_ECDHE_ECDSA_with_AES_128_GCM_SHA256 signifies the use of the TLS protocol, with ECDHE as the key exchange method, ECDSA as the authentication algorithm, AES as the symmetric algorithm with a 128-bit encryption key, GCM as the block cipher mode of operation, and SHA256 as the hash algorithm.

3.3. Decryption Methods

There are primarily two methods for decrypting encrypted data: the SSLKEYLOGFILE method and the MITM approach. The decryption process using SSLKEYLOGFILE involves enabling SSL logging to create a file that records information during the SSL process, which is then used for decryption. When accessing a domain, the client generates a pre-master secret key through a key exchange algorithm. This key is used to encrypt traffic, and browsers such as Chrome and Firefox can record this key. By registering the SSLKEYLOGFILE environment variable, the log of the pre-master key is saved at a specified path when HTTPS traffic occurs. By recording and storing such information through SSLKEYLOGFILE, it is possible to gain information about all cipher suites used in that session. This information is then used in the decryption process to decipher encrypted communications.

The MITM method, often used for intercepting information between a server and a client, leverages various proxy servers to decrypt encrypted traffic. A proxy server is commonly employed for security reasons, acting on behalf of the client to communicate with the server, and is utilized for decryption. During the communication between a client and a server, a handshake occurs in the order of client hello, server hello, certificate, and client key exchange. SSL/TLS employs a public key-based authentication method where a symmetric key is generated. During the client key exchange process, for example, the client sends the symmetric key it generated to the server, encrypted with the server's public key. In a MITM attack scenario, the client unknowingly sends the symmetric key to the interceptor (MITM). The interceptor then generates its own symmetric key and sends it to the server. The server, believing it to be communicating directly with the client, encrypts packets using the symmetric key provided by the interceptor and sends them back. The interceptor decrypts these packets, then re-encrypts them with the symmetric key initially shared by the client before forwarding them to the client. This process allows the interceptor to decrypt, read, and re-encrypt the communications, effectively conducting decryption through the MITM method.

4. Performance Analysis of Conventional Tools

This section presents the results after evaluating the performance of leading tools for decrypting existing network traffic data. The analysis offers detailed insights into performance variations based on data size, number of sessions, protocols, and cipher suites.

4.1. Analysis Environment

To generate a workload for measuring and comparing the performance of decryption tools, we captured encrypted communications between an Nginx web server and a client using Machine A, as shown in Table 1. Machine B was also used from Section 5 onwards to

evaluate performance with large pcap files. The captured data was saved in pcap format, and experiments were conducted with varying parameters, such as data size, sessions, clients, and TLS versions, to create a range of workloads, as outlined in Table 2. To ensure a diverse range of performance evaluations, we generated workloads using 18 supported cipher suites from the Nginx web server, excluding 12 that were not supported. The specific cipher suites used are listed in Table 3.

Table 1. Machine specifications used for evaluation.

	Machine A	**Machine B**
CPU	Intel(R) Xeon(R) CPU E5-2420 @ 1.90 GHz (12 Cores)	Intel(R) Xeon(R) CPU E5-2696 v4 @ 2.20 GHz (44 Cores)
Memory	DDR4 16 GB	DDR4 128 GB
Storage	PERC H310 1TB x2 (RAID 0)	Samsung SSD 850 250 GB
OS	Ubuntu 22.04.03 LTS	Ubuntu 22.04.03 LTS

Table 2. Workload combinations used for evaluation.

Specification	**Parameters**
Data size	1K, 16K, 256K, 4096K, 65,536K
Number of sessions	10, 100, 1000
Protocols	TLS v1.2, TLS v1.3
Clients	curl, firefox browser
Server	Nginx web server

Table 3. Cipher suites used for evaluation.

Unique ID	**Cipher Suites**
CS1	ECDHE-RSA-CHACHA20-POLY1305
CS2	ECDHE-RSA-AES256-GCM-SHA384
CS3	DHE-RSA-AES128-GCM-SHA256
CS4	DHE-RSA-AES256-GCM-SHA384
CS5	ECDHE-RSA-AES128-SHA256
CS6	ECDHE-RSA-AES256-SHA384
CS7	ECDHE-RSA-AES128-SHA
CS8	ECDHE-RSA-AES256-SHA
CS9	DHE-RSA-AES128-SHA256
CS10	DHE-RSA-AES128-SHA
CS11	DHE-RSA-AES256-SHA256
CS12	DHE-RSA-AES256-SHA
CS13	AES128-GCM-SHA256
CS14	AES256-GCM-SHA384
CS15	AES128-SHA256
CS16	AES256-SHA256
CS17	AES128-SHA
CS18	AES256-SHA

For each evaluation, we used the captured pcap files from all workloads along with the corresponding encryption key files stored in log format. Decryption was performed using the decryption tools, and execution times were measured. To assess differences in decryption execution times based on data size and session count, we analyzed the execution times of tshark and ssldump with varying data sizes and session counts among the cipher suites listed in Table 3. Following this, we conducted a performance analysis for each decryption tool. To represent data size and session count in the experiment results, we used a format like XK.NN, where "XK" indicates the data size in kilobytes and "NN" represents the number of sessions. For example, "1K.10" means a data size of 1 kilobyte with 10 sessions. All evaluations were conducted three times, and the average value was presented.

4.2. Execution Time Differences Based on File Size and Number of Sessions

The results showing the differences in execution times for decryption based on file size using tshark3 are presented in Figure 1. The ssldump shows a similar pattern to tshark, so we decided it was not necessary to include it. As expected, execution time increases with file size, which is logical given that larger files require more time for packet analysis and decryption. Notably, while file sizes were tested at 1K, 16K, 256K, and 4096K, representing a 16 times increase at each step, the increase in execution time was proportionally less than 16 times.

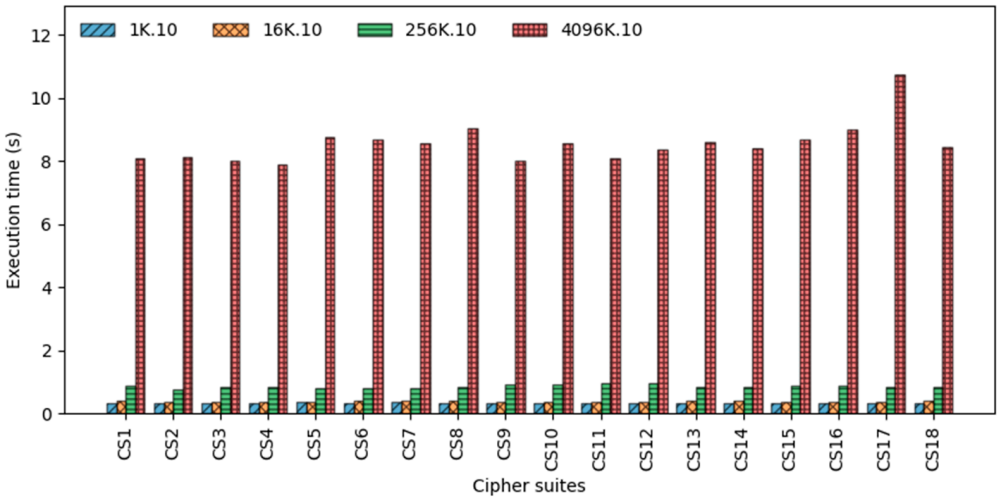


Figure 1. Execution time differences for file sizes.

Figure 2 illustrates the results of experiments based on the number of sessions. As the session count increases, the execution time also rises but, again, the increase is not strictly linear compared to the session count. Interestingly, there is no significant difference in performance based on the cipher suite used, which is worth noting.

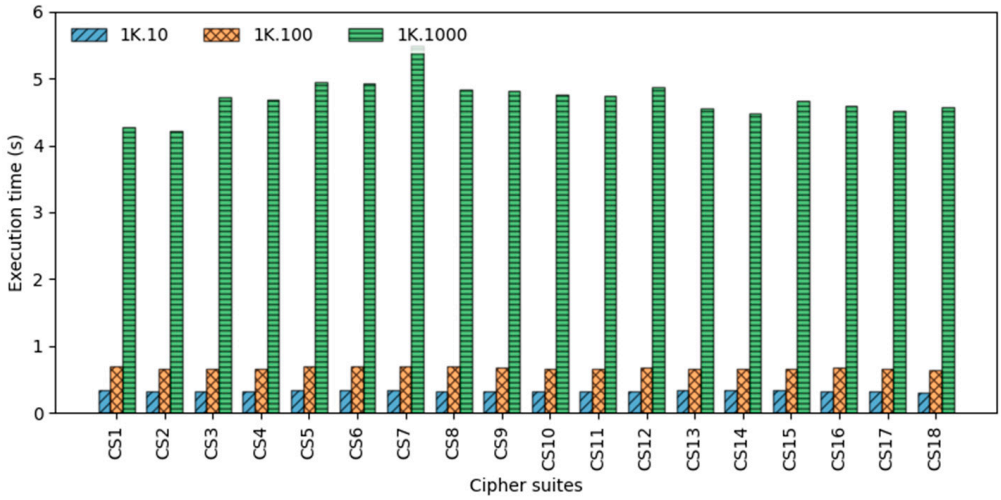


Figure 2. Execution time differences for number of sessions.

4.3. Decryption Tool Performance Analysis

Figure 3 shows the results of running tshark version 3 and version 4 with workloads 1K.10 and 4096K.10. As for the other parameters, we did not include them all because the patterns are similar. tshark3 refers to the stock version that can be installed as a package on Ubuntu server machines, while tshark4 is built from source code. The versions used were 3.2.3 for tshark3 and 4.0.6 for tshark4. While there are minor variations between cipher

suites, these are considered within the margin of experimental error. It is observed that tshark4 generally shows 10% to 20% better performance than tshark3. Therefore, for future experiments, tshark3 is excluded, and only tshark4 is used for further testing.

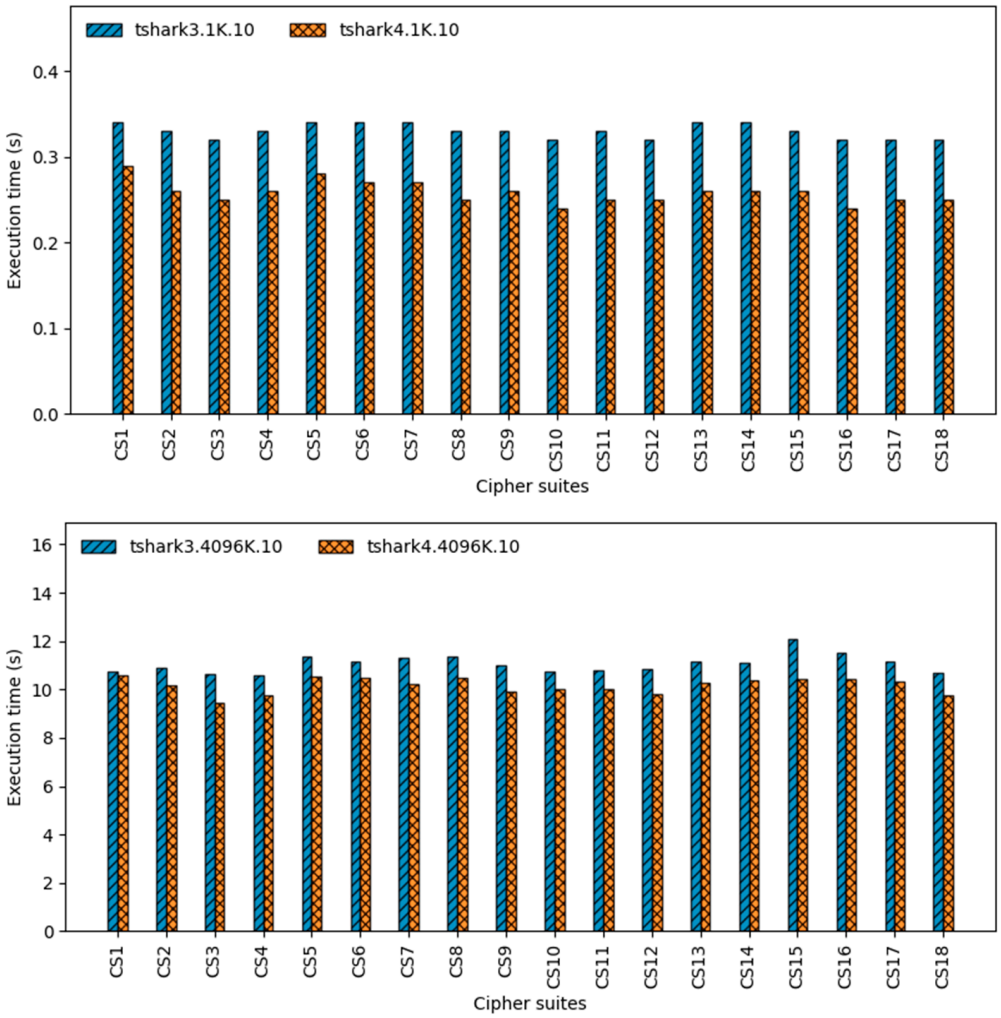


Figure 3. Execution time differences between tshark version 3 and version 4.

Next, to compare tshark and ssldump, we conducted experiments in the same way as before. Figure 4 presents the performance differences between tshark and ssldump. This experiment utilized the 1K.10 and 4096K.10 workloads, and the results show that the two tools exhibit opposite performance trends. When using the 1K.10 workload, tshark demonstrates significantly superior performance whereas, with the 4096K.10 workload, ssldump performs much better.

The completely opposite performance trends between the two decryption tools depending on file size differences are noteworthy. In the 1K.10 workload, tshark4 is much faster while, in the 4096K.10 workload, ssldump shows better results. Therefore, to further evaluate the performance of tshark and ssldump across all workloads, we conducted a comprehensive comparison. Since the differences between ciphersuites are not significant, the results of experiments conducted using only the ECDHE-RSA-AES256-GCM-SHA384 for all workloads are shown in Table 4. The execution time marked in red indicates better performance, and the ratio column indicates how much better one performs compared to the other. tshark exhibits up to 27 times better performance for 1K.1000 workloads, whereas ssldump shows up to 25 times improved performance in the 65,536K.10 workload. These experimental results suggest that ssldump tends to perform better when dealing

with larger single encrypted contents, whereas tshark yields superior performance when handling smaller size packages.

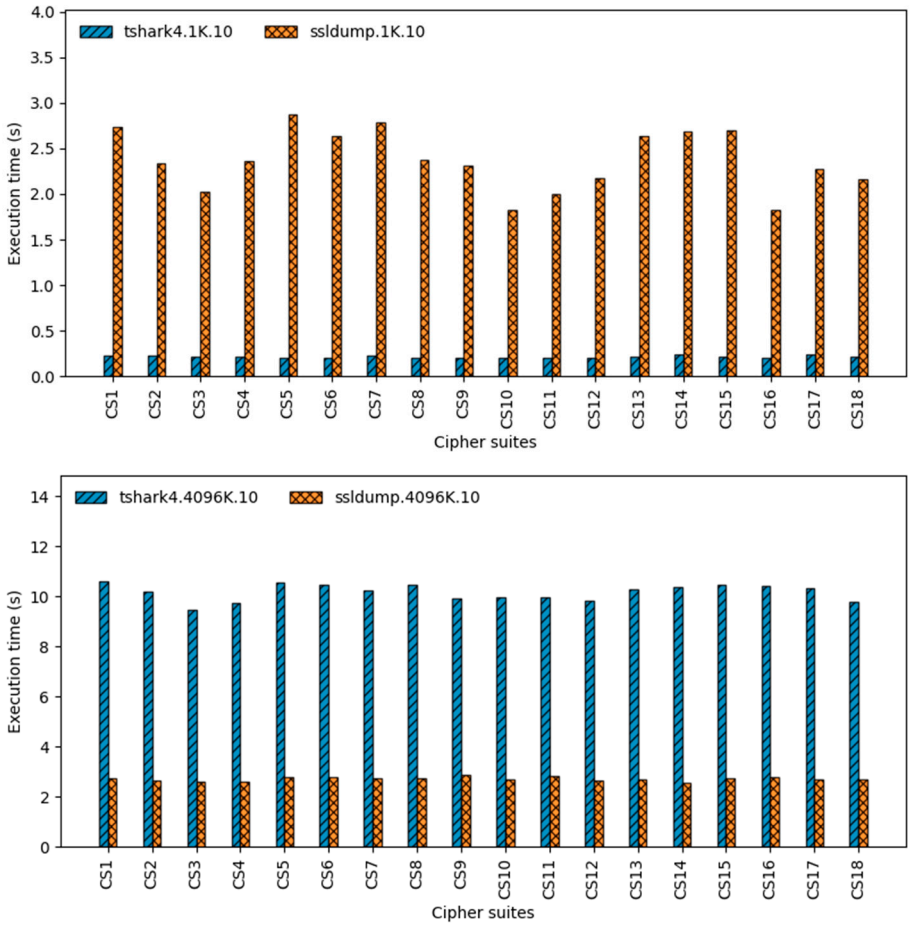


Figure 4. Execution time differences between tshark and ssldump.

Table 4. Execution time differences between tshark and ssldump.

File Size	Session	Tshark	Ssldump	Ratio
1K	10	**0.29**	2.74	9.45
1K	100	**0.82**	15.51	18.91
1K	1000	**5.72**	157.07	27.46
16K	10	**0.36**	1.97	5.47
256K	10	**0.97**	2.47	2.55
4096K	10	10.59	**1.69**	0.16
4096K	1000	1259.35	**279.22**	0.22
65,536K	10	119.99	**5.21**	0.04

The text in bold and red shows the minimum execution time within the same workload.

4.4. Performance Analysis of TLS Protocol Versions

To analyze performance across different TLS protocol versions, we conducted a comparative experiment with TLS versions 1.0, 1.1, 1.2, and 1.3. The evaluation was performed using both tshark4 and ssldump, with a 4096K.10 workload. However, TLS versions 1.0 and 1.1 are no longer supported by the industry and are not available in Nginx web servers. Attempts to establish communication with versions 1.0 and 1.1 resulted in failure.

The results comparing TLS 1.2 and TLS 1.3 are shown in Figure 5. Using tshark, we observed that TLS 1.3 takes slightly longer than TLS 1.2. However, when using ssldump, the performance was similar within the margin of experimental error. This suggests that there is no significant difference in decryption performance between TLS 1.2 and TLS 1.3.

Therefore, from a decryption performance perspective, the choice between TLS 1.2 and TLS 1.3 does not show substantial significance. Therefore, we used TLS 1.2 which is the most popular after Section 5.

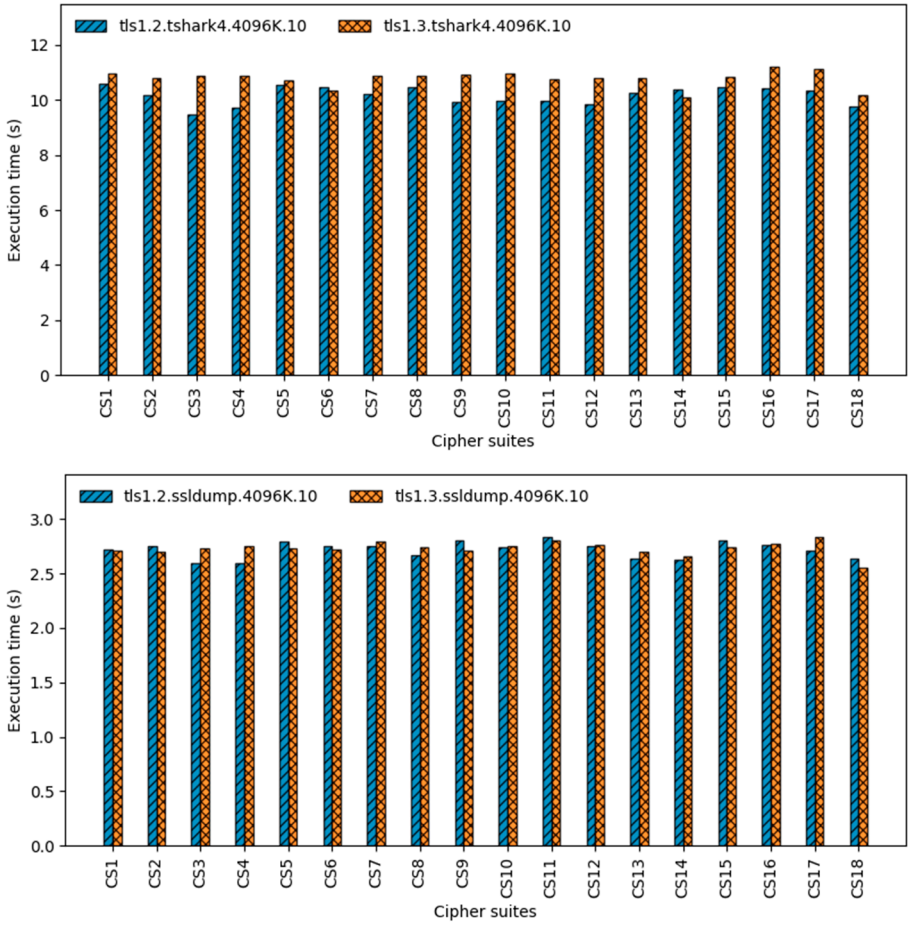


Figure 5. Execution time differences between TLS version 1.2 and 1.3.

4.5. Library and Function Levels Performance Analysis

To conduct a more detailed analysis of tshark and ssldump, we used the perf tool to examine performance at the library and function levels. Given that previous evaluations indicated minimal differences across cipher suites, we selected two representative cipher suites; AES256-GCM-SHA384 and ECDHE-RSA-AES256-GCM-SHA384. The 1K.1000 and 4096K.10 workloads were used for each cipher suite.

For tshark, we observed that the proportion of execution time attributed to the decryption library was less than 1%, indicating that decryption does not significantly impact the total execution time. This trend holds regardless of the cipher suite and workload. In contrast, ssldump showed a slightly higher proportion for the decryption library compared to tshark, but it was still relatively low at about 4%, suggesting that decryption is not a major factor in the overall execution time. Additionally, functions like print_data() or putc() accounted for a significant proportion of the execution time, indicating that output operations, such as printing results, are a substantial part of the total time, likely exceeding the time spent on packet analysis and decryption.

To determine the impact of decryption on overall execution time, we compared the time taken with and without decryption. Figure 6 shows the results. In the case of tshark, the difference is not significant, indicating that decryption does not consume significant CPU time. On the other hand, ssldump shows a more noticeable time difference, likely due to the additional time required to output the decrypted data. Thus, it can be concluded that the decryption part has a minimal impact on overall tshark or ssldump performance.

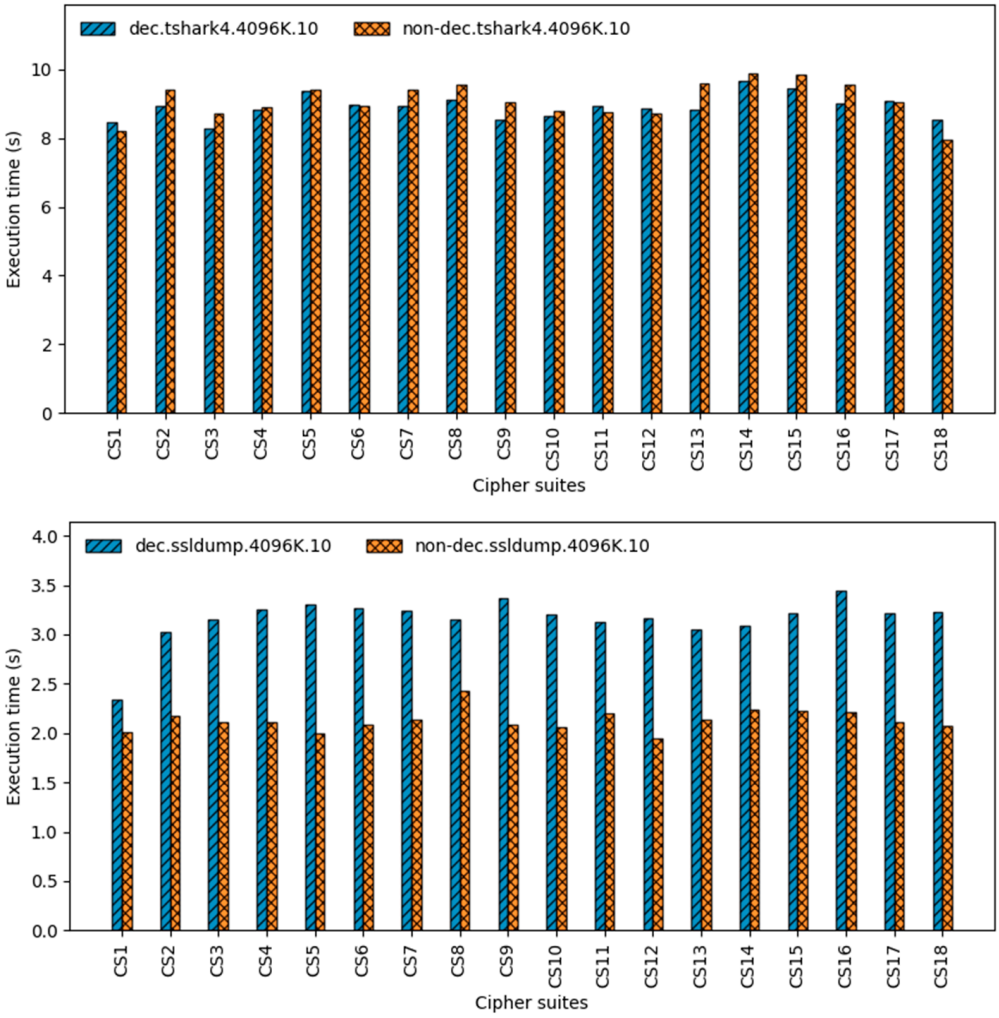


Figure 6. Execution time differences between decryption and non-decryption.

5. Enhancing Decryption Performance

5.1. Session Based Split and Conquer

To leverage the benefits of multicore processors and maximize performance, it is worth considering running tshark or ssldump in parallel on multiple cores. However, tshark and ssldump are not designed to operate in a multi-threaded manner. This design choice is based on the understanding that the overhead associated with packet disassembly and reassembly in a multi-threaded environment would result in similar performance to a single-threaded approach. Therefore, instead of implementing multi-threaded versions of tshark and ssldump, we propose a method to run them in parallel using a multi-process approach, thereby maximizing the advantages of multicore processors [18,19]. Specifically, we suggest splitting a single target pcap file into multiple smaller pcap files and running multiple instances of tshark or ssldump simultaneously to achieve parallelization.

There are various approaches to splitting pcap files, including by time, packet count, or session. After evaluating all these methods, we found that splitting by session was the most stable, because each session might contain all the payload required for decryption. If the file is split by time or packet count, parts of the payload could get cut off, preventing successful decryption. Therefore, we implemented a program to automate this process, naming the technique Session-based Split and Conquer (SSC).

For SSC, it is crucial to consider the ability to split by session and minimize overhead during the splitting process. The most widely used tool for splitting pcap files is editcap, which is part of the Wireshark tool package. However, editcap does not support session-based splitting. Another tool that supports session-based splitting is SplitCap, but it only

provides executables for Windows, making it unsuitable for SSC. Among open-source tools, PcapPlusPlus was ultimately selected, as it is open-source, supports session-based splitting, and is written in C++, offering excellent performance. We modified the source code of PcapPlusPlus to integrate it into SSC.

To implement session-based splitting and multi-process parallelization, we developed a Python program to automate the SSC technique. To optimize the process of splitting and parallelization, we configured the workflow to split in-memory and provide it as input to tshark or ssldump. This was achieved by using tmpfs, an in-memory filesystem, which completely eliminates disk I/O and avoids the overhead of accessing the OS-level page cache. This approach was deemed the most effective optimization strategy. Figure 7 shows the working differences between conventional process and SSC process. The number of splitted pcap files is automatically set to the same as the number of cores.

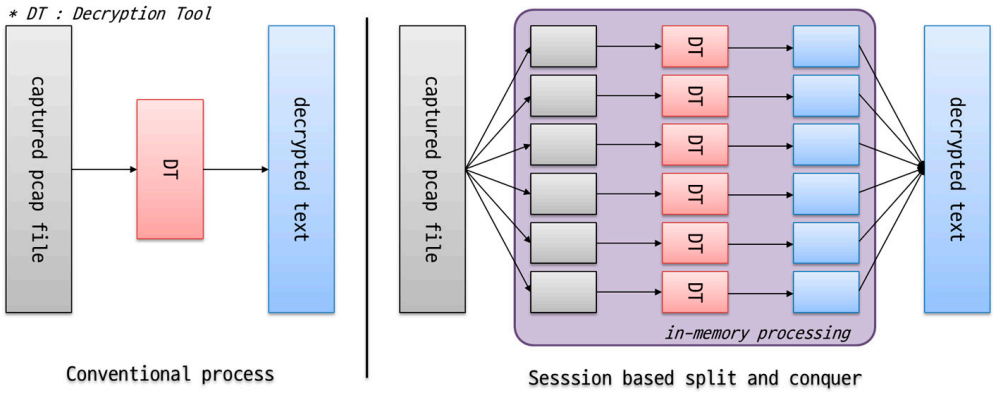


Figure 7. Process diagram of conventional tool and SSC.

5.2. Evaluation Environment

To compare the performance of parallel processing decryption of SSC with traditional decryption, the evaluation environment was set as previously described in Table 2. In this study, we compare the decryption performance of tshark in the native environment, SSC in the normal environment (denoted by disk), SSC in the asynchronous disk environment (denoted by async), and SSC in the in-memory environment (denoted by in-memory) using Machine A to see the performance differences between the disk environments that use SSC.

We also compare the decryption performance between native tshark4, SSC tshark, and SSC ssldump using Machine A and B to see the difference in performance between decryption tools using SSC. In this evaluation, to compare performance differences based on packet size and number, we additionally used workloads consisting of 1000 sessions of 1 KB each, 10 sessions of 4096 KB each, and 10 sessions of 65,536 KB each.

5.3. Performance Analysis between Disk Environments When Using SSC

Using Machine A, we compared decryption performance between tshark in a normal environment with SSC, tshark in an asynchronous disk environment with SSC, and tshark in an in-memory environment with SSC. Figure 8 shows the decryption times for ten 4096 KB packets encrypted with TLSv1.2_AES256_GCM_SHA384 under each condition. The split legend indicates the time taken to split the pcap file, while the decryption legend represents the time taken for decryption.

In this evaluation, compared to the execution time of tshark in a native single-processor environment, the use of SSC resulted in significant time reductions in different storage environments. Specifically, in a standard disk environment, SSC achieved a 40.2% reduction in time. In an asynchronous disk environment, the time reduction was 50.3%. The most significant improvement was observed in an in-memory environment, where the time reduction reached 68.2%.

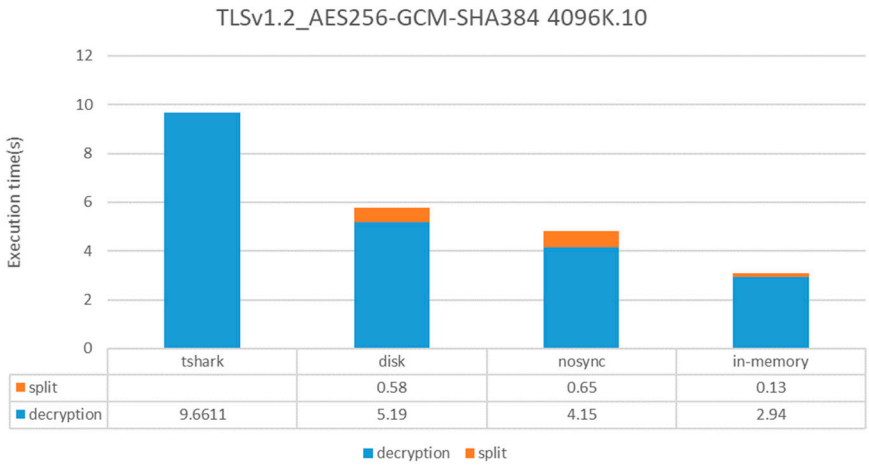


Figure 8. Detailed performance analysis of SSC in different disk environments compared to traditional tshark.

5.4. Performance Differences between SSC with Tshark and Ssldump

Initially, the decryption performance was compared among tshark, SSC with tshark, and SSC with ssldump for workloads consisting of 1000 sessions of 1 KB size and 10 sessions of 4096 KB size using machine A. As shown in Figure 9, for a large number of small-sized sessions, SSC with tshark proved superior performance across all cipher suites. SSC with ssldump showed lower performance, consistent with its inherent low performance for 1000 sessions of 1 KB size in the baseline.

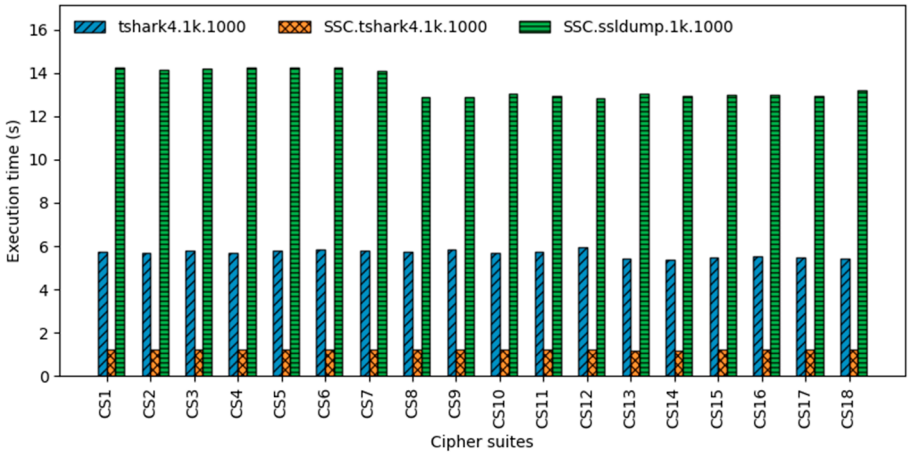


Figure 9. Performance comparison of SSC with tshark and ssldump (1K.1000).

However, as shown in Figure 10, for the workload decrypting 10 sessions of 4096 KB size, SSC with ssldump demonstrated exceptionally high performance. This indicates that the performance difference originates from the inherent performance disparity between tshark and ssldump, as previously described.

To further clarify the performance differences, we conducted evaluations using huge pcap files and Machine B. The same decryption methods were utilized to decrypt 1000 sessions of 4096 KB size and 10 sessions of 65,536 KB size. As described in Figure 11, when using significantly larger volumes of data, a reduction in time by up to 39 times was noted, demonstrating the efficiency of SSC.

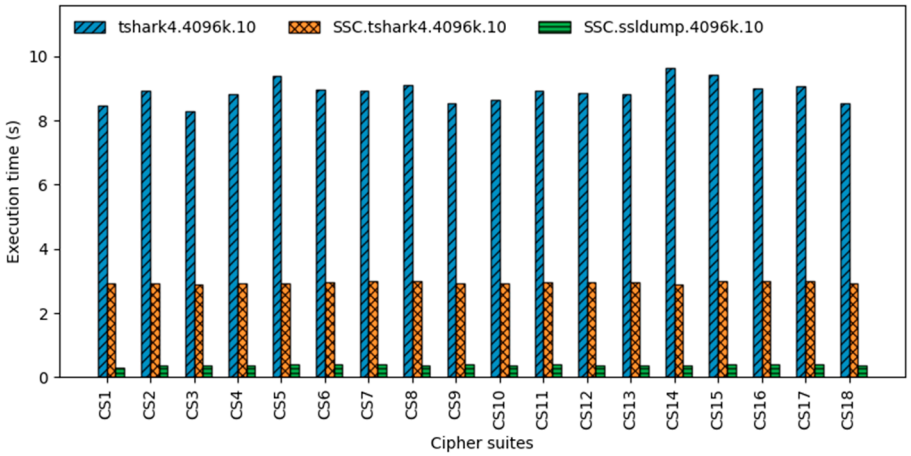


Figure 10. Performance comparison of SSC with tshark and ssldump (4096K.10).

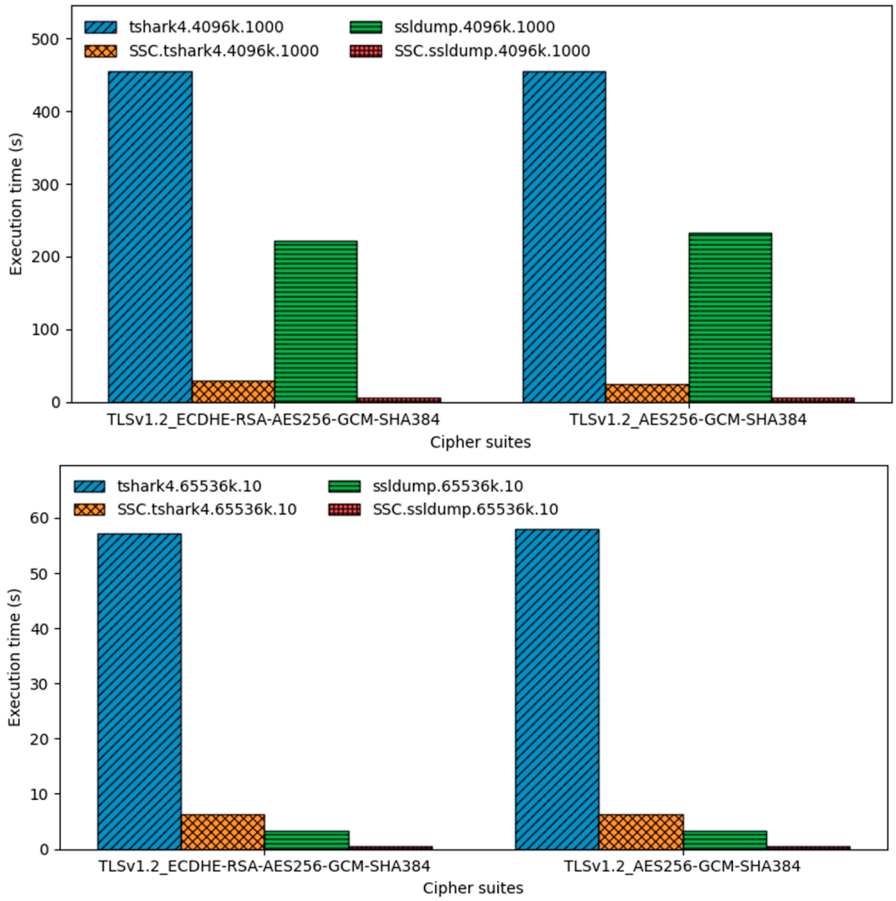


Figure 11. Performance comparison of SSC with tshark and ssldump (4096K.1000, 65,536K.10).

6. Discussion

SSC exhibits superior performance aspects but also presents drawbacks. To utilize the in-memory approach, a certain amount of memory must be allocated through tmpfs, and the memory occupied by the process itself must also be considered. A method to offset these memory requirements involves reducing the number of disk I/O operations at the program level. However, even with optimization through this in-memory method, it was proved that the advantages of the in-memory approach, which eliminates disk I/O and reduces the overhead of accessing the page cache at the OS Level, outweigh the disadvantages.

7. Conclusions

In this study, we present a detailed performance analysis using tshark and ssldump, which are commonly used for decrypting encrypted traffic. According to the research findings, tshark exhibited superior performance for small packet sizes, whereas ssldump showed excellent performance for large packet sizes. Based on this analysis, we propose SSC, a method that involves splitting pcap files by session and applying multiprocessing to perform TLS decryption. This method achieves performance improvements by utilizing conventional decryption tools, such as tshark and ssldump. Particularly, by optimizing in an in-memory environment, it has been able to enhance decryption performance compared to existing approaches. While the approach of SSC in this study has been primarily focused on TLS encrypted traffic, it is believed that similar performance improvement effects can be expected for other encryption methods. SSC achieved up to 39 times improvement in performance when using CS2 with the ssldump.4096K.1000 workload, reducing time from 221 s to 5.71 s. These outcomes demonstrate the potential value of SSC in the areas of network traffic and network security, suggesting efficient decryption for various encryption protocols.

As future work, the differences in decryption tools required by the workload pattern must also be considered. As observed in Section 5, tshark shows significant performance regardless of the number of small-sized packets it handles. However, ssldump consistently outperforms tshark when dealing with large-sized packets. The reason for this performance divergence depending on the workload pattern requires a detailed analysis based on the structure of each encryption library, libgcrypt for tshark and libcrypto for ssldump.

Author Contributions: Conceptualization, M.J. and H.J. (Hayong Jeong); Software, B.S.; Writing—original draft, M.J., H.J. (Hayong Jeong), B.S. and H.J. (Heeseung Jo). All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government [24ZS1100, Research on High-Performance Computing to overcome Limitations of AI].

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rijmen, V.; Daemen, J. Advanced encryption standard. *Proc. Fed. Inf. Process. Stand. Publ. Natl. Inst. Stand. Technol.* **2001**, *19*, 22.
2. Diffie, W.; Hellman, M.E. Exhaustive cryptanalysis of the NBS data encryption standard. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*; ACM: New York, NY, USA, 2022; pp. 391–414.
3. Schneier, B. Description of a new variable-length key, 64-bit block cipher (Blowfish). In Proceedings of the International Workshop on Fast Software Encryption, Cambridge, UK, 9–11 December 1993; Springer: Berlin/Heidelberg, Germany, 1993; pp. 191–204.
4. Schneier, B. The Twofish encryption algorithm. *Dr. Dobb's J. Softw. Tools Prof. Program.* **1998**, *23*, 30–34.
5. Rivest, R.L.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
6. Koblitz, N.; Menezes, A.; Vanstone, S. The state of elliptic curve cryptography. *Des. Codes Cryptogr.* **2000**, *19*, 173–193. [CrossRef]
7. Diffie, W.; Hellman, M.E. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*; ACM: New York, NY, USA, 2022; pp. 365–390.
8. TCPDump&Libpcap. Available online: https://www.tcpdump.org/ (accessed on 6 March 2024).
9. Anderson, B.; Chi, A.; Dunlop, S.; McGrew, D. 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, Richardson, TX, USA, 25–27 March 2019; pp. 267–278.
10. Wilkens, F.; Haas, S.; Amann, J.; Fischer, M. Passive, transparent, and selective TLS decryption for network security monitoring. In Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection, Openhagen, Denmark, 13–15 June 2022; Springer International Publishing: Cham, Switzerland, 2022; pp. 87–105.
11. Husák, M.; Čermák, M.; Jirsík, T.; Čeleda, P. HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. *EURASIP J. Inf. Secur.* **2016**, *2016*, 6. [CrossRef]

12. Moriconi, F.; Levillain, O.; Francillon, A.; Troncy, R. X-ray-TLS: Transparent decryption of TLS sessions by extracting session keys from memory. In Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS), Singapore, 1–5 July 2024.
13. Sosnowski, M.; Wiedner, F.; Hauser, E.; Steger, L.; Schoinianakis, D.; Gallenmüller, S.; Carle, G. The performance of post-quantum tls 1.3. In Proceedings of the Companion of the 19th International Conference on emerging Networking Experiments and Technologies, Paris, France, 5–8 December 2023; pp. 19–27.
14. Yildiz, B.V. The State of HTTPS Configuration of Public Websites Around the World. Bachelor's Thesis, University of Twente, Enschede, The Netherlands, 2023.
15. Yoon, J.; Do, S.; Kim, D.; Chung, T.; Park, K. Scaling the Performance of Modern TLS Traffic Monitoring without Compromising the Security. Available online: https://www.ndsl.kaist.edu/~kyoungsoo/papers/mmTLS_poster.pdf (accessed on 15 July 2024).
16. Naylor, D.; Li, R.; Gkantsidis, C.; Karagiannis, T.; Steenkiste, P. And then there were more: Secure communication for more than two parties. In Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, Incheon, Republic of Korea, 12–15 December 2017; pp. 88–100.
17. Lee, H.; Smith, Z.; Lim, J.; Choi, G.; Chun, S.; Chung, T.; Kwon, T.T. maTLS: How to Make TLS middlebox-aware? In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2019, San Diego, CA, USA, 24–27 February 2019.
18. Multi-Threaded Tshark. Available online: https://osqa-ask.wireshark.org/questions/56433/multi-threaded-tshark/ (accessed on 15 July 2024).
19. Does Wireshark Support Multi-Threading for Packet Parsing? Available online: https://ask.wireshark.org/question/33511/does-wireshark-support-multi-threading-for-packet-parsing/ (accessed on 15 July 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.