Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet

Nurullah Erinola and Marcel Maehren, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity23/presentation/erinola

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet

Nurullah Erinola[1], Marcel Maehren[1], Robert Merget[2], Juraj Somorovsky[3], and Jörg Schwenk[1]

[1]Ruhr University Bochum
[2]Technology Innovation Institute
[3]Paderborn University

Abstract

DTLS aims to bring the same security guarantees as TLS to UDP. It is used for latency-sensitive applications such as VPN, VoIP, video conferencing, and online gaming that can suffer from the overhead of a reliable transport protocol like TCP. While researchers and developers invested significant effort in improving the security of TLS, DTLS implementations have not received the same scrutiny despite their importance and similarity. It is thus an open question whether vulnerabilities discovered in TLS have been fixed in DTLS and whether DTLS-specific features open possibilities for new attacks.

To fill this gap, we extended the open-source tool TLS-Scanner with support for DTLS and implemented additional tests for DTLS-exclusive features. We evaluated twelve open-source DTLS server implementations and uncovered eleven security vulnerabilities, including a padding oracle vulnerability in PionDTLS and DoS amplification vulnerabilities in wolfSSL, Scandium, and JSSE. We then proceeded to scan publicly available servers. We discovered and analyzed more than 500,000 DTLS servers across eight ports providing detailed insights into the publicly accessible DTLS server landscape. Beyond cryptographic vulnerabilities and compatibility issues, our analysis showed that 4.4% of the evaluated servers could be used for DoS amplification attacks due to insufficient care when handling anti-DoS cookies.

1 Introduction

Transport Layer Security (TLS) [43] cannot be used to secure UDP connections since TLS requires reliable data transport. This led to the development of the Datagram Transport Layer Security (DTLS) [47, 39] protocol, which is based on TLS.

The security of TLS implementations has become a major concern for researchers, leading to critical attacks [38, 14, 9, 4, 10] and changes in the protocol to mitigate them. Since DTLS is based on TLS, problems that affect TLS typically also affect DTLS. On the other hand, it is unclear to what extent DTLS implementations have vulnerabilities that are unique to DTLS, as DTLS supports unique features like message retransmissions and message fragmentation. An example of such an issue was discovered by AlFardan and Paterson, who presented a padding oracle attack using DTLS-specific features [41]. This motivates our first research question:

> **RQ1:** What is the current state of DTLS server implementations? Are they vulnerable to known attacks? Which features do they implement?

To answer this question, we considered known attacks on TLS and analyzed DTLS RFCs to identify DTLS-specific features and possible vulnerabilities. We then practically evaluated these properties by adapting the open-source tools TLS-Attacker [1, 54] and TLS-Scanner [2] to analyze DTLS implementations. We extended the basic DTLS functionality in TLS-Attacker [24] and added the ability to modify fragments of handshake messages, proper handling of received retransmissions, and the dynamic sending of retransmissions. We then adapted the rich feature set of TLS-Scanner for DTLS and added additional tests for the identified DTLS-specific features.

Lab Evaluation. Our evaluation of twelve open-source libraries identified differences in the DTLS server implementations regarding our DTLS-specific tests, especially with anti-DoS cookies (Table 1). Additionally, we were able to identify security vulnerabilities, functional bugs, non-conformance issues, and unsupported mandatory DTLS-specific features:

- **Cryptographic Vulnerabilities.** We discovered a padding oracle vulnerability in PionDTLS that allows an attacker to decrypt DTLS traffic (Section 5.5) and a plaintext injection vulnerability in TinyDTLS[C] that allows an attacker to circumvent DTLS integrity protection (Section 5.4).
- **DoS Amplification.** We identified that Scandium (CVE-2022-2576), JSSE (CVE-2023-21835), and wolfSSL (CVE-2022-34293) can be tricked into executing a DTLS handshake without a cookie exchange by abusing the session resumption feature. It is possible to force the servers to fall back to a full DTLS handshake without cookie exchange, enabling DoS amplification attacks (Section 5.1).

- **DoS Vulnerabilities.** We found that MatrixSSL retransmits *HelloVerifyRequest* messages and keeps per-client state, making it susceptible to memory exhaustion attacks (Section 5.1). In addition, we found memory exhaustion attacks against Botan, JSSE, MatrixSSL, and PionDTLS with fragmented *ClientHello* messages (Section 5.3).
- **Buffer Over-Read.** We observed that MatrixSSL crashes after receiving an unencrypted *Finished* message or the *ChangeCipherSpec* and *Finished* message in the wrong order (Section 5.4).
- **Interoperability.** We discovered that TinyDTLS^C neither supports retransmissions nor fragmentation . In addition, Botan also does not support retransmissions. The lacking support influences their stability and interoperability with other implementations (Section 5.2 and Section 5.3).

Internet Scan. Typically, implementations on the Internet differ from those of standard libraries [14], as also private and proprietary implementations are used. A technique to analyze their server implementations is to perform large-scale Internet scans. For TLS, researchers were able to map the TLS ecosystem [33], quantify the impact of security vulnerabilities [38, 14, 9, 4, 10, 21, 11, 58, 57, 19, 30, 36], or even find new ones. Similar studies in the area of DTLS are missing, which leads to our second research question:

> **RQ2:** Where is DTLS deployed on the publicly accessible Internet and what is the current state of the DTLS server ecosystem? What vulnerabilities do servers suffer from?

For DTLS, it is unknown on which ports it is mostly deployed. Different application layer protocols specify ports but it is unclear to which extent they are really used in practice. To answer this question without significant impact caused by full IPv4 scans, we used ZMap [22] in a three-step approach. In the first step, we did a pre-scan of 2^{17} randomly sampled IPv4 addresses for each port. In the second step, we increased our sample size to 2^{20} for each port where we discovered at least one host. In step three, we chose the eight ports with the highest number of responses (Table 2) to scan the whole IPv4 range.

The results of our Internet evaluation is summarized in Table 4. Additionally, we collected supported cipher suites (Table 3), (D)TLS extensions (Table 8), and elliptic curves (Table 7). We also analyzed deployment-specific issues, like certificate handling (Table 6) and resistance to known TLS attacks [42, 11, 61, 13, 32, 10, 4, 15, 48, 37] (Table 5). We discovered 22,797 servers from Zscaler that can be used as amplifiers for DoS attacks (Section 6.5), self-signed certificates on nearly all 44,189 DTLS servers on port 1106 (Section 6.4), 472 servers with a directly observable padding oracle vulnerability and 28 servers vulnerable to Bleichenbacher attacks (Section 6.6).

Contributions. Our main contributions are as follows:

- We analyzed DTLS-specific features and created a catalog of threats and security pitfalls for DTLS implementations (Section 3).
- We conducted lab-based security evaluations on twelve open-source server implementations and tested for protocol features, known attacks, and our DTLS catalog. We identified eleven security vulnerabilities: a padding oracle, a plaintext insertion vulnerability, five memory exhaustion DoS attacks, three DoS amplification attacks, and a buffer over-read (Section 5).
- We conducted an Internet scan with an adapted scanning methodology and published the first comprehensive dataset based on 520,849 hosts which account for an estimated of 0.66% of all publicly available DTLS IPv4 hosts across all ports. Analysis of this dataset yields detailed and valuable insights into the current deployment of DTLS. Beyond cryptographic vulnerabilities and compatibility issues, we discovered that 4.4% of the evaluated servers could be used for DoS amplification attacks with an amplification factor of up to 33 (Section 6).

Ethical Considerations. Our scans respected the rules for Internet-wide scanning proposed by Durumeric et al. [22]. We shuffled the connections made to the servers during our scans to spread the computational load for each individual server. Finally, we established rDNS entries and a website that indicated the benign nature of our scans and offered the possibility of opting out of our study.

Responsible Disclosure. We have reported our findings from the local security evaluations to the respective projects in compliance with their security procedures. In addition, we have informed the vendors of the identified devices (AnchorFree and Zscaler) about our observed issues.

2 Background

The *Datagram Transport Layer Security* (DTLS) protocol is a variant of the TLS protocol for UDP. Version 1.0 was introduced in 2006 [44, 39] and is based on TLS 1.1 [17]. DTLS 1.1 was skipped to align TLS and DTLS versions. In 2012, DTLS 1.2 [45] was published based on TLS 1.2 [18]. The most recent version (from April 2022) is DTLS 1.3 [47], based on TLS 1.3 [43]; since many libraries do not support it yet, our work focuses on DTLS 1.0 and 1.2.

As with TLS, the main components of DTLS are the *DTLS handshake* and the *DTLS record layer*. The *handshake* negotiates cryptographic algorithms and keys while the *record layer* wraps the data from the upper layers into encrypted records sent over UDP.

Handshake. As shown in Figure 1, the client starts the DTLS handshake by sending a *ClientHello* message to the server that contains the highest supported protocol version and a list of supported cipher suites. A *cipher suite* defines a set of cryptographic parameters used to achieve confidentiality,

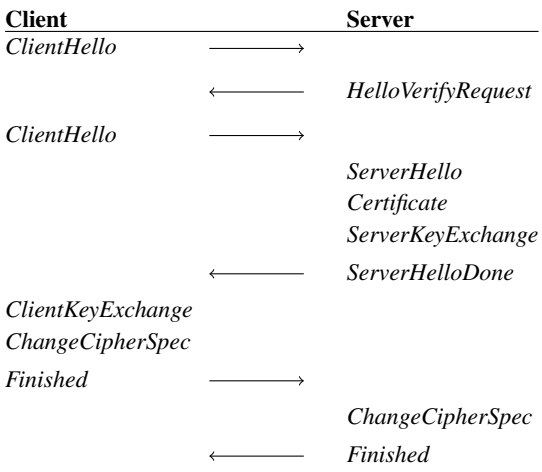

Figure 1: DTLS handshake with cookie exchange and Diffie-Hellman key exchange.

integrity, and authenticity for the DTLS channel. To mitigate DoS attacks, the server replies to the *ClientHello* with a *HelloVerifyRequest* that contains an anti-DoS *cookie*. The client then retransmits its *ClientHello* but includes the cookie value. Upon receiving the updated *ClientHello*, the server selects a cipher suite and announces it in the *ServerHello* message. The server proceeds by sending a *Certificate* message that contains the server's X.509 certificate. The server then sends a signed *ServerKeyExchange* message containing a Diffie-Hellman key share followed by a *ServerHelloDone* message to indicate the end of the flight. The client completes the key exchange with the *ClientKeyExchange* message which contains the client's Diffie-Hellman share. Both parties may now compute the *Premaster Secret*, then derive the *Master Secret*, and finally the symmetric keys. The client informs the server that subsequent messages will be encrypted by sending a *ChangeCipherSpec* message followed by a *Finished* message, which is a cryptographic checksum across all exchanged messages. The server likewise responds with a *ChangeCipherSpec* and *Finished*, which concludes the handshake.

Session Resumption. With session resumption, a *Master Secret* negotiated in a previous handshake can be used to omit public key operations. In essence, both client and server must agree on the *Master Secret* by referencing it via a *session ID* in the *ClientHello* and *ServerHello* messages, or by using *session tickets* [51].

Renegotiation. With the renegotiation feature, it is possible to negotiate fresh keys and new parameters for an existing connection by performing an additional handshake. Either party can request the renegotiation handshake once the initial handshake is completed. Up to the new *ChangeCipherSpec* message, all handshake messages must be encrypted using the previously established keys.

Extensions. Extensions can be used to negotiate additional (D)TLS features. Typically, a client proposes a set of extensions in the *ClientHello* and the server can confirm them by including selected extensions in the *ServerHello*. The *Application-Layer Protocol Negotiation* (ALPN) extension [26] is an example of such an extension. It contains a list of application protocols that the client is willing to use once the (D)TLS channel is established. The server chooses a supported protocol from the client's list and includes it in the ALPN extension in the *ServerHello* message.

Denial-of-Service. The use of an unreliable transport protocol makes DTLS susceptible to DoS attacks based on IP spoofing. Before the server should create a local per-client state or answer with big messages, it should first validate that the client can actually receive messages from the server. This validation is done with *stateless* anti-DoS cookies, which are transmitted in the *ClientHello*. RFC 6347 [45] gives vague recommendations on how the server should compute the cookie:

$$\text{Cookie} = \text{HMAC}(\text{Secret}, \text{Client-IP}, \text{Client-Parameters})$$

where *Secret* is a key only known to the server and the *Client-Parameters* are composed of the *version*, *random*, *session ID*, *cipher suites*, and *compression methods* of the *ClientHello*. This construction allows servers to validate cookies without retaining any per-client state. By replaying the cookie issued by the server, the client shows that it can receive messages under its IP address.

Packet Reordering. UDP datagrams can get delivered out of order, so DTLS uses *explicit* sequence numbers (SQN) in the record header. An *epoch* number in the record header is used to keep track of which set of cryptographic parameters must be used to decrypt a given record. The epoch value starts at 0 and is incremented after each *ChangeCipherSpec*. The SQN is reset with each *ChangeCipherSpec* and gets incremented by one with each record sent.

Packet Loss. For handshake messages, a simple retransmission mechanism solves the problem of lost or corrupted UDP datagrams: After sending a handshake message, the sender starts a timer and waits for the expected response from the peer. If the timer reaches a predefined timeout value, the sender retransmits the previous flight. When the peer sees an already received handshake message, it ignores the message and likewise retransmits its last messages.

Handshake Message Fragmentation. Handshake messages can be up to $2^{24} - 1$ bytes long. In TLS, TCP keeps track of fragments and reassembles the fragmented handshake messages. In DTLS, handshake messages that are longer than the maximum transmission unit (MTU, typically 1500 bytes) must be reassembled by DTLS itself. To identify (retransmitted) fragments and their position with the handshake message, each DTLS message contains a *message sequence*, *fragment offset*, and *fragment length* field.

MAC Errors. In contrast to TLS, the DTLS specifications allows for a more relaxed handling of MAC errors. In TLS, any invalid MAC must result in a connection termination, while in DTLS, peers may keep the connection open. This behavior can enable more efficient oracle attacks than in TLS, which was explored by AlFardan and Paterson [41].

2.1 TLS-Scanner

TLS-Scanner [2] is an open-source tool to assist pentesters and security researchers in evaluating TLS server implementations. It builds upon TLS-Attacker [54, 1], a well-established framework for a systematic analysis of TLS implementations. TLS-Attacker allows its users to generate arbitrary protocol flows and modify the structure of the included protocol messages. TLS-Scanner uses the flexibility of TLS-Attacker to automatically scan a TLS server and provides a report of supported features like protocol versions, cipher suites, extensions, and potential security issues. To perform large-scale scans with TLS-Scanner, TLS-Crawler [38] can be used. TLS-Crawler is a framework that utilizes multiple TLS-Scanner instances to scan a large number of servers in parallel and write the results to a database. In addition, it allows for distributing the scan tasks across multiple machines.

3 Analysis of DTLS Implementation Pitfalls

DTLS introduces new features to support stateless and unreliable transport. We carefully analyzed the specifications [44, 45] and previously known vulnerabilities to identify implementation pitfalls specific to DTLS.

3.1 Stateless Cookie Exchange

The cookie exchange is *optional*. If it is missing, the DTLS server can be abused for amplification attacks or can be targeted by DoS attacks via memory exhaustion themselves. The DTLS specification does not specify a fixed length for anti-DoS cookies – their length may range from 0 to 128 bytes. In addition, the RFCs only make a recommendation for the generation of the cookie but do not mandate strict compliance. If a server deviates from this recommendation, it may be possible for the attacker to predict a cookie and, therefore, skip the cookie exchange.

Even if servers implement the cookie exchange, they may be tempted to skip it when resuming an existing session. If an implementation consciously decides to skip the cookie exchange on session resumption, it has to be careful: An attacker may send *ClientHello* messages that look like they are resuming a session but intentionally violate RFC 5246 [18]. An implementation that simply checks if *some* session ID is present to determine if a cookie exchange is necessary may later find out that it can not resume the session. Thus by sending an *invalid* session ID, a server may be tricked into skipping the cookie exchange while falling back to a full handshake. This may result in a DoS amplification vulnerability, where an attacker sends a small *ClientHello* using a spoofed IP address of a victim to provoke a flight of large messages from the server returned to this address. However, this issue could also be used in a DoS attack that aims to exhaust the memory of the server itself, as it is forced to create a per-client local state.

Issued cookies have to be *stateless*. If issued cookies are *stateful* and have to be locally stored, the server again exposes a potential DoS vulnerability as the server risks getting flooded with requests from various spoofed IP addresses.

3.2 Renegotiation

Another interesting case in DTLS is renegotiation. Neither DTLS RFC ever mentions the renegotiation feature by name, but it is still implied that the feature is supposed to be present for DTLS via the *epoch* concept. This leaves it ambiguous whether peers should perform a cookie exchange when renegotiating. Interestingly, the renegotiation feature is mentioned in [39], but not with enough detail to settle the issue. This can result in severe interoperability issues if clients disagree with the server and enforce their interpretation of the specification during renegotiation.

3.3 Timeout and Retransmissions

For a DTLS implementation, it is mandatory to support retransmissions, as without it, interoperability is not guaranteed and handshaking might fail in the event of packet loss. However, the server must not retransmit the *HelloVerifyRequest*. Otherwise, the server would need to hold per-client state and thus would risk getting flooded with *ClientHello* messages.

3.4 Handshake Message Fragmentation

Each DTLS implementation must support the fragmentation of its handshake messages and also the processing of received handshake message fragments. Otherwise, the implementation is not entirely functional and can fail during benign use. Typically, a DTLS implementation should support the fragmentation of each handshake message; in that regard, the DTLS specification does not prohibit fragmentation for any of the handshake messages. However, that potentially creates a DoS vulnerability via memory exhaustion if the server allows fragmentation of *initial ClientHello* messages. If a server starts buffering the fragment before the cookie was validated, it is holding per-client state too early, which leads to a possible DoS vulnerability. An attacker can consume excessive resources on the server by sending a series of *ClientHello* fragments, causing the server to allocate state. To prevent this issue, developers must ensure that the fragmentation feature does not introduce DoS vulnerabilities.

3.5 Concurrent Cipher States

RFC 6347 [45] makes the following statement about renegotiation:

> Note that because DTLS records may be reordered, a record from epoch 1 may be received after epoch

> 2 has begun. In general, implementations SHOULD discard packets from earlier epochs, but if packet loss causes noticeable problems they MAY choose to retain keying material from previous epochs [...] to allow for packet reordering.

This is somewhat vague, and misunderstandings may lead to vulnerabilities in the implementation of DTLS; accepting records with a *previous* epoch number after completing the handshake can be dangerous. For example, an attacker could inject clear application data with epoch 0 after a successful DTLS handshake, which would have a similar impact as the Renegotiation attack [42]. Generally, a server should always reject application data from DTLS records with epoch 0.

4 Methodology

The goal of our research is to assess the state of DTLS implementations and to publish the first comprehensive dataset on the DTLS ecosystem on the publicly accessible Internet. To accomplish this, we first extend the tools TLS-Attacker and TLS-Scanner to evaluate DTLS implementations automatically. We then evaluate open-source DTLS implementations and perform the large-scale scan on the DTLS ecosystem. In our evaluation, we focus only on DTLS server implementations. We do not consider co-located attackers, access to the library's internals, or timing side channels. Additionally, we excluded issues related to client authentication or pre-shared keys (PSK) from our large-scale study.

Extending TLS-Attacker. Although generic DTLS support was already introduced to TLS-Attacker by Brostean et al. [24], the implementation was not designed to work reliably outside a lab environment. TLS-Attacker lacked proper support for the receiving of retransmissions, dynamic sending of retransmissions itself, and the ability to modify fragments of handshake messages. These features are crucial for our tests and large-scale scans that evaluate real servers deployed in different environments and had to be implemented by us.

Extending TLS-Scanner. We integrated the new TLS-Attacker version in TLS-Scanner and adjusted existing testing strategies to fit DTLS. We describe the adapted tests in Section 4.1. To test for the DTLS-specific problems identified in Section 3, we implemented new tests in TLS-Scanner and introduce them in Section 4.2. For retransmissions, we used the following configuration: Whenever TLS-Scanner receives an unexpected response (or no response at all), it performs a retransmission of the last flight up to three times to ensure that the response it got from the server is indeed correct.

4.1 Tested (D)TLS Properties

Supported Features. We collect the supported protocol versions, cipher suites, compression algorithms, and elliptic curves to determine if obsolete and insecure algorithms are still in use. To do so, we send *ClientHello* messages that first offer an extensive list of features, i.e., all specified cipher suites. Subsequently, we remove entries that the server negotiates until the server does not find suitable parameter choices anymore. At the time of writing, around 60 officially defined extensions exist, some of which are only applicable in specific use cases. In order to analyze extensions supported by DTLS hosts, we chose a subset of common (D)TLS extensions and offered them in our *ClientHello*. We specifically selected extensions that the server must include in the *ServerHello* message upon acting on them. This way, we have definitive proof that the server supports an extension. The tested extensions can be found in Table 8.

Certificate Ecosystem. We probe a server for all certificates by performing handshakes with different cipher suites and key exchange groups. We evaluate how many self-signed or expired certificates are used and the public key and signature types.

Identification of Application Protocols. In an uncontrolled environment, it is typically not clear which application protocol is used by a server. We use the ALPN extension [26] to gather hints of the used application protocol. For ALPN, we consider 40 standardized ALPN values,[1] which we evaluate using the same strategy as for cipher suites and extensions. However, a server misconfiguration may result in a mismatch between the announced protocol and the actual application protocol. Therefore we performed smaller pre-scans for our study and manually identified some of the applications that were running on the scanned ports. We then implemented a set of application layer test vectors for STUN [50], TURN [34], CoAP [53], and VPN from Citrix and Fortinet.

Attacks. We test each server for known protocol vulnerabilities. Some vulnerabilities can be identified solely based on supported protocol features (e.g., Sweet32 [11]), while others require dedicated tests. We evaluate the following attacks: Renegotiation attack [42], CVE-2020-13777 (GnuTLS session ticket bug), CBC padding oracle [61], Bleichenbacher [13], invalid curve [32], Sweet32 [11], Logjam [4], FREAK [10], ALPACA [15], Raccoon [37], and CRIME [48]. Details about how we scan for these vulnerabilities can be found in Appendix A.

4.2 Tested DTLS-Specific Features

We extended TLS-Scanner to perform tests for the DTLS-specific implementation pitfalls introduced in Section 3. Below, we define categories for our catalog of DTLS-specific tests.

Stateless Cookie Exchange (A). We first check whether the server performs the cookie exchange during a new handshake (A1) and determine the cookie length (A8). We then test whether the server skips the cookie exchange in three cases: resumption via session ID (A2), resumption via session

[1] https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids

ticket (A3), and renegotiation (A4). We also check whether a server unnecessarily holds per-client state by waiting three seconds for retransmitted *HelloVerifyRequest* messages (A5). Further, we check whether the server performs the cookie generation as recommended (A6): For each *ClientHello*, we modify a different field that should be included in the computations but keep the cookie static and check if the server is still accepting the cookie. To test whether the *client IP address* and *source port* influence the cookie generation we first request a cookie and then subsequently send a *ClientHello* with the cookie from a different IP address and source port. Finally, to test if cookies are validated at all, we send a *ClientHello* with flipped cookie bits (A7).

Timeout and Retransmissions (B). To test retransmissions, we first start a handshake but do not send a *ClientKeyExchange* after receiving the *ServerHelloDone*. Instead, we wait for three seconds. If we receive the last flight of server messages again, we consider the server to support the sending of unprovoked retransmissions (B1). In a second test, we intentionally retransmit the *ClientHello* with the cookie instead of sending a *ClientKeyExchange*. If the server responds with a retransmission of the *ServerHello* flight, we consider it to support client-requested retransmissions (B2).

Handshake Message Fragmentation (C). To test if the server supports fragmentation, we execute several handshakes in which we fragment the *initial ClientHello* without cookie (C1, C2) and the *ClientKeyExchange* message (C3, C4). Each handshake is tested twice. During the first handshake, the fragments are serialized in a *single* UDP datagram, and in the second handshake, each fragment is serialized in an *individual* UDP datagram. This test approach also allows us to test if the server supports fragmentation of initial *ClientHello* messages in multiple UDP packets. If a server allows this kind of fragmentation, it is potentially vulnerable to DoS attacks since the server starts buffering the fragments and creates per-client state before the cookie is evaluated.

Multiple Cipher States (D). We implemented three tests to test the *epoch* mechanism. In the first test, the *Finished* message, which must be encrypted with epoch 1, is sent unencrypted with epoch 0 (D1). In the second test, the *Finished* is sent encrypted with epoch 1, and then application data is sent unencrypted with epoch 0 (D2). Finally, we start a handshake and intentionally send the encrypted *Finished* before the *ChangeCipherSpec* (D3). We note that we excluded D2 from our large-scale study. In contrast to the lab environment, determining if the unencrypted application data gets processed is challenging during black-box tests. The application protocol deployed on the server may serve as an indicator; however, determining a definitive result requires a more sophisticated implementation of each possible application protocol which we consider out of scope for this study.

5 Evaluation of Software Libraries

In total, we analyzed twelve different implementations. We focused on open-source implementations, as this allowed us to investigate the root causes of the discovered issues. This includes well-known libraries like OpenSSL (v1.1.1m), LibreSSL (v3.4.2), Mbed TLS (v3.0.0), GnuTLS (v3.7.2), wolfSSL (v5.0.0), Botan (v2.19.3), MatrixSSL (v4.3.0), and JSSE (Sun JSSE provider of Java, v17.0.3). Furthermore, we analyzed PionDTLS (v2.1.3), Scandium (v3.0.0), and two variants of TinyDTLS (one from Eclipse (98e2cd7) and one from Contiki-NG (42356d9)), which exclusively implement DTLS and are, therefore, of particular interest. We refer to Eclipse's variant of TinyDTLS as TinyDTLS^E and to Contiki-NG's as TinyDTLS^C. Whenever possible, we used the utilities to configure and launch DTLS servers that are provided by the developers (e.g., for OpenSSL `openssl s_server`). For JSSE, we used the DTLS server from Fiterau-Brostean et al [24][2]. We excluded BoringSSL and NSS from our study as these libraries do not provide working DTLS example server utilities.

We evaluated each implementation in a lab environment with the extended TLS-Scanner. In our evaluation, we enabled the key exchange algorithms RSA, DH, and ECDH. Where possible, we additionally performed our tests for PSK cipher suites and both with and without client authentication. Table 1 summarizes the results regarding our DTLS-specific tests, and the following subsections provide an overview of all results. We observed no differences when using PSK cipher suites or client authentication. Therefore, we do not differentiate between these cases in our results.

5.1 Cookie Exchange

DoS Amplification Vulnerability in Session Resumption (A2, A3, ↯). All tested servers perform a cookie exchange during a new handshake. However, during a session resumption and renegotiation, the individual implementations behave differently. JSSE, Scandium, and wolfSSL do not execute a cookie exchange during a session resumption. A possible reason is that the server's first response in a resumption is small, rendering amplification attacks ineffective. However, we *manually* identified tricks to force all three servers to fall back to a full handshake without a cookie exchange, making practical DoS amplification and memory exhaustion attacks viable again.

Interestingly, a slightly different strategy is needed for every server to trigger a full handshake without a cookie exchange: In wolfSSL (CVE-2022-34293), an attacker only needs to send a *ClientHello* with *any* 32-byte session ID or a *ClientHello* with a *non-empty* session ticket extension. This causes wolfSSL to fall back to a full handshake and skip the cookie exchange, allowing it to be used as an amplifier in a DoS attack. In Scandium (CVE-2022-2576), an attacker

[2] https://github.com/pfg666/jsse-dtls-server

Label	Test	Botan	GnuTLS	JSSE	LibreSSL	MatrixSSL	Mbed TLS	OpenSSL	PionDTLS	Scandium	TinyDTLS^C	TinyDTLS^E	wolfSSL
A1	Issues a cookie during a new handshake	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A2	Issues a cookie during a resumption with session ID	✓	✓	↯	✓	✓	✓	✓	-	↯	-	-	↯
A3	Issues a cookie during a resumption with session ticket	✓	✓	↯	✓	-	-	✓	-	-	-	-	↯
A4	Issues a cookie during a renegotiation	-	✗	-	✓	-	-	✗	-	-	-	✓	-
A5	Performs no *HelloVerifyRequest* retransmissions	✓	✓	✓	✓	↯	✓	✓	✓	✓	✓	✓	✓
A6	Performs recommended cookie computation	✓	✗[b]	✗[a]	✗[b]	✗[c]	✗[d]	✗[b]	✗[b]	✓	✓	✗[c]	✓
A7	Validates the received cookie	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A8	Cookie length	32	16	32	20	16	32	20	20	32	16	16	32
B1	Sends retransmissions without requesting	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
B2	Processes client-requested retransmissions	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
C1	Processes fragmented *ClientHello* in a single datagram correctly	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
C2	Processes fragmented *ClientHello* in cross datagrams correctly	↯	✓	↯	✗	↯	✓	✗	↯	✓	✗	✗	✓
C3	Processes fragmented *ClientKeyExchange* in a single datagram	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
C4	Processes fragmented *ClientKeyExchange* in cross datagrams	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
D1	Rejects unencrypted *Finished*	✗	✓	✓	✓	↯	✓	✓	✓	✓	✓	✓	✓
D2	Rejects unencrypted *Application Data*	✓	✓	✓	✓	✓	✓	✓	✓	✓	↯	✓	✓
D3	Processes reordered *ChangeCipherSpec* and *Finished* correctly	✗	✗	✓	✓	↯	✓	✓	✗	✓	✗	✗	✗

✓ Applies ✗ Does not apply ↯ Does not apply and results in a vulnerability - Resumption/Renegotiation not supported

a: Excludes only the port.
b: Includes only IP address and port.
c: Includes only IP address, port, and the values *version*, *random*, and *session ID* from the *ClientHello*.
d: Includes only IP address.

Table 1: Overview of the introduced DTLS-specific tests regarding (A) anti-DoS cookie handling, (B) retransmissions, (C) fragmentation, and (D) reordering and epochs.

needs to send a *ClientHello* that contains a *valid* session ID combined with an *invalid* cipher suite list missing the cipher suite negotiated in the previous session. The valid session ID in the *ClientHello* message tricks Scandium into skipping the cookie exchange, and the enforced cipher suite mismatch results in the fallback to a full handshake. In JSSE (CVE-2023-21835), an attacker also requires a *valid* session ID or session ticket. However, JSSE always falls back to a full handshake if the session ID or session ticket used for the resumption has been negotiated in a session without the *Extended Master Secret* extension [12].

Possible Interoperability Issue with Renegotiation (A4). OpenSSL and GnuTLS do not perform a cookie exchange upon renegotiation. In contrast, LibreSSL, for example, does issue a cookie. As described in Section 3.2, we do not see a vulnerability in either behavior; however, the cookie exchange does not serve a purpose during a renegotiation and simply slows down the connection. The results indicate that clients should always be prepared to do a cookie exchange to prevent interoperability issues.

DoS Vulnerability via *HelloVerifyRequest* Retransmission (A5, ↯). MatrixSSL responds to an initial *ClientHello* message with multiple *HelloVerifyRequest* messages. While investigating this faulty behavior, we found that MatrixSSL will retransmit the *HelloVerifyRequest* until a timeout (32s) is reached or it receives the second *ClientHello* with the cookie. Within these 32 seconds, the server sent six *HelloVerifyRequest* messages, each with a size of 44 bytes. This means that MatrixSSL creates per-client state after receiving the initial *ClientHello* and holds it until the timer expires. By holding state, the server risks getting flooded, as an attacker could quickly exhaust the server's memory by requesting many cookies. Further, the server can be abused as an amplifier by sending initial *ClientHello* messages with an amplification factor of 3.8. A similar issue was discovered by Fiterau-Brostean et al. in PionDTLS [24].

Disregarded *ClientHello* Parameters (A6). We observed a deviation from the recommended cookie computation in eight of the tested implementations. However, none of the deviations does result in interoperability issues or vulnerabilities since the cookie computation includes the client IP address and therefore remains unpredictable for an attacker with spoofed addresses.

Different Cookie Lengths (A8). In our evaluation, we could identify three different cookie lengths: 16, 20, and 32 bytes. None of these cookie lengths are small enough that an attacker could reasonably guess a cookie.

5.2 Retransmissions

Epoch Confusion. During the scanning process, we observed that MatrixSSL sends faulty retransmissions after completing the handshake. If a client does not send any message after a completed handshake, MatrixSSL retransmits its *ChangeCipherSpec* and *Finished* messages. Upon sending the new *ChangeCipherSpec* message, MatrixSSL increases its epoch, resulting in an invalid value for the retransmitted *Finished*. Since the client cannot process this epoch number, all further communication will fail.

Missing Support for Retransmissions (B). Botan and TinyDTLS^C do not implement a retransmission mechanism.

They can neither process nor send retransmissions. In contrast, TinyDTLS^E only processes received retransmissions correctly but does not send retransmissions itself. This causes connection attempts to fail if a packet gets lost in transmission.

DoS in Example Server. In a handshake with RSA key exchange, the example server of OpenSSL and LibreSSL does not always close the connection correctly. If the server receives a malformed *ClientKeyExchange* followed by a *ChangeCipherSpec*, it is impossible to close the connection with an *Alert* message. Both implementations respond to new handshake attempts from different source ports by retransmitting the *ServerHello*, *Certificate*, and *ServerHelloDone* messages from the aborted connection. The bug prevents both servers from performing further connections until restarted.

5.3 Fragmentation

DoS by Exploiting Fragmentation (C2, ⚡). Botan, JSSE, MatrixSSL, and PionDTLS process fragmented *ClientHello* messages *before* a cookie has been exchanged. After receiving a *ClientHello* fragment, they wait for the remaining fragments of the *ClientHello* message. This behavior forces the server to create per-client state, enabling DoS attacks via memory exhaustion as an attacker can flood the server with *ClientHello* fragments.

How exploitable these vulnerabilities are depends on the size of the generated state and its lifetime. In the case of MatrixSSL, the library restricts the maximum size of the *ClientHello* to 2^{10} bytes and the fragmentation of a message to 16 fragments. These limitations weaken the attack by constraining the size of the state and its lifetime. For JSSE and PionDTLS, the strength of the vulnerability varies based on the server configuration. In the example server of JSSE, the user can define a maximum size for the *ClientHello* message, while in PionDTLS, the user can set a timer to determine the lifetime for the generated state.

We additionally evaluated whether these implementations disable the fragmentation functionality when attacked, but no such countermeasure is implemented.

In OpenSSL and LibreSSL, we observed the same behavior. However, when investigating this behavior, we found that the API allows users to implement *no* cookie exchange, a *stateful* cookie exchange, or a *stateless* cookie exchange. The example server uses the stateful cookie exchange by default, but stateless mode can be requested through command line parameters. Developers using either library should make sure they use the stateless implementation.

Missing Fragmentation Support (C). We discovered that both TinyDTLS variants do not process any fragmented messages and can not fragment their own messages. However, at the time of writing, the developers of TinyDTLS^E have started the development of this feature.

5.4 Message Order and Epochs

Possible Injection of Unencrypted Application Data (D2, ⚡). TinyDTLS^C processes unencrypted application data sent with epoch 0 after completing a handshake. This bug has severe consequences as it allows an attacker to inject arbitrary application data at any point once a handshake between two peers has been completed. An attacker can simply send unencrypted application data with epoch 0 with a spoofed IP to the victim after a benign handshake with two honest peers has been established. The victim will accept this application data as part of the benign connection. In 2020, a similar issue was discovered in PionDTLS [24].

Processing of unencrypted *Finished* messages (D1). Botan accepts unencrypted *Finished* messages delivered with epoch 0 during a handshake. In comparison to D2, this bug does not result in an immediate vulnerability since an attacker is unable to construct a valid *Finished* message.

Crashes on Receiving Unexpected Messages (D1, D3, ⚡). Upon receiving an unencrypted *Finished* message or the *ChangeCipherSpec* and *Finished* messages in the wrong order within a regular handshake, MatrixSSL attempts to retransmit its *initial* handshake messages. While doing so, MatrixSSL wrongfully assumes it is performing a session resumption. Consequently, it sends a new *ServerHello*, followed by an early *ChangeCipherSpec* and *Finished*. It sends the *Finished* message within a record with *epoch 1* but does not encrypt the record's payload. Then, before the client can send any other messages, MatrixSSL crashes with a segmentation fault.

Delayed Completion of the Handshake (D3). We observed that Botan, GnuTLS, PionDTLS, TinyDTLS^E, and wolfSSL completed the handshake delayed when they received the *ChangeCipherSpec* and *Finished* messages out of order. In that event, they do not buffer out of order messages and wait for a retransmission from the client. This behavior can lead to a less efficient connection establishment when packets are received out of order due to the unreliable transport protocol.

Failed Handshakes (D3). In addition, we observed that TinyDTLS^C can not handle receiving the *ChangeCipherSpec* and *Finished* messages in the wrong order. It fails to find the key material to decrypt the *Finished* message and sends an *Alert* to abort the whole connection establishment. This bug is a rediscovery from [24] and is still unfixed.

5.5 Cryptographic Vulnerabilities

Padding Oracle Vulnerability in PionDTLS (⚡). Our evaluation showed that PionDTLS is vulnerable to a *direct* CBC padding oracle attack. The vulnerability results from an *observable* difference in the processing of different padding values [38]. We found that if the plaintext of an encrypted message contains valid padding and the unpadded message is

too short to contain a MAC, the server throws an internal exception and closes the connection. In all other cases, the server correctly processes an invalid message, omits it, and keeps the connection alive as prescribed in the DTLS specification. This behavior is exploitable and can be abused to extract confidential data if CBC cipher suites are negotiated [41, 61].

Insecure Renegotiation in TinyDTLS^C (↯). TinyDTLS^C supports insecure renegotiation and is therefore vulnerable to the Renegotiation attack [42]. This bug is also a rediscovery from [24] and is still unfixed.

5.6 Vendor Feedback

We responsibly disclosed all discovered bugs and vulnerabilities listed in Table 1, except for deviations from the recommended cookie computation (A6) to the respective vendors. The developers of JSSE, Scandium, and wolfSSL promptly addressed and fixed the DoS amplification vulnerabilities (CVE-2022-34293, CVE-2022-2576, CVE-2023-21835) (A2, A3, ↯). However, at the time of writing, among the memory exhaustion vulnerabilities (C2, ↯), only the JSSE implementation has been fixed. For the vulnerabilities in MatrixSSL, the developers acknowledged the receipt of our report and started the investigations. For the two TinyDTLS variants, the developers informed us about their plans to discontinue the TinyDTLS^C variant, indicating that the discovered issues will likely not be fixed and that this implementation must not be evaluated in the future. Regarding D3, the developers of wolfSSL and TinyDTLS^E replied to us that they intentionally do not cache records of *future* epochs and expect the peer to send the *ChangeCipherSpec* and *Finished* messages in one UDP datagram. Furthermore, the developers of Botan mentioned that their implementation is primarily utilized with other transport protocols that emulate reliable packet exchange. As a result, they do not consider missing support for retransmissions (B) a limitation for their customers.

6 Analysis of the DTLS Ecosystem

Using the knowledge gained from evaluating the open-source libraries, we conduct the first large-scale study on the state of the server-side DTLS ecosystem on the Internet from May 2022 to June 2022.

6.1 Host Discovery

In contrast to the TLS ecosystem, the DTLS ecosystem has not been analyzed yet. It is still unknown on which ports DTLS servers are mostly deployed. To make an independent port selection, port scans must be conducted. Since we aim to avoid noise and high load for deployed systems, we applied a scanning methodology that minimizes the necessary scans: First, we used ZMap [22] to scan 2^{17} random IP addresses on every port. If at least 100,000 DTLS servers are deployed on a specific port, by scanning 2^{17} hosts, we can expect to find at least one among them in our sample with a probability of more than 95%. We configured ZMap to send a DTLS *ClientHello* and saved each received response. We dismissed all ports where no DTLS record was received, which left us with 2343 ports and a total of 2403 hosts. From this, we estimate that there are around 78.7 million DTLS IPv4 hosts on the Internet across all ports. To determine the ports where the largest number of DTLS servers are located, we repeated our scans for these ports and increased our sample size to 2^{20}. For eight of the 2343 ports, we received at least four responses with DTLS records. For our study, we scanned the entire IPv4 address space for each of these eight ports using ZMap again. Table 2 provides an overview of the results of our scan.

Port	Hosts Found	Hosts Evaluated	
443	273,140	168,924	61.85%
10443	262,724	236,519	90.03%
1106	47,654	44,189	92.73%
3391	36,719	34,636	94.33%
4433	17,874	15,027	84.07%
12346	15,334	13,712	89.42%
12446	9,388	7,842	83.53%
12681	1,368	-	-
Σ	664,201	520,849	78.42%

Table 2: Results of our scan of 2^{20} random IP addresses showing the top eight DTLS ports by hosts that sent DTLS records. We evaluated all hosts that sent at least a *ServerHello* message. Since our scans interfered with the running services on port 12681, we excluded the port from our study.

Evaluability. We evaluated all hosts for which we could receive a *ServerHello* message from the server. However, on all ports from Table 2, some hosts could not be thoroughly evaluated. This is mainly due to two reasons: (i) The server did not respond after some time, for example, due to blocking our requests or (ii) bugs in the DTLS implementation prevented a server from completing the DTLS handshake with our scanner.

Port 443 had the highest quota of unevaluated hosts, with 38.15%. As the primary cause, we could identify that many of these hosts implement the OpenConnect VPN protocol [35]. To complete a DTLS handshake with these hosts, the *ClientHello* message requires a session ID from a previous *TLS* handshake with the server. In addition, the client must authenticate to the server to switch to the DTLS channel. We thus excluded these servers prompting client authentication from our study.

We also excluded all hosts found on port 12681 from the study. The owner of a large number of hosts on that port contacted us shortly after we started our evaluation and informed us that our scans were interrupting services due to a buffer overflow in custom logging code that was triggered by benign protocol messages sent by our scanner. We stopped our evaluation and did not attempt to reevaluate these hosts to avoid interrupting any running services.

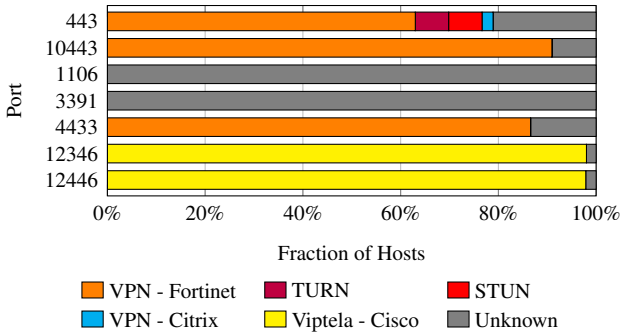

Figure 2: Distribution of the identified services. Note that services with $\leq 0.01\%$ support are not included.

6.2 Identified Services

Based on the ALPN extension, we were only able to negotiate an application protocol with 12,074 of the 168,924 hosts on port 443 and one host on port 4433. On the remaining ports, no host negotiated any of our offered protocols using ALPN. This can be either due to a lack of support for ALPN in general or due to the specific protocols offered in the extension. Since protocols are listed as ASCII strings, some servers might use custom values instead of the values specified by IANA. Among the 12,074 hosts, we were able to negotiate the following three application protocols: 11,958× TURN, 11,593× STUN, and 11466× HTTP 1.1. With 92% of the 12,074 hosts with identified ALPN support negotiating all three protocols. It is surprising that HTTP 1.1 was negotiated by so many servers, given that HTTP usually requires TCP. We attempted to trigger an HTTP reply from these servers to confirm that HTTP is used but were unable to do so. We thus assume that this is a misconfiguration of ALPN.

Independent of the ALPN extension, we tried to identify the service over the server's response. The distribution of the identified services is summarized in Figure 2. We could detect a VPN application from Fortinet as the most popular service on ports 443, 10443, and 4433. In contrast, the majority of hosts (97%) on ports 12346 and 12446 are running *Viptela*, a WAN management tool acquired by Cisco in 2017. [3] We identified this service based on the certificates we obtained from these hosts. Additionally, we found 11,522 STUN and 11,549 TURN services which are mainly located on port 443. Deeper analyses showed that almost all of these hosts also negotiate TURN or STUN as application layer protocols.

6.3 Data Collection Results

Here we analyze the collected protocol features and compare configuration differences between the individual ports.

Supported Protocol Versions (Table 3). On port 3391, DTLS 1.0 was used almost exclusively, while on ports 12346 and 12446, DTLS 1.2 was dominant. While analyzing excluded hosts that sent a DTLS record but did not send a *ServerHello*, we were able to identify a third version through the protocol version field of received alert records: DTLS 0.9, which is based on a pre-release draft of DTLS 1.0. We largely attributed this version to older Cisco servers by analyzing the certificates used by these hosts. A reference in a mailing list[4] indicates that Cisco used this pre-release version of DTLS in their *AnyConnect VPN* protocol implementation. We identified support for this pre-release draft version for 24,455 hosts on port 443.

Supported Cipher Suites (Table 3). On each considered port, most servers share a similar DTLS configuration. For example, on ports 10443 and 4433, the Camellia, ARIA, and ChaCha20-Poly1305 encryption schemes are supported by more than 87% of hosts. In contrast, hosts rarely use them on ports 3391, 12346, and 12446. This also applies to widespread support for weak algorithms; on port 443, we noticed that 13.5% of the hosts accept at least one cipher suite with NULL encryption, which does not achieve confidentiality if negotiated. In addition, we also determined support for old 64-bit block ciphers like 3DES and IDEA, with the highest ratios on port 1106, with 99.67%, and 3391, with 86.4% of hosts. Almost all hosts on port 1106 supported the RC4 stream cipher. RC4 is forbidden in DTLS as continuous internal state is required, which can not be maintained if datagrams of unknown size are lost in transit. Interestingly, the same hosts also supported modern AEAD cipher suites, such as ChaCha20 with Poly1305 authentication tags, which have much stronger security properties than the deprecated RC4 cipher [7, 28, 60]. Generally, support for AEAD cipher suites has been high across all ports except for port 3391, where block ciphers in CBC mode have almost exclusively been supported. While block ciphers in CBC can be implemented securely, minor implementation mistakes can make the servers susceptible to padding oracle attacks [38].

Regarding the preferred key exchange mechanism, we found that ECDHE key exchange is preferred over RSA and DHE across all ports. Hosts on ports 12346 and 12446 often enforced cipher suites that achieve Perfect Forward Secrecy (PFS). Static (EC)DH cipher suites have not been supported by any host in our scan.

Supported Elliptic Curves (Table 7). Similarly to the distribution of cipher suites, we see different elliptic curve configurations across the analyzed ports. For example, curves over binary fields were generally not supported, except on ports 12346 and 12446, where approximately 39% of hosts supported each binary field curve.

The most common curves we observed across all ports were *secp256r1*, *secp384r1*, and *secp521r1*. This is unsurprising as these curves are also very common among TLS implemen-

[3] https://www.cisco.com/c/en/us/solutions/collateral/enterprise/design-zone-security/sase-viptela-cvd.html

[4] https://lists.unix-ag.uni-kl.de/pipermail/vpnc-devel/2008-September/002585.html

Port	Total	v1.0	v1.2	RSA	DHE	ECDHE	RSA-PSK	DHE-PSK	ECDHE-PSK	PSK	Only PFS	mTLS
443	168,924	142,417	164,535	146,849	9,963	165,970	0	4,770	4,013	4,752	17,028	3,596
10443	236,519	235,148	236,060	234,579	69	236,402	0	66	66	66	1,779	47
1106	44,189	43,685	43,692	10	4	44,158	0	3	3	3	0	1
3391	34,636	34,636	15	33,398	16,410	34,582	0	3	3	3	1,210	34,520
4433	15,027	14,933	14,974	14,278	571	14,994	0	545	545	541	183	43
12346	13,712	432	13,709	13,707	13,681	13,404	0	3	3	3	12,424	13,697
12446	7,842	248	7,839	730	7,808	7,679	0	3	3	3	7,105	7,827

Port	Total	NULL	RC4	DES	3DES	IDEA	CAMELLIA	ARIA	AES	CHACHA	CBC	AEAD
443	168,924	22,792	0	13	9,256	250	119,772	105,583	168,627	113,214	154,474	162,563
10443	236,519	0	0	10	3,529	0	235,203	219,371	236,425	232,823	236,113	235,912
1106	44,189	0	44,075	0	44,041	0	11	6	43,872	41,039	44,052	41,653
3391	34,636	0	0	0	29,928	0	14	6	34,611	7	34,611	14
4433	15,027	3	0	3	501	0	14,891	13,125	15,006	13,902	14,969	14,939
12346	13,712	0	0	0	4	0	30	26	13,710	27	1,286	13,702
12446	7,842	0	0	0	4	0	32	28	7,838	29	733	7,832

Table 3: Distribution of the supported protocol features. The data shows that the ecosystem on a given port is mostly heterogeneous and dominated by a few configurations, likely caused by individual products deployed on these ports.

tations [58]. The sparse support for most other curves over prime fields and all curves over binary fields in general also led to their deprecation with RFC 8422 [40, Section 5.1.1]. Support for the newer curves, *x25519* and *x448*, was minimal across all ports, except for port 3391, where 74.4% supported *x25519*.

Supported Extensions (Table 8). The *Renegotiation* extension [46] is the only one almost all hosts support. The three ports 10443, 1106, and 4433 show similarly high support for the *EC Point Format* [40] (>99%), *Encrypt Then Mac* [29] (>91%), *Extended Master Secret* [12] (>96%), *Max Fragment Length* [23] (>91%) and *Session Ticket* [51] (>95%) extensions. For port 443, the support was slightly lower for all of these extensions, with 95.2%, 67.2%, 77%, 81.1%, and 94.4%, respectively. Surprisingly, the *Heartbeat* [52] extension is supported by only 5.18% of all evaluated servers. We expected higher support since the extension allows DTLS peers to determine if a peer is still alive and since more than 34% of TLS servers offered this extension in 2018 on port 443 [33]. We suspect that disabling the *Heartbeat* extension is a reaction to the discovered Heartbleed bug (CVE-2014-0160) in OpenSSL.

6.4 Certificate Details

We were able to get certificates from 98.55% of the evaluated servers (see Table 6). Some servers were exclusively supporting cipher suites with pre-shared keys (PSK) and therefore do not own any X.509 certificates for authentication.

Self-Signed Certificates on Port 1106. 99.75% of all servers on port 1106 use a self-signed certificate to authenticate themselves. Self-signed certificates cannot be verified based on a PKI and must *manually* be configured as trusted. Consequently, they are not suitable for communication with public peers. Additionally, all except one of these servers used an identical self-signed certificate. For this certificate, the *issuer* and *subject* fields contain the following content: `C = XX, L = Default City, O = Default Company Ltd`. These servers are also the ones that wrongfully support the RC4 stream cipher. We assume they use the same software in a default configuration or are hosted by the same operator.

Expired Certificates. Across all evaluated ports, we identified widespread use of expired certificates. In total, 28,569 (5.49%) hosts sent expired certificates, the majority on port 443. All DTLS client implementations should reject such certificates.

Keys and Signatures. On port 1106, we found 44,079 certificates that contained an ECDSA public key and were signed with ECDSA. Only nine certificates[5] contained an RSA public key and were signed using RSA. On all other ports, RSA public keys and signatures were used almost exclusively.

6.5 DTLS-Specific Features

For our DTLS-specific tests introduced in Section 3, the results are summarized in Table 4.

Cookie Exchange. The results show that the received cookie is mostly verified by the hosts across all ports except port 443 (A7). On that port, 13.5% of the servers did not validate the received cookie and accepted a *ClientHello* message with a *random* cookie. An analysis of their certificates indicated that the servers belong to the company Zscaler and likely use the same DTLS implementation. Further analysis revealed that their DTLS implementation always issues a *static* cookie. Both behavior patterns pose a significant threat because the servers can be abused as amplifiers by sending *ClientHello* messages with a *random* cookie. The server's response size varied depending on the server's *Certificate* message. Most often, the received DTLS data was bigger than 2,300 bytes and reached an amplification factor of 33. In addition, we identified that their servers are among those that support cipher suites with NULL encryption. We could not

[5] We identified 10 hosts that support RSA key exchange, but stopped responding during the certificate extraction.

Label	Test	443	10443	1106	3391	4433	12346	12446
	Total Hosts	168,924	236,519	44,189	34,636	15,027	13,712	7,842
A1[a]	No cookie exchange during a new handshake	461	526	566	147	60	6	8
A2[a]	Supports session resumption with session ID	48,681	102,451	4	3	5,775	4	4
	No cookie exchange during a resumption with session ID	0	0	0	0	0	0	0
A3[a]	Supports session resumption with session ticket	56,172	101,327	4	7	5,696	3	2
	No cookie exchange during a resumption with session ticket	3	10	0	0	0	0	0
A4[a]	Supports renegotiation	23,088	0	0	0	1	0	0
	No cookie exchange during a renegotiation	16,048	0	0	0	0	0	0
A5[a]	Performs *HelloVerifyRequest* retransmissions	10	22	4	84	1	24	14
A6	Performs recommended cookie computation	4,264	394	1	33,938	10	3	3
A7[a]	Does not validate the received cookie	22,797	0	3	0	2	0	0
A8	Cookie length: 8 bytes	0	0	0	0	2	0	0
	Cookie length: 16 bytes	22,831	0	43,611	5	0	0	0
	Cookie length: 20 bytes	140,811	235,921	8	6	14,377	13,700	7,828
	Cookie length: 24 bytes	762	0	0	0	0	0	0
	Cookie length: 32 bytes	4,059	72	4	34,478	588	6	6
B1	Sends retransmissions without requesting	15,163	432	43,605	34,478	903	48	47
B2	Processes client-requested retransmissions	140,517	235,344	43,566	34,463	15.254	13,696	7,830
C1[a]	Does not process fragmented *ClientHello* in a single datagram	15,980	17,699	509	34,629	1,763	13,704	7,834
C2[a]	Does not process fragmented *ClientHello* in cross datagrams	12,136	17,410	20,327	34,629	1,762	13,704	7,834
C3[a]	Does not process fragmented *ClientKeyExchange* in a single datagram	11,679	16,783	509	34,628	1,696	13,704	7,834
C4[a]	Does not process fragmented *ClientKeyExchange* in cross datagrams	11,962	17,254	20,327	34,629	1,279	13,704	7,834
D1[a]	Processes unencrypted *Finished*	0	0	0	0	0	0	0
D2[a,b]	Processes unencrypted *Application Data*	-	-	-	-	-	-	-
D3	Processes reordered *ChangeCipherSpec* and *Finished*	133,640	219,603	409	8	13277	7	7

a: Test description was inverted to emphasize interesting server behaviors.
b: Excluded from our large-scale study.

Table 4: Detailed results on our DTLS-specific tests of the Internet scan. Note that we inverted test descriptions compared to Table 1 to emphasize interesting server behaviors. The cookie lengths listed above cover all lengths we observed during our evaluation.

complete a handshake with these hosts regardless of which cipher suite was selected, indicating a severe deviation from the specification. We reported the observed behaviors to Zscaler, who acknowledged that the servers do not validate the cookie, as well as the intentional support of the NULL encryption.

Regarding session resumption, we discovered that less than 0.01% of all evaluated hosts do not execute a cookie exchange during resumption (A2, A3). We did not further evaluate whether a server can be forced to fall back to a full handshake without a new cookie exchange. We generally found that servers deviate from the RFC's recommended cookie computation but prevent DoS attacks by including the client's IP address (A6) and enforcing cookie exchanges for new DTLS sessions (A1).

Retransmissions. Across all ports, the majority of hosts supported retransmissions (B). However, most hosts do not implement a timer but only send retransmissions themselves if they receive retransmissions from the client. A notable exception are the hosts on ports 1106 and 3391, which also actively implement a timer. This is a behavioral difference from the server implementations evaluated in Section 5. However, for DTLS, it is sufficient that one peer implements the timer for the connection to be functional.

Missing Features. Almost all servers on ports 3391, 12346, and 12446 could not process fragmented messages (C) and messages received in the wrong order (D3). We expect these services to have connectivity problems from time to time. In contrast, most hosts on port 10443 supported both fragmentation (92.9%) and message reordering (92.8%).

6.6 Vulnerability Analysis

Subsequently, we analyze which known protocol attacks selected in Section 4.1 affect real server deployments. An overview of the tested attacks and the number of vulnerable hosts provides Table 5.

ALPACA [15]. Almost no hosts used *strict* ALPN validation. We could only identify 63 hosts on port 443 that implemented the mitigation, indicating that the DTLS ecosystem is still potentially affected by cross-protocol attacks. However, at the time of writing, there is no known DTLS cross-protocol attack exploitable through ALPACA. We did not evaluate if those servers are also using compatible certificates on other DTLS services.

Sweet32 [11]. We identified 16.75% of the servers using the 3DES and IDEA encryption schemes, making them potentially vulnerable to Sweet32. Most of these servers are located on two ports: 99.67% of the servers on port 1106 and 86.41% of the servers on port 3391. However, the support of 64-bit block ciphers only poses a real threat if those ciphers get negotiated and are used to transmit large amounts of data.

Raccoon [37]. The analysis of the supported cipher suites revealed no support for *static* DH key exchange. However,

Port	Total	Insecure Renegotiation	CVE-2020-13777	CRIME	Logjam	FREAK	Sweet32	Bleichenbacher	Padding Oracle	Invalid Curve	Raccoon	ALPACA not mitigated[1]
443	168,924	1,124	0	11	0	0	9,256	28	129	0	4004	168,861
10443	236,519	0	0	0	0	0	3,529	0	318	0	63	236,519
1106	44,189	0	0	0	0	0	44,041	0	0	0	3	44,189
3391	34,636	0	0	0	0	0	29,928	0	0	0	31	34,636
4433	15,027	0	0	0	0	0	501	0	25	0	540	15,027
12346	13,712	0	0	0	0	0	4	0	0	0	12,301	13,712
12446	7,842	0	0	0	0	0	4	0	0	0	7,016	7,842
Σ	520,849	1,124	0	11	0	0	87,263	28	472	0	23,958	520,768

1: We evaluated the potential attack surface by testing for a mitigation through the *Application-Layer Protocol Negotiation (ALPN)* extension.

Table 5: Overview of the number of hosts vulnerable to the evaluated attacks.

we detected the support of *ephemeral* DH key exchange in 48,506 servers, among which 23,958 *reused* their keys for more than one connection, making them likely vulnerable to the Raccoon attack. This vulnerability is especially common among ports 12346 and 12446 with around 89% each, whereas port 1106, despite its larger sample size, only had approximately 0.01% vulnerable hosts. While the vulnerability is quite common, it is hard to exploit as it requires very precise timing measurements and can (for the most part) only attack connections that naturally negotiate FFDHE cipher suites.

Insecure Renegotiation [42]. We were able to execute a renegotiation with 23,089 of the hosts across all ports. Among these, we discovered 1,124 servers that support insecure renegotiation. The affected servers are willing to renegotiate while the client is not presenting the *Renegotiation Info* extension or the `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` cipher suite [46]. This makes them potentially vulnerable to a renegotiation attack. However, the severity of this flaw depends on the concrete application.

Padding Oracle Vulnerabilities [61]. We identified 472 hosts with a CBC padding oracle vulnerability. They are distributed among three of the studied ports. These vulnerabilities manifested in two distinct response patterns that were both *observable* [38], meaning that the differences in the server responses were directly visible (e.g., different number of alerts). Further, the low number of vulnerable hosts indicates that the DTLS ecosystem is less affected by padding oracle vulnerabilities than the TLS ecosystem. In 2018, a study by Merget et al. [38] revealed vulnerabilities in 1.83% of the Alexa Top Million hosts by scanning their TLS implementation. Similiar to the Raccoon attack, these flaws can only be exploited if CBC cipher suites get naturally negotiated.

Bleichenbacher Attack [13]. For 28 hosts on port 443, we discovered a Bleichenbacher vulnerability. We identified two distinct subgroups based on their specific alerts. In both cases, the behavioral differences between a valid and invalid PKCS#1 padding could allow an attacker to recover RSA encrypted *Premaster Secrets* or forge RSA signatures. Overall, compared to the results of Böck et al. [14] on the TLS ecosystem, the Bleichenbacher vulnerability is nearly mitigated in DTLS. They discovered vulnerabilities in 2.8% of the Alexa Top Million hosts. While the Bleichenbacher attack is especially severe if RSA gets negotiated by real clients naturally, it is also potentially possible to perform a downgrade attack if the client supports any RSA cipher suite.

CRIME [48]. We found only eleven servers that support *DEFLATE* compression and are vulnerable to the CRIME attack if an attacker can (partially) control the encrypted content. Note that the client connecting to the server also needs to support compression and naturally negotiate it with the server. This result indicates that the CRIME attack is mostly mitigated in the DTLS ecosystem.

Other Attacks. None of the evaluated hosts support the intentionally weakened EXPORT ciphers indicating that the FREAK [10] and Logjam [4] attacks do not pose a threat to the DTLS ecosystem. Despite high support for the *Session Ticket* extension [51] of 87%, none of the hosts were vulnerable to the GnuTLS session ticket bug (CVE-2020-13777). Further, we did not find hosts vulnerable to an invalid curve [32] attack.

6.7 Malformed Application Data

We identified 10,875 servers on port 443 that send malformed encrypted application data after a successful handshake. Their records contain a *random* epoch and sequence number in the record header. We were unable to decrypt these records with the negotiated keys. Surprisingly, the *Finished* messages of these servers contained the *expected* epoch values and sequence numbers and could be decrypted correctly. We attribute this behavior to servers of the company AnchorFree, which did not respond to our reports.

7 Applicability to DTLS 1.3

DTLS 1.3 is primarily based on TLS 1.3 but incorporates specific modifications to work on unreliable transport protocols. Consequently, there are significant cryptographic divergences between DTLS 1.3 and its predecessors, while generic DTLS concepts mostly stayed the same.

DTLS 1.3 still uses the same anti-DoS cookie concept, which means that the same implementation pitfalls (Section 3.1) might be present. In contrast to previous RFCs, the DTLS 1.3 RFC explicitly mentions that *HelloRetryRequest*

messages should not be retransmitted to avoid the creation of a local state (Section 3.3). Ignoring this advice may obviously result in a vulnerability again. The renegotiation feature was replaced with *key updates*, which resolves our concerns in Section 3.2. The epoch concept is still present, which potentially enables the implementation flaws in Section 3.5; the RFC does not explicitly warn about this flaw. Fragmentation and message sequence numbers are still present, potentially enabling flaws from Section 3.4. The RFC does not explicitly warn about not keeping state for fragmented *ClientHello* messages.

In general, most of our potential flaws and tests can be adapted to DTLS 1.3 in the future and should be considered by developers.

8 Related Work

Flaws in DTLS. In 2012, AlFardan and Paterson presented a DTLS-specific technique to exploit timing-based CBC padding oracles in DTLS [41]. They could amplify minor timing differences by sending the same record with different sequence numbers multiple times. In 2013, the authors extended their work by discovering Lucky 13 - a timing-based CBC padding oracle vulnerability in the TLS specification. They evaluated the vulnerability also in the context of DTLS [5]. The attack was later reanalyzed [49, 6, 56], where the vulnerability reappeared multiple times.

Analysis of DTLS Implementations. DTLS implementations have already been evaluated in a controlled lab environment utilizing *protocol state fuzzing* (or *state learning*) [59, 55, 24, 25]. During state learning, DTLS messages are sent in different orders to the peer, to construct a mealy machine model of the state machine implementation. This model can then later be analyzed to find flaws related to the state machine of the analyzed implementation. While Van Drueten [59] and Tåkvist [55] could not find new vulnerabilities, Fiterau-Brostean et al. examined thirteen server implementations [24] using protocol state fuzzing and uncovered four security vulnerabilities. A time-consuming manual analysis of the obtained state machines was required to identify flaws. In [25], the analysis process was automated such that the previously discovered bugs could now be confirmed without further manual analysis. While state machine learning is a powerful tool, it is not suitable to extract general properties of DTLS implementations or to test for known cryptographic vulnerabilities. For example, Fiterau-Brostean et al. [24, 25] did not test for all cryptographic attacks [11, 61, 13, 32, 10, 4, 15, 48, 37] nor for the following issues and properties: Cookie exchange (A2, A3, A4, A7, A8), retransmissions (B), fragmentation (C), and reordering (D3). Additionally, previous state learning approaches require careful tuning of the tested implementation and the learner to fully automate the process and avoid false positives, making them unsuitable for large-scale scans.

Asadian et al. applied symbolic execution to analyze four server implementations for violations of the DTLS specification [8]. They focused only on handshakes with a pre-shared key to reduce the execution time of their evaluation and uncovered vulnerabilities and non-conformance issues in OpenSSL and TinyDTLS.

Large-Scale Scans on the TLS Ecosystem. In contrast to DTLS, the TLS ecosystem has been the subject of multiple large-scale studies, which tracked its development over time [33], evaluated provided HTTPS certificates [20], compared deployment for different ports and application protocols [30, 36], and even analyzed clients based on passively collected data [27]. Additionally, studies estimated the impact of vulnerabilities such as Heartbleed [21], weak Diffie-Hellman parameters [57, 19], ROBOT [14], padding oracles [38], Logjam [4], DROWN [9], Curveswap [58], Raccoon [37], and ALPACA [15]. Further, Dahlmanns et al. performed an Internet-wide study of ten (D)TLS-based industrial IoT protocols [16]. They evaluated the configuration of 705 DTLS servers found on ports 5683 and 5684 as part of their scan for CoAP hosts. They analyzed the negotiated protocol versions, cipher suites, and certificates obtained using ZGrab2 [3].

9 Conclusions and Future Work

In this work, we analyzed DTLS server implementations of well-known open-source libraries in a controlled environment and presented the first large-scale study of the DTLS server ecosystem by scanning an estimated 0.66% of publicly available IPv4 servers on the Internet.

Regarding our first research question, we determined that several (D)TLS libraries show unsupported features, functional bugs, or non-conformance issues. While the libraries are mostly secure, the discovered issues limit their robustness. Further, we observed that DoS attacks are still a real threat to DTLS implementations. We found five DoS vulnerabilities via memory exhaustion and three DoS amplification vulnerabilities across well-known libraries.

As for our second research question, we found that few DTLS configurations dominate every evaluated port. We attribute this to a few individual pre-configured products deployed on a given port. We also found that DTLS servers are relatively secure against known attacks, except for the Raccoon [37] and ALPACA [15] attacks which have only been discovered recently. The most pressing issues in the DTLS ecosystem appear to be DoS amplification vulnerabilities, with 4.4% of servers being affected.

While our evaluation focused on the analysis of DTLS servers, the state of the ecosystem regarding clients is still widely unknown, as a thorough evaluation requires access to real-world network traffic collected by passive scans. Another related area not covered by our study is WebRTC, which is used for real-time communication over the Internet, such as video conferencing and also uses DTLS. To evaluate WebRTC, a dedicated setup is required, as server instances are short-lived and only started on demand after a signaling phase

has been executed.

However, it is questionable how relevant DTLS will be for the community, even with the recent specification of DTLS 1.3 [47]. DTLS competes directly with QUIC [31] as both protocols try to achieve similar goals. Today, it is unclear which protocol will dominate the ecosystem in the future and whether the newly specified DTLS 1.3 protocol will be widely adopted. The methodologies used in our work will be useful when answering questions about the prevalence and security of these two standards in the future.

Acknowledgements

We thank the anonymous reviewers and shepherd for their valuable feedback. This research was partially funded by Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the German Federal Ministry of Education and Research (BMBF) through the project KoTeBi. Nurullah Erinola was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 450197914.

References

[1] TLS-Attacker. https://github.com/tls-attacker/TLS-Attacker.

[2] TLS-Scanner. https://github.com/tls-attacker/TLS-Scanner.

[3] ZGrab2. https://github.com/zmap/zgrab2.

[4] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[5] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy, SP*, 2013.

[6] Martin Albrecht and K.G. Paterson. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT*, 2016.

[7] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the Security of RC4 in TLS. In *22th USENIX Security Symposium*, 2013.

[8] Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, and Konstantinos Sagonas. Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification. In *IEEE Conference on Software Testing, Verification and Validation, ICST*, 2022.

[9] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium*, 2016.

[10] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security and Privacy, S&P*, 2015.

[11] Karthikeyan Bhargavan and Gaëtan Leurent. On the Practical (In-)Security of 64-Bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2016.

[12] K. Bhargavan (Ed.), A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627 (Proposed Standard), September 2015.

[13] Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Advances in Cryptography - 18th Annual International Cryptology Conference*, 1998.

[14] Hanno Böck, Juraj Somorovsky, and Craig Young. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *27th USENIX Security Symposium*, 2018.

[15] Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk, and Sebastian Schinzel. ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication. In *30th USENIX Security Symposium*, 2021.

[16] Markus Dahlmanns, Johannes Lohmöller, Jan Pennekamp, Jörn Bodenhausen, Klaus Wehrle, and Martin Henze. Missed Opportunities: Measuring the Untapped TLS Support in the Industrial Internet of Things. 2022.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006.

[18] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[19] Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. Indiscreet Logs: Diffie-Hellman Backdoors in TLS. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[20] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Conference on Internet Measurement Conference, IMC*, 2013.

[21] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Conference on Internet Measurement Conference, IMC*, 2014.

[22] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, 2013.

[23] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), January 2011.

[24] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri

de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *USENIX Security Symposium*, 2020.

[25] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. In *30th Annual Network and Distributed System Security Symposium, NDSS*, 2023.

[26] S. Friedl, A. Popov, A. Langley, and E. Stephan. Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (Proposed Standard), July 2014.

[27] Sergey Frolov and Eric Wustrow. The use of TLS in Censorship Circumvention. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[28] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS. In *USENIX Security Symposium 2015*, 2015.

[29] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), September 2014.

[30] Ralph Holz, Johanna Amann, Olivier Mehani, Mohamed Ali Kâafar, and Matthias Wachs. TLS in the Wild: An Internet-wide Analysis of TLS-based Protocols for Electronic Communication. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[31] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000 (Proposed Standard), May 2021.

[32] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical Invalid Curve Attacks on TLS-ECDH. In *20th European Symposium on Research in Computer Security, ESORICS*, 2015.

[33] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of Age: A Longitudinal Study of TLS Deployment. In *Internet Measurement Conference, IMC*, 2018.

[34] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010.

[35] N. Mavrogiannopoulos. The OpenConnect VPN Protocol Version 1.2. draft-mavrogiannopoulos openconnect-03 (Internet-Draft), October 2020.

[36] Wilfried Mayer, Aaron Zauner, Martin Schmiedecker, and Markus Huber. No Need for Black Chambers: Testing TLS in the E-mail Ecosystem at Large. In *11th International Conference on Availability, Reliability and Security, ARES*, 2016.

[37] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E). In *USENIX Security Symposium*, 2021.

[38] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities. In *USENIX Security Symposium*, 2019.

[39] Nagendra Modadugu and Eric Rescorla. The Design and Implementation of Datagram TLS. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2004.

[40] Y. Nir, S. Josefsson, and M. Pegourie-Gonnard. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier. RFC 8422 (Proposed Standard), August 2018.

[41] Kenneth G. Paterson and Nadhem J. AlFardan. Plaintext-Recovery Attacks Against Datagram TLS. In *19th Annual Network and Distributed System Security Symposium, NDSS*, 2012.

[42] M. Ray and S. Dispensa. Authentication gap in TLS renegotiation, 2009.

[43] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[44] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006.

[45] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.

[46] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[47] E. Rescorla, H. Tschofenig, and N. Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147 (Proposed Standard), April 2022.

[48] Juliano Rizzo and Thai Duong. The CRIME attack. In *Ekoparty Security Conference*, 2012.

[49] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018.

[50] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[51] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), January 2008.

[52] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520 (Proposed Standard), February 2012.

[53] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.

[54] Juraj Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2016.

[55] Fredrik Tåkvist. Analysis of DTLS Implementations Using State Fuzzing. August 2020.

[56] Ye Tang, Huiyun Li, and Guoqing Xu. Cache Side-Channel Attack to Recover Plaintext against Datagram TLS. In *Inter-*

national Conference on IT Convergence and Security, ICITCS, 2015.

[57] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[58] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In Search of CurveSwap: Measuring Elliptic Curve Implementations in the Wild. In *IEEE European Symposium on Security and Privacy, EuroS&P*, 2018.

[59] N. van Drueten. Security analysis of DTLS 1.2 implementations. January 2019.

[60] Mathy Vanhoef and Frank Piessens. All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium*, 2015.

[61] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, 2002.

A Configuration of TLS-Scanner

We now describe how we test specific vulnerabilities with TLS-Scanner:

ALPACA [15]: At the time of writing, there is no known DTLS cross-protocol attack exploitable through ALPACA. To estimate the *potential* attack surface, we evaluate if *strict* ALPN is deployed as a countermeasure. This is the case if the server only accepts *ClientHello* messages that propose the extension.

CRIME [48]: We consider any server as vulnerable as soon as the server supports (D)TLS compression.

CVE-2020-13777: For this vulnerability, we decrypt the issued session ticket with the encryption algorithm AES-128-CBC using an all-zero key. We then consider any server as vulnerable as soon as the established *Master Secret* can be found in the decrypted session ticket.

FREAK [10] & Logjam [4]: We consider any server as vulnerable that supports intentionally weakened EXPORT cipher suites.

Raccoon Attack [37]: We do not evaluate if the code on the server is running in constant time but consider it vulnerable as soon as the server *reuses* a DH key during any observed handshake during the scan.

Renegotiation Attack [42]: We consider any server as vulnerable as soon as the server allows insecure renegotiation. In DTLS the vulnerability is only exploitable if the server wraps epoch numbers. We do not evaluate if the server wraps epoch numbers to keep the load on the scanner and the server low, as this would require 2^{16} handshakes per tested server.

Side-Channel Attacks [13, 61, 32]: For Bleichenbacher and CBC padding oracle attacks, we use the adapted statistical test technique from [37] to test potentially observed side channels for statistical significance ($p \leq 0.0001$). For CBC padding oracles, we use the four vectors of Merget et al. [38] with one to ten repetitions, while for Bleichenbacher, we use twelve different vectors and evaluate three different message flows inspired by Böck et al. [14]. For the invalid curve [32] evaluation, we perform 19 attempts to compute a valid *Finished* message with a point of order five, resulting in an expected false negative rate of less than 0.01%.

In all cases, we only test a *single* version, cipher suite (and key exchange group) combination instead of exhausting the input space due to the large number of tests we are already performing in this study. As indicated by previous research, this choice will likely undercount the discovered vulnerabilities, as some vulnerabilities only show in particular combinations [14, 37, 38].

Sweet32 [11]: We consider any server as vulnerable that supports 64-bit block ciphers. We note that to perform the attack, this is generally not enough, as also long-lived connections with large amounts of known data are required that also frequently transmit secret values.

B Results of the Large-Scale Study

Below we present the results regarding the collected certificates in Table 6, the supported elliptic curves in Table 7, and the supported extensions in Table 8.

		General		Public Key Type		Signature Algorithm	
Port	**Total**	**Self-signed**	**Expired**	**RSA**	**ECDSA**	**RSA**	**ECDSA**
443	168,924	9,752	17,695	162,908	519	163,291	136
10443	236,519	4,026	6,549	234,803	628	235,368	63
1106	44,189	44,080	7	9	44,079	9	44,079
3391	34,636	3,981	3,892	34,419	92	34,458	53
4433	15,027	509	422	14,316	85	14,390	11
12346	13,712	1	2	13,635	0	13,632	3
12446	7,842	1	2	7,795	0	7,795	0

Table 6: Overview of the collected certificates. Almost all hosts on port 1106 used a self-signed certificate to authenticate themselves. Note that since some servers only support pre-shared key cipher suites, the certificates do not add up to 100%.

Port	Total	sect233k1	sect233r1	sect283k1	sect283r1	sect409k1	sect409r1	sect571k1
443	168,924	0	0	470	470	470	470	471
10443	236,519	0	0	0	0	0	0	0
1106	44,189	0	0	0	0	0	0	0
3391	34,636	0	0	0	0	0	0	0
4433	15,027	0	0	1	1	1	1	1
12346	13,712	5,300	5,297	5,296	5,295	5,295	5,294	5,293
12446	7,842	3,080	3,078	3,077	3,077	3,076	3,076	3,076

Port	Total	sect571r1	secp192k1	secp192r1	secp224K1	secp224r1	secp256k1	secp256r1
443	168,924	471	0	22	0	4,002	7,664	52,946
10443	236,519	0	0	0	0	380	0	2,908
1106	44,189	0	0	0	0	0	0	43,966
3391	34,636	0	0	0	0	0	2	34,164
4433	15,027	1	20	20	20	22	21	119
12346	13,712	5,293	0	0	0	0	0	5,316
12446	7,842	3,076	0	0	0	0	0	3,101

Port	Total	secp384r1	secp521r1	bpP256r1[1]	bpP384r1[1]	bpP512r1[1]	x25519	x448
443	168,924	142,370	37,629	472	474	474	4,327	4,130
10443	236,519	212,042	402	0	0	0	25	22
1106	44,189	9	0	0	0	0	1	0
3391	34,636	34,148	1,352	2	2	2	25,767	0
4433	15,027	12,920	25	21	21	21	9	2
12346	13,712	5,323	13,315	0	0	0	23	20
12446	7,842	3,108	7,623	0	0	0	25	22

1: "bp" indicates a Brainpool curve.

Table 7: Distribution of the supported elliptic curves. The most common curves across all ports were *secp256r1*, *secp384r1*, and *secp521r1* which are also very common among TLS implementations. Interestingly, a large share of hosts on ports 12346 and 12446 support *sect* curves, which are rare for TLS and have been deprecated by RFC 8422 [40, Section 5.1.1]. The newer curves, *x25519* and *x448* were only supported by a small share of hosts, except for port 3391, where 74% supported curve *x25519*. These findings again indicate mostly homogeneous configurations on some of the ports.

Port	Total	ALPN	Certificate Status Request	Certificate Status Request V2	EC Point Format	Encrypt Then Mac	Extended Master Secret
443	168,924	12,074	36	0	160,734	113,551	130,039
10443	236,519	0	2	0	235,922	220,263	233,760
1106	44,189	0	0	0	44,105	43,615	44,107
3391	34,636	0	26,962	0	10	9	34,531
4433	15,027	1	5	0	14,967	13,762	14,560
12346	13,712	0	2	0	10	29	5,947
12446	7,842	0	2	0	10	31	3,501

Port	Total	Heartbeat	Max Fragment Length	Renegotiation Info	Session Ticket	Truncated HMAC
443	168,924	12,275	136,935	166,332	159,449	0
10443	236,519	2,207	220,717	235,970	235,897	0
1106	44,189	0	44,104	44,107	44,103	0
3391	34,636	3	9	34,609	10	0
4433	15,027	428	13,805	14,991	14,408	0
12346	13,712	7,755	5,943	13,702	131	0
12446	7,842	4,337	3,497	7,838	84	0

Table 8: Overview of the prevalence of the tested extensions. Surprisingly, *ALPN* was supported by 7% of hosts on port 443 while hosts on all other ports indicated no support except for a single server on port 4433. The *Heartbeat* extension is supported by only 5% of all evaluated servers although the extension allows DTLS peers to determine if a peer is still alive.