TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries

Marcel Maehren[1], Philipp Nieting[1], Sven Hebrok[2], Robert Merget[1], Juraj Somorovsky[2], and Jörg Schwenk[1]

[1]Ruhr University Bochum
[2]Paderborn University

Abstract

Although the newest versions of TLS are considered secure, flawed implementations may undermine the promised security properties. Such implementation flaws result from the TLS specifications' complexity, with exponentially many possible parameter combinations. Combinatorial Testing (CT) is a technique to tame this complexity, but it is hard to apply to TLS due to semantic dependencies between the parameters and thus leaves the developers with a major challenge referred to as the *test oracle problem*: Determining if the observed behavior of software is correct for a given test input.

In this work, we present TLS-Anvil, a test suite based on CT that can efficiently and systematically test parameter value combinations and overcome the oracle problem by dynamically extracting an implementation-specific input parameter model (IPM) that we constrained based on TLS specific parameter value interactions. Our approach thus carefully restricts the available input space, which in return allows us to reliably solve the oracle problem for any combination of values generated by the CT algorithm.

We evaluated TLS-Anvil with 13 well known TLS implementations, including OpenSSL, BoringSSL, and NSS. Our evaluation revealed two new exploits in MatrixSSL, five issues directly influencing the cryptographic operations of a session, as well as 15 interoperability issues, 116 problems related to incorrect alert handling, and 100 other issues across all tested libraries.

1 Introduction

Transport Layer Security (TLS) is a cryptographic protocol that provides encryption, authentication, and integrity to application data. Due to many attacks and weaknesses discovered in the recent years [3, 7, 8, 10, 18, 19, 53, 59, 65], the currently recommended versions of TLS are 1.2 and 1.3 [40, 51]. Besides the main standards, there is a multitude of accompanying RFCs, which define further TLS extensions and cryptographic algorithms for the protocol [41–50, 52]. These RFCs are the result of continuous refinements throughout multiple public draft versions. Especially in the case of TLS 1.3, the development of drafts was strongly influenced by feedback from the community. The knowledge of the research community has been used to emphasize security and interoperability critical statements in the RFCs using the terminology for absolute requirements from RFC 2119 [39], which are marked with the keywords MUST, SHALL, or REQUIRED. These requirements prescribe specific TLS behavior, for example, by defining protocol flows, record layer processing, or exact alert messages. These requirements go far beyond solely functional aspects, and TLS libraries must adhere to the specifications in all critical aspects. Otherwise, missing compliance can lead to interoperability issues or even critical security bugs.

On the Complexity of TLS Libraries The high number of protocol versions, extensions, and cryptographic algorithms increases the complexity of TLS and makes implementing a secure TLS library very challenging. For backward compatibility reasons, standard TLS libraries need to support multiple TLS versions starting from TLS 1.0 and also outdated cryptographic algorithms such as 3DES. The complexity of TLS and the backward compatibility requirements led to several critical attacks. For example, Böck et al. discovered that the at the time almost 20-year-old Bleichenbacher [8] vulnerability was still widespread among TLS implementations of the most prominent websites on the internet; some of the discovered vulnerabilities had subtle dependencies to seemingly unrelated parts of the code, like the mode of operation [10]. Similarly, a study by Merget et al. [36] showed that sometimes CBC padding oracle vulnerabilities only surface for specific negotiated parameters; a different key exchange algorithm in a cipher suite could already change the code flow enough to reveal or hide a vulnerability in the otherwise unrelated record layer. Thus despite the huge amount of research and careful specification of the protocol, TLS *libraries* are still vulnerable to subtle attacks exploiting complex parameter combinations. This leads to the following research question:

> **RQ1**: How well do current TLS libraries perform in regards to security, interoperability, and conformance considering the complexity of requirements from the TLS specification and scientific literature?

Test Oracle Problem Software tests stimulate a *system under test* (SUT) with inputs and observe the reaction of the system. To provide a meaningful test result, software tests must overcome the *test oracle problem*. Barr et al. define the test oracle problem as follows [4]:

> Given the input for a system, the challenge of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior is called the "test oracle problem".

For a complex system, this problem is already hard for functional tests, where the goal is to determine if a program computes the correct values. For the TLS protocol, this would be tests that check if an implementation is interoperable and finishes the handshake correctly when provided with valid cryptographic messages. It becomes even harder if the non-functional property "security" should be tested – this property must be preserved even for all combinations of invalid inputs.

Previous attempts to solve the test oracle problem for the TLS protocol defined the "desired, correct behavior" of a TLS implementation as having exactly the same output as a reference implementation – miTLS in [56] and nqsb-TLS in [57] and [23]. This approach practically failed because it produced a very high false positive rate as in TLS, there are often multiple valid responses to a specific input, but every deviation from the reference implementation was treated as a flaw. Ultimately, the authors only attested that behavioral *differences* between libraries could be observed [23, Sec. 6][56, Sec. 7][57, Sec. 7].

A trivial approach to overcome false positives is that a human specifies the correct behavior for every combination of input parameters. While this may be feasible to check some basic security properties (e.g., the strongest encryption algorithm is always chosen during the TLS handshake or that dangerous features are deactivated), the exponential number of tests needed for a thorough coverage cannot be generated manually. For example, for the parameters in the scope of this paper (Table 3), in the worst case, we would have to manually write 18,743,296 variations of the same server test.

Test Oracle Automation Test oracles can be derived from formal specifications of the system, from assertions in the source code, from pseudo-oracles, from regression tests, or from invariants in the program code detected automatically [4]. These approaches either need special forms of formal specifications not available for TLS, or may only be used to test functional properties. Approaches to automatically generate test oracles from human-readable specifications either suffer from exponential growth of the resulting formal description [37], or need a restricted natural language [55]. Without these restrictions, human interaction seems necessary to define the correct outcome. In this case, automation can be achieved by defining abstract test templates from which many test cases can be generated automatically.

Combinatorial Testing CT is a widely known technique to select combinations from a set of *parameters*. In CT, a test is hence designed such that the same test can be invoked with different values of these parameters. These tests are referred to as *parameterized tests* or *test templates*. Designing such test templates can be far from trivial as each template requires an analysis that identifies which parameters can be parameterized. Even given such test templates, testing all combinations of test parameters is usually still infeasible since the number of test cases grows exponentially. One way to reduce the number of possible parameter combinations is a CT method called t-way testing [31]. In t-way testing, all possible combinations of up to t parameters out of the n parameters are used for testing. If $t < n$, the number of tests can be further reduced by including two or more t-sets in the same test vector. If $t << n$, this results in a significant reduction in the number of tests.

In general, CT is a suitable choice for developing precise test inputs with high flexibility to trigger deep TLS corner cases. However, using this approach for TLS evaluations yields further challenges which have been left unsolved in the previous scientific studies [23, 56, 57].

Challenging Parameter Interactions A naive approach to utilize CT would blindly combine different parameter values. However, in a complex protocol like TLS, parameters have dependencies, and not all potential parameter values are useful for a given test. For example, a test that evaluates if a library can complete the handshake with a given cipher suite should choose a cipher suite that is supported by the SUT. A test that analyzes the elliptic curve computations performed by an SUT must further limit its choice to an *elliptic curve* cipher suite. While these two examples are still reasonably manageable, the parameter value interactions can quickly become complex depending on the test context. For example, consider a test for the client-side validation of signatures generated by a server. In that case, there is an interaction between the selected cipher suite, signature algorithm, and server certificate that ultimately affects signature validity. A chosen combination of these three parameters may be invalid regardless of the specific byte values of the signature. If these dependencies are ignored, the test result on the SUT does not correspond to the validation of the digital signature but to the invalid parameter choice, which would render the test oracle unreliable. These challenges result in another research question:

> **RQ2**: Can reliable test oracles for parameterized test templates with a large parameter space for TLS

be defined with a low false positive rate?

Solving the Test Oracle Problem for TLS We propose a novel approach for solving the test oracle problem for TLS consisting of four components:

1. We use *test templates* which can be parameterized to automatically generate many test cases. Each test template tests a requirement based on an RFC. The parameterization of the template enables the use of CT.

2. We define a test oracle for each test template that can decide if the selected requirement is fulfilled for a given input and output; this reflects the flexibility inherent to TLS and mitigates the high false positive rate of previous approaches.

3. We systematically restrict the Input Parameter Model (IPM) of test templates to 'reasonable' values. In the first step, feature extraction is used to restrict the parameter values to values supported by the SUT. Then semantic dependencies between test parameters are taken into account to derive a semantically sound subset for the IPM.

4. We use *t-way combinatorial testing* to minimize the number of test cases while still covering all t-combinations of parameters.

This answers **RQ2** in the affirmative but leads to a new research question:

> **RQ3**: How practical and effective is our methodology when considering complex TLS libraries?

TLS-Anvil To demonstrate the effectiveness of our methodology, we implemented a TLS test suite called TLS-Anvil that uses CT to derive test cases from test templates. TLS-Anvil is capable of testing the compliance of a server or client implementation with the protocol specification. It can be used by developers to test their implementations as well as penetration testers to estimate the quality of a TLS stack. We built TLS-Anvil upon TLS-Attacker [60], TLS-Scanner [61], JUnit5 [27], and coffee4j [17, 21]. TLS-Attacker is a well-established framework for the analysis of TLS libraries, while the coffee4j framework allows us to use t-way testing with a configurable strength t, enabling us to find hidden bugs in the implementation. To create test templates, we carefully analyzed TLS-related RFCs [40–52] for absolute requirements which are marked with the keywords MUST, SHALL or REQUIRED [39]. Additionally, we integrated known state machine vulnerabilities from the literature [54] to guide our state machine tests beyond the implicit definition of a TLS state machine in the RFCs. We then created test templates for the extracted requirements for our test suite.

To demonstrate the effectiveness of TLS-Anvil, we developed a library of docker images that allows researchers to quickly start TLS clients and servers in different versions, which is of general interest for TLS developers and researchers independent of the test suite. Our docker library contains around 700 versions of 23 different implementations and provides a Java interface to start and stop TLS implementations easily. We used our docker library to evaluate 13 widely used TLS libraries.

Results Although TLS is arguably the mostly researched cryptographic protocol, TLS-Anvil was able to find five issues that affect cryptographic computations and three immediately exploitable vulnerabilities in the newest TLS library versions (Table 2). These included a TLS server authentication bypass (reported independently), a CBC padding oracle resulting from invalid buffer boundary validation, and a Denial-of-Service vulnerability. In addition, TLS-Anvil successfully found 231 other RFC violations, including 15 interoperability issues. Especially the security and cryptography issues indicate a blind spot in current testing approaches as the considered libraries are generally well-tested yet these issues are present. Thus, in answering **RQ1** we must state that although TLS libraries are primarily well-maintained, the security of the overall ecosystem will profit from systematic testing. Our results also answer **RQ3** since various RFC violations could be found within a reasonable execution time. Additionally, the extent of the test suite can be scaled based on the use case by adjusting the t parameter used for CT.

Contributions We make the following contributions:

- We propose a novel methodology to solve the test oracle problem for TLS through a technique based on parameterized test cases derived from test templates and careful constraints for the input space. This methodology can be adapted to other cryptographic protocols.
- We develop TLS-Anvil, a test suite for TLS 1.2 and TLS 1.3 libraries, which provides a high degree of flexibility, an extendable architecture, and application programming interfaces (APIs) to write test templates for the TLS protocol efficiently. TLS-Anvil uses t-way testing to enhance performance.
- We demonstrate that t-way testing is an effective technique for testing complex security protocols as it found issues even for already highly tested TLS libraries.
- We found and disclosed two new vulnerabilities (the third vulnerability was already reported independently) and five cases where cryptographic functionalities in TLS could be illictly affected as well as additional 231 RFC violations.
- We provide all of our developed tools as open-source software.

Responsible Disclosure We responsibly disclosed all of our findings to the respective developers.

2 Background

2.1 Transport Layer Security (TLS)

The TLS protocol allows two communicating peers to establish a secure channel. To this end, the peers perform a TLS handshake to negotiate cryptographic parameters – defined in so-called cipher suites (e.g., `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`) – and derive TLS session keys. The negotiated parameters and keys are then used within the channel to protect the exchanged messages' confidentiality, integrity, and authenticity.

There are two widely used and recommended TLS versions: TLS 1.2 [40] and TLS 1.3 [51].

TLS 1.2 A regular TLS 1.2 handshake with DH key exchange and server authentication requires two round trips [40]. The client starts the handshake with the `ClientHello` message, which contains the TLS version, a list of proposed TLS cipher suites, named groups for the key exchange, and extensions. The server responds with four handshake messages. With the `ServerHello` message, the server selects the TLS version and other cryptographic parameters from the proposals of the client. The server then sends a `Certificate` message that contains the X.509 certificate chain with the server's public key. The server uses its private key to authenticate a freshly generated ephemeral DH public key and sends it within the `ServerKeyExchange` message. Finally, the server sends the `ServerHelloDone` message to indicate the end of the flight. The client continues the handshake with a `ClientKeyExchange` message, which contains the client's DH share. From now on, both parties can compute a shared secret, called premaster secret, which is used to derive all cryptographic keys. The client then sends a `ChangeCipherSpec` message to notify the server about switching to an encrypted state and sends a `Finished` message. The server finalizes the handshake by sending its own `ChangeCipherSpec` and `Finished` messages.

TLS 1.3 The newest version of TLS is TLS 1.3 [51]. In contrast to TLS 1.2, it usually only needs one RTT for its handshake to establish cryptographic keys. This is achieved by reordering the messages and adding new extensions.

TLS 1.3 attempts to negotiate a shared secret already with the `ClientHello` and `ServerHello` messages that contain protocol parameters along with DH shares. Both parties then directly compute additional secrets and begin to encrypt and protect messages. The server then sends an `EncryptedExtensions`, `Certificate`, and `CertificateVerify` message. The `CertificateVerify` message contains a signature over previously exchanged messages, verifiable with the public key from the server certificate. The last message the server sends in the handshake is a `Finished` message, which is similar to the `Finished` message in TLS 1.2. The client finally also sends a `Finished` message to complete the handshake.

Alert Protocol The TLS protocol has a mechanism to communicate errors to the peer called *alerts*. TLS RFCs specify more than 30 different alerts and how to use them in specific exceptional cases. For example, the `Close Notify` alert notifies the peer about closing the underlying connection, or `Bad Record MAC` informs the peer that the received record message contains an invalid authentication code. Correct handling of alert messages is critical for the security of a TLS library, as they can provide information to a potential attacker that can be used to break the security goals of TLS [36].

2.2 Software Testing

Testing In the general testing terminology, the evaluated software is referred to as the System Under Test (SUT). Testing an SUT can be performed on various levels. *Unit tests* are performed on a very low level and focus on testing the functionality of isolated functions. To do so, Unit tests need access to software internals. *System tests* evaluate software as a whole, with all of its interacting submodules. The test inputs are the external inputs of the software; thus, system tests are suitable for black-box testing. A complete set of inputs for an SUT is called a *test case*. The collection of all test cases which are executed against the SUT is called *test suite*. For parameterized tests, test cases are instantiated from *test templates*, which the test developer writes.

Test Oracles To classify the result of a test case, the developer of the test template defines a *test oracle* [24]. The test oracle decides if the behavior of an SUT is appropriate for a given input and if the SUT passed or failed the test. In practice, this decision can be challenging, which is also known as the *test oracle problem* [4].

Combinatorial Testing Some test failures may only occur due to an interaction between different input parameters, so they can only be detected by performing tests that cover *all* combinations of parameters. In practice, this is often not feasible as complex software can have a multitude of parameters with many different values, resulting in an exponential growth of generated test cases. However, research in the field of software testing has shown that most observed failures are not the result of an interaction of all possible parameters but rather a subset of them [32, 38]. This led to the concept of t-way tests.

A t-way test covers all combinations of parameter values for each subset of t parameters. All possible test parameters and their values are predefined in a so-called Input Parameter Model (IPM).

Generating an ideal test suite, i.e., the minimum set of test cases required to cover all t-way interactions, is challenging.

Still, modern algorithms are often able to provide a good test suite within an acceptable timeframe. To avoid the creation of nonsensical test inputs and to speed up the generation of the test suite, many CT algorithms provide the option to define constraints on specific parameter value combinations manually and thus exclude combinations that do not produce meaningful test cases. If the test inputs are chosen carefully, even a CT with a small value for t can achieve a high bug detection rate while maintaining a comparatively small test suite [30].

3 Methodology

In this paper, we propose a new testing methodology for TLS libraries that is of general interest for testing cryptographic protocols. It leverages the power of t-way testing to systematically evaluate the complex parameter interactions of the libraries. The test oracle problem is solved by defining test templates, from which the different test cases for t-way testing can be generated automatically. Many values of TLS parameters are optional: E.g., not all cipher suites, and extensions must be supported. Hence, for each tested library, we perform feature extraction using TLS Scanner. Upon execution, an Input Parameter Model (IPM) is created for each test template that is specific to the tested implementation such that the oracle problem is solvable for all possible test cases of the template.

3.1 Test Templates for Reliable Test Oracles

A test template defines the desired outcome of *all* test cases derived from it – thus, it represents a test oracle that is applicable specifically to its resulting test cases. Each test template checks a specific requirement. These requirements are derived from TLS-related RFCs and known state machine bugs. For the test templates to be reliable, the conditions listed below must be met. In Section 4 we give the details on which requirements have been considered and how to fulfill these requirements.

Combinatorial Testing (CT) To enable CT, the template must allow for the insertion of all reasonable parameter combinations, even in seemingly unrelated aspects of the test input.

General IPM In order to create an IPM for CT, concrete parameter values have to be chosen for each parameter for a given test template. For parameters that only have a limited number of values, all values can be considered, while for parameters with potentially many different values, individual, *interesting* values have to be chosen. For example, a parameter could model the addition of a specific extension as a boolean parameter. Here, both possible values 'true' and 'false' can be considered as parameter values. In another example, the specific fragmentation of a message as a parameter has a combinatorial explosion in possible parameter values, such that a limited number of specific, likely fault-inducing, parameter values have to be chosen.

Semantically Sound IPM For a given parameter, all possible parameter values must have the same semantic, e.g., they are either all valid or all invalid *within the context of the tested requirement*. However, some combinations of parameter values might change the semantics of another parameter value. For example, in TLS, RSA signatures can only be used if an RSA certificate is used in the connection. We, therefore, carefully model constraints of parameter values and exclude combinations of parameters where the interaction of specific values changes their semantics.

SUT-Specific IPM Real-world implementations commonly do not implement all possible features of a protocol, but only a subset of them. While a specific value might objectively be valid for a given parameter, in the context of a concrete System Under Test (SUT) the parameter might be invalid because the respective feature is not supported. The SUT-specific IPM is therefore generated after the feature extraction.

3.2 Limiting the number of test cases

The subset of inputs defined by the SUT-specific IPM is still too large for exhaustive testing. However, now t-way tests can be used to automatically explore interesting parameter value combinations of the input space while still being able to solve the oracle problem at any time, as the test template knows the semantics of the parameter values.

3.3 Test Suite Execution

The execution of the test suite then follows three phases for each test template: An IPM creation, execution, and validation phase.

IPM Creation In this phase, the test suite determines which features an implementation supports and chooses concrete values for the IPM. After that, constraints are placed on the parameter values of the IPM to exclude semantic changing parameter value combinations.

Template Execution With the previously created IPM, the test suite can then create individual test cases using t-way testing for a provided strength. The generated test cases are then executed.

Validation In the validation phase, the test oracle is executed, which evaluates whether the execution of a test case was successful or not. Due to the constraining steps made before, the evaluation of the test oracle can achieve high reliability. The determined results are then gathered and presented to the user.

4 TLS-Anvil

We implemented the proposed methodology in a test suite called TLS-Anvil. Our test suite is capable of performing black box system tests with TLS clients and servers to evaluate protocol compliance. TLS-Anvil is based on TLS-Attacker [60], TLS-Scanner [61], coffee4j [17], and JUnit 5 [27]. TLS-Attacker was chosen as it provides a large TLS feature set and was written with testing in mind, allowing access to various internals of a TLS connection [58]. We use TLS-Scanner to determine the supported features of a tested library. We then use this information to constrain the parameter values to derive an SUT-specific IPM. Coffee4j was chosen as a framework for t-way testing, as it already implements many popular algorithms. JUnit 5 was chosen as a testing framework because it is state of the art for software testing in Java.

TLS-Anvil consists of an execution framework and a collection of *test templates*. Each test template is implemented as an individual JUnit test that evaluates an individual requirement, such as the compliance to one specific RFC statement. In our analyses, we considered a range of TLS related RFCs [40–52] (see Table 7), as well as known state machine vulnerabilities from scientific literature [54].

4.1 Architecture

The overall architecture of TLS-Anvil is visualized in Figure 1 and follows our methodology from Section 3.

Test Templates The test template defines the messages to be sent to the SUT and their fields based on the requirement. To classify the observed behavior of the SUT for a test case derived from the test template, the template contains a validation function that effectively implements the logic of its test oracle. Since test templates are independent, they can be run in parallel. The body of a test template is focused on the actions of the test, i.e., the exchanged messages, while the parameters and their constraints are provided as Java annotations. Additional metadata, such as a reference to an RFC section, can also be defined using annotations. For the interested reader, we provide an example for the creation of a test template in the TLS-Anvil GitHub repository.[1]

Feature Extraction Before any test templates get executed, TLS-Anvil runs pretests based on TLS-Scanner that collect the supported features of the SUT. This information guides the selection of applicable test templates and the constraining process for the IPMs.

Template Selection Based on the determined capabilities, we filter out test templates that can not be applied at all to the SUT due to a missing feature. After determining applicable test templates, TLS-Anvil continues with the selection of appropriate parameter values for the IPM.

Parameter Value Selection The values for each parameter are selected based on the supported features, as well as on the selected template. For example, if a template wants to test an ECDH feature, the parameter values for the cipher suite parameter will only contain cipher suites supported by the implementation with ECDH key exchange.

Constraining IPM After the values for each parameter were selected, the constraints are added upon the different parameter values, such that the semantics of parameter values do not change. For example, RSA certificates should not be used with ECDSA signature algorithms, as this combination is an illegal selection. Subsequently, the IPM only results in benign values. However, if a test template aims to enforce a specific illegal value (combination), it is possible to disable individual constraints. It is further possible to model a non-conformity as a parameter of the IPM, for example, to test different undefined values for a field. Table 3 summarizes the parameters that typically form an IPM in TLS-Anvil.

Test Case Input Creation Given the final and constrained IPM, we use coffee4j to generate the test inputs. We configured coffee4j to use the `IPOG` algorithm to determine the test inputs, as it was designed for efficient t-way tests with $t > 2$ [35].

Test Template Execution Once coffee4j determined the test inputs, JUnit repeatedly triggers an instantiation of the test template, each time with a different parameter set. This creates many individual test cases. For each test case, TLS-Anvil sets up a configuration to enforce the chosen parameters and a workflow trace for TLS-Attacker, i.e., a sequence of TLS messages to be sent and received. TLS-Anvil then executes the workflow.

Validation Afterward, the template's validation logic is called for the test case, which yields a result of the test oracle. This validation usually consists of two components: a template-specific validation and a call to the validator. On the template level, the test oracle can evaluate message fields specifically affected by the underlying requirement. For example, the test oracle could validate a received signature to ensure that the peer computed it correctly. The validator, in contrast, bundles evaluation steps shared among a larger number of test templates. This includes evaluating the observed message flow with respect to the protocol specification, which is crucial for a comprehensive test oracle. The validator also accounts for the influence of parameter values on the message flow and allowed deviations such as optional messages. The test template can also influence the validator by requesting additional analysis steps, like checking for specific alert messages or the termination of the TCP connection.

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

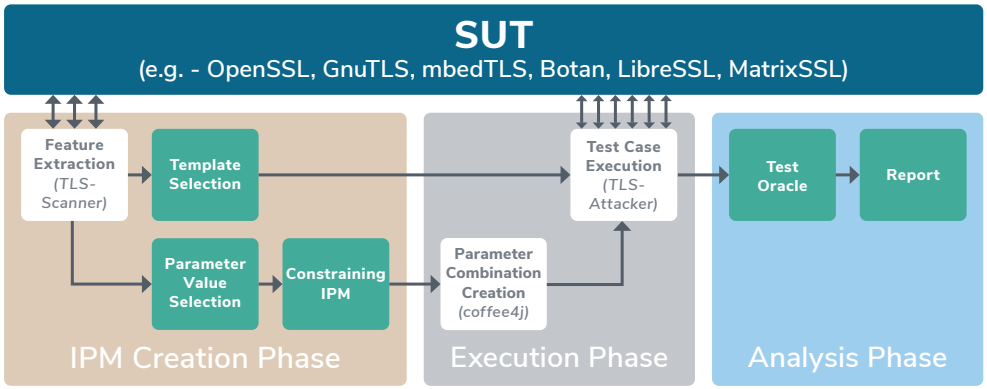

Figure 1: Overview of the execution of a test template in TLS-Anvil beginning with the feature extraction and selection of the template followed by the creation of the constrained IPM, the generation of concrete test inputs based on it, and finally the execution with TLS-Attacker and evaluation through the test oracle. Feature extraction and test case execution steps use state-of-the-art tools – TLS-Scanner and TLS-Attacker – to send TLS messages to the SUT. Elements marked in green depict the contributions of this paper.

When the last set of parameters for a test template has finished, TLS-Anvil composes a conclusive result for the test.

Report Evaluation Once all test templates have been executed, the individual results are collected in a final step that yields a test report. TLS-Anvil provides a small web application to analyze the results of the individual test cases, with the ability to inspect each performed TLS handshake on TCP message level.

4.2 Requirement Selection

In order to write test templates for TLS-Anvil, we had to select requirements for the evaluation.

Explicit Requirements from RFCs As the main source for our test requirements, we manually analyzed 13 TLS RFCs including those that define the most prominent protocol versions, TLS 1.2 and 1.3. A complete list with summaries of the scopes of the individual RFCs can be found in Table 7. Throughout these RFCs, we specifically focused on instances where the standard defines, according to RFC 2119 [39], absolute requirements by using the terms MUST, SHALL, REQUIRED, or absolute prohibitions by using the terms MUST NOT or SHALL NOT. These requirements reduced the risk of misinterpretation by us, as they explicitly describe demands that have to be met by the implementation.

We then filtered out statements of the RFCs that solely affect future extensions of the protocol and are not directed at implementations of the protocol. Furthermore, some requirements may be deprecated by newer RFCs, and therefore were excluded as well. As an example, the TLS 1.2 RFC imposes restrictions on how an implementation has to process SSL2 messages. However, RFC 6176 deprecated SSL2 entirely due to its weak cryptography, rendering these requirements meaningless. Additionally, we filtered requirements that are not testable with our approach. This mostly applies to requirements that can not (reliably) be tested without access to the internals of a library, such as timing-related vulnerabilities [19] but also restrictions imposed on the sender which do not mandate a reaction by the peer if violated. For the scope of the study, we excluded all requirements related to certificates and certificate validation with modified X.509 structures and chains as these have already been in the scope of previous research [11, 16, 29, 66]. We also excluded requirements related to renegotiation and 0-RTT related tests since not all sample implementations provide the required level of control over the client or server to test these requirements properly. We provide an overview of the number of covered requirement keywords per RFC in Section 4.3. Note that we only depict the numbers for MUST and MUST NOT keywords as the other mandatory keywords, REQUIRED and SHALL (NOT), are seldomly used throughout the considered RFCs. Only RFC 6066 makes extensive use of SHALL to impose restrictions on certificate handling.

Length Field Tests A single requirement may mandate the creation of multiple test templates. A prime example of this are tests with manipulated length fields within messages. RFC 5246 defines the requirement to reject such invalid messages only once, but a test template can be written for each individual length field of any TLS 1.2 message. Due to the increased number of semantically very close resulting tests, we subsequently group these length field tests in our overviews.

Implicit Requirements As expected for any standard, not all crucial aspects of the TLS protocol are sufficiently pointed out in the RFCs. For example, the TLS 1.2 RFC states that an

implementation must validate the peer's `Finished` message, including its contained cryptographic checksum. However, it does not mark the verification of a Message Authentication Code (MAC) as mandatory in terms of RFC 2119. Yet the RFC does define an alert to be used upon receiving a message with an invalid MAC resulting in an implicit requirement. We, therefore, also included specific implicit requirements in our test templates. These requirements account for 47 test templates of the 408 test templates we implemented overall.

State Machine Tests While the TLS RFCs define that messages sent outside of the expected order must be rejected, it does not define a concrete state machine for an implementation. For the most recent version, TLS 1.3, a figure of a state machine was included in the RFC but it is far from complete. Ultimately, requirements spread across different RFCs exist that affect the state machine. As explicit requirements exist that mandate tests for the correctness of the state machine, we wrote a set of test templates based both on RFC statements and on known state machine bugs from the literature [54]. As for length field tests, we subsequently group these tests in a separate category.

4.3 Resulting Test Templates

After processing all of the RFCs and related literature mentioned in Section 4.2, our test suite consisted of 408 unique test templates of which 361 were able to use combinatorial testing. The other templates account for requirements for which the behavior of an SUT can be evaluated but not influenced through parameter choices, such as the evaluation of a received `ClientHello`, which is the very first message of the protocol. Table 1 provides a detailed breakdown of the mandatory keywords of the considered RFCs according to our selection process. In Table 6, we also provide an overview of the number of test templates implemented for clients and servers for each RFC.

For the parameters in our test templates we considered various aspects of a TLS connection. These parameters can be simple properties, like the tested cipher suite or if an optional extension is added or not, but can also cover specific properties, for example, the bit position at which the test template invalidates a MAC. In Table 3, we list the parameters that we consider within our IPM's in client and server test templates. A default set of parameters is used if a template does not explicitly define its parameters. A test template can further define value constraints for all parameters available if only some of their values are meaningful in their context. All test templates share a general set of constraints that filter out generally invalid *combinations* of values.

RFC	Σ	✓	×	⊞	⊟	∉	⊠	**%**
5246	118	37	81	10	15	43	13	31.4
8446	255	156	99	2	0	60	37	61.2
8701	7	4	3	0	0	3	0	57.1
7507	3	2	1	0	0	1	0	66.7
6066	22	6	16	1	1	12	2	27.3
7568	3	3	0	0	0	0	0	100.0
7919	12	11	1	1	0	0	0	91.7
7465	1	1	0	0	0	0	0	100.0
7366	2	1	1	0	0	1	0	50.0
8422	34	16	18	1	0	15	2	47.1
7685	1	0	1	0	0	0	1	0.0
6176	1	0	0	0	0	0	1	0.0
7457	0	0	0	0	0	0	0	-
5246	18	11	7	1	2	1	3	61.1
8446	73	38	35	4	0	11	20	52.1
8701	4	2	2	0	0	1	1	50.0
7507	2	1	1	0	0	1	0	50.0
6066	6	2	4	0	0	3	1	33.3
7568	4	4	0	0	0	0	0	100.0
7919	4	3	1	0	0	1	0	75.0
7465	2	2	0	0	0	0	0	100.0
7366	3	1	2	0	0	1	1	33.3
8422	10	7	3	0	0	3	0	70.0
7685	1	1	0	0	0	0	0	100.0
6176	3	2	1	0	0	0	1	66.7
7457	0	0	0	0	0	0	0	-

Table 1: Overview of the categorized MUST (top) and MUST NOT (bottom) keywords. Σ: number of keywords contained overall; ✓: covered; ×: not covered; ⊞: protocol extensions; ⊟: deprecated; ∉: out of scope; ⊠: not testable with our approach; **%**: percentage covered - The other mandatory keywords, REQUIRED or SHALL (NOT), are rarely used in RFCs. Note that RFC 7457 summarizes known attacks without explicit requirements based on these keywords

5 TLS-Docker-Library

Testing TLS libraries, especially in different versions, can result in complex setups and may require conflicting dependencies installed alongside each other. To ease the evaluation process and keep the results reproducible, we created Docker images for open-source libraries which can start the provided example client and server applications. Additionally, we created a Java library that provides an abstraction layer to start these Docker containers using a unified API that works for every implementation. This allowed us to abstract away from the command line parameters each server or client implementation requires to start successfully. These are, for example, the port on which the server listens or the server to which a client implementation should connect to.

The TLS-Docker-Library contains Docker images of 23 different TLS libraries with around 700 different versions in

total. We believe that this project is a valuable resource for the community, independently of the developed test suite. We release TLS-Docker-Library alongside TLS-Anvil. For the interested reader, we provide an example of how to start an OpenSSL server in the TLS-Docker-Library GitHub repository.[2]

6 Evaluation

In order to evaluate TLS-Anvil, we tested open-source TLS libraries with their respective newest labeled versions at the time as listed in Table 4. To remain consistent with related literature, we selected libraries that have been analyzed in various other publications. We further guided our selection to cover implementations in different programming languages, such as Rust and Python. We then tested the provided example client and server implementations in their default configuration.

6.1 Performance and Code Coverage

To benchmark TLS-Anvil, we measured the performance based on the server evaluation of the libraries. We used a virtual machine with 16 cores with a clock speed of 2800 MHz and 16 GB RAM. We provide the complete overview of the execution time and the number of connections required for each library for testing strengths one to three in Table 4. For strength three, the execution time varied between 5.9 and 67.2 hours among the libraries. As expected, the execution time of strength one is much smaller, varying between 0.1 and 2.6 hours.

The benchmark shows two significant outliers. wolfSSL had a very high execution time of 50.4 hours with comparatively few 64079 handshakes. This is due to the behavior of wolfSSL's example server, which frequently terminated, requiring a time-consuming restart of the docker container. mbedTLS had the most individual connections while only supporting TLS 1.2. This is the result of mbedTLS' extensive default configuration that accepted 44 cipher suites and 13 named groups. Since these capabilities significantly define the extent of the implementation-specific IPM, a given test template, in general, required more individual connections for mbedTLS than for any other library to achieve the coverage guarantees of the t-way test.

We also measured the code coverage TLS-Anvil reached for OpenSSL and compared it to the tlsfuzzer of Hubert Kario [62] and TLS Inspector [1] from Achelos to give an idea of the extent of our test suite beyond the number of our test templates. To determine the code coverage, we instrumented the library with kcov [28]. Both TLS Inspector and tlsfuzzer (despite the name) are test suites, while tlsfuzzer sometimes uses a fuzzing approach within its tests. For comparability, we used the OpenSSL server as tlsfuzzer does not support client tests. As for the rest of the evaluation, we used strength $t = 3$ for the combinatorial testing in TLS-Anvil. tlsfuzzer reached a code coverage of 17.4% while TLS-Anvil reached 17.3% and TLS Inspector 14.6%. Note that OpenSSL consists of various cryptographic tools. Consequentially, large parts of the code can not be reached using TLS sessions.

[2] `https://github.com/tls-attacker/TLS-Docker-Library`

6.2 Findings

We divided the test results into four categories: tests that strictly succeeded, tests that conceptually succeeded, and tests that fully or partially failed. Succeeded tests indicate that the SUT performed actions that are in conformance with the specification. Failed tests, on the other hand, indicate that the SUT directly violated the specification and was acting in direct contradiction.

Strictly succeeded A strictly succeeded test means that a library behaved exactly as expected. If multiple test cases are performed during the execution of a test template, the SUT must have behaved correctly across all of them.

Conceptually succeeded A conceptually succeeded test means that an implementation did not precisely fulfill the RFC requirements or did not do so in all test cases but effectively behaved correctly. This usually applies to tests where a fatal alert was expected, but the library either only closed the connection but did not send an alert, or the alert description did not match the RFC's specification.

Partially failed When multiple handshakes are performed for a test template, the partially failed result indicates that not all test inputs failed for a specific test template.

Fully failed A fully failed result means that the SUT did not behave correctly for any test input.

For failed tests, we further analyzed possible reasons using TLS-Anvil's Report Analyzer to ascertain our findings.

6.2.1 Overall Results

We count all tests as 'passed' that either succeeded strictly or conceptually and include the percentage of passed tests in Table 5. We further list the ratio of conceptually to strictly succeeded tests as an additional metric to compare how close an implementation is to the RFC for our tests. Rustls, for example, passed many tests, but most of them only succeeded conceptually as Rustls often did not send any alerts. Botan, in contrast, often fulfilled the expectations of our tests and, at the same time, was very accurate with alert descriptions. We generally found that most libraries pass a high ratio of the test templates, with NSS, BoringSSL, tlslite-ng, and OpenSSL passing around 97% of their applied server tests. Among the client tests, BearSSL, BoringSSL, and Botan have the highest ratio of passed tests with 97.3%, 96.8%, and 96.2%,

respectively. We further expand upon the results of libraries with significantly worse ratios in Section 6.2.2.

In Table 6 we list how many test templates of an RFC passed and how many were executed for each library. We grouped the results of test templates based on similar error cases and identified a total of 239 issues. We further categorized these findings based on their impact and determined that three immediately led to exploits in wolfSSL and MatrixSSL. Additionally, we found five issues affecting the cryptography of a handshake. As an example, the clients of MatrixSSL, s2n, and wolfSSL are willing to negotiate parameters they did not offer. While none of the parameters negotiated are (sufficiently) weakening the security to pose an immediate threat now, parameter negotiation is a basic security property of every cryptographic protocol to prevent current and potential future attacks. We further identified 15 issues affecting the interoperability to an extent where a peer that operates within the boundaries of the RFC may not be able to complete a handshake. Note that this may also include intentional deviations by the developers if they break the implementation's correctness in regards to the specification. 100 issues account for various likely uncritical cases where a library deviated from the RFC beyond alert codes and where interoperability should not be affected. Examples of these findings are a bug in OpenSSL, which allowed multiple TLS 1.3 `HelloRetryRequest` messages, which can keep the client in a handshake loop, or the support of deprecated curves by mbedTLS. Finally, we grouped 116 cases where a library did not send an alert or sent a different alert than requested by the RFC. These are minor deviations from the standard. However, in the past, information gained from the type of alert sent by an implementation has been used to mount side-channel attacks [10, 36]. To avoid such deviations, great care must be taken when designing the alert handling of an implementation. We hence chose to include these findings in our reports to the developers. We describe the more severe findings in Section 6.2.3 and present the number of findings for each evaluated library based on the above categorization in Table 2.

Library	Exploit	Crypto	Interop.	Alerts	Other
BearSSL	0	0	1	15	4
BoringSSL	0	0	0	6	3
Botan	0	0	0	3	3
GnuTLS	0	0	1	9	10
LibreSSL	0	1	1	7	6
MatrixSSL	2	2	7	6	16
mbed TLS	0	0	1	14	5
NSS	0	0	0	7	6
OpenSSL	0	0	0	6	7
Rustls	0	0	1	15	7
s2n	0	1	0	13	12
tlslite-ng	0	0	0	2	10
wolfSSL	1	1	3	13	11
	3	5	15	116	100

Table 2: Overview of the findings for each library with results categorized based on their impact. We distinguish between immediately exploitable issues (Exploit) and issues that illicitly affect the cryptographic computations in a session (Crypto). The category 'Other' accounts for a vast range of findings that do not affect the security or interoperability, such as minor state machine bugs or support for deprecated (but safe) features. 'Alerts' refers to alert handling and is a category separated from these findings as the deviations are marginal but may result in a vulnerability under certain circumstances.

6.2.2 Outliers of the Evaluation

As can be seen in Table 5, GnuTLS, MatrixSSL, s2n, and wolfSSL clients only passed comparatively few tests. In the case of GnuTLS and wolfSSL this is due to an intolerance towards record fragmentation. Record fragmentation is a mechanism that allows an implementation to split TLS messages into smaller fragments, containing at least a single byte [40]. These fragments are sent in independent TLS records. Both implementations, GnuTLS and wolfSSL, failed to process tiny records, particularly records that held only a single byte of the message payload. As record fragmentation is a parameter of the combinatorial testing, not all inputs resulted in a success for a test case. This is evident from the large number of tests that only failed partially. Excluding record fragmentation, the results are closer to other implementations, as seen by the number of entirely failed tests.

s2n has a large number of tests that failed completely. This is due to s2n's error handling; when an error occurs, or misbehavior of the peer is detected, s2n sends an alert but does not close the connection. As far as we evaluated this state, it is impossible to send further handshake messages or application data, but this requires a detailed analysis for each test case. The connection should be closed properly to facilitate the evaluation and ensure that it is impossible to continue a session.

Due to incorrect handling of `HelloRetryRequest` message flows with the `TLS_AES_256_GCM_SHA384` cipher suite, the MatrixSSL client often failed to complete handshakes. Since tests in TLS 1.3 usually require a `HelloRetryRequest` to enforce a key exchange group other than the default group of the client, this intolerance affects the results of many test cases.

6.2.3 Detailed Findings

wolfSSL We found a directly exploitable authentication bypass for wolfSSL clients when TLS 1.3 is used. An attacker can force the client to accept a TLS connection by sending a `Certificate` message with an empty certificate list. The wolfSSL client then ignores the `CertificateVerify` mes-

sage and accepts any unauthenticated key share it received. wolfSSL's behavior is a direct violation of an RFC requirement that demands a fatal alert upon receiving an empty `Certificate` message in TLS 1.3. While our evaluation was still ongoing, the bug had already been independently discovered, reported[3] and fixed in version 4.7.0.

Our evaluation also revealed that the wolfSSL client accepted a signature algorithm it did not offer. This only applied to ECDSA_SHA224 and hints towards an internal misunderstanding, as the RSA_SHA224 algorithm and the named group secp224r1 are among the offered parameters. Additionally, the server implementation showed an intolerance towards `ClientHello` messages that contain more than 150 cipher suites. This may become an interoperability issue for feature-rich clients.

MatrixSSL We discovered a segmentation fault[4] in MatrixSSL. The problem occurred only in cipher suites that use HMAC SHA256 CBC, due to incorrect initialization of SHA256 for the Lucky13 mitigation [19] when a record with invalid padding is received. An attacker could exploit this vulnerability to retrieve information about the validity of padding, leading to a padding oracle attack [64].

Our test results further showed that the MatrixSSL client mishandled length fields. In TLS 1.3, messages with an up to two-byte reduced handshake message length field resulted in an infinite loop during the message parsing. This behavior could be exploited in a denial of service attack. In TLS 1.2, MatrixSSL proceeded to parse handshake messages correctly despite the invalid handshake message length field. Because negotiating an extended master secret was modeled as a parameter of our combinatorial testing, we noticed an interesting behavior in how the length field was used. When both parties send the corresponding extension, all handshake messages exchanged influence the key calculation. While MatrixSSL seemingly ignored the length field for parsing, its value determined which bytes influence the digest causing a discrepancy between the actual parsed message and the message included in the digest, which we categorize as a 'Crypto' finding in our overview. However, we currently do not see a way to exploit this behavior.

The MatrixSSL client also accepted `ServerKeyExchange` messages that negotiate a group that has not been offered in the `ClientHello`. This applies to the groups secp192r1 and secp224r1, which are deprecated by RFC 8422 and 8446 and have comparatively weak parameters. For a developer, it is not evident that the client accepts these weaker curves based on an analysis of the `ClientHello` message. This misbehavior results in the second 'Crypto' finding for MatrixSSL

Furthermore, we discovered an interoperability issue in MatrixSSL server where it could not process a `ClientHello` if a client offered more than 32 named groups for the key exchange.

LibreSSL We noticed that it is possible to send an additional encrypted `ChangeCipherSpec` message after completing a TLS 1.2 handshake with Libressl. Further analysis revealed that this triggered a bug in LibreSSL, which caused it to start decrypting incoming messages with its own write key, such that the same key is used in both directions. The bug is very similar to a bug found in OpenSSL by de Ruiter and Poll in 2015 [54]. This is another example of a 'Crypto' finding. We further observed that the LibreSSL server is not able to process a `SignatureAlgorithms` extension that includes more than 32 algorithms.

s2n Like MatrixSSL, the s2n client accepts a `ServerKeyExchange` message that negotiates an unproposed named group. In this case, this only applies to the groups X25519 and secp521r1, which are not deprecated and have strong parameters. Nevertheless, we stress that even these particular groups could be targeted in the future. Secure parameter negotiation is a fundamental security property of every cryptographic protocol that prevents future attacks on particular algorithms.

6.3 t-way Testing

Among the failed tests of individual libraries, we found multiple cases where a test template only failed partially. Effectively, this means that a failure only surfaced for specific parameter choices. This concerns less crucial compliance and interoperability issues, the negotiation of unproposed cryptographic primitives, but most importantly, the padding oracle vulnerability in MatrixSSL. Our evaluation identified 29 of these partially failed tests across all evaluated libraries.[5] We argue that these test templates in particular justify the concept of CT, as a traditional software test would require multiple manually written tests aimed at these specific parameter choices to uncover the same issue.

Testing Strength While our evaluation used a testing strength of three, all of the failures we found were caused by a combination of at most two parameters. This means that a lower test strength with fewer handshakes overall can be used to identify the same faults. Given the extent of our parameters, even an execution with strength one may uncover all faults, but it is not guaranteed that all relevant combinations appear in the test inputs, and therefore some flaws might be missed. At the same time, a strength above three may uncover additional, more complex faults not yet observed in our evaluation.

[3] CVE-2021-3336
[4] CVE-2022-23809

[5] We excluded partially failed tests of wolfSSL, MatrixSSL, and GnuTLS caused by prevalent bugs.

7 Discussion

Our evaluation revealed a vast range of implementation failures which proves that our methodology can be beneficial to the quality and security of future versions of the analyzed libraries. Still, other testing approaches remain necessary for a complete picture of an implementation's quality.

Limitations Since our methodology focuses on black box system tests, limitations to the testable requirements apply. This mostly affects requirements that can not or not reliably be tested without access to the internals of the implementation. Additionally, the SUT must have all features enabled that should be tested. In contrast to software verification, CT can only find flaws that affect code that is actually executed. In order to do so, it is essential that all relevant parameters are modeled and that the values considered by CT also cover relevant corner cases. Both parameters and their values, require *some* amount of intuition of which aspects of an input can potentially cause software faults. Additionally, the design of a test template for a given requirement is not bijective, leaving some decisions to the tester.

Another drawback of our IPM constraining methodology is that it always tests with only one invalid parameter. A combination of multiple invalid parameters may yield a more severe implementation error than the one detected with a single nonconformity but makes the implementation of a reliable test oracle harder. The applicability of our approach is eventually also limited by the precision of the tested requirement. If implicit limitations exist in the standard, which are not reflected by an explicit requirement, there is a chance that they are missed in the testing process.

Finally, we created our test templates manually after systematic analyses of TLS RFCs. To extend TLS-Anvil to consider new tests (e.g., tests with modified X.509 certificates), further manual analysis is necessary. Unfortunately, there are no practical, fully automatic algorithms to generate tests from specifications [13]. However, we want to stress that while the test template generation demands a significant amount of work, this work only needs to be performed once. The resulting test templates can then be used to test different libraries in different versions and can be included in continuous testing frameworks.

RFC Discussion In our case study, we noticed shortcomings which ultimately hindered tests for some requirements and contribute to the difficulty of automating the derivation of test templates. First of all, the RFCs use the keywords from RFC 2119 to emphasize absolute requirements but do not do so consistently. Coming back to the example from Section 4.2, the TLS 1.2 RFC never explicitly uses a mandatory keyword to point out the importance of MAC validation. The same applies to the validation of signatures computed by a TLS server. However, these values are a crucial part of the security of TLS and, therefore must be validated. Shortcomings like these must be considered to obtain an extensive and meaningful test suite. The newer TLS 1.3 RFC is already clearer in this regard. Within the considered RFCs, restrictions were often only imposed on the sender of a message without specifying what the recipient must do upon detecting a violation. A requirement to enforce these restrictions whenever possible would reduce the leeway and hence enable meaningful tests.

Ultimately, each requirement necessitates an analysis if it is suitable for the testing approach used. This task can be far from trivial as deep domain knowledge may be necessary to determine this categorization correctly.

Responsible Disclosure and Feedback We responsibly disclosed all of our findings to the respective developers. During the disclosure process, multiple developers stated that they intentionally violate RFC requirements in specific cases. As an example, in TLS 1.2 peers are not allowed to resume a session that has been terminated by a fatal alert. However, when multiple sessions take place in parallel, this requirement is difficult to implement. Multiple developers also stated that they intentionally send different alerts or no alerts at all. One reason was to minimize the risk of creating an alert-oracle for attacks. We do, however, stress that the specified alert handling of current TLS RFCs does not result in a known, exploitable oracle but is considered to be secure and refer to our discussion in Section 6.2. Our original test suite contained 18 additional test templates, which we removed from the test suite after discussions with different library developers. Their reasoning convinced us that in these cases our interpretation of the RFC was too strict and that their library behavior was indeed valid. Our presented evaluation does not contain these additional test templates anymore. There were also some cases where the developers argued that it is unreasonable to follow the specification. For example, in some tests, a server that supports TLS 1.3 and TLS 1.2 would need different alert handling for the same effective test. The situation becomes even more tricky when the server has not decided which protocol version to speak yet. For example, a server that receives a malformed or illegal `ClientHello` message would first need to evaluate the supported protocol version of the client to decide upon the correct alert handling rules. Correctly handling these nuances can be very complex, and it is arguable if the strict RFC conformance across all supported versions is worth the added complexity.

The security bugs we reported and most of our other reports have been acknowledged by the developers and will be considered for future releases. Since most failed test templates only failed for single or very few libraries, we conclude that the developers in general share our understanding of the RFCs.

8 Related Work

TLS Testing There are several tools to test TLS libraries. TLS-Attacker is a framework to analyze TLS libraries. So-

morovsky published the first version of TLS-Attacker in 2016 [58]. He presented basic concepts behind the framework and showed how to implement known TLS attacks and simple fuzzing strategies. Somorovsky also showed that it is generally possible to develop test suites with TLS-Attacker, by implementing proof of concept system tests. The preliminary testing strategies did not include CT techniques or comprehensive test oracles for them. We use an extended version of TLS-Attacker in TLS-Anvil. Similar to TLS-Attacker, FLEXTLS [6, 20] is a framework to build tools to test TLS libraries. It was used to find vulnerabilities like FREAK or SKIP [5]. FLEXTLS is no longer maintained.
Another testing library is Boarssl [9], which is a test suite that is developed alongside of Bearssl. Boarssl is used in the Twrch framework, mainly to test Bearssl. 'tlsfuzzer' [62] is, despite its name, a test suite for TLS implementations that implements test cases for TLS servers. Among those test cases are explicit RFC requirements and specific test cases to prevent regressions for previous discovered bugs. Another test suite is implemented in the commercial tool 'TLS Inspector' [1] by the company Achelos. TLS Inspector implements test cases derived from selected requirements of the RFCs and the guideline TR-03116-4[25] of the German Federal Office for Information Security, which focuses on additional explicit requirements for secure TLS configurations.

Combinatorial Testing The approach of t-way testing of TLS libraries is not new. In [23, 56, 57] t-way testing was used to generate variations of the TLS handshake with different parameters. The authors then used a reference implementation as a test oracle and compared the behavior of different implementations. These approaches were able to show that different implementations behaved differently in response to the generated inputs. This approach is generally not suitable for complex protocols like TLS, as oftentimes multiple answers should be considered valid by the test oracle for a given input. For example, if the SUT and the reference library support different algorithms, connection termination or handshake continuation are both valid responses to a given `ClientHello`. This produces unmanageable amounts of false positives during an automated evaluation, which showed in the authors' inability to convert their observed behavior differences into actionable findings. In [29] combinatorial testing techniques were used together with differential testing methods to test X.509 certificate validation. They then manually analyzed the discovered differences for vulnerabilities.

Further Automatic Test Input Generation Techniques Since generating test inputs is a crucial aspect of testing, various studies analyzed techniques to fulfill this task. Over the recent years, fuzzing has been widely used in practice to find inputs that crash software or lead to memory corruption. Current research in this field focuses on optimization strategies [14, 34] that yield a higher coverage within less execution time. Another technique for the generation of test inputs is based on symbolic execution. In contrast to using specific test inputs, symbolic execution first uses an interpreter that analyzes the branches within the software using symbolic variables. This ultimately allows for exhaustive branch coverage, but the efficiency of the approach is limited by the complexity of the software [33]. In the context of TLS, symbolic execution has been successfully used by Chau et al. [12] to evaluate compliance of the X.509 certificate validation logic in TLS libraries.

Requirements Engineering A sub-discipline in software engineering is Requirements Engineering (RE) [26], with the goal to collect, document, validate, and manage requirements for a given system. Automatically extracting requirements from documents written in Natural Language and the deduction of program code from the extracted requirements has proven to be very difficult. A study for RFC-based X.509 testing in the TLS landscape by Chen et al. [13] developed algorithms to generate tests from RFCs but ultimately had to fall back to performing steps manually due to limitations in the current state of Natural Language Processing. Their approach also focused on passages of the RFCs marked as absolute requirements based on the terminology of RFC 2119 [39]. A subsequent study by Chen et al. tested the applicability of the methodology to developer guides of payment services [15]. Again, a fully automated approach was not feasible, and manual work was required.

9 Conclusions and Future Work

In this work, we presented a methodology to conduct combinatorial testing for complex protocols such as TLS. Our evaluation of the technique revealed 239 issues overall, including 15 interoperability issues, five issues that illicitly affect cryptographic computations, and three immediately exploitable vulnerabilities. Coming back to our first research question, we determine that multiple TLS libraries still show a range of unintended deviations from the protocol specification, even to the extent of exploitable vulnerabilities. While these individual findings are surprising, given the extensive research and testing that has been conducted for the considered libraries in the past, we also found that the overall compliance to the specification is high. As for our second and third research question that concern the applicability of CT and effectiveness of our approach of utilizing CT, we found that our test oracles based on test templates produced actionable results and are thus better suited than the reference implementation-based approaches of previous studies. These resulted in unmanageable differences between TLS libraries requiring further manual analysis, which ultimately could not be converted into real software faults.

While TLS-Anvil already considers a large number of requirements from the RFCs, the TLS protocol contains even more features with testable requirements that could be in-

corporated. Additionally, our evaluation only considered the default configuration of TLS libraries. These can often be configured at build time to use certain features and optimizations which might influence their behavior. It may be of interest to include such build parameters in the IPMs used for the combinatorial testing to also find flaws that are only present in specific configurations. Furthermore, TLS related protocols like DTLS and QUIC could be tested with TLS-Anvil in the future. Generally, the presented methodology is suitable for standardized protocols with many different implementations (like TLS, SSH or IPSEC). While it is also applicable to protocols with only a single implementation, it is more cost efficient to write individual tests or test templates for that specific implementation since the presented black box approach requires preliminary steps, such as the feature extraction.

Our evaluation showed that security-critical bugs, such as the authentication bypass for wolfSSL, can be found in widely used TLS libraries based on testing RFC requirements. The ability to test these requirements strongly depends on the tester's ability to identify these requirements. We, therefore, recommend that future specifications continue the trend set by TLS 1.3 to mark requirements as precise as possible. The TLS 1.3 specification was also already implemented alongside of the writing process; for future specifications, it may prove valuable to also develop a test suite like TLS-Anvil alongside of the specification. The developed test suite could serve as an authority on how the RFCs, which are written by humans for humans, have to be interpreted by machines, and ensure that all implementations are, and stay interoperable to one another. It would also increase the confidence of developers in their implementation, which would speed up the development process and ultimately help to adapt newer standards quicker.

Acknowledgments

We would like to thank Malena Ebert, Jan Kaiser, Joeri de Ruiter, Jan Drees, Nimrod Aviram, and Filippo Valsorda for their contribution to the TLS Docker Library. We further thank Achelos for providing us access to TLS Inspector. This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 450197914 and under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Availability

TLS-Anvil and TLS-Docker-Library are available as open-source projects under the Apache 2.0 license at `https://github.com/tls-attacker/TLS-Anvil` and `https://github.com/tls-attacker/TLS-Docker-Library`.

References

[1] *Achelos TLS Inspector*. `https://www.achelos.de/de/tls-inspector.html`.

[2] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. "On the Security of RC4 in TLS". In: *22nd USENIX Security Symposium (USENIX Security 13)*.

[3] 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: *USENIX Security Symposium 2016*.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Trans. Software Eng.* 5 (2015).

[5] 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: *2015 IEEE Symposium on Security and Privacy*.

[6] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. "FLEXTLS: A Tool for Testing TLS Implementations". In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*.

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

[8] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference*.

[9] *BoarSSL*. `https://www.bearssl.org/boarssl.html`.

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

[11] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*.

[12] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations". In: *2017 IEEE Symposium on Security and Privacy*.

[13] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. "RFC-directed differential testing of certificate validation in SSL/TLS implementations". In: *Proceedings of the 40th International Conference on Software Engineering ICSE 2018*.

[14] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy, SP 2018*.

[15] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. "Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis". In: *USENIX Security Symposium 2019*.

[16] Yuting Chen and Zhendong Su. "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*.

[17] *coffee4j*. `https://coffee4j.github.io/`.

[18] Thai Duong and Juliano Rizzo. *Here come the XOR ninjas*.

[19] N. J. Al Fardan and K. G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*.

[20] *FlexTLS*. `https://github.com/DinoTools/python-flextls`.

[21] Konrad Fögen, "Combinatorial Robustness Testing based on Error-Constraints", PhD thesis, RWTH Aachen University, Germany, 2021

[22] 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*.

[23] Bernhard Garn, Dimitris E. Simos, Feng Duan, Yu Lei, Josip Bozic, and Franz Wotawa. "Weighted Combinatorial Sequence Testing for the TLS Protocol". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

[24] William E. Howden. "Theoretical and Empirical Studies of Program Testing". In: *Proceedings of the 3rd International Conference on Software Engineering, 1978*.

[25] German Federal Office for Information Security (BSI). "TLS nach TR-03116-4 - Checkliste für Diensteanbieter". In: *TR-03116-4*.

[26] "ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes –Requirements engineering". In: *ISO/IEC/IEEE 29148:2011(E)* (2011).

[27] *JUnit 5*. `https://junit.org/junit5/`.

[28] *kcov*. `https://github.com/SimonKagstrom/kcov`.

[29] Kristoffer Kleine and Dimitris E. Simos. "Coveringcerts: Combinatorial Methods for X.509 Certificate Testing". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*.

[30] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. "A Model for T-Way Fault Profile Evolution during Testing". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017*.

[31] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. "Software Fault Interactions and Implications for Software Testing". In: *IEEE Trans. Software Eng.* 6 (2004).

[32] D.R. Kuhn and M.J. Reilly. "An investigation of the applicability of design of experiments to software testing". In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*

[33] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*.

[34] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. "Constraint-guided Directed Greybox Fuzzing". In: *USENIX Security Symposium 2021*.

[35] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. "IPOG: A General Strategy for T-Way Software Testing". In: *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007)*.

[36] 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*.

[37] Stacy J. Prowell and Jesse H. Poore. "Foundations of Sequence-Based Software Specification". In: *IEEE Trans. Software Eng.* 5 (2003).

[38] Zachary B. Ratliff, D. Richard Kuhn, Raghu N. Kacker, Yu Lei, and Kishor S. Trivedi. "The Relationship between Software Bug Type and Number of Factors Involved in Failures". In: *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*.

[39] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119

[40] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246

[41] D. Eastlake 3rd, *Transport Layer Security (TLS) Extensions: Extension Definitions*, RFC 6066

[42] S. Turner and T. Polk, *Prohibiting Secure Sockets Layer (SSL) Version 2.0*, RFC 6176

[43] P. Gutmann, *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*, RFC 7366

[44] Y. Sheffer, R. Holz, and P. Saint-Andre, *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*, RFC 7457

[45] A. Popov, *Prohibiting RC4 Cipher Suites*, RFC 7465

[46] B. Moeller and A. Langley, *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*, RFC 7507

[47] R. Barnes, M. Thomson, A. Pironti, and A. Langley, *Deprecating Secure Sockets Layer Version 3.0*, RFC 7568

[48] A. Langley, *A Transport Layer Security (TLS) ClientHello Padding Extension*, RFC 7685

[49] D. Gillmor, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, RFC 7919

[50] 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

[51] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446

[52] D. Benjamin, *Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility*, RFC 8701

[53] Juliano Rizzo and Thai Duong. "The CRIME attack". In: *Ekoparty Security Conference 2012*.

[54] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *USENIX Security Symposium 2015)*.

[55] Rolf Schwitter. "English as a Formal Specification Language". In: *13th International Workshop on Database and Expert Systems Applications (DEXA 2002)*.

[56] Dimitris E. Simos, Josip Bozic, Feng Duan, Bernhard Garn, Kristoffer Kleine, Yu Lei, and Franz Wotawa. "Testing TLS Using Combinatorial Methods and Execution Framework". In: *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017*.

[57] Dimitris E. Simos, Josip Bozic, Bernhard Garn, Manuel Leithner, Feng Duan, Kristoffer Kleine, Yu Lei, and Franz Wotawa. "Testing TLS using planning-based combinatorial methods and execution framework". In: *Softw. Qual. J.* 2 (2019).

[58] Juraj Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[59] *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. `https://www.openssl.org/~bodo/ssl-poodle.pdf`.

[60] *TLS-Attacker*. `https://github.com/tls-attacker/TLS-Attacker`.

[61] *TLS-Scanner*. `https://github.com/tls-attacker/TLS-Scanner`.

[62] *tlsfuzzer*. `https://github.com/tomato42/tlsfuzzer`.

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

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

[65] David Wagner and Bruce Schneier. "Analysis of the SSL 3.0 Protocol". In: *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*.

[66] Jiayu Zhu, Chengcheng Wan, Pengbo Nie, Yuting Chen, and Zhendong Su. "Guided, Deep Testing of X.509 Certificate Validation via Coverage Transfer Graphs". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Appendix

Parameter	SUT	Version	Values
Cipher Suite	●	●	3 - 44
Named Group	●	●	2 - 13
Signature Algorithm[5]	◐	●	5 - 15
Certificate[2,5]	◐	●	4 - 18
Record Fragment Length	●	●	4
TCP Fragmentation	●	●	bool
Session Ticket[1,3]	◑	●	bool
Extended Master Secret[1,3]	●	●	bool
Encrypt-then-MAC[1,3]	●	●	bool
ALPN[3]	◑	●	bool
GREASE Cipher Suites[4]	◑	●	bool
GREASE Named Groups[4]	◑	●	bool
GREASE Signature Algs[4]	◑	●	bool
Heartbeat[3]	◑	●	bool
Padding[3]	◑	●	bool
Renegotiation Indication[3]	◑	●	bool
PSK Key Exch. Modes[3]	◑	◓	bool
Send legacy CCS	●	◓	bool
Additional Padding Bytes	●	◓	3
Alert Description	●	●	34
Application Data Length	●	●	16
Manipulated AEAD Tag Byte	●	●	16
Manipulated Ciphertext Byte	●	●	16
Manipulated MAC Byte	●	◒	32 - 48
Manipulated PRF Byte	●	●	32 - 48
Manipulated Signature Byte	◐	●	5 - 10
Manipulated Bit in Chosen Byte	●	●	8
Unproposed Compression Method	◐	●	2
Selected Grease Cipher Suite	●	●	16
Selected Grease Extension	●	●	16
Selected Grease Named Group	●	●	16
Selected Grease Protocol Version	●	●	16
Selected Grease Signature Algo	●	●	16
Invalid `ChangeCipherSpec`	●	◒	4
Protocol Message Type	●	●	6
Invalid Protocol Version	●	●	1 - 4

Used to test connection endpoint ◐ client ◑ server ● both
Used in version ◒ TLS 1.2 ◓ TLS 1.3 ● both
1: TLS 1.2 extensions have only been included in TLS 1.3 test templates for server tests, as they are generally not applicable to client tests in this version
2: For client tests, we used only valid certificates with different public key types, signatures, and key lengths
3: Extension included or excluded from handshake messages - extensions have only been negotiated in client tests if the client offered them
4: Appended to offered parameters
5: Parameter is only used if any message it affects gets sent by the test template

Table 3: The parameters included commonly (top) and in specific (bottom) IPMs created by TLS-Anvil. The number of parameter values varies, depending on the TLS features supported by the SUT.

		Strength $t = 3$		Strength $t = 2$		Strength $t = 1$	
Library	**Version**	**Execution Time**	**Connections**	**Execution Time**	**Connections**	**Execution Time**	**Connections**
BearSSL	0.6	19.1h	61253	3.7h	12088	0.5h	1825
BoringSSL	3945	14.8h	48929	3.4h	10587	0.6h	1844
Botan	2.17.3	6.1h	26394	1.3h	5485	0.3h	965
GnuTLS	3.7.0	31.2h	88730	6.1h	17328	0.9h	2726
LibreSSL	3.2.3	38.4h	121650	7.7h	25600	1h	3869
MatrixSSL	4.3.0	20.8h	57598	5.1h	12777	1.1h	2541
mbed TLS	2.25.0	67.2h	181265	9.6h	35087	0.9h	4041
NSS	3.60	33.6h	91521	7h	18774	1h	2922
OpenSSL	1.1.1i	31.2h	95379	5.7h	18522	0.8h	2861
Rustls	0.19.0	13.6h	30761	3.4h	7517	0.1h	568
s2n	0.10.24	5.9h	26669	1.4h	5640	0.3h	1023
tlslite-ng	0.8.0-a39	55.2h	118167	8.7h	22784	1.2h	3389
wolfSSL	4.5.0	50.4h	64079	11.5h	14618	2.6h	2986

Table 4: Overview of the number of connections and the execution time for each server implementation with different testing strengths.

	Library	**Tests**	**Passed**	**Passed [%]**	**Strictly succeeded**	**Conceptually succeeded**	**Ratio conceptually**	**Failed partially**	**Failed fully**
Servers	BearSSL	109	102	93.6	48	54	52.9	1	6
	BoringSSL	219	213	97.3	202	11	5.2	0	6
	Botan	101	97	96.0	93	4	4.1	0	4
	GnuTLS	240	229	95.4	221	8	3.5	3	8
	LibreSSL	222	213	95.9	204	9	4.2	0	9
	MatrixSSL	222	173	77.9	159	14	8.1	10	39
	mbed TLS	113	105	92.9	80	25	23.8	1	7
	NSS	225	220	97.8	205	15	6.8	0	5
	OpenSSL	237	230	97.0	223	7	3.0	0	7
	Rustls	219	209	95.4	191	18	8.6	1	9
	s2n	104	95	91.3	44	51	53.7	1	8
	tlslite-ng	239	233	97.5	228	5	2.1	1	5
	wolfSSL	219	98	44.7	63	35	35.7	112	9
Clients	BearSSL	75	73	97.3	27	46	63.0	0	2
	BoringSSL	190	184	96.8	172	12	6.5	0	6
	Botan	78	75	96.2	74	1	1.3	2	1
	GnuTLS	201	128	63.7	120	8	6.3	62	11
	LibreSSL	193	178	92.2	169	9	5.1	1	14
	MatrixSSL	193	133	68.9	113	20	15.0	40	20
	mbed TLS	82	78	95.1	66	12	15.4	0	4
	NSS	197	184	93.4	181	3	1.6	0	13
	OpenSSL	197	184	93.4	181	3	1.6	0	13
	Rustls	188	178	94.7	128	50	28.1	2	8
	s2n	73	38	52.1	26	12	31.6	0	35
	tlslite-ng	203	180	88.7	173	7	3.9	3	20
	wolfSSL	198	105	53.0	67	38	36.2	85	8

Table 5: Overview of the results of the test templates for each tested library for strength $t = 3$. The columns on the left summarize the number of passed test templates, while the columns on the right state more detailed results.

		8446	5246	8701	8422	7919	7568	6066	7507	7465	7366	6176	7685	7457	State Machine	Length Field
	Test Templates	97	46	10	14	5	3	6	2	2	2	1	1	0	25	44
Servers	BearSSL	0/0	41/43	5/5	9/13	1/1	3/3	3/4	2/2	0/0	0/0	1/1	1/1	0/0	17/17	19/19
	BoringSSL	77/82	42/43	10/10	13/13	1/1	3/3	1/1	2/2	0/0	0/0	1/1	1/1	0/0	24/24	38/38
	Botan	0/0	34/35	5/5	10/13	4/4	3/3	1/1	0/0	0/0	0/0	1/1	1/1	0/0	18/18	20/20
	GnuTLS	88/91	39/43	10/10	11/13	4/4	3/3	3/4	2/2	0/0	2/2	1/1	1/1	0/0	24/24	41/42
	LibreSSL	78/82	41/43	10/10	13/13	2/2	3/3	1/1	2/2	0/2	0/0	1/1	1/1	0/0	23/24	38/38
	MatrixSSL	58/83	35/43	10/10	9/12	1/1	2/3	4/4	0/0	0/0	0/0	1/1	1/1	0/0	20/22	32/42
	mbed TLS	0/0	40/43	5/5	10/13	2/2	3/3	2/4	2/2	0/0	2/2	1/1	1/1	0/0	18/18	19/19
	NSS	80/83	42/43	10/10	13/13	4/4	3/3	1/1	2/2	1/2	0/0	1/1	1/1	0/0	24/24	38/38
	OpenSSL	86/90	43/43	10/10	10/13	2/2	3/3	4/4	2/2	0/0	2/2	1/1	1/1	0/0	24/24	42/42
	Rustls	83/88	34/36	10/10	11/12	1/1	3/3	1/1	0/0	0/0	0/0	1/1	1/1	0/0	24/24	40/42
	s2n	0/0	37/41	5/5	10/12	1/1	3/3	1/1	2/2	0/0	0/0	1/1	1/1	0/0	17/18	17/19
	tlslite-ng	88/91	42/43	10/10	11/13	4/4	3/3	1/1	2/2	0/0	2/2	1/1	1/1	0/0	24/24	44/44
	wolfSSL	27/82	18/39	0/10	3/12	1/4	3/3	0/1	2/2	0/0	0/2	1/1	0/1	0/0	7/24	36/38
	Test Templates	90	35	14	11	4	1	7	1	1	1	2	1	1	15	29
Clients	BearSSL	0/0	35/35	5/5	9/10	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	7/7	10/11
	BoringSSL	80/86	35/35	14/14	9/9	0/0	1/1	0/0	1/1	1/1	0/0	2/2	0/0	1/1	15/15	25/25
	Botan	0/0	31/31	5/5	8/9	3/4	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	6/7	15/15
	GnuTLS	33/88	31/35	6/14	9/9	4/4	1/1	1/1	1/1	1/1	1/1	2/2	0/0	1/1	9/15	28/28
	LibreSSL	78/86	33/35	10/14	9/9	2/2	1/1	0/0	1/1	0/1	0/0	2/2	0/0	1/1	15/15	26/26
	MatrixSSL	51/87	30/35	8/14	8/10	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	14/15	15/25
	mbed TLS	0/0	33/35	4/5	9/10	2/2	1/1	1/1	1/1	1/1	1/1	2/2	0/0	1/1	7/7	15/15
	NSS	80/87	34/35	10/14	8/9	2/2	1/1	0/0	1/1	1/1	1/1	2/2	0/0	1/1	15/15	28/28
	OpenSSL	80/87	34/35	10/14	8/9	2/2	1/1	0/0	1/1	1/1	1/1	2/2	0/0	1/1	15/15	28/28
	Rustls	78/87	31/31	14/14	9/9	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	14/15	25/25
	s2n	0/0	19/35	2/5	3/8	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	7/7	0/11
	tlslite-ng	72/88	34/35	9/14	9/9	4/4	1/1	1/1	1/1	1/1	1/1	2/2	1/1	1/1	14/15	29/29
	wolfSSL	30/87	22/35	6/14	5/8	2/4	1/1	0/0	1/1	1/1	0/1	2/2	0/0	1/1	7/15	27/28

Table 6: Overview of the number of implemented, executed, and passed test templates for each RFC or category for strength $t = 3$. Client and server test templates are not disjoint sets. Our test suite consists of 408 unique test templates.

RFC	Description
5246 [40]	Primary TLS 1.2 standard, defining handshake, record layer, mandatory cipher suites, basic extensions, and client/server behavior in unexpected situations
8446 [51]	Primary TLS 1.3 standard, defining handshake, record layer, mandatory cipher suites, basic extensions, and client/server behavior in unexpected situations
6066 [41]	Specifies new extensions for TLS
6176 [42]	Deprecates SSL 2 and forbids the fallback to this protocol
7366 [43]	Specifies an Encrypt-then-MAC extension in response to CBC padding oracle attacks [19]
7457 [44]	Summarizes known attacks on TLS and DTLS
7465 [45]	Prohibits usage of RC4 following papers on breaking the stream cipher [2, 22, 63]
7507 [46]	Specifies downgrade protection mechanisms
7568 [47]	Deprecates SSL 3 and forbids the fallback to this protocol
7685 [48]	Specifies a padding extension to inflate the `ClientHello` to account for servers with parser intolerances
7919 [49]	Specifies the negotiation of finite field Diffie-Hellman parameters
8422 [50]	Specifies handling of elliptic curve cipher suites for TLS 1.2 and earlier versions
8701 [52]	Specifies a set of 'GREASE' constants that are meant to ensure interoperability between peers with different protocol features

Table 7: Brief overview of the considered RFCs and their contents