POLITECNICO DI TORINO
Repository ISTITUZIONALE

Detecting TLS Protocol Anomalies Through Network Monitoring and Compliance Tools

Original
Detecting TLS Protocol Anomalies Through Network Monitoring and Compliance Tools / Berbecaru, Diana Gratiela; De Santo, Marco. - In: FUTURE INTERNET. - ISSN 1999-5903. - ELETTRONICO. - 18:1(2026). [10.3390/fi18010062]

Availability:
This version is available at: 11583/3006954 since: 2026-01-26T18:24:44Z

Publisher:
MDPI

Published
DOI:10.3390/fi18010062

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

13 May 2026

future internet

MDPI

Article

Detecting TLS Protocol Anomalies Through Network Monitoring and Compliance Tools

Diana Gratiela Berbecaru * and Marco De Santo

Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Torino, Italy
* Correspondence: diana.berbecaru@polito.it

Abstract

The Transport Layer Security (TLS) protocol is widely used nowadays to create secure communications over TCP/IP networks. Its purpose is to ensure confidentiality, authentication, and data integrity for messages exchanged between two endpoints. In order to facilitate its integration into widely used applications, the protocol is typically implemented through libraries, such as OpenSSL, BoringSSL, LibreSSL, WolfSSL, NSS, or mbedTLS. These libraries encompass functions that execute the specialized TLS handshake required for channel establishment, as well as the construction and processing of TLS records, and the procedures for closing the secure channel. However, these software libraries may contain vulnerabilities or errors that could potentially jeopardize the security of the TLS channel. To identify flaws or deviations from established standards within the implemented TLS code, a specialized tool known as TLS-Anvil can be utilized. This tool also verifies the compliance of TLS libraries with the specifications outlined in the Request for Comments documents published by the IETF. TLS-Anvil conducts numerous tests with a client/server configuration utilizing a specified TLS library and subsequently generates a report that details the number of successful tests. In this work, we exploit the results obtained from a selected subset of TLS-Anvil tests to generate rules used for anomaly detection in Suricata, a well-known signature-based Intrusion Detection System. During the tests, TLS-Anvil generates .pcap capture files that report all the messages exchanged. Such files can be subsequently analyzed with Wireshark, allowing for a detailed examination of the messages exchanged during the tests and a thorough understanding of their structure on a byte-by-byte basis. Through the analysis of the TLS handshake messages produced during testing, we develop customized Suricata rules aimed at detecting TLS anomalies that result from flawed implementations within the intercepted traffic. Furthermore, we describe the specific test environment established for the purpose of deriving and validating certain Suricata rules intended to identify anomalies in nodes utilizing a version of the OpenSSL library that does not conform to the TLS specification. The rules that delineate TLS deviations or potential attacks may subsequently be integrated into a threat detection platform supporting Suricata. This integration will enhance the capability to identify TLS anomalies arising from code that fails to adhere to the established specifications.

Keywords: TLS vulnerabilities; network monitoring; Suricata rules; threat detection; TLS-Anvil

Check for updates

Academic Editor: Gianluigi Ferrari

Received: 28 November 2025
Revised: 29 December 2025
Accepted: 14 January 2026
Published: 21 January 2026

Copyright: © 2026 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.

1. Introduction

In today's digital landscape, securing communications over networks is more crucial than ever. The TLS protocol provides confidentiality, message authentication, and integrity

 https://doi.org/10.3390/fi18010062

for the application data exchanged between two end nodes, as well as entity authentication for the communicating parties. Consequently, TLS is extensively utilized across diverse contexts to facilitate secure communication channels, encompassing networked applications as well as IoT and embedded systems. Nevertheless, the intricate nature of TLS architecture, coupled with implementation challenges faced by various versions of the libraries that support this protocol, has led to vulnerabilities that threaten the security of TLS-enabled communications [1–4].

In the initial stages, various design factors contributed to the emergence of TLS vulnerabilities, specifically the complexity of the protocol and backward compatibility. The layered structure of TLS, which includes the handshake, key exchange, cipher negotiation, and compression created multiple attack surfaces. For instance, the rollback attack (first identified in the mid-1990s) against the SSL 2.0 protocol (ancestor of TLS) is one of the earliest examples of how backward compatibility in cryptographic protocols can be exploited. A rollback, or downgrade, attack deceives one party into believing that the other party only supports less secure options. In this scenario, the attacker intercepts and alters the handshake messages, forcing the connection to revert to SSL 2.0 or weaker ciphers, despite both parties having the capability to utilize stronger versions. The rollback attack affecting SSL 2.0 enabled subsequent exploits such as the DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) attack [5], which was disclosed in March 2016. In this attack, servers supporting SSL 2.0 could be used to decrypt TLS sessions. The DROWN attack showed that even obsolete protocols like SSL 2.0 could compromise modern TLS if backward compatibility was left enabled.

As ciphers become outdated, weak algorithms must be avoided in TLS, such as the MD5 or SHA1 hash algorithms, or the symmetric block algorithms in CBC mode. To mitigate design issues affecting the TLS protocol specification, the algorithms employed within TLS, or some attacks against CBC mode, new versions of the protocol have been proposed, namely TLS 1.2 [6] and TLS 1.3 [7]. Nonetheless, many systems need to support old versions (SSL 3.0, TLS 1.0, TLS 1.1) and outdated ciphers for legacy clients. Consequently, standard TLS libraries have retained compatibility with multiple TLS versions and legacy algorithms, such as 3DES and SHA1. Historically, the complexity of the TLS protocol and its emphasis on backward compatibility have led to other significant security vulnerabilities, including the POODLE [8], BEAST [9], and CRIME [10] attacks, which are summarized in Section 2.1.

Another significant cause of TLS attacks lies in the implementation errors. For example, minor coding mistakes, timing leaks and padding checks, have created vulnerabilities that have resulted in various high-impact attacks, such as Heartbleed [11], which are frequently more dangerous than the protocol flaws themselves. As an example, shortly after the Heartbleed vulnerability was publicly disclosed, there have been "exploit attempts from almost 700 sources, beginning less than 24 h after disclosure" [12]. This situation arises from the fact that libraries, such as OpenSSL, have accumulated years of complexity, rendering subtle bugs unavoidable. Additionally, detailed error reporting has frequently provided attackers with "oracles" to exploit cryptographic vulnerabilities.

1.1. Motivation

To counter TLS attacks, several mitigations can be adopted. For instance, old versions of the protocol (i.e., SSL 3.0, TLS 1.0, and TLS 1.1) have been deprecated and (only) TLS 1.3 is recommended nowadays [13], although TLS 1.2 can still be used. Only strong cipher suites must be used, while weak cryptographic algorithms and modes, like RC4 [14], DES, SHA1, and the CBC mode, are disabled. TLS 1.3 no longer supports RSA for key exchange, but only ephemeral Diffie–Hellman, ensuring perfect forward secrecy. Additionally, TLS

1.3 includes mechanisms like "downgrade sentinels" in the ServerHello to prevent forced rollbacks. To support secure renegotiation, standardized fixes prevent injection attacks.

Typically, shortcomings in the implementations are solved in subsequent versions of the released TLS libraries. However, if an adversary successfully substitutes a non-vulnerable version of a TLS library at a designated node with a vulnerable one, the result is a compromised (TLS-aware) node. In other use cases, the user might install TLS interception software [15], which could lower the security of the TLS channels. This node will receive and transmit data over a TLS channel, though it will be operating over a low-security channel, or it could be subject to compelled certificate creation attack [16]. Outdated TLS-enabled IoT (Internet of Things) or embedded devices [17,18] as well as malformed TLS-aware nodes (clients or server) could also run vulnerable TLS software. In all these cases, it is recommended to use an Intrusion Detection System (IDS) able to analyze the traffic to detect weak or faulty TLS software run by network nodes or devices.

Several signature-based IDS or network monitoring tools, such as Suricata [19], Zeek [20], or the ones proposed by Qualys [21], can already analyze in depth the TLS connections. They are able to assess negotiated protocol versions, cipher suites, and various certificate fields and extensions, including issuer, subject, expiration date, and subject alternative names, among others, as well as TLS extensions. The Qualys SSL Server Test is designed to simulate client connections, scrutinize the handshake process, and evaluate the resilience of the tested server against known TLS attacks. Other tools, like TLS-Monitor [22] and Threat-TLS [23], utilized Suricata or Zeek to detect more specific patterns of TLS attacks within network traffic. For instance, these tools exploited Zeek v5.0.2 and Suricata v6.0.6 to analyze the traffic, searching for the Heartbeat extension in the TLS messages as a possible indicator of the Heartbleed attack, or the presence of the RSA algorithm or a symmetric block algorithm in CBC mode within the cipher suites negotiated during the TLS handshake, which may be susceptible to vulnerabilities such as POODLE, ROBOT, or those related to Bleichenbacher.

Although certain IDS, including Suricata, have been updated to analyze TLS network traffic, they have yet to account for deviations in TLS implementations from the relevant RFCs, as performed for instance by TLS-Anvil [24]. Although there have not yet been any documented instances of actual attacks resulting from non-compliant TLS software, we believe it is imperative to promptly identify such anomalous TLS implementations. These may suggest the presence of malware, a malfunctioning TLS interception proxy, outdated embedded devices, or a misconfigured TLS-enabled client or server.

1.2. Contribution

In order to detect and correct the deficiencies observed in the TLS implementations, researchers have proposed tools that can identify errors in TLS libraries or deviations from the standards. One notable tool is TLS-Anvil, which checks the compliance of several TLS libraries (supporting TLS 1.2 and 1.3) with industry standards by executing a significant number of security tests. The developers of TLS-Anvil have created a collection of Docker images, enabling researchers to efficiently initiate TLS clients and servers across various versions. This Docker library encompasses approximately 700 versions of 23 distinct implementations and offers a Java interface for the straightforward starting and stopping of TLS implementations. This library has been employed to evaluate 13 widely utilized TLS libraries, specifically BearSSL (v. 0.6), BoringSSL (v. 3945), Botan (v. 2.17.3), GnuTLS (v. 3.7.0), LibreSSL (v. 3.2.3), MatrixSSL (v. 4.3.0), mbedTLS (v. 2.25.0), NSS (v 3.60), OpenSSL (v. 1.1.1i), Rustls (v 0.19.0), s2n (v. 0.10.24), tlslite-ng (v. 0.8.0-a39), and wolfSSL (v 4.5.0).

In this work, we explore how such TLS-Anvil tests can be leveraged to enhance security measures through anomaly detection, specifically within the framework of Suricata. By

exploiting the TLS-Anvil tests, we aim to identify other anomalies in TLS communications that arise from non-compliant TLS implementations, besides the ones already supported by Suricata. A tailored IDS, equipped with a more complete set of specific rules for detecting anomalous TLS connections, is particularly advantageous in situations where it is impractical to assess the software integrity of the node, such as in cases involving remote attestation techniques [25], due to the limited resources available on the node itself. Ultimately, the capability to identify anomalous TLS connections within networking nodes, including routers and switches, is essential for evaluating whether the machines within a local network have been compromised through the installation of non-compliant TLS libraries. Additionally, such an IDS may prove useful in detecting TLS interception proxies that may lower the security capabilities of a node without the knowledge of the user managing the node [26], to detect outdated TLS-enabled IoT or embedded devices, or misconfigured TLS web servers.

1.3. Organization

The paper is organized as follows. Section 2 provides details of key attacks due to TLS complexity, backward compatibility, and implementation errors. Section 3 examines several tools that have been proposed for monitoring TLS connections, as well as for detecting TLS attacks, anomalies, and suspicious connections. Section 4 introduces TLS-Anvil and Suricata, whereas Section 5 outlines our methodology, which is based on the tests conducted with TLS-Anvil to develop Suricata rules intended to identify inconsistencies or errors within TLS traffic. The ultimate objective is to pinpoint compromised nodes operating vulnerable TLS software. We describe in detail the experimental testbed and the commands utilized for deriving and validating the Suricata rules aimed at detecting certain TLS anomalies associated with the OpenSSL library. Lastly, Section 6 offers a summary of the paper and highlights potential future research.

2. Related Work

2.1. Key Attacks Stemming from TLS Complexity & Backward Compatibility

TLS downgrade attack, also referred to as version rollback or bidding-down attacks, occur when attackers deceive clients and servers into utilizing outdated and less secure versions of TLS or SSL. For instance, forcing a connection to revert from TLS 1.2 to SSL 3.0 exposes the system to established vulnerabilities such as POODLE (Padding Oracle On Downgraded Legacy Encryption), which takes advantage of the flawed padding mechanisms in SSL 3.0's block ciphers. Even when a system is equipped to support more current TLS versions, its backward compatibility with SSL 3.0 permits attackers to enforce a downgrade and potentially harvest session cookies. BEAST (Browser Exploit Against SSL/TLS) attack specifically targeted the predictable initialization vectors employed in CBC mode by TLS 1.0, enabling attackers to decrypt secure HTTPS traffic. CRIME (Compression Ratio Info-leak Made Easy) is an attack wherein an adversary can take advantage of TLS compression to extract confidential information, such as authentication cookies, by scrutinizing compressed traffic. The Lucky 13 Attack is a timing attack targeting CBC-mode ciphers in TLS versions 1.1 and 1.2, which exploits slight discrepancies in MAC and padding validation. Another recognized form of attack is the RC4 Bias attack. The RC4 encryption algorithm was maintained to ensure backward compatibility; nevertheless, the statistical biases inherent in its keystream allowed attackers to extract plaintext from encrypted sessions. A different category of attacks arose from vulnerabilities related to TLS renegotiation. Previous iterations of the TLS protocol allowed for insecure renegotiation, thereby enabling man-in-the-middle attacks. This facilitated the injection of malicious commands into sessions that were presumed to be secure.

https://doi.org/10.3390/fi18010062

2.2. Key Attacks from TLS Implementation Errors

TLS implementation errors have led to several high-impact attacks, often more dangerous than the flaws in the protocol specification themselves. These arise when developers misapply cryptographic checks, do not check the size of data received [27], mishandle memory, or introduce timing leaks, creating exploitable vulnerabilities even in otherwise secure versions of TLS. A list of known TLS attacks, together with an explanation of their origin and severity is provided in Table 1, while a short discussion on the software implementations that have been subject to these attacks is given below.

One of the most notorious TLS vulnerabilities is Heartbleed, which arose from a flaw (CVE-2014-0160 [28]) present in specific versions of the OpenSSL library. The implementation(s) that were susceptible did not adequately manage the heartbeat extension during the TLS handshake, thereby enabling remote attackers to extract sensitive information from process memory through specially crafted packets.

In particular, the flaw was found in OpenSSL versions 1.0.1 through 1.0.1f and 1.0.2-beta1. Heartbleed represented one of the most extensive Internet security vulnerabilities recorded, impacting millions of servers, devices, and applications globally. At its peak in April 2014, estimates indicated that over 20% of the world's secure web servers were compromised, including significant platforms such as Yahoo, GitHub, and Cloudflare.

The Lucky 13 attack instead (CVE-2013-0169) [29] was a timing side-channel issue in how padding and MAC checks were implemented. It affected multiple TLS libraries, including OpenSSL, GnuTLS, PolarSSL (later renamed mbedTLS), OpenJDK (Java Secure Socket Extension–JSSE), Network Security Services (NSS) [30]. All the affected libraries were implementing CBC mode in TLS 1.1 and 1.2.

Bleichenbacher's attack is another famous one, which affected nearly all major TLS libraries implementing RSA key exchange with PKCS#1 v1.5 padding. The vulnerability arises when implementations leak information (via error messages or timing) about invalid padding, creating a "padding oracle" that attackers can exploit to decrypt TLS traffic. OpenSSL was vulnerable to Bleichenbacher's original attack and later variants, and multiple CVEs (including those tied to the ROBOT attack in 2017) required patches to fix RSA padding oracles. The GnuTLS library was also susceptible to Bleichenbacher-style padding oracle attacks, since researchers demonstrated practical exploits against certain versions. NSS (Network Security Services—used by Mozilla) has also been found vulnerable to Bleichenbacher oracles. Specifically, all NSS versions prior to 3.41 were affected by a cached side-channel vulnerability [31] so it required fixes to ensure constant-time RSA decryption. The JSSE (Java Secure Socket Extension, part of OpenJDK) was also impacted by this attack [32] and it was updated to mitigate oracle leaks. The mbedTLS / PolarSSL library was vulnerable to Bleichenbacher attacks in earlier versions, and it was later patched to enforce stricter RSA decryption checks. Finally, Cisco confirmed that some on its products, namely Cisco Firepower Threat Defense [33], were affected by Bleichenbacher variants (ROBOT attack, CVE-2017-12373, CVE-2017-15533, CVE-2017-17428).

The ROBOT vulnerability significantly impacted numerous widely utilized TLS implementations and software that continued to support RSA PKCS#1 v1.5 key exchange. Systems exposed to this vulnerability permitted attackers to execute RSA decryption or signing operations utilizing the server's private key, thereby compromising the confidentiality of TLS. The flaw was present in various implementations, including OpenSSL, Cisco TLS stacks, F5 BIG-IP, Fortinet FortiGate, JSSE, NSS, and numerous web applications such as Facebook and PayPal, thereby demonstrating that Bleichenbacher-style vulnerabilities have persisted throughout decades of TLS implementations [34].

The FREAK attack (Factoring RSA Export Keys, CVE-2015-0204) impacted both TLS clients and servers that continued to support obsolete 'export-grade' RSA cipher suites. This

vulnerability was notably prevalent among major operating systems, web browsers, and libraries. It affected OpenSSL, Apple SecureTransport, Microsoft SChannel, the TLS stack in Android, as well as prominent browsers such as Safari, Internet Explorer, Chrome, and Android Browser, in addition to any server configured with RSA_EXPORT cipher suites [35].

Table 1. TLS Attacks due to implementation errors.

TLS Attack (Year)	Origin	Impact	Severity/Cause/Persistence
Heartbleed (2014)	A buffer over-read has been identified in OpenSSL's implementation of the TLS heartbeat extension.	Allows attackers to read up to 64 KB of memory from servers, potentially exposing sensitive information such as private keys, passwords, and session data.	Approximately 20% of widely utilized HTTPS servers were affected at that time.
Lucky 13 (2013)	Timing side-channel in CBC padding and MAC verification.	Allowed attackers to gradually recover plaintext from TLS 1.1/1.2 connections.	Subtle differences in how implementations handled padding errors.
Bleichenbacher's Oracle Attacks (1998)	Incorrect handling of RSA PKCS#1 v1.5 padding in TLS implementations.	Enabled decryption of TLS sessions by exploiting error messages or timing differences.	Resurfaced repeatedly. Variants reappeared in different TLS libraries over decades.
ROBOT (Return Of Bleichenbacher's Oracle Threat) (2017)	Modern TLS stacks still vulnerable to Bleichenbacher-style RSA padding attacks.	Allowed decryption and impersonation attacks against servers using RSA key exchange.	The attack demonstrated that Bleichenbacher-style padding oracle vulnerabilities persisted for nearly 20 years after the original discovery in 1998.
FREAK (Factoring RSA Export Keys) (2015)	Implementation allowed forced downgrade to weak "export-grade" RSA keys.	Attackers could break 512-bit RSA keys and decrypt traffic.	Mismanagement of backward compatibility in libraries.
DROWN (2016)	Cross-protocol attack exploiting SSLv2 support in TLS implementations.	Attackers could decrypt TLS sessions if servers shared keys with SSLv2.	Poor isolation between protocol versions

3. Tools for Detecting TLS Attacks, Anomalies or Suspect Connections

With the rise in TLS vulnerabilities, ensuring the integrity and safety of TLS connections has thus become a primary concern. On one hand, intensive work has been performed to update the specification, resulting in new versions, and the latest one (TLS 1.3) has been fully redesigned with respect to the previous versions. On the other hand, there have been created specialized network monitoring tools, scanners, and tools for detecting TLS anomalies, attacks, or suspicious connections. They range from research frameworks for protocol testing to enterprise-grade monitoring solutions that analyze certificates, cipher suites, and traffic patterns. Table 2 illustrates some of these tools whose characteristics are briefly described below.

Table 2. Categories of TLS Anomaly/Vulnerability Detection Tools.

Category	Example Tools	Focus
Intrusion Detection Systems & Network monitoring tools	Suricata, Zeek	Deep inspection of packet (flows). Allow the identification of many TLS vulnerabilities in the network traffic, such as weak cipher suites, outdated algorithms (like MD5, DES), and obsolete protocol versions, like SSL v3.0, TLS 1.0, or TLS 1.1. Support the inspection of X.509 certificate(s) exchanged in the TLS handshake, allowing the identification of expired or self-signed certificates, as well as other deviations in the certificate format.
TLS Protocol Testing	TLS-Attacker, testssl.sh [36], TLSAssistant [37–39]	Simulate TLS attacks, detect protocol flaws.
Certificate Analysis	TLS hunting (Databricks), x509 monitoring (Censys)	Spot malicious certificate anomalies. Censys provides detailed information about all certificates (for web domains) on the Internet, including the expired ones and the untrusted ones.
TLS Server Scanners	Qualys SSL Labs, Pentest-Tool SSL/TLS Scanner, SSLyze, Nmap	Check configs, cipher suites, cert validity for target server(s), typically for web servers. Provide an indication of TLS vulnerabilities when the scanners are executed, either as required or at designated times or intervals. May suggest remediation strategies for the identified vulnerabilities, including applicable patches.
Continuous Monitoring (TLS attack detection)	Threat-TLS, TLS-Monitor	Detect evolving TLS threats in real time by exploiting network monitoring tools, like Suricata or Zeek. Exploit other tools (e.g, Metasploit [40], Nmap, and TLS-Attacker) to validate alarms indicating potential TLS attacks.
TLS Compliance Tools	TLS-Anvil, FLEXTLS [41]	Test compliance of a TLS server or client implementation with the protocol specification. May be used also by penetration testers to estimate the quality of a TLS stack.

TLS Scanner tools such as Qualys SSL Labs Server Test, Pentest-Tools SSL/TLS Scanner, testssl.sh, SSLyze [42], and Nmap [43], are frequently used for scanning servers looking for SSL/TLS vulnerabilities, misconfigurations, and weak cryptographic settings. Typically, they generate a graded report to facilitate error remediation. They generally can provide the following information related to the scanned host(s): (a) identify supported TLS/SSL versions and cipher suites; (b) identify where the target is vulnerable to TLS attacks including Heartbleed, ROBOT, weak ciphers, and risks associated with protocol fallback; (c) retrieve

the certificate (and the related chain) of the target, validate the chain, checks the expiration, and issuer trust; (d) detect weak keys, insecure renegotiation, or fallback mechanisms.

TLS-Attacker [44] is a Java-based framework designed for the analysis of TLS libraries. It can simulate man-in-the-middle attacks, downgrade attempts, and protocol vulnerabilities. This framework is frequently utilized by researchers and penetration testers to assess the integrity of TLS implementations.

Threat-TLS [23] and TLS-Monitor [22] are research tools developed to identify vulnerable, malicious, or suspicious TLS connections by analyzing the network traffic. By exploiting IDS tools like Suricata and Zeek, these tools aim to detect anomalies that can generate TLS attacks against a target, and validate the raised alarms by running TLS attack tools against the target. They are useful in various contexts both in networked environments as well as in other contexts, like IoT, mobile, and embedded systems environments.

Certificate-based Threat Hunting [45] utilizes TLS certificates obtained from the handshake process as a data source for anomaly detection. This approach aids in the identification of malicious infrastructures, botnets, or malware that utilize TLS to disguise their traffic. For instance, it can be employed to detect ransomware such as LockBit or Remote Access Trojans (RATs) that depend on TLS channels.

TLS-Anvil is designed to assess the compliance of server or client implementations with the specifications of the protocol. It serves as a valid tool for penetration testers to evaluate the robustness of a TLS stack, as well as for developers to test their applications that utilize TLS.

4. Tls-Anvil and Suricata: Presentation

4.1. TLS-Anvil

TLS-Anvil, developed in Java, serves as a test suite that employs combinatorial testing [46,47] to identify violations of protocol specifications by TLS servers or clients through systematic testing of various parameter value combinations. TLS-Anvil is a tool that utilizes software testing methodologies to stimulate a System Under Test (SUT) using specified inputs while monitoring the system's responses. In the development of test templates, the authors meticulously examined TLS-related RFCs to identify essential requirements, which are denoted by the keywords MUST, SHALL, or REQUIRED. Furthermore, the tool incorporates known state machine vulnerabilities from related research [48]. From a practical standpoint, TLS-Anvil is built upon several libraries, including JUnit 5 (test framework), TLS-Attacker (TLS stack), TLS-Scanner (scanner derived from TLS-Attacker), and coffee4j (combinatorial testing library).

Figure 1 illustrates the overall architecture of TLS-Anvil, encompassing the various phases, as detailed in [24], that are executed during the testing process.

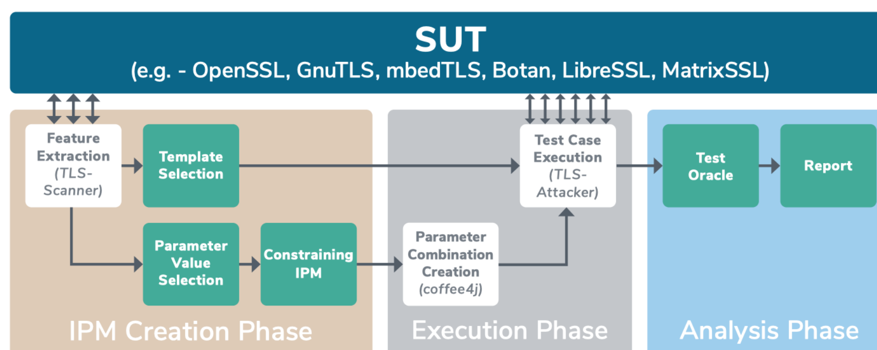

Figure 1. TLS-Anvil Architecture (source: [24]).

Each test template constitutes a JUnit test function designed to validate a specific requirement and establishes the expected outcome for all associated test cases. Essentially, it serves two primary functions: (1) It delineates the TLS messages that the test suite is responsible for sending and receiving; (2) It determines the criteria for the success or failure of a test case.

The template has additional Java annotations that define an IPM (Input Parameter Model), which contains all relevant test parameters and their values. The IPM is used to generate test inputs (a value is assigned to each parameter) using t-way combinatorial tests. Different IPMs are defined for each test template, depending on the requirement it verifies.

TLS-Anvil consists of two primary Java modules: TLS-Testsuite and TLS-Test-Framework. The TLS-Testsuite serves as the main module, holding all templates located in the `de.rub.nds.tlstest.suite.tests` package. Within this package, three test types are available: server, client, and both, containing tests applicable to both peers. The tests are categorized further based on RFC standards. In contrast, the TLS-Test-Framework module contains all JUnit extensions and the logic required for test execution.

4.2. Suricata

Suricata is a high-performance intrusion detection and prevention system, which is also extensively utilized as a network security monitoring engine. This software is open-source and managed by a community-driven nonprofit organization known as the Open Information Security Foundation (OISF). Similar to other intrusion detection systems, Suricata offers capabilities for threat detection. Furthermore, it enables traffic filtering and monitoring. Suricata is capable of identifying prevalent attack types, including port scanning and denial-of-service attacks. This software employs a comprehensive set of rules for threat detection and analysis, granting network administrators the capability to formulate and implement their own detection rules. Suricata operates in a multi-threaded manner, enabling it to execute multiple threads concurrently, provided that the system's capabilities permit. Furthermore, by configuring the `max-pending-packets` parameter within the `suricata.yaml` file, one can determine the quantity of packets that Suricata may process concurrently. This value can vary from one to several tens of thousands of packets, thereby preventing the loss of packets resulting from system saturation.

We will explain how we derived some new Suricata rules to detect anomalous TLS implementations by analyzing the traffic generated in TLS-Anvil tests. In order to comprehend the derived rules, we will present a concise overview of the semantics associated with Suricata rules, exemplified by a specific Suricata rule (rule/signature).

A Suricata rule is made up of three parts: action, header, and options.

Action. This term represents the initial component of the rule, highlighted in red in Figure 2. The potential actions that can be specified are as follows:

```
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request
Containing Rule in URI"; flow:established,to_server; http.method;
content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-
unknown; sid:123; rev:1;)
```

Figure 2. Example Suricata rule.

- `alert`—triggers an alert.
- `pass`—prevents packet inspection.
- `drop`—discards the packet and triggers an alert.
- `reject/rejectsrc`—sends an RST/ICMP unreachable error to the sender of the packet.
- `rejectdst`—functions similarly to reject, but targets the recipient.
- `rejectboth`—operates like reject, but affects both peers.

Header. This part is shown in green in Figure 2 and consists of the following:

1. Protocol—the protocol the rule refers to; the main ones are TCP, UDP, ICMP, and IP.
2. Source/Destination—IP address (or range of addresses) of the packet's sender/recipient.
3. Source/Destination Port—port (or range of ports) of the sender/recipient.
4. Direction—direction of the traffic to be analyzed, usually `->`.

The address and port fields can be replaced with `any` if you do not want a specific restriction.

Rule Options. This section is depicted in blue in Figure 2 and delineates the specifications of the rule. It encompasses keywords from diverse categories, including Meta and Payload. The Meta keywords do not directly influence the inspection of network traffic; however, they do impact the manner in which events and alerts are communicated. The primary Meta keywords are as follows:

- `msg` (message)—provides information about the rule and the potential alert, such as the RFC it references or the potential attack in progress.
- `sid` (signature ID)—gives each rule its ID (a number greater than 0).
- `rev` (revision)—often appears after `sid`; represents the version of the rule, i.e., a number that is incremented each time it is modified.

The Payload keywords are used to inspect the contents of a packet's payload. The keywords used in deriving the Suricata rules in our work are as follows:

- `content`—indicates in quotation marks what you want to be present in the packet; if the content is ASCII, it is simply written, while hexadecimal is delimited by two slashes, "`|`", for example, `content:‘‘|16 03 03|’’;`.
- `offset`—indicates the byte of the payload from which a match will be sought; for example, `offset:3;` checks from the fourth byte onwards.
- `depth`—indicates how many bytes from the beginning of the payload will be checked. `offset` and `depth` can be combined and are often used together; for example, in Figure 3, `offset:3; depth:3;` checks the fourth, fifth, and sixth bytes of the payload, allowing to find/match the indicated content (as emphasized with the red sign).
- `distance`—unlike the previous modifiers, it is a relative modifier, meaning it refers to the previous `content`. The value given to `distance` determines how many bytes away from the previous match to start searching for the new match.
- `within`—specifies within how many bytes from the previous `content` to search for a match. `distance` and `within` are often used together to search within a specific range of bytes; in the example shown in Figure 4, `distance:0; within:3;` limits the search to the 3 bytes following the content `"abc"` determining a 'no match' for the content "def".
- `byte_test`—is a keyword with several input values:
    1. `<num of bytes>`—the number of bytes to be examined.
    2. `<operator>`—is the operator used to compare the bytes with the test value; it can be `<, >, =, <=, >=, &` (if preceded by an exclamation point, they are negated).
    3. `<test value>`—value to compare the bytes against, using the operator; can be decimal or hexadecimal.
    4. `<offset>`—indicates the position in the payload of the bytes to be selected. By adding `relative`, the offset will not be absolute, but will be considered as the distance from the previous content.

https://doi.org/10.3390/fi18010062

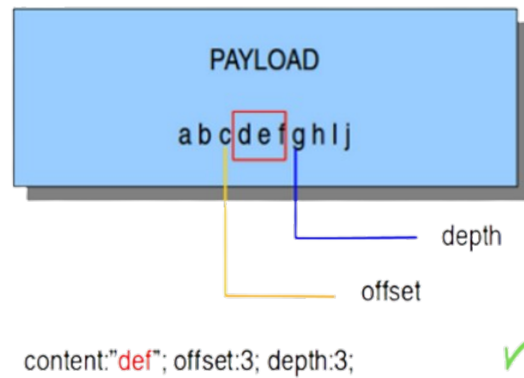

Figure 3. Example for specifying an "offset" and "depth" in a Suricata rule (match) [49].

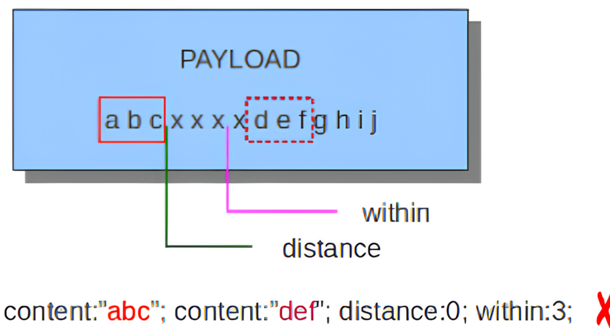

Figure 4. Example for specifying a "distance" and "within" in a Suricata rule (no match) [49].

Finally, the keyword `flow` indicates the flow direction, that is, one between `from_server` and `from_client`.

4.3. Suricata TLS Detection Capabilities

By default, Suricata is unable to decrypt TLS traffic. Nevertheless, it can examine various X.509v3 certificate fields and extensions, as well as TLS handshake extensions [50] by utilizing a specific set of SSL/TLS keywords: (a) the subject and issuer of TLS certificates (keywords: tls.cert_subject and tls.cert_issuer); (b) the validity period of the certificate (keywords: tls_cert_notbefore, tls_cert_notafter, tls_cert_expired); (c) the subject alternative name extension (keyword: tls.subjectaltname); (d) the length of the certificate chain (keyword: tls.cert_chain_len); (e) the random bytes transmitted in the Client Hello and Server Hello messages (keywords: tls.random and tls.random_bytes); (f) the Server Name Indication (SNI) extension found in the Client Hello message (keyword: tls.sni); (g) TLS versions and cipher suites (keywords: tls.version and tls.cipher); (h) JA3 and JA3S fingerprints (available in Suricata version 6 and later); and (i) anomalies in X.509v3 certificates, which may include self-signed, expired, or mismatched certificates.

By using these SSL/TLS keywords, it is possible to write Suricata rules for the following: (a) detecting TLS connections that use weak or deprecated cipher suites; (b) detecting TLS ClientHello with malformed or missing SNI (this case is commonly encountered in malware software); (c) detecting self-signed certificates; (d) detecting TLS traffic to suspicious or newly registered domains (by using SNI); (e) detecting anomalies related to TLS certificate chain length, such as very long chains when the common length is 2 or 3; (f) detecting TLS connections with a handshake containing invalid or zeroed random bytes; (g) detecting TLS handshake with timestamp = 0 (this case is commonly encountered in malware); (h) detecting TLS ClientHello with unusual cipher suite ordering (this case is commonly encountered in malware); (i) detecting TLS handshake with self-signed cert

and mismatched SNI (this case is commonly encountered in malware); (j) detecting TLS handshake with invalid or empty ALPN, where typically the legitimate web browsers send ALPN (e.g., h2, http/1.1).; (k) detecting TLS handshake with malformed extensions, since malware often sends incomplete or malformed extension blocks; (l) detecting TLS version mismatch between record layer and handshake. Some examples of such rules are already provided in the official Suricata documentation [50].

5. Deriving Suricata Rules from TLS-Anvil Tests

In this section, we provide further details regarding the objectives of our research. By utilizing the extended Suricata TLS inspection engine, enhanced with the derived rules, we aim to identify anomalous TLS implementations potentially run by man-in-the-middle (MITM) tools, TLS interception proxies, as well as malicious or misconfigured clients and servers that might compromise the security of secure channels. On one hand, security threats, including malware and botnets, are increasingly exploiting TLS (1.3 and 1.2) to avoid detection, thereby necessitating urgent analysis and identification of their traffic behavior. On the other hand, network operators and service providers are still required to identify and manage TLS traffic in a legal and compliant manner to ensure network optimization and facilitate anomaly diagnosis. Furthermore, in particular domains such as cybercrime investigation and national security, there persists a demand for the compliant analysis of TLS 1.2 & TLS 1.3 traffic and the detection of suspect behavior [51,52].

Threat Model. We consider the following potential adversaries: authors of commodity malware, operators of custom command and control (C2) systems, malicious insiders, and active MITM operators including TLS interception software installed with or without end-user's awareness. The considered adversaries have the following capabilities: they could craft custom TLS ClientHello/ServerHello messages, present forged or self-signed certificates, utilize deprecated TLS versions or ciphers, use protocol legacy versions, send empty TLS records, try to use non-existent signature algorithms, try to use TLS implementations that slightly deviate from the TLS standard RFCs and install them on the target (victim).

We provide a comprehensive description of several new Suricata rules that we have created by exploiting a subset of TLS-Anvil tests and the resultant traffic. We will specify the TLS-Anvil test along with the relevant file(s) used to create each TLS anomaly, followed by the presentation of the corresponding Suricata rule. All files pertaining to the TLS-Anvil tests can be found in the tests folder of the GitHub repository [53]. Consequently, all file paths referenced in this section are derived from that specific folder.

5.1. TLS 1.2 Anomaly: Empty TLS Records

The TLS protocol specification does not prohibit the transmission of zero-length *Application Data* records, as such empty records may serve a function as a countermeasure for traffic analysis. An adversary could potentially infer the content of the communication by scrutinizing the length of the packets. Conversely, receiving a substantial number of empty records without accompanying legitimate data may suggest an attempted denial-of-service attack. Consequently, it is prudent to trigger an alert to flag this suspicious message. For this reason, we define a specific Suricata rule to identify empty *Application Data* records.

TLS-Anvil test description and files. In order to establish the rule, we utilize the TLS-Anvil test, which transmits an empty *Application Data* record, that is with a length set to zero. The test is located in TLS-Anvil in the file named `Fragmentation.java` available in TLS-Anvil tests folder [54]. Figure 5 shows a Wireshark screenshot of a dump file generated by the test, where the *Application Data* record with length zero is highlighted.

Derived rule. The derived Suricata rule for detecting a TLS Record of type *Application Data* with length zero is shown below:

```
alert tls any any -> any any (msg:"RFC 5246: Empty Application Data";
flow:from_client;
content:"|17 03 03 00 00|"; depth:5;
sid:1000004;)
```

The rule searches for an empty *Application Data* record, where the string "|17 03 03 00 00|" indicates a record of type *Application Data* (0x17) with TLS version 1.2 (0x0303) and length 0 (0x0000); this sequence must be at the beginning of the record (`depth:5;`).

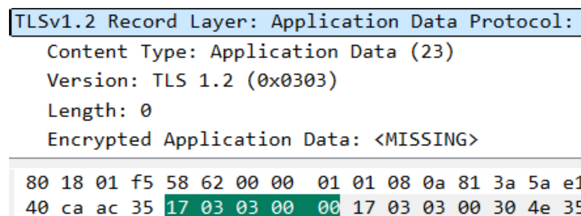

Figure 5. Wireshark dump showing the traffic generated in the test for TLS Record of type *Application Data* with length 0.

Sending TLS *Handshake* record fragments of length 0 is instead forbidden (by the RFC), and in this case the server should respond with a fatal alert.

TLS-Anvil test description and files. In this test, the client (i.e., TLS-Anvil) sends an empty TLS handshake message, namely a *Finished* record. Figure 6 shows a Wireshark screenshot of a dump file generated by the test, where the *Finished* record having the length 0 is highlighted. The test is located at the path `both/tls12/rfc5246/Fragmentation.java` in the TLS-Anvil repository. Figure 6 illustrates the screenshot captured with Wireshark for a dump file generated in this test, where it can be noticed the *Handshake* message of length 0. In Figure 6, we observe that Wireshark also flags the TLS record as "not allowed" due to its length of 0, through using a specific error message.

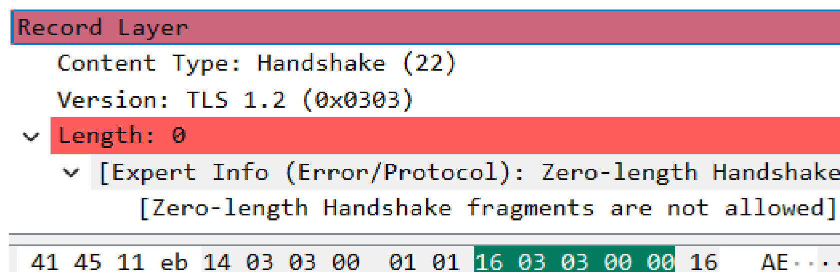

Figure 6. Wireshark dump showing the traffic generated in the test for TLS Record of type Handshake with length 0.

Derived rule. We derived a new Suricata rule for an empty Finished TLS handshake record, as shown below:

```
alert tls any any -> any any (msg:"RFC 5246: Empty Finished Record";
flow:from_client;
content:"|16 03 03 00 00|"; depth:5;
sid:1000005;)
```

The rule searches for a message of type Handshake with length 0, where |16 03 03 00 00| indicates a *Handshake* message (0x16) with TLS version 1.2 (0x0303) and length equal to 0 (0x0000); this sequence must be found at the beginning of the TLS record (`depth:5;`).

5.2. TLS 1.2 Anomaly: Undefined TLS Record Content Type

According to the TLS 1.2 specification, any TLS record must contain information about the type of the higher-level protocol carried inside the TLS record. The type, corresponding to application data, TLS handshake message, change cipher spec message, or alert messages, can be one of the following:

```
enum {
change_cipher_spec(20), alert(21), handshake(22),
application_data(23), (255)
} ContentType;
```

In hexadecimal, these values correspond to `0x14, 0x15, 0x16` and `0x17`.

We consider the following anomaly: the presence of an undefined TLS Record type within the Client Hello TLS handshake message. We run the following test with TLS-Anvil.

TLS-Anvil test description and files. TLS-Anvil sends a Client Hello with an incorrect `Content Type`, specifically set to `0xFF` instead of `0x16`, as shown in Figure 7. A server implementing the TLS 1.2 specification should detect the presence of an invalid content type and should respond with a fatal alert of type `unexpected_message`. The test called `sendNotDefinedRecordTypesWithClientHello` is found in the file `server/tls12/rfc5246/TLSRecordProtocol.java` located at [54]. Another test, named `sendNotDefinedRecordTypesWithCCSAndFinished` test, found in the same file, performs a similar operation, but with the ChangeCipherSpec and Finished handshake messages. In Figure 7, we show a Wireshark screenshot of a dump file generated by these test(s), where the first 3 bytes of the record are marked, representing the content type used in this test (0xff) and the TLS version (0x0303).

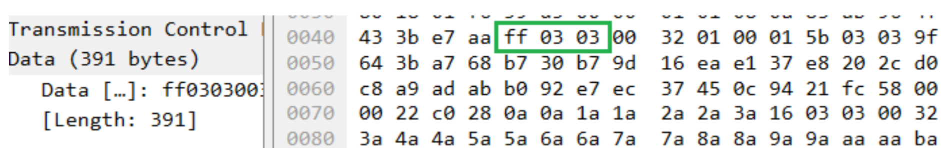

Figure 7. Wireshark dump showing the traffic generated in the tests for undefined TLS Record content type.

Derived rule. We have formulated the following Suricata rule pertaining to an unspecified TLS Record type:

```
alert tls any any -> any any (msg:"RFC 5246: Not Defined ContentType";
flow:from_client;
byte_test:1,!=,0x14,0;
byte_test:1,!=,0x15,0;
byte_test:1,!=,0x16,0;
byte_test:1,!=,0x17,0;
content:"|03 03|"; offset:1; depth:2;
sid:1000009;)
```

This rule searches for a TLS record with an undefined content type, which refers to any content type that deviates from the four permissible values specified in RFC 5246 [6]. More in detail:

- the `byte_test` functions check the first byte of the record (`offset 0`), verifying whether it is different from the four content type values allowed for a TLS record.
- |03 03|: This sequence corresponds to the version field of the record and indicates TLS 1.2 (0x0303); the two bytes are the second and third of the record, i.e., 2 bytes deep with an offset of 1 (`offset:1; depth:2;`).

5.3. TLS 1.2 Anomaly: TLS Record with Excessive Size (Too Large CipherText)

As per the TLS 1.2 specification [6], each TLS record fragment with encrypted application data should not exceed $2^{14} + 2048$ bytes. We consider the anomaly for a TLS record fragment (containing encrypted application data) whose exceeds $2^{14} + 2048$ bytes. We run the following test using TLS-Anvil.

TLS-Anvil test description and files. TLS-Anvil transmits a fragment of encrypted application data (`TLS_Ciphertext`) that exceeds the size of $2^{14} + 2048$ (18,432 bytes) specified by the RFC. The server should respond with a fatal alert of type `record_overflow`. Figure 8 shows a Wireshark screenshot of a dump file generated by the test, highlighting the length field, which is greater than `0x4800`. We observe thhat Wireshark also reports the prohibited length of the record by displying a specific error message. The test is found at the path `both/tls12/rfc5246/Fragmentation.java` in the TLS-Anvil repository.

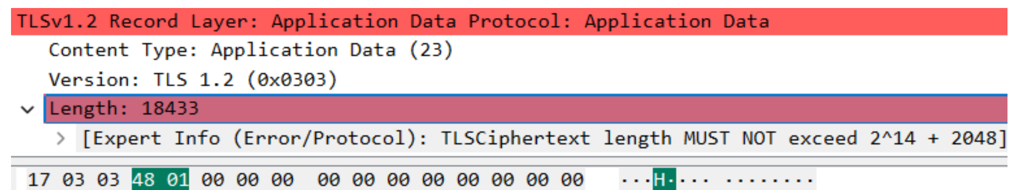

Figure 8. Wireshark dump showing the traffic generated in the test for a TLS Record whose size exceeds the limited allowed by the RFC ($2^{14} + 2048$ bytes).

Derived rule. We have created the following Suricata rule for a TLS Record of type *Application Data* with an excessive size:

```
alert tls any any -> any any (msg:"RFC 5246: Too Large CipherText
(>2^14+2048 bytes)";
flow:from_client;
content:"|17 03 03|"; depth:3;
byte_test:2,>,0x4800,0,relative;
sid:1000007;)
```

The above rule detects a TLS record of type *Application Data* with a length field whose size is greater than $2^{14} + 2048$. The meaning of the fields in the Suricata rule is explained next:

- `|17 03 03|`: indicates a record of type *Application Data* (0x17) for the version TLS 1.2 (0x0303); this sequence is found at the beginning of the TLS record (`depth:3;`).
- the function `byte_test` verifies if the number represented by the immediately following 2 bytes (`0,relative`) is greater than $2^{14} + 2048$, or 4800 in hexadecimal (`>,0x4800`).

5.4. TLS 1.3 Anomaly: TLS Record with Excessive Size

In TLS 1.3, unlike earlier TLS versions only Authenticated Encryption with Associated Data (AEAD) ciphers are required. AEAD algorithms combine encryption and authentication, converting plaintext into authenticated ciphertext and vice versa. Each encrypted record has a cleartext header and an encrypted body, which includes a type and optional padding.

```
struct {
ContentType opaque_type = application_data; /* 23 */
ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
uint16 length;
opaque encrypted_record[TLSCiphertext.length];
```

```
} TLSCiphertext;

struct {
opaque content[TLSPlaintext.length];
ContentType type;
uint8 zeros[length_of_padding];
} TLSInnerPlaintext;
```

The length (in bytes) of the TLSCiphertext.encrypted_record is the sum of the following:

- length of the internal plaintext, maximum equal to 2^{14} bytes;
- 1 byte for the ContentType;
- any expansion added by the AEAD algorithm, which must not exceed 255 bytes.

The maximum length must not exceed $2^{14} + 256$ bytes. If a peer obtains a record that surpasses this limit, it must close the connection with a `record_overflow` alert. We will perform a test using TLS-Anvil.

TLS-Anvil test description and files. TLS-Anvil sends a record with a ciphertext length surpassing the RFC 8446 limit of $2^{14} + 256$ bytes (16,640 bytes). The server should send a fatal alert of type `record_overflow` and close the connection. The test is located at the (file) path `both/tls13/rfc8446/RecordProtocol.java`. Figure 9 displays a Wireshark screenshot of a dump file from the test, emphasizing that the length field exceeds `0x4100`, with Wireshark also indicating that it surpasses the size of the TLS record as specified by the RFC.

```
TLSv1.3 Record Layer: Application Data Protocol: Application Data
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
  v Length: 16641
      > [Expert Info (Error/Protocol): TLSCiphertext length MUST NOT exceed 2^14 + 256]
    Encrypted Application Data […]: 0000000000000000000000000000000000000000000000000

 17 03 03 41 01 00 00 00  00 00 00 00 00 00 00 00   ···A···· ········
 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
```

Figure 9. Wireshark dump showing the traffic generated in the test for a TLS Record exceeding the maximum allowed size (in TLS 1.3).

Derived rule. We created the following Suricata rule for a TLS Record exceeding the limit allowed by the RFC:

```
alert tls any any -> any any (msg:"RFC 8446: Too Large CipherText
 (>2^14+256 bytes)";
flow:from_client;
content:"|17 03 03|"; depth:3;
byte_test:2,>,0x4100,0,relative;
sid:1000019;)
```

This rule searches for a record of type *Application Data* whose size exceeds $2^{14} + 256$ bytes.

- |17 03 03|: indicates a record of type *Application Data* (0x17) with TLS 1.2 as record version (0x0303); this sequence must be at the beginning of the record (`depth:3;`).
- the function `byte_test` verifies whether the number represented by the next 2 bytes (`0,relative`) is greater than $2^{14} + 256$, or 4100 in hexadecimal (`>,0x4100`).

5.5. TLS 1.3 Anomaly: Send Empty TLS Records

TLS 1.3 implementations must not send TLS handshake message fragments with a length of 0, regardless of whether such fragments include padding. On the other hand,

application data fragments of length 0 may be sent, as they are potentially useful as a traffic analysis countermeasure.

TLS-Anvil test description and files. The test sends empty records, which are defined as records with a length field equal to zero, encompassing various types, specifically including empty records of the Handshake, Application Data, and ChangeCipherSpec types. We note that ChangeCipherSpec is an obsolete type in TLS 1.3 and is utilized solely for compatibility purposes. The test also sends records with a content type not used in TLS 1.3, such as `0x18` (corresponding to the Heartbeat extension).

In accordance with the specification, a TLS server must recognize fragments of length zero and terminate the connection by transmitting a fatal alert. test is found at the path `both/tls13/rfc8446/RecordProtocol.java` in the TLS-Anvil repository. Figure 10 illustrates the traffic captured using Wireshark for this test, specifically for a TLS record whose content type is Handshake. It is important to highlight that Wireshark also reports a length of 0, which is not allowed by the RFC. The same methodology was followed for a TLS 1.2 record of type *Application Data*, as described in Section 5.1.

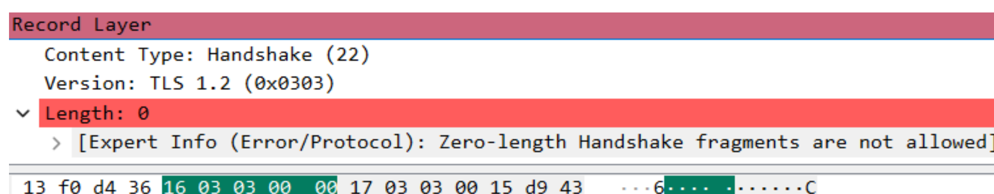

Figure 10. Wireshark dump showing the traffic generated in the test for a zero-length TLS Record of type Handshake.

Derived rule. The new Suricata rule we have derived by analyzing the captured traffic for this test is shown below:

```
alert tls any any -> any any (msg:"RFC 8446: Empty Record";
flow:from_client;
byte_test:1,>,0x13,0;
byte_test:1,<,0x20,0;
content:"|03 03 00 00|"; offset:1; depth:4;
sid:1000020;)
```

The rule looks for a record of length 0, where the record can have one of the allowed content types.

- the `byte_test` functions verify whether the first byte, which indicates the content type, is in a range between `0x14` and `0x19` (`>,0x13 && <,0x20`), since these are the types of records used in the test.
- |03 03 00 00|: indicates a record of version TLS 1.2 (0x0303) and the length equal to 0 (0x0000); This sequence must start at the second byte of the record (`offset:1; depth:4;`).

5.6. TLS 1.3 Anomaly: Invalid Legacy Version (Higher)

In earlier versions of TLS, the `legacy_version` field within the Client Hello handshake message was utilized for version negotiation, representing the highest version number that the client supported. In TLS 1.3, the client communicates its version preferences through the `supported_versions` extension, and the `legacy_version` field must be set to `0x0303`, which corresponds to TLS 1.2. In the context of TLS 1.3, it is imperative that clients consistently set the value of `legacy_version` to 0x0303 and include the `supported_versions` extension, specifying `0x0304` as the highest supported version.

TLS-Anvil test description and files. The test transmits a Client Hello message utilizing a higher legacy version of TLS 1.2, specifically one that exceeds the value of `0x0303`. The RFC states that the legacy version of Client Hello must always be `0x0303` and TLS 1.3 must only be declared in the `supported_versions` extension, so the server must send a fatal alert and close the connection.

The test is located at the path `server/tls13/rfc8446/ClientHello.java` in the TLS-Anvil repository. Figure 11 illustrates the traffic captured using Wireshark for this particular test, with the Client Hello version emphasized (0x0304). It is noteworthy that Wireshark additionally indicates the irregular `legacy_version` field.

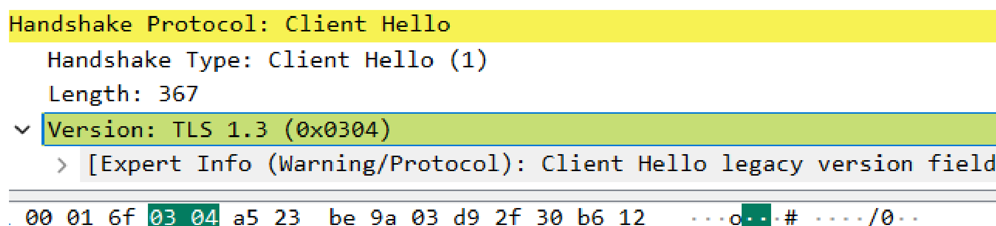

Figure 11. Wireshark dump showing the traffic generated in the test for invalid legacy version (higher than `0x0303`.)

Derived rule. Based on this test and by analyzing the traffic intercepted with Wireshark we have defined the followiing Suricata rule:

```
alert tls any any -> any any (msg:"RFC 8446: v1.3 Client Hello with a
 wrong Legacy Version (higher than TLS 1.2)";
flow:from_client;
content:"|16 03 03|"; depth:3;
content:"|01|"; distance:2; within:1;
content:"|03 04|"; distance:3; within:2;
sid:1000021;)
```

The following derived Suricata rule searches for a Client Hello with the version legacy equal to `0x0304`.

- |16 03 03|: indicates a Handshake message (0x16) with the version of the record layer equal to TLS 1.2 (corresponding to the value 0x0303); this sequence must be at the beginning of the record (`depth:3;`).
- |01|: indicates a Client Hello handshake message; it is located 2 bytes away from the first sequence.
- |03 04|: this is the legacy version of Client Hello, set to TLS 1.3; it is 3 bytes away from the Handshake type.

5.7. TLS 1.3 Anomaly: Invalid Legacy Version (Lower)

The test sends a Client Hello using a lower legacy version, which is below `0x0303`. The RFC stipulates that the legacy version of Client Hello must consistently be set to `0x0303`. Consequently, if the server identifies a lower legacy version, it is required to issue a fatal alert and terminate the connection. The test is located at the path `server/tls13/rfc8446/ClientHello.java` in the TLS-Anvil repository. Figure 12 shows the traffic captured with Wireshark for this test, in which the legacy version of the Hello Client (0x0302) and the `supported_versions` extension are highlighted.

```
Extension: supported_versions (len=3) TLS 1.3
    Type: supported_versions (43)
    Length: 3
    Supported Versions length: 2
    Supported Version: TLS 1.3 (0x0304)

01 00 00 87 03 02 97 10  71 d1 95 4d 1d ee 76 53
dd da e3 1e 70 f8 a8 47  18 07 ea 13 12 f0 98 24
9d af e1 46 39 2a 00 00  02 13 02 01 00 00 5c 00
0b 00 02 01 00 00 0a 00  04 00 02 00 1d 00 01 00
01 02 00 0d 00 14 00 12  08 05 08 04 08 06 04 01
05 01 06 01 04 03 05 03  06 03 00 2b 00 03 02 03
04 00 33 00 26 00 24 00  1d 00 20 b1 e8 23 6b 63
```

Figure 12. Wireshark dump showing the traffic generated in the test for invalid legacy version (lower than `0x0303`).

Derived rule. The Suricata rule that we developed following the analysis of the traffic generated during this test is presented below:

```
alert tls any any -> any any (msg:"RFC 8446: v1.3 Client Hello with a
 wrong Legacy Version (lower than TLS 1.2)";
flow:from_client;
content:"|16 03 03|"; depth:3;
content:"|01|"; distance:2; within:1;
byte_test:2,<,0x0303,3,relative;
content:"|00 2b 00|";
content:"|03 04|"; distance:2; within:2;
sid:1000022;)
```

The above rule searches for a Client Hello with the legacy version lower than `0x0303` and with TLS 1.3 declared in the `supported_versions` extension.

- |16 03 03|: indicates a Handshake message (0x16) with Record Layer version equal to TLS 1.2 (0x0303); this sequence must be at the beginning of the record (`depth:3;`).
- |01|: indicates a Client Hello handshake message; it is located 2 bytes from the first sequence.
- the `byte_test` function checks whether the two octets of the legacy version of the Client Hello are less than the value corresponding to TLS 1.2 (`<,0x0303`); they are 3 bytes away from the Handshake type (`3,relative`).
- |00 2b 00|: represents the `supported_versions` extension (0x002b) and the first byte of the length field, usually equal to 0.
- |03 04|: identifies TLS 1.3 within the extension; it is included in the rule to ensure that the Client Hello is actually version 1.3.

5.8. Testing the Derived Suricata Rules

We have established a testbed for the purpose of conducting tests with TLS-Anvil and for validating the newly developed Suricata rules. A local TLS server has been established, utilizing the OpenSSL library version 1.1.1i, which serves as the System Under Test (SUT).

We used the TLS-Docker-Library [55], which provides Docker images for different versions of TLS libraries. A dedicated Docker network has been established for the TLS-Anvil containers and the server. Specific tags were created to facilitate the execution of individual tests within TLS-Anvil. An OpenSSL server has been configured with an RSA certificate, and TLS-Anvil, functioning as a TLS client, was initiated for each specific test using custom tags, such as S7507incFallSCSV. The specific installation and execution commands are detailed below, accompanied by a concise explanation:

https://doi.org/10.3390/fi18010062

```
1 $ docker run \
2 -d \
3 --rm \
4 --name openssl-server \
5 --network tls-anvil \
6 -v cert-data:/certs/ \
7 ghcr.io/tls-attacker/openssl-server:1.1.1i \
8 -port 8443 \
9 -cert /certs/rsa2048cert.pem \
10 -key /certs/rsa2048key.pem
```

Lines 2–6 are command flags related to Docker, line 7 specifies the Docker image from the library, and lines 8-10 contain the flags passed as parameters to the OpenSSL server. Next, TLS-Anvil is started. The current directory is mounted in the Docker container and is used to store the results.

```
1 $ docker run \
2 --rm \
3 -it \
4 --name tls-anvil \
5 --network tls-anvil \
6 -v $(pwd):/output/ \
7 tlsanvil \
8 -identifier openssl-server \
9 -tags S7507incFallSCSV \
10 server \
11 -connect openssl-server:8443
```

- Lines 2–5 are Docker-related command flags.
- Line 6 sets the output directory (the current one).
- Line 7 specifies the TLS-Anvil image. The image from the offial TLS-Anvil site (`ghcr.io/tls-attacker/tlsanvil:latest`) cannot be used because it would be downloaded at the moment, so it would not contain user-added tags (necessary to perform an individual test) such as `S7507incFallSCSV`.
- Line 9 defines the textbftag to use.
- Line 11 determines how TLS-Anvil connects to the server.

Following the derivation of the Suricata rules, we proceeded to validate them by initially installing Suricata version 7.0.10. Subsequently, we updated the rules file with the derived rules, which were inserted into a distinct file named `derivedsuricatarulesTLSanomalies.rules`. We have updated the file `suricata.yaml` in the `rule-files` part as shown below:

```
rule-files:
- suricata.rules
- /path/to/derivedsuricatarulesTLSanomalies.rules
```

Afterwards, we started Suricata with the command:

```
$ sudo /usr/local/bin/suricata -c \
/usr/local/etc/suricata/suricata.yaml -i $INTERFACE$
```

Instead of the `INTERFACE` parameter, it can be indicated as `any` (to analyze all traffic), or a specific interface, such as `eth0`, or `docker0`, to analyze traffic coming from a TLS-Anvil Docker container.

To read the Suricata logs in real time and check its operation, in another terminal the following command is used:

```
$ tail -f /usr/local/var/log/suricata/fast.log
```

The path for the logs is found in `suricata.yaml`, under `default-log-dir`. Finally, the OpenSSL test server and the TLS-Anvil Docker test container are started, as explained above, to verify whether the Suricata rules are triggered correctly.

Figure 13 illustrates a segment of the file named `fast.log`, which comprises a default Suricata rule alongside a derived rule addressing a TLS 1.2 anomaly. This anomaly pertains to a cipher suite that includes TLS_FALLBACK_SCSV within the cipher suites listed in the Client Hello handshake message, specifically corresponding to the TLS-Anvil test documented in the file `SCSV.java`. It is noteworthy that the log captures the date and time when the rule was activated, as well as the signature ID, accompanying message, classification, priority level, and both the sender and recipient of the message that triggered the event.

```
06/16/2025-06:55:39.096462  [**] [1:2230010:1] SURICATA TLS invalid record/traffic [**] [Classifica
tion: Generic Protocol Command Decode] [Priority: 3] {TCP} 172.17.0.2:50016 → 10.0.2.15:4433
06/16/2025-06:55:39.096462  [**] [1:1000001:0] RFC 7507: TLS 1.0 Client Hello with Cipher Suite TLS
_FALLBACK_SCSV [**] [Classification: (null)] [Priority: 3] {TCP} 172.17.0.2:50016 → 10.0.2.15:4433
06/16/2025-06:55:39.483699  [**] [1:1000001:0] RFC 7507: TLS 1.0 Client Hello with Cipher Suite TLS
_FALLBACK_SCSV [**] [Classification: (null)] [Priority: 3] {TCP} 172.17.0.2:50022 → 10.0.2.15:4433
```

Figure 13. Example of Suricata log indicating the test for Cipher Suite TLS with Fallback SCSV.

6. Conclusions and Future Work

The aim of this study was to conduct a comprehensive review of the vulnerabilities that have affected various TLS implementations and to evaluate the feasibility of identifying flawed TLS-aware nodes through the analysis of intercepted network traffic. This capability enables network administrators and security managers to receive notifications regarding TLS software flaws or anomalies present in end nodes that may lead to performance deterioration, compromise system integrity, or result in communication errors. Our work exploits TLS-Anvil, a famous tool that can be employed to conduct TLS compliance evaluations against target systems. In particular, this tool executes tests that enable to identify TLS implementations that deviate from the protocol standard specified in several different RFCs.

Using the information collected in a set of selected TLS-Anvil tests, we created rules for the Suricata TLS inspection engine, modeled to recognize patterns for subtle software anomalies in the intercepted TLS messages. These rules were designed to send alarm messages in the presence of an anomaly related to an RFC standard. They are therefore intended as an add-on support for detecting anomalous TLS connections with a Suricata IDS. The primary objective was achieved because the rules proved effective in detecting irregularities introduced by the testing tool, reporting alerts promptly.

One potential advancement of this work includes the definition of rules for all 400 tests performed with TLS-Anvil for the various versions of TLS libraries. Once defined, such rules can be integrated into a more complex intrusion detection and prevention system to identify possible violations in TLS-aware nodes or systems. Furthermore, comprehensive testing may be conducted to assess the scalability of the enhanced Suricata IDS through real-time evaluations, which would encompass the rates of false positives and false negatives. Thus, it should be feasible to outline strategies aimed at addressing false alarms, and, in the event of legitimate alarms, to conduct further investigations into the underlying causes and any possible malicious intent related to the connection.

Additionally, strategies can be explored to optimize the Suricata code to further reduce the number of computational operations performed. This can be particularly advantageous for systems with low computational capacity, such as IoT devices. Finally,

a methodology for the automatic generation of Suricata rules warrants further exploration. It is well established that signature-based IDS requires continuous updates in response to the occurrence of new anomalies and attack signatures, as well as updates to RFCs resulting from changes in protocols.

Author Contributions: Conceptualization, D.G.B.; methodology, D.G.B. and M.D.S.; software, D.G.B. and M.D.S.; data curation, Berbecaru, D.G.B. and M.D.S.; writing—original draft preparation, D.G.B.; writing—review and editing, D.G.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union—NextGenerationEU.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

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

References

1. Wazan, A.S.; Laborde, R.; Chadwick, D.W.; Barrere, F.; Benzekri, A. Which web browsers process ssl certificates in a standardized way? In *IFIP International Information Security Conference*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 432–442.
2. Wazan, A.S.; Laborde, R.; Chadwick, D.W.; Barrere, F.; Benzekri, A. Tls connection validation by web browsers: Why do web browsers still not agree? In Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Turin, Italy, 4–8 July 2017; Volume 1, pp. 665–674.
3. Durumeric, Z.; Kasten, J.; Bailey, M.; Halderman, J.A. Analysis of the HTTPS certificate ecosystem. In Proceedings of the 2013 Conference on Internet Measurement Conference (IMC'13), Barcelona, Spain, 23–25 October 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 291–304. [CrossRef]
4. Kim, D.; Cho, H.; Kwon, Y.; Doupé, A.; Son, S.; Ahn, G.-J.; Dumitras, T. Security analysis on practices of certificate authorities in the HTTPS phishing ecosystem. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, Virtual, 7–11 June 2021; pp. 407–420.
5. CVE-2016-0800 Detail. Available online: https://nvd.nist.gov/vuln/detail/CVE-2016-0800 (accessed on 27 November 2025).
6. Dierks, T.; Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246. 2008. Available online: https://datatracker.ietf.org/doc/html/rfc5246 (accessed on 27 November 2025).
7. Rescorla, E. The Transport Layer Security protocol Version 1.3. August 2018, RFC 8446. Available online: https://www.rfc-editor.org/rfc/rfc8446 (accessed on 27 November 2025).
8. Möller, B.; Duong, T.; Kotowicz, K. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Google, September 2014. Available online: https://openssl-library.org/files/ssl-poodle.pdf (accessed on 27 November 2025).
9. Duong, T.; Rizzo, J. Here Come The ⊕ Ninja. 13 May 2011 . Available online: https://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf (accessed on 27 November 2025).
10. Rizzo, J.; Duong, T. Crime (Compression Ratio Info-Leak Made Easy). Available online: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit (accessed on 27 November 2025).
11. Heartbleed Bug. Available online: https://owasp.org/www-community/vulnerabilities/Heartbleed_Bug (accessed on 27 November 2025).
12. Durumeric, Z.; Li, F.; Kasten, J.; Amann, J.; Beekman, J.; Payer, M.; Weaver, N.; Adrian, D.; Paxson, V.; Bailey, M.; Halderman, J.A. The Matter of Heartbleed. In Proceedings of the 2014 Conference on Internet Measurement Conference (IMC'14), Vancouver, BC, Canada, 5–7 November 2014; Association for Computing Machinery: New York, NY, USA, 2014; pp. 475–488. [CrossRef]
13. Holz, R.; Hiller, J.; Amann, J.; Razaghpanah, A.; Jost, T.; Vallina-Rodriguez, N.; Hohlfeld, O. Tracking the deployment of TLS 1.3 on the web: A story of experimentation and centralization. In *ACM SIGCOMM Computer Communication Review*; Association for Computing Machinery: New York, NY, USA, 2020; Volume 50, pp. 3–15. [CrossRef]
14. Vanhoef, M.; Piessens, F. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15), Washington, DC, USA, 12–14 August 2015; pp. 97–112. Available online: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-vanhoef.pdf (accessed on 27 November 2025).
15. Durumeric, Z.; Ma, Z.; Springall, D.; Barnes, R.; Sullivan, N.; Bursztein, E.; Bailey, M.; Halderman, J.A.; Paxson, V. The Security Impact of HTTPS Interception. In Proceedings of the NDSS'17, San Diego, CA, USA, 26 February–1 March 2017. [CrossRef]

16. Soghoian, C.; Stamm, S. Certified lies: Detecting and defeating government interception attacks against SSL (short paper). In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security (FC'11), Gros Islet, St. Lucia, 28 February–4 March 2011*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 250–259. [CrossRef]
17. Samarasinghe, N.; Mannan, M. Short paper: TLS ecosystems in networked devices vs. web servers. In *International Conference on Financial Cryptography and Data Security (FC 2017)*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2017; Volume 10322, pp. 533–541. [CrossRef]
18. Samarasinghe, N.; Mannan, M. *Another Look at TLS Ecosystems in Networked Devices vs. Web Servers, Computers & Security*; Elsevier: Amsterdam, The Netherlands, 2019; Volume 80, pp. 1–13. ISSN 0167–4048. [CrossRef]
19. Suricata. Available online: https://suricata.io/ (accessed on 27 November 2025).
20. Zeek. Available online: https://zeek.org/ (accessed on 16 January 2025).
21. Qualys SSL Labs—SSL Server Test. Available online: https://www.ssllabs.com/ssltest/ (accessed on 27 November 2025).
22. Berbecaru, D.G.; Petraglia, G. TLS-Monitor: A Monitor for TLS Attacks. In Proceedings of the 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2023; pp. 1–6. [CrossRef]
23. Berbecaru, D.G.; Lioy, A. Threat-TLS: A Tool for Threat Identification in Weak, Malicious, or Suspicious TLS Connections. In Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES'24), Vienna, Austria, 30 July–2 August 2024; Association for Computing Machinery: New York, NY, USA ; pp. 1–9. [CrossRef]
24. Maehren, M.; Nieting, P.; Hebrok, S.; Merget, R.; Somorovsky, J.; Schwenk, J. TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries. In Proceedings of the 31th Usenix Security Symposium, Boston, MA, USA, 10–12 August 2022; pp. 215–232. Available online: https://www.usenix.org/system/files/sec22-maehren.pdf (accessed on 27 November 2025).
25. Coker, G.; Guttman, J.; Loscocco, P.; Herzog, A.; Millen, J.; O'Hanlon, B.; Ramsdell, J.; Segall, A.; Sheehy, J.; Sniffen, B. Principles of remote attestation. *Int. J. Inf. Secur.* **2011**, *10*, 63–81. [CrossRef]
26. de Carné de Carnavalet, X.; van Oorschot, P.C. A Survey and Analysis of TLS Interception Mechanisms and Motivations: Exploring how end-to-end TLS is made "end-to-me" for web traffic. *ACM Comput. Surv.* **2023**, *55*, 269. [CrossRef]
27. Berbecaru, D.; Lioy, A. On the Robustness of Applications Based on the SSL and TLS Security Protocols. In *Public Key Infrastructure. EuroPKI 2007*; Lecture Notes in Computer Science; Lopez, J., Samarati, P., Ferrer, J.L., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4582. [CrossRef]
28. CVE-2014-0160. Available online: https://nvd.nist.gov/vuln/detail/CVE-2014-0160 (accessed on 27 November 2025).
29. CVE-2013-0169: Understanding the "Lucky Thirteen" Attack on TLS and DTLS Protocols. Available online: https://www.cve.news/cve-2013-0169/ (accessed on 27 November 2025).
30. Lucky 13 Vulnerability. Available online: https://brandsek.com/kb/books/ssl-vulnerability/page/lucky-13-vulnerability (accessed on 27 November 2025).
31. CVE-2018-12404. Available online: https://nvd.nist.gov/vuln/detail/CVE-2018-12404 (accessed on 27 November 2025).
32. Meyer, C.; Somorovsky, J.; Weiss, E.; Schwenk, J.; Schinzel, S. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks, In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 733–748. Available online: https://research.utwente.nl/files/24317210/revisiting.pdf (accessed on 27 November 2025).
33. CVE-2022-20940 Detail. Available online: https://nvd.nist.gov/vuln/detail/CVE-2022-20940 (accessed on 27 November 2025).
34. Böck, H.; Somorovsky, J.; Young, C. Return of Bleichenbacher's Oracle Threat (ROBOT). In Proceedings of the 27th Usenix Security Symposium, Baltimore, MD, USA, 15-17 August 2018. Available online: https://www.usenix.org/conference/usenixsecurity18/presentation/bock (accessed on 27 November 2025).
35. What Is the FREAK Vulnerability? How to Prevent SSL FREAK Attacks. Available online: https://certpanel.com/resources/what-is-the-freak-vulnerability-how-to-prevent-ssl-freak-attacks/ (accessed on 27 November 2025).
36. testssl – Testing TLS/SSL encryption. Available online:https://testssl.sh/ (accessed on 27 November 2025).
37. TLSAssistant. Available online: https://st.fbk.eu/tools/TLSAssistant/ (accessed on 27 November 2025).
38. Manfredi, S; Ranise, S.; Sciarretta, G.; Tomasi, A. TLSAssistant Goes FINSEC A Security Platform Integration Extending Threat Intelligence Language. In *Proceedings of the Cyber-Physical Security for Critical Infrastructures Protection, Guildford, UK, 18 September 2020*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 16–30. [CrossRef]
39. Manfredi, S.; Ranise, S.; Sciarretta, G. Lost in TLS? no more! assisted deployment of secure TLS configurations. In Proceedings of the 33th IFIP Annual Conference on Data and Applications Security and Privacy (DBSec), Charleston, SC, USA, 15–17 July 2019; pp. 201–220.
40. Metasploit. Available online: https://www.metasploit.com/ (accessed on 27 November 2025).
41. Beurdouche, B.; Delignat-Lavaud, A.; Kobeissi, N.; Pironti, A.; Bhargavan, K. FLEXTLS: A Tool for Testing TLS Implementations. In Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT), Washington, DC, USA, 10–11 August 2015. Available online: https://www.usenix.org/conference/woot15/workshop-program/presentation/beurdouche (accessed on 13 January 2026).

42. SSLyze—Fast and Powerful SSL/TLS Scanning Library. Available online: https://github.com/nabla-c0d3/sslyze (accessed on 27 November 2025).
43. Nmap Security Scanner. Available online: https://nmap.org/ (accessed on 13 January 2026).
44. TLS-Attacker. Available online: https://deepwiki.com/tls-attacker/TLS-Attacker (accessed on 27 November 2025).
45. Hunting Anomalous Connections and Infrastructure with TLS Certificates, 20 January 2022. Available online: https://www.databricks.com/blog/2022/01/20/hunting-anomalous-connections-and-infrastructure-with-tls-certificates.html (accessed on 27 November 2025).
46. Simos, D.E.; Bozic, J.; Garn, B.; Leithner, M.; Duan, F; Kleine, K.; Lei, Y.; Wotawa, F. Testing TLS using planning-based combinatorial methods and execution framework. *Softw. Qual. J.* **2019**, *27*, 703–729. [CrossRef]
47. Simos, D.E.; Bozic, J.; Duan, F.; Garn, B.; Kleine, K.; Lei, Y.; Wotawa, F. Testing TLS Using Combinatorial Methods and Execution Framework. In *Testing Software and Systems*; ICTSS 2017; Lecture Notes in Computer Science; Yevtushenko, N., Cavalli, A., Yenigün, H., Eds.; Springer: Cham, Switzerland, 2017; Volume 10533. [CrossRef]
48. de Ruiter, J.; Poll, E. Protocol State Fuzzing of TLS Implementations. In Proceedings of the 24th USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015. Available online: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter (accessed on 27 November 2025).
49. Suricata User Guide Release 9.0.0-dev, Jan 17, 2026. Available online: https://app.readthedocs.org/projects/suricata/downloads/pdf/latest/ (accessed on 13 January 2026).
50. Suricata Rules. Available online: https://docs.suricata.io/en/latest/rules/tls-keywords.html (accessed on 27 November 2025).
51. Zhou, J.; Fu, W.; Hu, W.; Sun, Z.; He, T.; Zhang, Z. Challenges and Advances in Analyzing TLS 1.3-Encrypted Traffic: A Comprehensive Survey. *Electronics* **2024**, *13*, 4000. [CrossRef]
52. Mathews, N.; Holland, J.K.; Oh, S.E.; Rahman, M.S.; Hopper, N.; Wright, M. SoK: A critical evaluation of efficient website fingerprinting defenses. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–25 May 2023; pp. 969–986.
53. TLS-Anvil GitHub Repository. Available online: https://github.com/tls-attacker/TLS-Anvil/ (accessed on 27 November 2025).
54. TLS-Anvil Test Files. Available online: https://github.com/tls-attacker/TLS-Anvil/tree/main/TLS-Testsuite/ (accessed on 27 November 2025).
55. TLS-Docker-Library GitHub Repository. Available online: https://github.com/tls-attacker/TLS-Docker-Library (accessed on 27 November 2025).

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.