Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--25/091--SE

Enabling Dynamic Symbolic Execution for TLS Implementation Analysis

Möjliggörande av dynamisk symbolisk exekvering för analys av TLS-implementationer

Johannes Pettersson
Oscar Nyman

Supervisor : Johannes Wilson
Examiner : Mikael Asplund

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

Ensuring TLS implementations behave as outlined in their specification is essential for secure communication over networks. Security flaws in protocol implementations can have severe consequences, as demonstrated by incidents like the Heartbleed vulnerability and the POODLE attack, leading to data breaches, man-in-the-middle attacks, and unauthorised access to sensitive information. This thesis investigates the use of dynamic symbolic execution to analyse WolfSSL, an open-source implementation of the TLS 1.3 protocol, with a specific focus on its ServerHello message parser. To uncover potential implementation errors, we derive conditions from the TLS 1.3 specification and use them to constrain symbolic execution to program paths where nonconformances may occur. We also investigate how the placement of such constraints, either directly when the message is received, before validation logic, or after message acceptance, affects time overhead, input space coverage, and generalisability. Additionally, we propose an alternative approach to accelerate the testing of multiple requirements by inserting them as logical queries into KQuery files, the intermediate representation of symbolic constraints generated by KLEE, after symbolic execution has completed. This approach eliminates the need to re-run KLEE for each requirement. Our methods are evaluated against a previous test-case–based approach by Wilson and Asplund, showing that our approaches manage to find the same number of nonconformances. The results indicate that the placement of assume statements affects execution time and state space exploration when running KLEE on WolfSSL, and these factors along with generalisability affect how the assume statements should be placed to obtain more desirable results in different scenarios. Our results also indicate that modifying KQuery files can significantly reduce time overhead while still allowing complete exploration of all input cases where a requirement might be violated.

Acknowledgments

We would like to express our sincere gratitude to our supervisor, Johannes Wilson, for his continuous support throughout the project. His guidance and willingness to brainstorm ideas with us when we were stuck have been invaluable.

We are also thankful to Mikael Asplund for his insightful feedback and for helping us explore new directions for our project.

Finally, we would like to extend our appreciation to Niklas Johansson and Sectra Communications for the opportunity to collaborate during the course of this thesis. It has been a rewarding experience, and we are truly grateful for the trust and support we received.

Contents

List of Figures

List of Tables

1 Introduction

The introduction chapter begins by presenting the motivation behind the thesis (Section 1.1), which establishes the foundation for the study. This is followed by a discussion of the thesis's aim (Section 1.2) and the specific research questions that the work tries to address (Section 1.3). The chapter then presents a short description of the methodological approach taken in the study (Section 1.4) and details the delimitations that define the scope of the research (Section 1.5). Finally, the key contributions of the thesis are presented in Section 1.6.

1.1 Motivation

In modern society, there is a significant demand for secure communication over the internet. With the increasing volume of data transmitted across distributed systems, ensuring data integrity, confidentiality, and authenticity is crucial. One of the most widely used protocol for encrypting and securing digital communication is the Transport Layer Security (TLS) protocol. TLS plays a central role in securing web browsing, email communication, and many other network-based services.

However, like any protocol, TLS is only as secure as its implementations. Security flaws in implementations of the TLS protocol can have severe consequences, leading to data breaches, man-in-the-middle attacks, and unauthorised access to sensitive information. A well-known example is the Heartbleed vulnerability [7] which resulted from improper memory handling in OpenSSL's TLS implementation, exposing critical private key information. This forced many major companies to temporarily shut down their businesses and in total around 24-55% of popular HTTPS sites were affected [10].

Incidents like POODLE [16] (Padding Oracle On Downgraded Legacy Encryption) also show the importance of secure protocols. The POODLE attack exploits the SSL 3.0 protocol's vulnerability, allowing attackers to decrypt secure HTTP cookies and other sensitive data. This attack is possible because many TLS clients support downgrade negotiation mechanisms to maintain compatibility with legacy systems, which can be exploited by attackers to force a connection to use the insecure SSL 3.0 protocol. The attack highlights the critical need to have protocols that are well-tested and conform to specification.

Verifying protocol implementations against their specifications is essential to ensure both correctness and security. However, testing cryptographic protocol implementations presents significant challenges due to their reliance on complex cryptographic functions and state

transitions. Additionally, the vast range of configuration options can result in state space explosion when attempting to test all possible configurations exhaustively. Traditional testing approaches, such as fuzzing or unit testing, often fall short in such scenarios, as they rely on either randomised inputs or predefined test cases, which may not provide sufficient coverage.

Symbolic execution is a program analysis technique used to extract execution paths from a program by treating variables as symbolic instead of concrete values [13]. This method is often characterised by its ability to achieve high code coverage and, in the ideal case, explore all possible paths within the program's logic. As a result, symbolic execution is particularly well-suited for exhaustively testing TLS protocol implementations against their specifications.

Our research builds on the work of Wilson and Asplund [26]. They use symbolic execution to generate test cases, which they compare to requirements derived from the TLS 1.3 specification. While their method prioritises performance, it sacrifices some precision, potentially missing nonconformances due to the numerous possible concretisations within a single path. Comparing to a single test case, which is the case in their work, which represents one concretisation of that path, may overlook edge cases that exist within the path conditions but are not captured by the specific concretisation chosen for the test case.

1.2 Aim

The aim of this thesis is to evaluate how symbolic execution can be used to test protocol conformance. To achieve this, we conduct an experimental study and comparative analysis of different symbolic execution strategies. We use KLEE [5], a symbolic execution engine, in order to symbolically analyse the WolfSSL TLS 1.3 implementation. To address the limitations of the approach taken by Wilson and Asplund [26], this thesis explores an alternate approach based on the method used by Asadian et al. [3] in which we constrain paths using *assume statements*. These statements are inserted into the implementation and contain the negation of the condition needed for a requirement from RFC 8446 [19] to be fulfilled. This method allows us to exhaustively explore faulty paths that could lead to nonconformances, focusing on bad input and ensuring that no errors are overlooked due to varying concretisations within a single path [3].

However, their approach focuses on making the specific fields they check symbolic. While they justify this by claiming that it enhances scalability, they also acknowledge that this approach can overlook requirements that are only violated when other bytes are also symbolic [3]. To address this, we combine the method of instrumenting *assume statements* with the full-message symbolic execution approach. This enables a more thorough analysis, allowing us to also identify errors related to the relationships between fields, at the expense of some performance.

We showcase the method on two versions of the WolfSSL TLS 1.3 implementation: version 5.6.0, which contains two known bugs, and version 5.6.6, which is not known to contain any bugs. We exclusively focus on the ServerHello message parser and explore how different placement strategies affect time overhead, input space coverage, and generalisability of the approach. Our goal is to enhance the testing process and document our experience using these different placement strategies. We compare the strategies against each other and against the approach taken by Wilson and Asplund [26] to evaluate the advantages and disadvantages of each placement strategy and method.

In addition, the study introduces a second approach to enable verification of various requirements by directly modifying the files containing the logical paths of the system under test (SUT), which are generated by KLEE as KQuery files. This method aims to enable complete exploration of all concretisations of input data while maintaining low time overhead during the verification process.

1.3 Research Questions

The questions that this thesis aims to answer are as follows:

1. How can TLS 1.3 requirements from RFC 8446, specifically those concerning the ServerHello message, be translated into KLEE assume statements for symbolic execution?
2. How do different placement strategies for assume statements: early placement, before validation, and after message acceptance, affect the symbolic execution performance and ability to detect nonconformances?
3. How can post-execution modification of KQuery files be used to verify multiple TLS requirements efficiently, and what are the benefits and limitations of this approach compared to in-code instrumentation?
4. How does the proposed method, including both in-code assume statements and KQuery file modifications, compare to Wilson and Asplund's test-case–based method in terms of nonconformance detection capability and time overhead?

1.4 Approach

To answer these research questions we take the following approach:

1. We identify relevant requirements from the TLS 1.3 specification that apply to the ServerHello message. These requirements are then translated into conditions, which we negate and use to guide the symbolic execution toward program paths that violate expected behaviour. We explore multiple strategies for placing these constraints, represented as assume statements, at different locations in the code, such as early in input handling, before validation logic, or just after message acceptance.
2. We investigate the efficiency and effectiveness of each placement strategy by collecting and comparing metrics such as time and the number of explored program paths for every symbolic execution run.
3. We propose an alternative method for verifying multiple requirements more efficiently by modifying KQuery files after execution, eliminating the need to re-run the analysis multiple times.
4. We conduct a comparative analysis of our method against the previously proposed approach by Wilson and Asplund [26]. This comparison helps us evaluate how well our strategy performs in terms of precision and resource consumption.

1.5 Delimitations

In this thesis, we only test the WolfSSL TLS 1.3 implementation. Specifically, we limit our scope to WolfSSL versions 5.6.0 and 5.6.6. These versions are picked due to the fact that the method proposed by Wilson and Asplund [26] finds two vulnerabilities when testing version 5.6.0 and finds no bugs in version 5.6.6. Therefore, it is possible to compare our methods with theirs and we are able to verify that our two approaches finds the same vulnerabilities, or possibly even additional bugs. Also, these versions are stable and compatible with the setup we had available. Furthermore, experiments are performed only on ServerHello messages that comply with the strict TLS specification proposed by Wilson and Asplund [26]. Although such configurations may be useful in security-critical implementations, they do not represent all variations of TLS 1.3 ServerHello messages. Other configurations may contain bugs or nonconformances that we do not find. Also, newer versions may be more secure than those tested in this work and do not necessarily contain the bugs we find here. Lastly, we only

consider TLS version 1.3 in order to limit our scope. By no means is this the only version that is in use at the time of writing this thesis, but it is the latest, most secure version, which makes it arguably the most relevant to test. Older and future versions of the protocol may or may not have to adhere to the requirements tested in this work.

1.6 Contributions

The main contributions of the work are outlined as follows:

1. **An approach for translating TLS specification requirements into assume statements that restrict symbolic execution paths:**

   This thesis uses a verification approach based on assume statements, allowing more exhaustive testing of the TLS implementation. We show how requirements derived from the RFC can be implemented as negated constraints in the source code.

2. **A comparison of approaches to verify TLS implementations:**

   We perform a comparison of an approach based on assume statements to verify the TLS implementation. We do this by both comparing different ways of inserting assume statements into the source code, and comparing this approach with the approach used by Wilson and Asplund[26], in order to find out how feasible the approach is.

3. **Experimental validation of TLS implementation correctness:**

   By using assume statements to test WolfSSL we exhaustively evaluate its adherence to the TLS specification and explore all possible concretisations for a given path. We also analyse the feasibility of symbolically verifying protocol rules using this approach.

4. **Optimising the time if takes to validate multiple requirements by modifying KQuery files:**

   By modifying the KQuery files generated by KLEE, we present a way to check multiple requirements without having to re-run the symbolic execution process for every requirement.

2 Background and Related Work

In this chapter, the background and related work of the thesis is presented, in order to give the reader the necessary information to understand the rest of the thesis.

2.1 Background

The thesis covers various topics, which will be briefly explained in this section.

2.1.1 Transport Layer Security

Transport Layer Security (TLS) is a widely adopted protocol designed to secure communications between clients and servers over the Internet [24]. Originally developed as the Secure Sockets Layer (SSL) by Netscape in the mid-1990s, TLS has evolved through several versions, with TLS 1.0 introduced in 1999, followed by TLS 1.1 in 2006, TLS 1.2 in 2008 and TLS 1.3 in 2018 [19]. The protocol aims to prevent eavesdropping, tampering, and message forgery, ensuring the integrity and confidentiality of data transmitted over networks. TLS operates primarily over the Transmission Control Protocol (TCP) and is implemented in user space, which allows for quicker updates compared to kernel-based protocols like IPsec. The protocol supports various cipher suites, which are a simplified set of algorithms that define the encryption and integrity mechanisms used to secure a connection, as well as key exchange algorithms, enabling flexibility in securing different types of applications. Notably, TLS 1.2 introduced features such as session resumption and support for authenticated encryption. The ongoing development of TLS, particularly with TLS 1.3, focuses on enhancing security by encrypting more of the handshake process, reducing latency, and addressing known vulnerabilities in previous versions. Despite its robustness, TLS implementations in some cases require diligent management by server administrators to mitigate risks from vulnerabilities like the Heartbleed bug.

The TLS 1.3 Handshake

The TLS 1.3 handshake is responsible for negotiating protocol parameters, establishing shared secrets, and authenticating endpoints. There are several steps in this process, which are specified in RFC 8446 [19].

The handshake proceeds through a sequence of messages: the initial ClientHello, the ServerHello, EncryptedExtensions, Certificate, CertificateRequest, CertificateVerify, and Finished [19]. The process can be divided into three phases:

1. **Key Exchange**
   The client initiates the handshake with a ClientHello message containing the TLS versions and cipher suites it supports, together with key exchange parameters. The server determines the cryptographic parameters to be used for the connection and then responds with its own ServerHello message, which indicates these parameters. The combination of these two messages determines the shared keys, and after these two messages are sent the cryptographic context has been defined. After this phase, all data is encrypted.

2. **Server Parameters**
   The server sends at least one message to establish the server parameters, The EncryptedExtensions message, which contains responses to the ClientHello extensions that are not needed to establish the cryptographic parameters. This message is empty if the client did not send any additional extensions. If certificate-based client authentication is to be used, the server also sends a CertificateRequest message containing parameters for the certificate.

3. **Authentication**
   Lastly, the server, and potentially the client, are authenticated depending on whether certificate-based client authentication is in use. This phase may include a Certificate, a CertificateVerify, and a Finished message. If certificate-based client authentication is not used, the client omits the Certificate and CertificateVerify messages, sending only the Finished message. However, the server always sends the Certificate and CertificateVerify messages to authenticate itself, provided that pre-shared keys are not in use. This phase ensures key confirmation and establishes handshake integrity.

After these phases, the handshake is complete and the client and server can exchange application data which is now encrypted. The full TLS 1.3 handshake without the use of certificate-based client authentication is illustrated in Figure 2.1.

The ServerHello Message

The ServerHello message is responsible for negotiation with the client based on the parameters in the ClientHello message.

A ServerHello message contains multiple different fields. It is essentially a sequence of bytes that together make up a message that is sent back to the client. A visual representation of the ServerHello message can be seen in figure 2.2.

Firstly, it contains a record header, with record type which is set to 0x16 to represent a handshake message, a record version set to 0x0303 to represent TLS legacy version 1.2, and a record length specifying the length of the handshake message that follows. The next four bytes make up the handshake header, consisting of the handshake type and the handshake message length. The handshake type should be 0x02 to indicate that it is a ServerHello message, and the next three bytes is the length of the message.

Following the record header, the two following bytes represent the legacy version, which should be set to 0x0303 (indicating TLS 1.2) for backward compatibility. The next 32 bytes is the random string, which should be generated independently of the ClientHello random, and is crucial for security and uniqueness of handshakes. The following bytes are the Session ID, which was used in TLS 1.2 by the client and server as an id of the connection and was used to reconnect at a later stage without having to do the entire handshake process all over again, thus saving time. However, for TLS version 1.3 this field is no longer used, and should

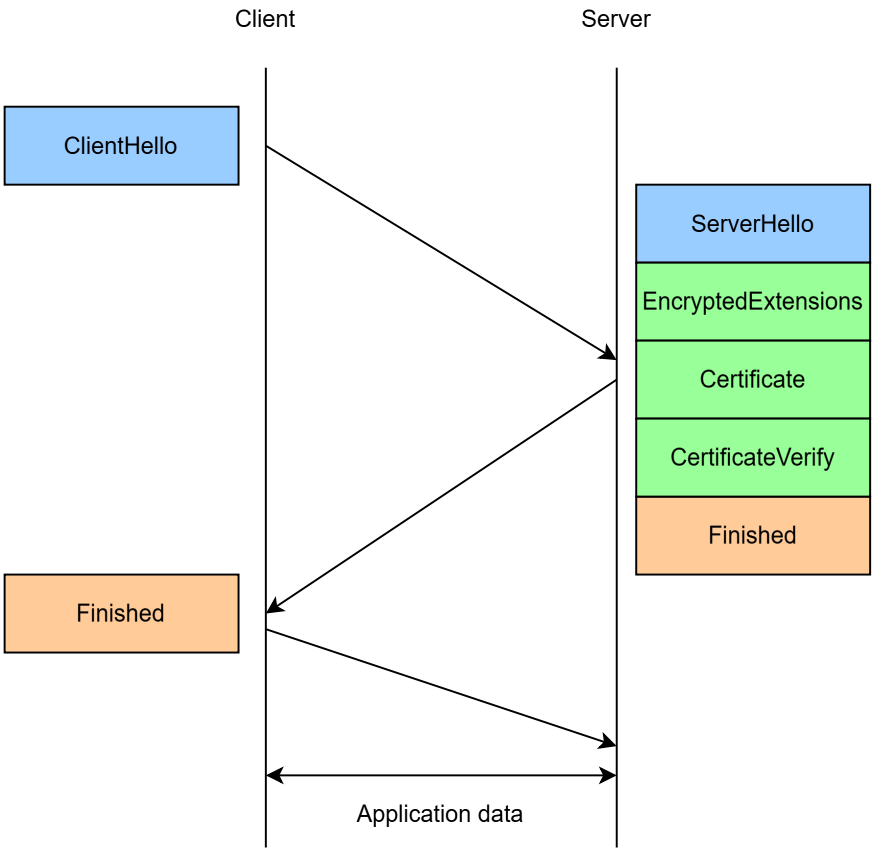


Figure 2.1: The full TLS 1.3 handshake without certificate-based client authentication. Messages highlighted in blue are part of the Key Exchange phase (phase one), the message highlighted in green is part of the Server Parameters phase (phase two), messages highlighted in orange are part of the authentication phase (phase three).

therefore have the value 0. After the Session ID there are two bytes containing the Cipher Suite, which is chosen by the server from the list of ciphers sent in the ClientHello message. The next byte is the Legacy Compression Method, which according to the RFC should be set to 0. The final bytes of the message together make up the extensions. Here, both the length of the extensions and possible extensions used in the message are contained.

2.1.2 WolfSSL

WolfSSL is an open-source, embedded SSL/TLS implementation in the form of a library written in C, designed for applications across various environments, including embedded systems, real-time operating systems (RTOS), and resource-constrained devices [27]. Its compact size and efficient performance make it suitable for desktop, enterprise, and cloud applications as well. Supporting industry standards up to TLS 1.3 and DTLS 1.3, WolfSSL is up to 20 times smaller than OpenSSL, offering a simple API and an OpenSSL compatibility layer. It includes features such as Online Certificate Status Protocol (OCSP) and Certificate Revocation List (CRL) support, and is backed by the robust WolfCrypt cryptography library. WolfSSL is the TLS implementation used in our experimental evaluation of the two approaches studied in this thesis.

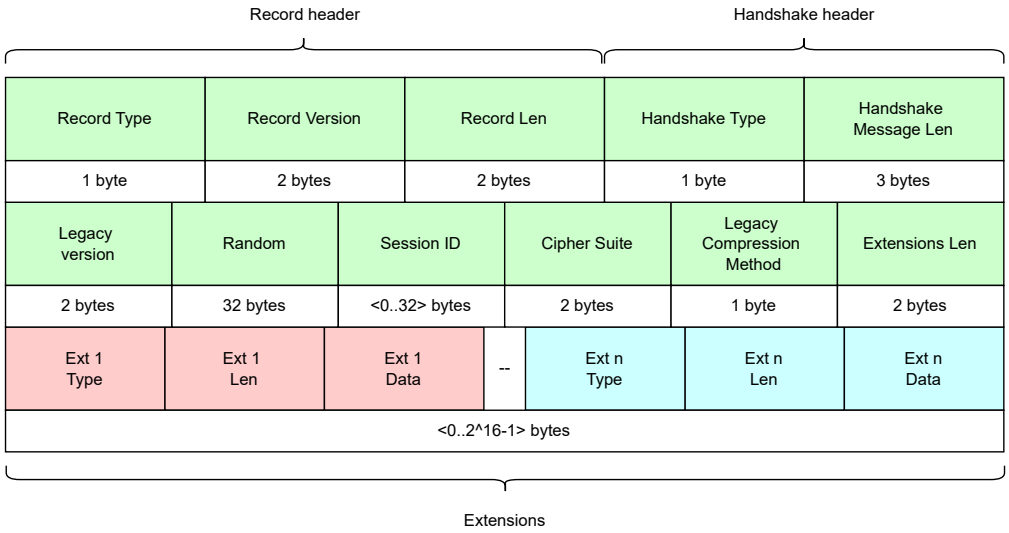


Figure 2.2: The structure of a ServerHello message. The Session ID and Extensions fields may vary in length, as illustrated by the notation <*n*..*m*> where *n* denotes the shortest length and *m* denotes the longest length the fields can be. Note that the structure is shown in three rows for improved readability. In practice, the ServerHello message is transmitted as a single continuous byte stream.

2.1.3 SAT/SMT

The Boolean satisfiability problem (SAT) is a constraint satisfaction problem which involves trying to find solutions to a logical expression [9]. The variables in a propositional logic formula is given boolean values, true or false, in order to try to make the whole expression true. Satisfiability Modulo Theories (SMT) extends traditional SAT to address more complex logical formulas by incorporating additional theories such as arithmetic, arrays, and bit-vectors. This approach enables the analysis of formulas that capture intricate relationships and conditions found in real-world applications. By combining multiple theories, SMT allows for richer expressions that are essential for tackling a variety of problems.

Recent advancements in SMT have led to significant interest across multiple fields, including software verification, scheduling, static program analysis, and test-case generation [17]. The integration of these theories has facilitated more accurate representations of programming concepts, ultimately enhancing the capabilities of various analysis and verification tools. The ability to efficiently manage complex instances highlights the evolution of these techniques in solving constraint satisfaction problems.

To demonstrate a simple SAT problem, consider the following expression:
$C = (A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B)$
To solve this, every clause needs to evaluate to true. This can be achieved by letting $A = \text{True}$ and $B = \text{True}$, which satisfies C.

To determine the validity of an SMT, SMT solvers can be used. They are powerful tools designed to determine the satisfiability of logical formulas with respect to various theories, such as integers, bitvectors, arrays, and more [18]. These solvers extend the capabilities of propositional SAT solvers by incorporating decision procedures for these underlying theories, allowing them to handle quantified formulas and reasoning about complex data types. One popular SMT solver is the STP solver, which is widely used as the underlying solver

for KLEE (see Section 2.1.5) [20]. This is also the SMT solver used in this thesis in order to determine the validity of the query-expression in a KQuery file, presented in Section 2.1.7.

2.1.4 Symbolic Execution

Symbolic execution is a technique used in program analysis to systemically explore execution paths by treating inputs as symbolic rather than concrete values [13]. These symbolic values represent all potential input values and are subsequently propagated through the program's computations and decision points. This process forms path constraints, which describe the conditions required to reach a specific state in the program's execution. By enabling the exploration of multiple execution paths in a single analysis, symbolic execution aims to achieve high code coverage and identify unwanted behaviour in systems.

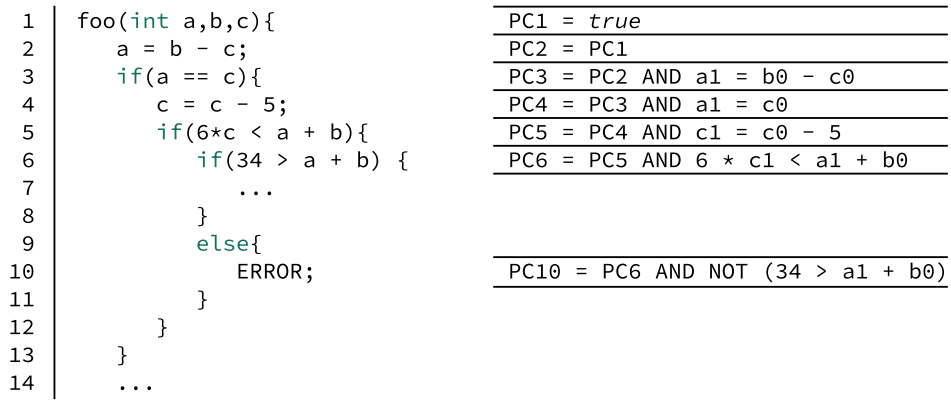


Figure 2.3: Simple symbolic execution example.

Consider the example in figure 2.3. The function foo takes three integer parameters: a, b, and c. Depending on the values of these parameters, execution of the function may look different as different paths are followed. For example, to reach the error at line ten, the corresponding path constraint (PC) would be:

$PC6 \wedge \neg(34 > a1 + b0)$

which expands to the SMT formula

$PC\_ERROR = (a1 = b0 - c0) \wedge (a1 = c0) \wedge (c1 = c0 - 5) \wedge (6 * c1 < a1 - b0) \wedge \neg(34 > a1 + b0)$

Each of the path constraints *PC1 - PC10* accumulates the conditions of all previous branches into the final path constraint for reaching the error (*PC_ERROR*). As these conditions are added, the symbolic execution engine (KLEE in our case, see Section 2.1.5) uses an SMT solver to check whether the path is still logically consistent (i.e., satisfiable). If a combination of constraints is unsatisfiable, the path is pruned, meaning it is not explored further. This process helps reduce the number of irrelevant or unreachable paths, focusing the analysis only on feasible ones.

Despite the promise of being able to (in the ideal case) reach full code coverage, symbolic execution faces some challenges. In particular, the number of execution paths grows exponentially with program complexity and can lead to path explosion, meaning that there are too many paths to feasibly analyse. Path explosion makes it infeasible to analyse large complex programs, especially programs where nonlinear operations (such as cryptographic functions) are involved, in which case symbolic execution would get stuck trying to solve the corresponding constraint. Often, approximations or abstractions of such functions are required in order for symbolic execution to run efficiently.

2.1.5 KLEE

KLEE is a symbolic execution engine designed to analyse programs through symbolic execution [5]. Built on the LLVM compiler infrastructure, KLEE systematically explores all possible execution paths of a program and produces corresponding test cases to explore these paths. Its primary goal is to improve software testing by automatically creating high-coverage test suites. KLEE has been applied successfully to detect bugs, verify program correctness, and evaluate the reliability of software systems. Compared to other symbolic execution tools, KLEE is notable for its efficiency, ease of use, and compatibility with a wide range of real-world software. KLEE is used in this project to systematically explore the paths in the WolfSSL TLS 1.3 implementation and to ensure that it conforms to the specification.

2.1.6 Assume Statements

The constructs we refer to as assume statements are essentially conditions used to constrain and guide symbolic execution. In the context of SMT, these assume statements act as additional path constraints that are sent to the SMT solver for validation. Specifically, our assume statements contain the negation of the condition that leads to the requirement in question being fulfilled. This approach allows for exhaustive testing of faulty paths where the requirement is violated. By focusing on invalid paths and pruning correct ones, we can thoroughly explore the invalid state space.

To instrument these constructs in the test suite so that KLEE can analyse them, we use the KLEE intrinsic function *klee_assume* [14]. This function restricts the values that a symbolic variable can take during execution. The remainder of the symbolic execution will only consider paths where the condition passed to the klee_assume function is satisfied. Essentially, it acts like wrapping the rest of the program in an if(condition) statement. An example of how klee_assume can be used is shown in Listing 2.1.

The negated conditions we pass to this function constrain the symbolic bytes for the specific field in the ServerHello message that the requirement concerns, preventing them from taking correct values. These conditions are added as path constraints, which KLEE sends to the underlying SMT solver, STP in our case. Again, this allows us to explore how the implementation handles ServerHello messages where the field in question does not contain its expected value.

Listing 2.1: An example of how the KLEE intrinsic function klee_assume can be used to constrain symbolic input. In this example, KLEE will only explore paths where x > 5, and therefore only reach the second printf call.

```
int main() {
    int x;
    klee_make_symbolic(&x, sizeof(x), "x");

    // Constrain x to be greater than 5
    klee_assume(x > 5);

    if (x <= 5) {
        printf("x is 5 or less\n");
    } else {
        printf("x is more than 5\n");
    }

    return 0;
}
```

2.1.7 Symbolic Execution Queries / KQuery

Symbolic execution queries are specific questions or checks made during symbolic execution to determine the feasibility of certain paths or conditions within a program [5]. These queries are typically handled by solvers, such as SMT solvers, which determine if there exists an assignment of values to the symbolic inputs that satisfies the conditions of the query.

In the context of KLEE, KQuery is a textual representation format used as input to Kleaver, KLEE's solver for KQuery files. KQuery files are generated from the symbolic execution process. These files contain queries and expressions [15]. Kleaver is a logical solver that aims to find solutions to the queries contained in the KQuery files. The structure of a KQuery file in the context of this thesis is as follows:

> **kquery** ::= "(" "query" **constraint-list query-expression** [**eval-expr-list** [**eval-array-list**]] ")"

Here, the constraint-list contains the path conditions collected during symbolic execution, and the query-expression is a logical condition whose satisfiability is being checked under the given constraints. In other words, the solver tries to determine whether the query-expression can be true given the current set of path constraints.

The eval-expr-list and eval-array-list describe optional components used for value evaluation, which are not relevant in the context of this thesis.

2.2 Related Work

Research within similar topics of the thesis is presented in this section.

2.2.1 Contributions by Asadian et al.

Asadian et al. have made several contributions toward symbolic and hybrid approaches to protocol testing. In [4], they compare symbolic execution and fuzzing on CoAP implementations. Their later works [2], [3] extend this by introducing assertion-driven testing and monitor-based approaches for DTLS and QUIC. While these studies test individual fields or packets, our work builds on their assertion concept and adapts it for full-message symbolic execution on TLS 1.3, using both assume statements and post-analysis of KQuery files.

In one of their works, they compared two techniques, symbolic execution and fuzzing, to provide insights into the benefits of each technique[4]. They applied these techniques to implementations of the Constrained Application Protocol (CoAP) and identified numerous nonconformances. Fuzzing is a technique where input is generated to trigger unexpected behaviour. The authors found fuzzing to be relatively fast and effective in finding bugs in the code. However, when compared to KLEE and symbolic execution, they concluded that symbolic execution provides more comprehensive tests, although it requires more time to perform. When testing the implementations using KLEE, the authors manually added assertions to the source code to uncover bugs. How the authors use symbolic execution is similar to how it is used in this paper, but here it is applied to a different protocol, focusing on TLS implementations. Also, we treat the entire message symbolic, and not just individual bytes. Lastly, the aim of the article presented is to compare symbolic execution to fuzzing, which is not the case in this paper.

Asadian et al. also conducted tests on network protocol implementations using a monitor-based approach [2]. They developed a component external to the System Under Test (SUT) for each test case and applied this methodology to both DTLS and QUIC protocols. Their approach successfully identified several unresolved vulnerabilities in the existing implementations, as well as newly discovered bugs, which they reported to the respective developers. In contrast, our research does not employ a monitor-based approach. Instead, we utilise symbolic execution with KLEE using assume statements and by modifying the KQuery files to investigate WolfSSL. With the strict TLS we use, presented in Section 3.1.1, and by looking at one message, we limit the packets sent, which reduce the need for a monitor-based approach.

Lastly, they also used symbolic execution to thoroughly test implementations of network protocols against their specification [3]. Their methodology involves extracting requirements from the RFC, which are subsequently formulated as logical assertions. Symbolic execution is then used to identify execution paths that do not conform to these requirements by systematically exploring possible input sequences. As opposed to this thesis, their work focused on DTLS protocols whereas we consider only non-datagram TLS. Furthermore, and more importantly, our approach considers full-message symbolic execution rather than performing symbolic execution on individual fields, enabling multiple fields to be analysed holistically. However, our work follows a similar method to the aforementioned authors', and is inspired by their work. For example, the idea of negating requirements and inserting them as conditions into KLEE assume statements stems from their classification of an input-validity requirement.

2.2.2 Comparative Approaches to Protocol Testing

Researchers have explored a range of testing techniques to evaluate protocol implementations, including fuzzing, concolic execution, and hybrid symbolic methods. Somorovsky [21] performed fuzzing on multiple TLS implementations in order to find cryptographic failures

and boundary violation vulnerabilites. By dynamically modifying variable content in TLS messages and enabling creation of custom protocol flows, the author was able to find several bugs in the TLS libraries studied. His approach depends on generating semi-random input dynamically, while this thesis focuses on verifying compliance with the TLS specification by manually inserting relevant asssumptions. As a result, our approach should achieve higher coverage and aims to uncover a broader range of potential vulnerabilities.

In the Internet of Things (IoT) domain, where data protection is critical, protocol testing plays an essential role. Tempel et al. [23] performed testing on the RIOT and Zephyr operating systems to uncover bugs and nonconformances. While the systems under test are OSes, the target of testing is their protocol behaviour, with the focus lying on their network protocol stack implementations. Their approach is specification-based, utilising concolic testing, a method that combines concrete execution with symbolic execution. While symbolic execution is powerful, it often suffers from path explosion in larger programs due to the vast number of possible execution paths. Concolic testing mitigates this challenge by using real inputs to simplify symbolic constraints and dynamically generate new test inputs to explore previously uncovered paths, making it more efficient. Using this approach, the authors successfully detected previously unknown bugs in the RIOT operating system. Their method differs from the work described in this research, where symbolic execution is carried out using KLEE, instead of SymEx-VP. Also, the authors implement a multipacket exploration algorithm to analyse multiple packets symbolically, whereas we make an entire ServerHello message symbolic, and test whether it conforms to its specification.

Wen et al. [25] proposed a novel approach for detecting vulnerabilities in network protocol implementations using model-guided symbolic execution. Recognising that traditional testing methods, such as fuzzing, might not achieve full testing coverage, the authors introduced a finite state machine (FSM) model to guide symbolic execution, which significantly improved the effectiveness of vulnerability detection compared to conventional fuzzing tools. However, their work focuses on vulnerabilities linked to the deep states of the File Transfer Protocol (FTP), rather than the specific requirements of the protocols. In contrast, our research investigates vulnerabilities related to the requirements of a TLS implementation.

There are ways to improve symbolic execution to make it more efficient. Zhang et al. [29] presented one way in which they proposed a method to combine symbolic execution with fuzzing and concolic execution. This method includes using symbolic execution to traverse through the program states symbolically to explore different paths in the program, and then utilise fuzzing to test the program with inputs generated from the symbolic execution. Concolic execution was used to guide fuzzing through complex checks, which eventually yielded a result that was more efficient than symbolic execution. However, this method was applied to test for vulnerabilities in binary programs, and not in a TLS implementation.

2.2.3 Symbolic Execution for Protocol Verification

Back in 2005, Goubault-Larrecq and Parrennes [11] proposed a methodology for analysing cryptographic protocol implementations written in C. Their approach combines static program analysis techniques with a symbolic model of cryptographic protocols, specifically by translating C code into Horn clauses and leveraging the Dolev-Yao model. Their main focus lies in addressing the challenges of analysing low-level C code, including handling memory allocation, aliasing, and external libraries. These program states are treated as black boxes to reason about protocol security properties using trust assertions.

Wilson and Asplund [26] use full-message symbolic execution to verify TLS implementations. By providing abstractions of implementation primitives, the authors handle many problems that would typically cause state space explosion. They also propose a strict TLS configuration with limited configuration to make symbolic analysis tractable. Using their method, the authors managed to find vulnerabilities caused by incorrect handling of ServerHello messages in several versions of WolfSSL that the developers were not previously aware

of. Their work acts as the foundation for this thesis. Thereby, the problem of handling cryptographic and other non-linear operations symbolically and making full-message symbolic execution tractable is handled the same way in our work. All changes to KLEE made in the aforementioned work are reused in this work. We also adopt their proposed strict TLS specification and limit the amount of configurations possible when sending messages. However, in this thesis we alter the approach taken by Wilson and Asplund. Instead of comparing traces (test cases) to specification, we utilise assume statements in the source code of the SUT to enable a more thorough search. We also propose a stand-alone method based on modifying KQuery files.

SYMBEXNET [22] combines symbolic execution and rule-based specifications to detect security flaws, semantic errors, and interoperability issues in network protocol implementations. The framework generates test input packets using symbolic execution, which are then checked against manual rules extracted from protocol specifications. To analyse deeper execution paths, SYMBEXNET utilises both single-packet symbolic execution for stateless protocols and multi-packet symbolic execution for stateful protocols. Additionally, the framework tests for interoperability issues by replaying generated test cases across multiple implementations to detect inconsistencies. This thesis focuses exclusively on the TLS protocol, meaning that stateless protocols are not within our scope. Furthermore, we do not consider interoperability issues as our research is limited to a single implementation.

Flaws in implementations of the certificate chain validation logic can also lead to serious security issues, such as accepting invalid certificate chains, which may allow impersonation attacks. Recent work has introduced SymCerts, a method that leverages symbolic execution to analyse X.509 certificate chain validation implementations, successfully uncovering numerous instances of noncompliance with the X.509 standard [8]. This work is particularly relevant because it addresses the security assurance needed for SSL/TLS implementations in a nuanced way, utilising a technique that can uncover subtle flaws that traditional testing methods might miss. In contrast, while both approaches seek to enhance the security of SSL/TLS implementations, our research narrows in on TLS handshakes where certificate-based client authentication is not in use, hence leaving this type of research out of scope for the thesis.

EXE, a tool developed by Cadar et al. [6] is a bug-finding tool that aims to find bugs in protocols by generating inputs that crashes the code. It uses symbolic execution to test the code paths more thoroughly, and uses the constraint solver STP to solve these constraints. It focuses on finding bugs in low-level software, especially network protocols, to find vulnerabilities and issues like buffer overflows. However, it does not focus on finding bugs related to the requirements of the protocols, and is not focused on TLS implementations, unlike our research.

3 Prestudy

This section presents the preparatory work needed to evaluate our methods. We start by describing the underlying implementations done by Wilson and Asplund [26] that lay the foundation of our work in Section 3.1. We then describe which requirements we consider in this research and explain in short how they are extracted in Section 3.2. Lastly, in Section 3.3 we explain shortly how nonconformances can be verified as true positives by feeding the data produced by KLEE or Kleaver back to the implementation.

3.1 Setup for Symbolic Execution of WolfSSL

This section details the experimental setup. We adopt the strict TLS limitations and overall experimental framework from Wilson and Asplund [26], including the reuse of their code for handling cryptographic functions. Nonetheless, to guide the reader, we present them in short in the coming subsections.

3.1.1 Strict TLS

As mentioned in Section 1.5, the number of TLS options are limited in this research. The TLS 1.3 specification already limits the amount of different configurations by for instance reducing the amount of allowed cipher suites in TLS 1.3 [1], removing several suites that are deemed unsafe to use. Still, if no further limitations are applied to message configuration, there would be too many different combinations of its structure and contents. We therefore only allow a strict ServerHello message with limited configuration in order to make symbolic execution manageable. These limitations are reused from the work done by Wilson and Asplund [26]. Following that work, the limitations of a ServerHello message can be seen in Appendix A.

3.1.2 Handling Cryptographic Functions

The WolfSSL source code is highly complex and includes cryptographic functions that pose significant challenges to symbolic analysis. These cryptographic functions must be addressed in a manner that minimises interference with the symbolic execution of requirements. To achieve this, we simply reuse the approach outlined by Wilson and Asplund [26], using the

modifications introduced in their experiments. For detailed explanations of these modifications and how they are implemented, we refer the reader to their paper, as these aspects are beyond the scope of this thesis and will not be discussed further here.

3.1.3 Symbolic Sockets

In their work, Wilson and Asplund [26] introduce symbolic sockets as part of their modifications to KLEE to facilitate systematic analysis of how TLS implementations handle a wide range of possible message contents. Their methodology involved converting all reads from TCP sockets into symbolic bytes, which serve as the primary symbolic variables during the execution of the program. We adopt this symbolic socket implementation to be a part of our KLEE build, utilising the symbolic sockets to make all bytes of the incoming message symbolic immediately upon receipt.

3.1.4 Running KLEE

Once a requirement is represented symbolically, it is verified through symbolic execution to assess its compliance with the specified standards. Utilising KLEE enables a systematic analysis of the program to identify potential implementation bugs and nonconformances. Through symbolic reasoning, KLEE can effectively detect a range of issues, including index-out-of-range violations, memory access errors, and infinite loops, ensuring robustness and adherence to protocol specifications.

We run KLEE using the same overall setup as Wilson and Asplund [26]. This means that we run symbolic execution on an echoclient example provided by WolfSSL, which implements a client that establishes a connection to an echoserver through a socket. The data transmitted by the client is echoed back from the server to verify basic communication functionality. To analyse the handling of ServerHello messages, an abort statement is placed immediately after the implementation accepts a ServerHello message, serving as a termination point for the symbolic execution runs. In this context, reported abort errors indicate execution paths where the ServerHello message is deemed valid by the implementation.

To aid in the testing process, we script the execution of KLEE to automate the symbolic analysis. Since KLEE operates on bitcode files (.bc), the program must be recompiled for each requirement being tested. When requirements are placed directly within KLEE, KLEE itself must be rebuilt to include the updated assume statements. In contrast, when requirements are placed directly within the WolfSSL implementation, WolfSSL must be rebuilt for the specific version under test to reflect the instrumentation of the constraints.

3.2 Extracting Requirements

In order to analyse the ServerHello messages, it is essential to investigate whether the implementation adheres to the rules defined for the specified TLS version, which in this case is TLS 1.3. This process is critical to determining whether the implementation behaves as intended and conforms to the protocol's specifications. To achieve this, requirements outlined in the protocol's Request for Comments (RFC) [19] are extracted, with particular emphasis on the section defining the ServerHello message. Identifying these requirements involves performing a keyword search within RFC 8446 to locate terms commonly associated with mandatory behaviours, such as "MUST" or "SHOULD." For this work, a subset of requirements tied to the "MUST" keyword is compiled for further analysis.

The requirements of interest align with those identified by Wilson and Asplund [26]. Consequently, this work focuses on the specific ServerHello requirements listed in Table 3.1.

Configuration	Reason
Legacy Version 1.2	The field for legacy version should indicate TLS version 1.2.
Not Retry	The field for the random should not be the same as the static string that indicates a HelloRetryRequest.
Session ID Match	This field should be empty to match the one sent in the ClientHello.
Cipher Match	The field for the cipher should indicate AES-128-GCM-SHA256, since it should match the one sent in the ClientHello.
Empty Compression	This field should be zero, as it represents the legacy compression.
Requires Version	Extensions should contain the value 0x002b to indicate that there is a SupportedVersion extension.
Version 1.3	Should be 0x0304 to indicate TLS version 1.3.
Requires Key Share	Since PSK is disabled, extensions should contain 0x0033 indicating that the key share extension is included.
Boundary Aligns	The boundary in the record header should match the one in the ServerHello message.

Table 3.1: Requirements for the ServerHello message extracted from the RFC.

3.3 Verifying Nonconformances

Tests that reach the abort statement without resulting in a crash represent instances of nonconformance. Conversely, if a crash occurs during execution, it is classified as a bug rather than a nonconformance. As the methodology ensures that only paths violating the requirements reach the symbolic execution's defined termination point (the abort statement), the inputs generated by KLEE can be reliably identified and extracted. These inputs, corresponding to invalid states, can then be used to test the implementation directly by supplying the data from the KLEE-generated test cases back into the program. If the previously identified nonconformance or bug persists upon execution of the unmodified SUT, it represents a true positive.

4 Design and Placement of Assume Statements

This section covers the first method, that is based on assume statements. It both covers how requirements can be inserted as KLEE assume statements, as well as the placement of the assume statements.

To systematically verify adherence to specific requirements, assume statements are instrumented into either KLEE or the implementation code base. If an execution path reaches the abort statement after applying these assume constraints, it signifies a nonconformance with respect to the tested requirement, as the implementation incorrectly accepts an invalid ServerHello message.

4.1 Inserting Requirements as KLEE Assume Statements

When the requirements of interest are extracted from the RFC, we want to verify them using symbolic execution. Therefore, we have to insert the negated requirements as assumptions into the source code, which will guide the program towards faulty paths, as explained in Section 2.1.6.

Our approach focuses on verifying that the symbolic variables cannot take values which should not be accepted by the implementation. Therefore, we negate the validity condition for the requirement in order to only explore paths where ServerHello messages should not be accepted, representing paths that lead to nonconformances to the specification. If the negated condition reaches the termination point (our abort statement), KLEE will generate paths that can then be further investigated, to see if a nonconformance has been found. The placement of these KLEE assume statements is also important and is discussed in Section 4.2.

The following sections describe the extracted requirements in a little more detail and contain listings showing how the different requirements were converted into KLEE assume statements, written in C code. In these listings, the variable *input_buf* represents the buffer containing the ServerHello message while the *idx* variable is the index in that buffer where the corresponding byte sequence for the message field of interest is expected to start.

Keep in mind that due to the strict TLS specification, it is fairly trivial to find the expected starting points for all fields. If we, for instance, do not require that extensions must follow a predefined order where the supported versions extension must be the first extension in the list, we would not know that the supported versions extension field is expected to start at a

set index. Normally, extensions can have any order, in which case finding the expected start index is not as straightforward.

4.1.1 Legacy Version 1.2

According to RFC 8446, the legacy_version field should be set to 0x0303, which corresponds to TLS 1.2. This requirement ensures backward compatibility with middleboxes when using TLS 1.3. The RFC specifies: "In TLS 1.3, the TLS server indicates its version using the supported_versions extension (Section 4.2.1), and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2."[19]. The assume statement to handle this requirement is displayed in Listing 4.1.

Listing 4.1: Implementation of the Legacy Version 1.2 requirement. The variable "input_buf" represents the buffer containing the ServerHello message, and "idx" is the index in that buffer where the legacy version bytes are expected to start.

```
#ifdef LEGACY_VERSION_12_KLEE
    klee_assume(!(input_buf[idx] == 0x03 &
                  input_buf[idx + 1] == 0x03));
#endif
```

4.1.2 Not Retry

According to our strict TLS that we defined using Wilson's and Asplund's approach [26], we do not accept any HelloRetryMessages. As a result of this, the ServerHello message should not contain the random string indicating a HelloRetryRequest. Therefore, the 32 bytes representing the random field in the serverHello message must be different from that string in order for it to follow the strict TLS specification and thereby conform to specification. We assume the negation of this requirement using an assume statement in Listing 4.2.

Listing 4.2: Implementation of the Not Retry requirement. "helloRetryRequestRandom" is an array of bytes representing the bytestring that indicates a helloRetryRequest. "RAN_LEN" is a static integer with the value 32.

```
#ifdef NOT_RETRY_KLEE
    for (int i= 0; i < RAN_LEN; i++) {
        klee_assume(input_buf[idx + i] == helloRetryRequestRandom[i])
    }
#endif
```

4.1.3 Session ID Match

The session ID should match the one sent in the ClientHello message. Since our strict TLS only allows for TLS 1.3, the client should send an empty session id field, represented by a single byte of value zero[19]. Therefore, this field in the ServerHello message should also be 0. We represent the negation of the requirement using an assume statement in listing 4.3.

Listing 4.3: Implementation of the Session ID Match requirement.

```
#ifdef SESSION_ID_KLEE
    klee_assume(!(input_buf[idx] == 0));
#endif
```

4.1.4 Cipher Match

According to our strict TLS, a cipher suite is chosen to be ES-128-GCM-SHA256. According to the RFC [19], it is stated that "cipher_suite: The single cipher suite selected by the server from the list in ClientHello.cipher_suites.". This means that we should only choose a cipher suite that is sent in the ClientHello, and with our strict TLS, this means that the cipher AES-128-GCM-SHA256 should be chosen. We can use an assume statement to verify this requirement, shown in listing 4.4.

Listing 4.4: Implementation of the Cipher Match requirement.

```
#ifdef CIPHER_CHECK_KLEE
    klee_assume(!(input_buf[idx] == 0x13 &
                  input_buf[idx + 1] == 0x01));
#endif
```

In this way, we assume that the two bytes should not be 0x1301, and check whether there are any feasible paths where the cipher suite is not AES-128-GCM-SHA256. If there is such a path, we would have a possible nonconformance or bug.

4.1.5 Empty Compression

Prior versions of TLS 1.3 supported compression methods that were defined in this field [19]. However, in TLS 1.3, these are not allowed and it must consist of exactly one byte set to zero, in order to indicate the "null" compression. In the RFC, it is stated that "legacy_compression_method: A single byte which MUST have the value 0.", which we represent using an assume statement shown in listing 4.5. In this way, we only explore the paths where the field is not zero.

Listing 4.5: Implementation and placement of the Empty Compression requirement

```
#ifdef COMPRESSION_CHECK_KLEE
    klee_assume(!(input_buf[idx] == 0));
#endif
```

4.1.6 Requires Version

According to the RFC [19], the SupportedVersions extension is used by the client to specify which TLS versions it supports and by the server to specify which version it is actually using. The RFC also states that this extension should be parsed before any other extensions. However, it may be placed anywhere within the list of extensions meaning that all other extensions before it need to be skipped by the parser until it has found the supported versions extension. To simplify the check, our strict TLS specification places extensions in a predefined order, with the SupportedVersions extension needing to be the first extension in the list. Due to this, we can perform the check by comparing the first two bytes of the extensions list in the input buffer to the byte representations of the extension ID. Therefore, we assume that the first and second bytes are NOT 0x00 and 0x2b respectively, as shown in Listing 4.6.

Listing 4.6: Implementation of the Requires Version requirement

```
#ifdef REQUIRES_VERSION_KLEE
    klee_assume(!(input_buf[idx] == 0x00 &
                  input_buf[idx + 1] == 0x2b));
#endif
```

4.1.7 Version 1.3

As our strict TLS specification only allows TLS 1.3 to be used, the SupportedVersions extension should simply indicate TLS 1.3 and therefore have the value 0x0304. As the SupportedVersions extension should always be the first in the list, the check is done by comparing the fifth and sixth bytes (byte three and four represent the SupportedVersions extension length) of the extension list in the input buffer with 0x03 and 0x04 respectively. We therefore assume that these bytes in the list of extensions are not the aforementioned values and check whether there are any feasible paths for this configuration. The implementation is shown in Listing 4.7.

Listing 4.7: Implementation of the Version 1.3 requirement. The idx variable in this case refers to the beginning of the extensions list.

```
#ifdef VERSION_13_KLEE
    klee_assume(!(input_buf[idx + 4] == 0x03 &
                  input_buf[idx + 5] == 0x04));
#endif
```

4.1.8 Requires KeyShare

The strict TLS specification states that pre-shared keys are not to be used. Therefore, the list of extensions should always include the KeyShare extension. In order to verify this, we check that the extension following the SupportedVersions extension is indeed they KeyShare extension. Hence, we check if KLEE explores paths where the seventh and eighth bytes of the extensions list in the input buffer do not correspond to the ID for the KeyShare extension and thereby have the values 0x00 and 0x33 respectively, as seen in Listing 4.8.

Listing 4.8: Implementation of the requires KeyShare requirement. The idx variable in this case refers to the beginning of the extensions list.

```
#ifdef KLEE
    klee_assume(!(input_buf[idx + 6] == 0x00 &
                  input_buf[idx + 7] == 0x33));
#endif
```

4.1.9 Boundary Aligns

The RFC mentions that, since it can precede a key change, a ServerHello message must be sent in alignment with a record boundary [19]. Since our strict TLS specification does not allow for records to be fragmented, this requirement can be checked fairly easily by comparing the record header length with the handshake header length. If messages were allowed to be fragmented, these two values would not necessarily correspond to each other since the handshake message may be fragmented across multiple records. However, as this is not the case in our configuration, the check is simply done by converting the bytes representing the length of the current record and the length of the ServerHello message into integers and comparing them. To account for the handshake header, which is included in the record length, we also add four to the value representing the length of the ServerHello message. The assume statement can be seen in Listing 4.9.

This is the only assume statement that is always placed before any message processing is done. Instead, it is inserted at a point in the code where a message has been received but not yet processed. This placement ensures that we validate the original message as it arrives, and test paths where the boundary condition violates the protocol rule.

Listing 4.9: Implementation of the *Boundary Aligns* requirement. Here, "curRL.length" is a two-byte array containing the length field of the record header.

```
#ifdef BOUNDARY_ALIGNS_KLEE
    word32 record_len = curRL.length[0] << 8 | curRL.length[1];
    word32 handshake_len = input_buf[idx] << 16 | input_buf[idx + 1] << 8 |
                           input_buf[idx + 2];
    klee_assume(!(record_len == (handshake_len + 4)));
#endif
```

4.2 Placement of Assume Statements

One of the initial challenges encountered when verifying that an implementation adheres to its specification using assume statements is determining the optimal placement of these checks. The issue of placement is nontrivial, as it can influence the behaviour of symbolic execution and lead to varying results. Although the method using assume statements allows checks to be performed at multiple points within the program, certain placements may yield distinct advantages over others depending on the context and testing goals. In the following subsections, we describe the placement strategies we found most straightforward and practical.

4.2.1 When the Message is Received

By leveraging the symbolic socket implementation made by Wilson and Asplund [26], which is shortly described in 3.1.3, we can insert assume statements at an early stage of symbolic execution. Specifically, we place them at the point where the ServerHello message is first processed and before it is sent to the implementation. By placing assume statements immediately after the input buffer is made symbolic, constraints can be enforced before the values propagate through the underlying TLS implementation, in our case WolfSSL. This enables a thorough analysis starting from the initial stages of symbolic execution, ensuring that constraints are enforced prior to these values propagating through the SUT (WolfSSL in our case).

The placement of assume statements in the symbolic socket implementation could offer distinct advantages over alternative placement strategies, such as adding checks directly within WolfSSL and adding constraints closer to particular validation logic (discussed in Subsection 4.2.2). Notably, placing constraints directly within the symbolic socket implementation enhances the replicability and generalisability of the testing methodology, as this approach does not rely on or adapt to the specific design choices of any particular TLS implementation. For example, while putting constraints early within WolfSSL might validate nonconformance or conformance within that implementation, such placement would need to be adjusted for other TLS libraries. By contrast, assume statements applied at the socket level allow this methodology to be reused across multiple implementations with minimal changes to the testing framework, assuming the implementation conforms to the strict TLS specification described in Subsection 3.1.1.

In this approach, assume statements are placed within the symbolic read function directly after the input buffer is modeled as symbolic, see 4.10.

Listing 4.10: A very simplified version of the symbolic_read function originally implemented by Wilson and Asplund [26], with assume statements added after making the buffer symbolic.

```
int symbolic_read ( void * buf , size_t max_read ){
    // fill buffer with max_read # symbolic bytes
    _array_symbolic ( buf , max_read , "sym_read" );

    // Assume statements are added here

    return klee_range (0 , count ); // and symbolic return
}
```

4.2.2 Before Validation Logic

Most often, in some way, the requirements are checked in the source code of the program, to ensure that it conforms to its specification. However, these checks may not be enough to guarantee the programs correctness, and sometimes the checks may not even exist at all. We try to identify the places where checks are made for the corresponding requirement.

To do this, we need to understand the structure of the code itself, and the different functions that together handle the ServerHello message. We identify two functions in the WolfSSL source code that are of special interest, DoTls13ServerHello() and DoTls13HandShakeMsg(). Both these two functions are responsible for handling ServerHello messages and contain the logic we want to check if it is enough to fulfill the requirements derived from RFC 8446. While DoTls13ServerHello() processes the ServerHello message on the client side, DoTls13HandShakeMsg() is called directly when the message is received, before processing.

We use idx, the index counter, which systematically goes through the bytes of the message. With this counter it is easier to find what bytes are being processed at the current time, and is helpful for detecting where checks happen for the specific bytes of the message.

Some of the checks made are straightforward, and require us to simply place an assume statement of the requirement we want to check right before the actual check is made. However, the assume statements do not have to be the same as the checks in the code, instead we look at specific bytes in the message, and assume the values of each individual byte that is involved in the requirement.

Other checks are not so straightforward, and can have additional functions that are called that handle the logic of the checks. In that case, we try to identify the parts of the code of interest, and spend time debugging until the actual checks are made.

The requirement *Boundary Aligns*, is different from the other requirements in this case. Since it involves looking at both the record header, and handshake header, we cannot simply look at the message alone. Therefore, this check is in the DoTls13HandshakeMsg(), which processes the entire message, and is executed before DoTls13Serverhello. According to the WolfSSL GitHub [28], this check needs to be done early, therefore it is separate and before the parsing of the actual ServerHello message.

4.2.3 After the Message is Accepted

Another interesting placement to investigate is right after the message is accepted. An abort statement is placed at the part in the code where the message would be accepted, halting the execution of the program. Since this is where a message would be completed and, we hope, valid, it makes sense to also place the checks in the end.

We run the program in the same way as we do for the other placements of the checks, still negating the expression we assume to hold. The logic for the expressions we check is the same, since we look at specific bytes in the message. If we reach the abort statement, which is right after the KLEE assume statement, this would mean that our requirement is not fulfilled, since we negate the requirement which is supposed to hold.

5 Modifying KQuery Files

The second method we use involves modifying the KQuery files to test a specific requirement. As previously mentioned, KLEE generates KQuery files as a result of symbolic execution. These files encapsulate the logic of each generated test case, including all constraints on symbolic variables. The goal is to allow testers to directly insert requirements as logical constraints on the bytes in the symbolic buffer. Once modified, the KQuery file can be fed into an SMT solver for verification. In our case, we run KLEE on an unmodified version of the implementation without inserted assume statements. We simply follow the approach by Wilson and Asplund [26] to obtain the test cases for every existing path up until the abort statement, represented as KQuery files. This allows us to freely check for the requirements we wish to investigate, simply by inserting them as a query-expression in the KQuery file. This approach is promising because it could potentially eliminate the need to rerun KLEE for each requirement.

5.1 Inserting the Requirements

Since the query-expression in the KQuery file represents the logical condition that must be validated, requirements are inserted into the file in a format consistent with the query-expression. Translating the requirements from C code into KQuery format is a fairly straightforward process following the instructions of the KQuery syntax found in [15], as the underlying logic often remains unchanged. However, in KQuery files, we typically deal with two symbolic buffers: one representing the record header and the other representing the message. This organisation simplifies validation of requirements such as *Boundary Aligns* as it only requires comparing corresponding bytes between the two buffers.

Other requirements are similarly easy to express, with the primary differences lying in syntax rather than logic. For example, Listing 5.1 demonstrates how the Legacy Version requirement is encoded in KQuery format within a KQuery file.

Listing 5.1: An example of the Legacy Version requirement validation in KQuery format

```
array const_arr10[512] : w32 -> w8 = [171 171 171 ... # Total of 512 numbers]
array const_arr15[9] : w32 -> w8 = [0 0 0 0 0 0 0 0 0]
array model_version[4] : w32 -> w8 = symbolic
array read_ret[4] : w32 -> w8 = symbolic
array read_ret_1[4] : w32 -> w8 = symbolic
array sym_read[5] : w32 -> w8 = symbolic
array sym_read_1[512] : w32 -> w8 = symbolic
(query [ (Eq 1 (ReadLSB w32 0 model_version))
         (Eq 22 (Read w8 0 sym_read))
         (Ule N0:(SExt w64 N1:(ReadLSB w32 0 read_ret)) 5)
         ... # Shortened for readability purpose
]
(Eq false (And(Eq 3 (Read w8 4 sym_read_1)) (Eq 3(Read w8 5 sym_read_1))))
)
```

The first part of the code contains declarations for the various arrays used by the program under symbolic execution. The arrays sym_read and sym_read_1 are particularly significant because they represent the two symbolic buffers. After the array definitions, the file includes the constraint-list, which consists of logical constraints applied to specific bytes in order to model the execution path of the program.

Following the constraint-list is the query-expression, which represents the logical condition to be validated. In this example, the query checks the Legacy Version requirement by verifying that the bytes at indices 4 and 5 are both equal to 0x03. All requirements relevant to this thesis in KQuery format can be found in Appendix B.

5.2 Validating the Requirements Using Kleaver

To determine the validity of the query-expression, we use Kleaver to evaluate the KQuery file. Unlike the first method, we do not test the negation of a requirement. Instead, we test whether the requirement always holds for a given path, as represented in the KQuery file. Kleaver does this by verifying whether the query-expression always holds given the constraint-list in the KQuery file. It is possible to negate the requirement and use Kleaver to check if the negation holds. However, if the negation does not hold, we know that the requirement is fulfilled in some cases or all cases, but we do not know which. By not negating the requirement, we can test its satisfiability across all possible assignments. Therefore, if it holds, we are sure it holds for all possible assignments.

When running Kleaver, if the solver returns INVALID, it indicates that not all possible assignments to the symbolic variables satisfies the provided constraints and requirements. This means a nonconformance is likely found. Conversely, if it returns VALID, that means that all possible assignments of values to the symbolic variables make the entire expression true, also indicating that no nonconformances are found.

6 Evaluation

In this chapter, we evaluate the input space coverage (for the assume statement based approach), nonconformance detection capabilities, time overhead, and generalisability of our different methods. In Section 6.1, we describe how this evaluation is done. We then present the results of using our methods in Section 6.2.

6.1 Evaluation Method

This section describes the approach we take to evaluate the different methods.

6.1.1 Evaluating the Assume Statement Approach

In this evaluation, negated conditions are implemented as klee_assume(condition) and instrumented into the test suite using different placement strategies, as described in Chapter 4. This section describes how we measure and compare performance for these different placement strategies. It is distinct from the verification method itself, which details how the assume statements are constructed and inserted into the code. For each experiment, we collect three key metrics from KLEE runs: the total execution time, the number of explored paths (states), and the amount of generated tests. The first two of these metrics are used by both Asadian et al. [3] and Wilson and Asplund [26], and the results vary greatly between the two approaches. Therefore, they constitute interesting metrics to examine in this work. While execution time is important for the feasibility aspect of the proposed method, the path count can be used to compare how much of the state space different placement strategies explore. This can give some insights into coverage and how big of a problem state space explosion is for the different strategies. The amount of generated tests are used to confirm that we find at least the same amount of nonconformances as Wilson and Asplund [26], evaluating the nonconformance detection capabilities of the approach.

We examine three placement strategies in the ServerHello message-processing code:

- **When the Message is Received**
  Insert the negated requirement as an assume statement at the very start of message-parsing, immediately after the raw ServerHello message is received and made symbolic. This forces KLEE to prune invalid inputs before any protocol logic is executed.

- **Before Validation Logic**
  Insert the assume statement just before the implementation's standard validation checks relating to the requirement. The ServerHello message is partially parsed by normal code, but the assume statement is applied right before any internal compliance tests.

- **After the Message is Accepted**
  Insert the assume statement at the final stage of processing, at the point where the implementation accepts the ServerHello message. This allows most of the normal parsing and state updates to proceed before enforcing the negated requirement.

These placements were chosen since they are distinct. The placements could potentially be anywhere in the code that is executed in the parsing of the ServerHello message, but in order to have some structure and for generalisability, we found these placements to be of greatest interest.

During the experiments, KLEE logs the execution time, path count, and amount of generated tests for each individual run, where one run corresponds to the verification of a single requirement. For each placement strategy, we run the symbolic execution and extract the values of the metrics for each requirement individually. We manually extract these values from the information output files for each run. These numerical results form the basis for the comparison of strategies and between methods.

To evaluate the generalisability of a placement strategy, we consider whether it can theoretically be applied to other TLS libraries that adhere to the same strict TLS specification. This involves evaluating the dependencies of the placement strategy. If the strategy for placing assume statements is reliant on WolfSSL's internal structure or naming conventions, it is deemed not generalisable. If, however, the strategy is independent of the specific implementation logic, it is considered highly generalisable.

For example, the early placement strategy involves inserting assume statements based solely on the ability to introduce constraints when bytes are made symbolic within the symbolic socket implementation. Given that this implementation is separate from the TLS library, this strategy is likely to be generalisable across different TLS 1.3 implementations. Evaluating generalisability is crucial for determining the broader applicability and usability of the approach.

6.1.2 Evaluating the KQuery Approach

To evaluate the method involving modified symbolic execution queries post-execution, we focus on the time overhead and if this approach finds the same nonconformances as other approaches. After KLEE completes a run, all branch constraints along each explored path are available in KLEE's KQuery format. We use these files to insert queries containing requirements translated into constraints in the KQuery format and then re-run KLEE's solver on the constraint set. Here, we focus on the time it takes to validate the queries using Kleaver, and make sure that we still find the same nonconformances.

6.1.3 Comparative Analysis

Finally, we compare our methods to our baseline, which is the results of replicating the approach of Wilson and Asplund [26] on our system. We apply their test-case-based symbolic strategy to the same ServerHello scenario and record its execution time, number of explored paths and amount of generated tests. This comparison highlights the effect of using assume statements to guide symbolic execution. An unguided, fully symbolic search tends to explore far more paths and incur greater time, whereas our assume-statement-based strategies focus the search on relevant violation conditions. Comparing these methods is important because it reveals which approach achieves the best trade-off between precision and efficiency.

As Asadian et al. note, requirement-driven symbolic execution can produce a wide variety of violation inducing inputs but its practical value depends on containing the search space. By measuring and contrasting execution time and path counts for each strategy (and the baseline), our evaluation provides concrete insight into how placement of assume statements affect symbolic analysis of TLS implementations.

6.1.4 Hardware and Software Used

For our experiments, we use a dedicated server running Linux Ubuntu 22.04.1 LTS. The server is equipped with a 5.6 GHz processor, 4 cores, and 15 GiB of RAM. Both our experiments and those conducted by Asplund and Wilson [26] are performed on the same server to ensure comparability of the results.

6.2 Results

Here we present the results from our experiments.

6.2.1 KLEE Execution Time for the Assume Statement Approach

Our results show that execution times vary between placement strategies. The overall results, summarised in Table 7.1, indicate that among the three placement strategies, the most time-efficient approach is placing assume statements close to the validation logic within the implementation. Placing assume statements early, when the ServerHello message is received, results in higher execution times compared to validation logic placement but is more efficient than placing checks after the message is accepted. The results for each placement strategy, with respect to execution time, are outlined in the following subsections.

Table 6.1: Comparison of KLEE execution times for Versions 5.6.6 and 5.6.0

Version	Checks Placement	Avg. Time Per Req. (mm:ss)	Total Time (mm:ss)
5.6.0	When the Message is Received	21:20	192:01
	Before Validation Logic	12:00	107:58
	After the Message is Accepted	24:43	222:29
5.6.6	When the Message is Received	19:42	177:22
	Before Validation Logic	08:57	80:35
	After the Message is Accepted	25:35	230:16

When the Message is Received

When assume statements are placed immediately upon receiving the ServerHello message in the symbolic socket implementation, the execution times are relatively uniform across requirements, with the exception of "Legacy Version 1.2" and "Not Retry" which have lower execution time. For all other requirements, the execution time is approximately 25 minutes, which closely aligns with the baseline time obtained when running KLEE without any assume statements and is comparable to the runtime obtained when running Wilson and Asplund's experiments on our system. The execution times for each requirement are presented in Figure 6.1 for WolfSSL version 5.6.0 and Figure 6.2 for WolfSSL version 5.6.6.

Additionally, execution times vary slightly between WolfSSL versions, with the biggest difference shown in the checks for the *Boundary Aligns* requirement. For WolfSSL version 5.6.0, it takes nearly five times longer to verify this requirement compared to version 5.6.6. This difference is expected, as the bug present in version 5.6.0 allows boundaries to misalign, causing nonconforming messages to propagate further through the implementation

and thereby requiring additional paths to be explored during symbolic execution. According to information obtained from the WolfSSL GitHub [28], later versions contain an early sanity check that is invoked early in the execution process, which likely explains the significantly reduced runtime for this requirement in version 5.6.6. Apart from these differences, execution times remain largely consistent across versions for all other requirements.

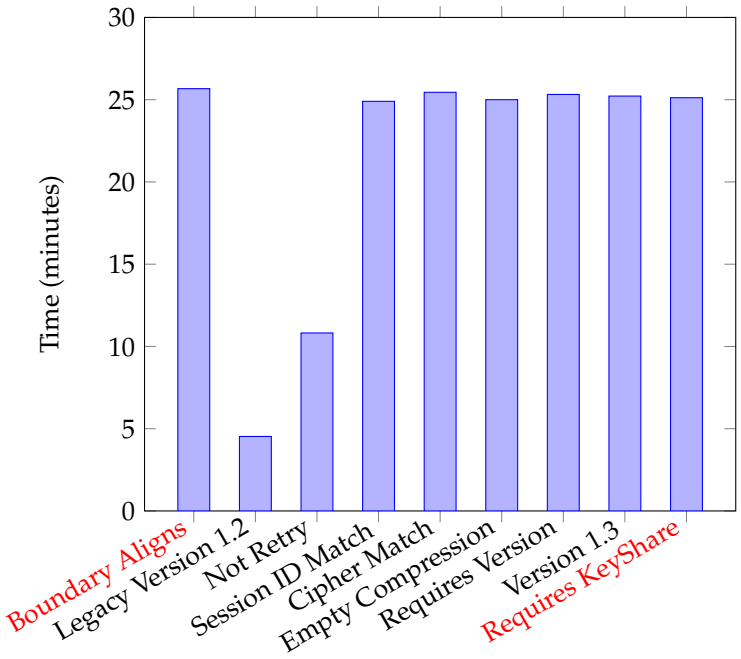


Figure 6.1: Time for version 5.6.0 with assume statements placed early (when the message is received).

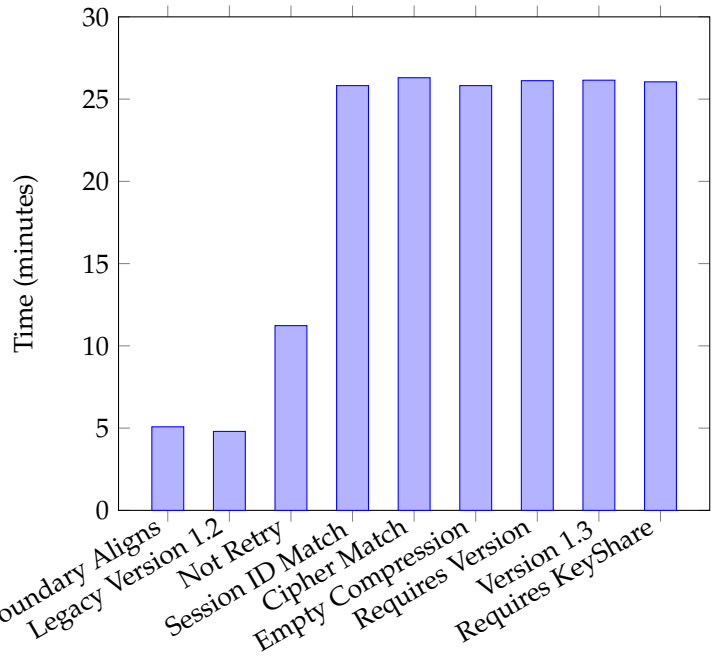


Figure 6.2: Time for version 5.6.6 with assume statements placed early (when the message is received).

Before Validation Logic

The execution times for checks placed immediately before the validation logic within WolfSSL vary greatly across different requirements, ranging from just a few minutes to around 25 minutes. This variation reflects the complexity and symbolic state space associated with each requirement. The execution times for WolfSSL version 5.6.0 are detailed in Figure 6.3, while the corresponding times for version 5.6.6 are presented in Figure 6.4.

This placement strategy demonstrates promising performance in terms of execution time for the majority of requirements. However, the "Session ID Match" and "Cipher Match" requirements have the longest execution times, collectively accounting for over 50% of the total runtime for WolfSSL version 5.6.6. For version 5.6.0, the execution time for the *Boundary Aligns* requirement is also around this range. As previously noted in 6.2.1, the significant difference in execution times between versions for the *Boundary Aligns* requirement can be attributed to the bug in version 5.6.0, which allows messages to bypass boundary alignment checks, resulting in more symbolic paths being explored. Notably, placing the checks closer to the validation logic leads to a substantial reduction in execution times for five out of the nine requirements tested.

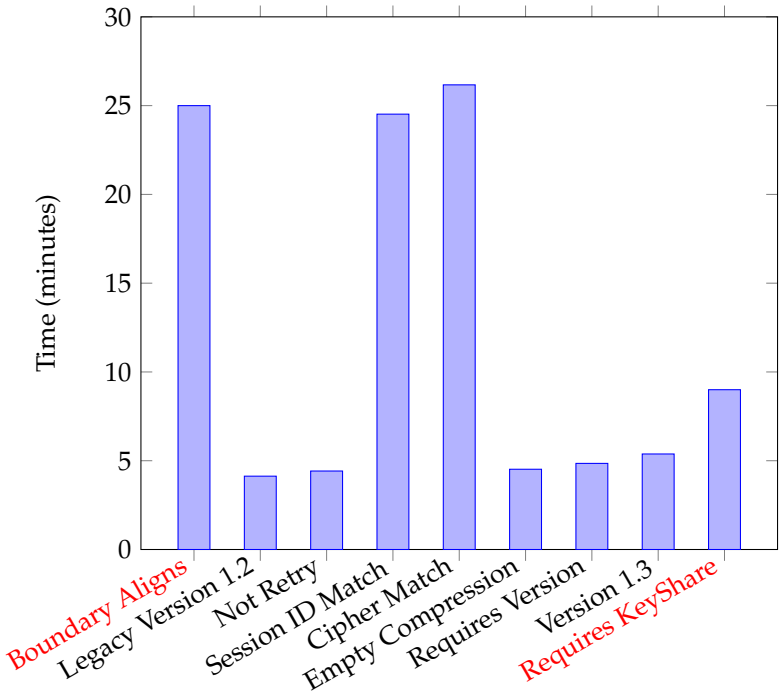


Figure 6.3: Time for version 5.6.0 with assume statements placed before internal validation logic.

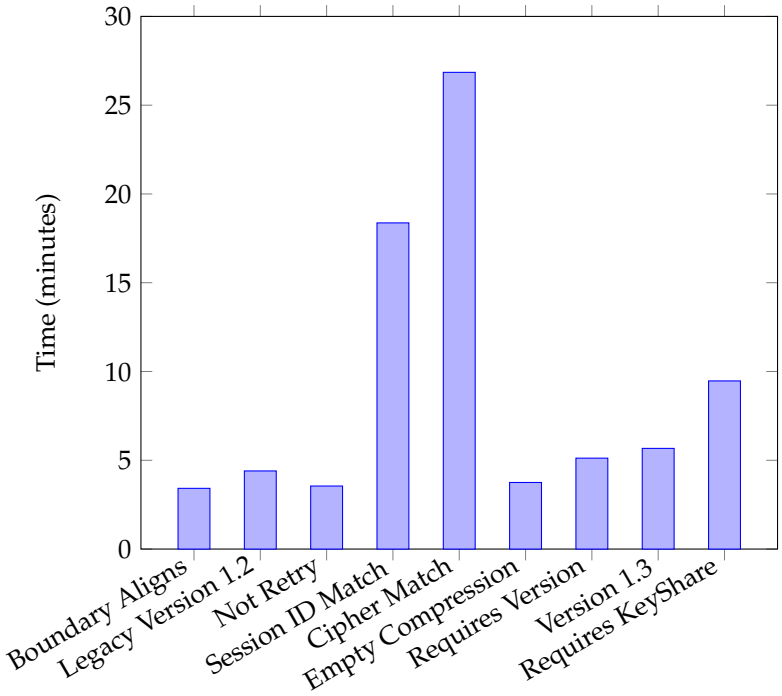


Figure 6.4: Time for version 5.6.6 with assume statements placed before internal validation logic.

After the Message is Accepted

The execution times for running KLEE across different requirements, when checks are placed after the message is accepted, are relatively consistent. For both WolfSSL versions 5.6.0 and 5.6.6, the time required per requirement has little variation. Specifically, the average execution time for version 5.6.0 is just under 25 minutes, while for version 5.6.6, it is slightly over 25 minutes.

For WolfSSL version 5.6.0, the abort statement was reached for both the *Boundary Aligns* and *Requires KeyShare* requirements, as expected, since these are known nonconformances in the implementation. For all other requirements, KLEE generated an "invalid klee_assume call (provably false)" message, which indicates that while a path leading to the abort statement exists, no feasible symbolic assignments could satisfy the negated condition. Therefore, these requirements are strictly enforced with no violations detected. The execution times for checking each requirement are presented in Figure 6.5.

In contrast, for WolfSSL version 5.6.6, the abort statement was not triggered for any of the requirements when checks were placed after the message was accepted. This result confirms that all tested requirements are fully satisfied in this version. The execution times for each requirement are presented in Figure 6.6.

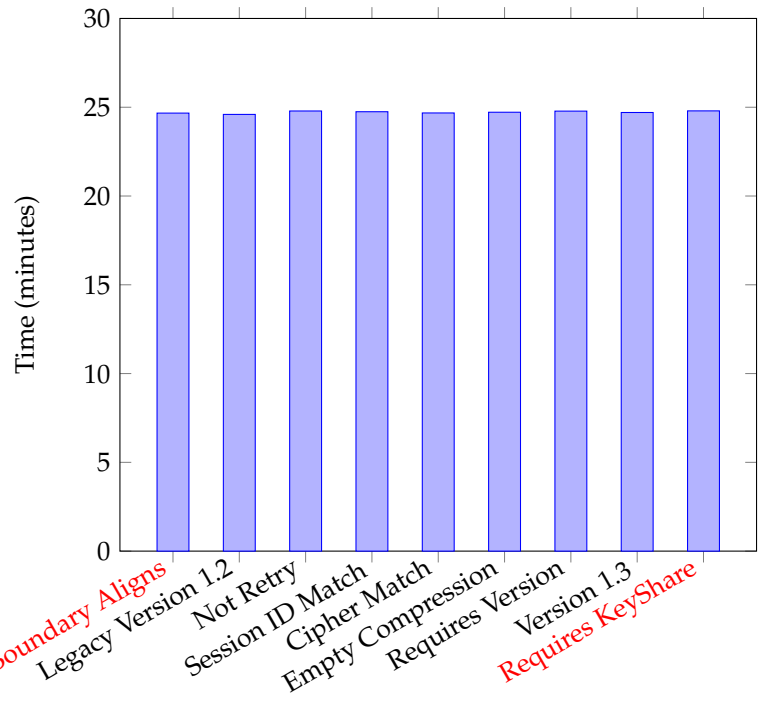


Figure 6.5: Time for version 5.6.0 with assume statements placed after the message is accepted.

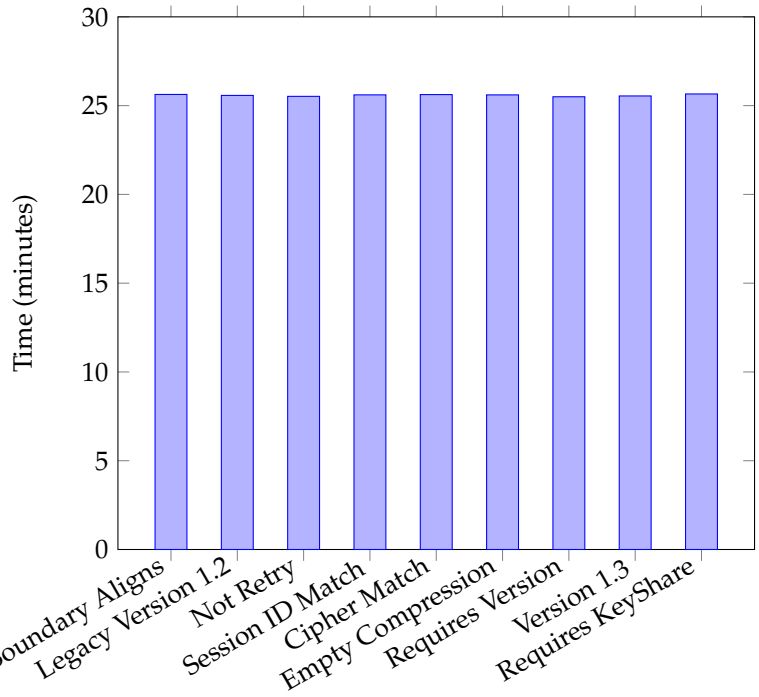


Figure 6.6: Time for version 5.6.6 with assume statements placed after the message is accepted.

6.2.2 Explored Paths and Generated Tests for the Assume Statement Approach

Using KLEE to analyse the ServerHello message parser, the different placement strategies produce different amounts of explored paths and generated tests. From the results, presented in Table 6.2, we can see that generally, placing messages once the message is accepted gives the most paths explored. Placement closer to the validation logic gives the least paths explored while an early placement lays somewhere in between for most requirements.

All identified nonconformances that Wilson and Asplund had were consistently detected across placement strategies. However, an interesting observation is that while placing the checks after the message has been accepted yields 16 test cases for the *Requires Key Share* requirement, it only yields 8 test cases for the other two placement strategies. We did not manage to find any additional nonconformances or bugs using our approach across placement strategies.

When the Message is Received

The strategy of inserting assume statements directly within the symbolic sockets implementation produced relatively consistent results, with the number of explored paths closely resembling that of a full run (i.e., without any assume statements) for most requirements. However, this strategy is sensitive to checks that occur early in the execution flow of the implementation. For example, in the case of the Legacy Version 1.2 requirement, early validation logic in WolfSSL significantly reduces the number of paths explored during symbolic analysis of the requirement's fulfilment. In this instance, the early placement strategy results in over 500 fewer explored paths compared to the full run. Notably, for some requirements, this strategy yields the same number of explored paths as the "Before Validation Logic" placement strategy.

When it comes to exploring paths where nonconformances exist, this placement yields the same amount of paths for the *Boundary Aligns* requirement, and nearly the same amount for the *Requires Key Share* requirement.

Before Validation Logic

Placing assume statements closer to the validation logic of the actual implementation manages to significantly reduce the amount of paths for nearly all requirements compared to performing a full run without assume statements inserted into the code. A prominent example of when this is not the case is for the *Boundary Aligns* requirement in version 5.6.0. This is because the constraint is still being inserted at the start of the ServerHello message handling and ultimately hits the abort statement due to nonconformance of the requirement in that version. In other words, KLEE symbolically executes the whole way from start to the abort statement, as if it were a "normal run". For other requirements, it is on par with the socket placement as it yields the same amount of explored paths.

After the Message is Accepted

The late placement strategy yields the same amount of paths for each requirement for both WolfSSL versions. This is expected since we are only constraining bytes after everything in the message has already been set, and the symbolic exploration is practically done. These checks become more of an assertion of correctness, seeing if messages crafted that are not supposed to be accepted indeed are discarded, as compared to the other placement strategies which only explores the state space where assume statements do not hold. Here, we are instead exploring the whole state space and then checking if it within that state space contains incorrect messages. Reasonably, the amount of paths correspond to the amount of paths that is explored when using Wilson and Asplunds approach, as we are now searching through the whole state space up until the abort statement. Due to this deeper symbolic exploration without any pruning, the strategy of placing checks after the message is accepted consistently resulted in a higher number of explored paths for all requirements.

Generated tests are also the same amount for every requirement. However, the tests themselves are generated due to different reasons. For all requirements except *Requires Key Share* and *Boundary Aligns*, all tests consist of test cases generated due to invalid klee_assume call errors. The interpretation of this is that there is no feasible assertions of symbolic variables that can validate the assume statement, due to prior constraints on the bytes. In other words, this is a positive result as it shows that it is impossible for an incorrect message to go through the implementation. For the *Requires Key Share* and *Boundary Aligns* requirements in WolfSSL version 5.6.0, however, the generated tests consist of output generated from both invalid klee_assume calls and abort errors. This shows that there can be both correctly and incorrectly formatted messages for these requirements.

Requirement	Version	Received		Validation		Accepted	
		Paths	Tests	Paths	Tests	Paths	Tests
Legacy Version 1.2	5.6.0	2094	0	2094	0	2638	16
	5.6.6	2098	0	2098	0	2658	8
Not Retry	5.6.0	2102	0	2102	0	2638	16
	5.6.6	2106	0	2106	0	2658	8
Session ID Match	5.6.0	2574	0	2574	0	2638	16
	5.6.6	2594	0	2594	0	2658	8
Cipher Match	5.6.0	2622	0	2622	0	2638	16
	5.6.6	2642	0	2642	0	2658	8
Empty Compression	5.6.0	2574	0	2126	0	2638	16
	5.6.6	2594	0	2130	0	2658	8
Requires Version	5.6.0	2574	0	2174	0	2638	16
	5.6.6	2594	0	2178	0	2658	8
Version 1.3	5.6.0	2574	0	2198	0	2638	16
	5.6.6	2594	0	2202	0	2658	8
Requires Key Share	5.6.0	2606	8	2390	8	2638	16
	5.6.6	2626	0	2402	0	2658	8
Boundary Aligns	5.6.0	2638	16	2638	16	2638	16
	5.6.6	2094	0	2090	0	2658	8

Table 6.2: Number of paths and test cases for all requirements across versions and placements. For the "Message Received" and "Before Validation Logic" placement strategies, zero tests signifies that no nonconformances or bugs were found. Yielded test cases for these placements mean that there are nonconformances or bugs found. For brevity, 'Received' refers to the placement at message receipt, 'Validation' refers to placement before validation logic, and 'Accepted' to placement after message acceptance.

6.2.3 Modifying KQuery Files

Modifying the KQuery files involves changing the query expressions to check for the satisfiability of the requirements extracted from the RFC. This process was carried out using KLEE to generate the KQuery files, which contain the logic of the paths in the program. We ran KLEE on the program free from any assume statements to obtain all possible paths, resulting in multiple KQuery files, each representing the logic of a single path in the program. We obtained 16 files from the symbolic execution for WolfSSL version 5.6.0 and eight for version 5.6.6. This took 24 minutes and 43 seconds, and 25 minutes and 26 seconds, respectively.

Using Kleaver, we solved the KQuery files for valid assignments of the variables used in the requirements we checked. Each requirement was checked individually for every KQuery file to ensure that all paths fulfill the requirements.

Using this method, we managed to validate the nonconformances identified by [26] and also confirmed that the other requirements were fulfilled. For version 5.6.0, Kleaver returned that the query was invalid for both the *Boundary Aligns* requirement and the *Requires Key Share* requirement, and also returned a counterexample. However, for version 5.6.6, we were able to verify that there are no assignments of variables that would cause the requirements to be unfulfilled, and that there are assignments that fulfill the requirements.

The time taken to run Kleaver for all requirements is negligible for this experiment, taking only around two seconds to solve each query. The time, however, varies between requirements and KQuery files, depending on the complexity of the query to be solved. However, the time did not surpass three seconds for our tests. Table 6.3 shows the time taken to run

Kleaver for each requirement for two selected KQuery files. Kleaver does not look at only one concretisation of the path, but rather the logic of the path itself to determine if it is possible to fulfill the query-expression. The only time-consuming part is manually inserting the requirements into each KQuery file after running KLEE once.

Requirement	**Version 5.6.0 (s)**	**Version 5.6.6 (s)**
Requires Key Share	2.143	2.103
Boundary Aligns	0.497	0.484
Legacy Version 1.2	2.138	2.131
Not Retry	0.062	0.054
Session ID Match	2.137	2.130
Cipher Match	2.219	2.186
Empty Compression	2.139	2.134
Requires Version	2.115	2.182
Version 1.3	2.214	2.130

Table 6.3: Time taken (in seconds) to run Kleaver per requirement and version for two selected KQuery files

6.2.4 Discovered Vulnerabilites

With our approach we achieved the same results as those reported by Wilson and Asplund for WolfSSL versions 5.6.0 and 5.6.6. This includes identifying two major nonconformances in version 5.6.0 related to the boundary alignment and KeyShare requirements. These vulnerabilities were assigned CVE identifiers CVE-2023-3724 and CVE-2023-6937 by WolfSSL.

The first vulnerability (CVE-2023-3724) arises from the client incorrectly accepting ServerHello messages that lack a KeyShare extension, resulting in the use of trivially computable encryption keys for subsequent communication. This vulnerability can compromise the security of the handshake, as an attacker may derive session keys by observing protocol messages. The second vulnerability (CVE-2023-6937) involves the bypassing of handshake message encryption through message coalescing at the record layer, which allows ServerHello to improperly encapsulate successive handshake messages in plaintext. This violates the mandatory TLS 1.3 requirement for proper record boundary alignment and encryption of all handshake messages after the ServerHello.

For further details regarding the vulnerabilities, their causes, and their potential impact, we refer the reader to the work of Wilson and Asplund [26]. These issues were fixed in WolfSSL starting from versions 5.6.2 and 5.6.6, respectively. Our tests validate that WolfSSL version 5.6.6 fully complies with the TLS 1.3 specification for the requirements investigated in this study and contains no known nonconformances or vulnerabilities related to these specific requirements.

7 Discussion

This chapter contains a discussion of the results obtained from the experiments and includes a dedicated section that discusses the work in a broader societal and ethical context.

7.1 Placement Strategies

In this section, we discuss the various placement strategies used in the study and explain their strengths and weaknesses in terms of state space coverage, execution times, and ease of implementation. Each placement strategy has its own distinct advantages and challenges depending on the testing goals and scenario.

Placement early in the execution process, such as within the symbolic socket implementation, offers high generalisability and low implementation complexity. This makes it suitable for testing across different implementations due to its minimal dependence on the internal structure of the SUT. However, an early placement results in slower execution times compared to strategies where assume statements are placed closer to the validation logic.

The strategy of positioning assume statements near the validation logic has better performance in terms of execution time, offering significant gains in efficiency in relation to other placements for most requirements. However, this approach is SUT-specific and can be hard to implement, as it requires knowledge of the internal validation workflow and careful consideration of the appropriate placement for the assume statements.

A late placement strategy, such as enforcing assume statements only after the message is accepted, uses KLEE's natural pruning capabilities to reduce the symbolic state space. This approach functions more as an assertion of correctness at the end of execution but has the highest runtime compared to the other strategies.

In the following subsections, we provide a more detailed analysis of the placement strategies and assess their viability in various testing scenarios. A summary of the discussion is found in Table 7.1.

7.1.1 When the Message is Received

Verifying the requirements using assume statements placed early in the execution process, specifically within the symbolic socket implementation directly in the KLEE library, has both advantages and disadvantages. As highlighted in Section 4.2.1, this approach provides a

more generalised method for testing requirements compared to the other placement strategies. However, it is not as time-efficient as placing requirements closer to the validation logic, as demonstrated by our results. Its key advantage is that it is generic and reusable across multiple TLS 1.3 implementations, requiring little to no knowledge of the internal structure of the system under test. This flexibility makes the approach particularly good for studies where several different implementations are tested.

It is important to note that the implementation may require modifications when testing alternative TLS 1.3 configurations, such as one allowing fragmentation, as the current implementation is specific to our strict TLS implementation. More concretely, the implementation assumes that symbolic bytes are located at predefined positions for e.g. extensions, which may not be true for all configurations. Apart from this, placing assume statements directly within the symbolic socket implementation in KLEE offers a fully generic and reusable approach. This placement is especially viable for testing several TLS 1.3 implementations, as it should be widely applicable and is not constrained by the design specifics of individual implementations.

7.1.2 Before Validation Logic

Placing the assume statements directly within the SUT, where implementation-specific checks are made, presents several challenges that should be considered. Similarly to the late placement strategy, it is dependent on the specific structure and design of the SUT. Due to this, the approach varies between different implementations, making it less generalisable. For each new SUT, one would need to adjust the placement and implementation of the assume statements. This process would typically involve identifying the internal representation of the TLS 1.3 ServerHello message, locating the corresponding variables, and ultimately instrumenting the assume statements. Differences between SUTs can lead to substantial overhead, as implementations often differ significantly in terms of their complexity and internal design. Given these limitations, this placement strategy is potentially ill-suited for scenarios where comparative studies across multiple implementations are performed. In situations where many different implementations are analysed, the effort needed to adapt the strategy for each SUT would likely outweigh the efficiency benefits of the approach. However, if the focus is on a single implementation, this strategy may offer advantages due to its generally faster execution times.

A related and major issue is the complexity of many TLS implementations, which often have large code bases that can be difficult to navigate. Locating the specific places where validation checks are performed can therefore prove to be a big challenge. In some cases, checks related to the same requirement may be located in multiple places within the implementation. This makes it important to carefully determine where to place the assume statements, as incorrectly placed statements could lead to incomplete or inconsistent results. Furthermore, areas of the code that appear to perform validation logic might simply be entry points to functions deeper within the implementation that actually handle the validation. This hierarchical structure can introduce unexpected consequences. For example, assume statements meant to constrain behaviour in connection with the validation logic may accidentally constrain behaviour earlier in the process, long before the validation occurs, reducing the efficiency of the testing process.

As demonstrated by our results, this placement strategy yields the best performance in terms of execution time. This is likely because it generally avoids applying constraints prematurely, unlike the early placement strategy. Instead, it allows paths to be naturally pruned by KLEE during execution until reaching the point where the validation logic is, at which constraints are applied, which increases efficiency. If assume statements are placed too early, long before the validation logic, the behaviour may closely resemble that of the early placement strategy, thereby reducing its performance advantages. This could explain the similar execution times for certain requirements, such as Cipher Match and Session ID Match, across these

two placement strategies. It is likely that constraints for these requirements were applied too early, resulting in the strategy producing results similar to the early placement strategy in terms of execution times and explored paths.

7.1.3 After the Message is Accepted

The late placement strategy ensures that no further code will execute to handle any checks of the message. All logic has already been performed, guaranteeing that the message will not be altered and no additional checks will follow. This placement tests the final accepted message, providing a relatively straightforward approach that does not require extensive understanding of the code.

However, running KLEE with checks placed after the message is accepted takes relatively longer compared to other placements (see Section 6.2.1). The execution time remains consistent for every requirement, as the same code is executed and the requirement check is performed at the very end.

Table 7.1: Summary of placement strategies

Placement strategy	Key characteristic	Implementation level	Execution time
When the Message is Received (Socket)	Reusable and generic	Not SUT specific	Medium
Directly Before the Validation Logic	Fast	SUT specific	Fast
When the Message is Accepted	Full state space coverage	SUT specific	Slow

7.2 Modifying KQuery Files

Modifying the KQuery files can be a way to improve the time it takes to run KLEE for multiple requirements, while still exploring all concretisations of symbolic bytes, and not just one concrete concretisation for a test case. Since Kleaver checks for satisfiability of the query expression, it will guarantee the fulfillment of the requirements for all concretisations. To the best of our knowledge, validating TLS requirements symbolically using Kleaver and modified KQuery files has never been done before.

Running Kleaver takes only a few seconds, making it a quick task to change the query expression and rerun Kleaver. This method is efficient and could potentially be used to verify requirements for other TLS implementations. Since all TLS 1.3 implementations follow the same specification, the requirements should be consistent. However, different implementations may have unique local differences in their requirements, depending on the choice of extensions, versions, etc., used during implementation. Ultimately, one could run KLEE on various TLS 1.3 implementations and extract different requirements from the RFC. As long as these requirements can be implemented in KQuery format, they can be tested using Kleaver.

7.3 Symbolisation Trade-offs: Entire ServerHello Messages vs. Specific Bytes

For the experiments conducted, the entire ServerHello message was made symbolic. One reason for this is to have the same prerequisites for our experiments as those conducted by Wilson and Asplund[26], to allow for a more comparable result. However, for our experiment, when we test specific requirements, and therefore look at specific bytes, we might not

need to make the entire message symbolic. Making the entire message symbolic allows for more paths to be tested, and more combinations of all the possible values of the message, but this also slows down the testing process. If we, for instance, wish to test the Version 1.3 requirement, it would most likely be enough to make the two bytes that represent the version symbolic, since these are probably the only relevant bytes for the requirement. According to Asadian et. al. [3], only making the relevant bytes symbolic and testing one requirement at a time allows for better scalability. On the other hand, it requires the person testing to be careful and attentive, and it also makes it easier to miss requirements that could only be violated when other bytes are also symbolic. Therefore, our approach may not be the best for scalability, but it allows for more thorough testing.

7.4 Comparison to Wilson and Asplund's Approach

Our original hypothesis and the core of this work was that assume statements would allow for a more thorough search where we explore all possible concretisations of symbolic paths. By not relying on a specific test case concretisation, we hoped to be able to possibly detect more nonconformances and/or bugs in the WolfSSL TLS 1.3 implementation. Our results, however, show that this is not the case as we obtain the same results as Wilson and Asplund [26], discovering exactly the same amount of nonconformances. This likely indicates that no further nonconformances exist for these requirements.

The test-case-only approach proposed by Wilson and Asplund [26] is both simpler and more computationally efficient compared to our methodology. Their approach requires only a single symbolic execution run and focuses exclusively on paths represented by concretised test cases. While our assume statement method successfully detects nonconformances in WolfSSL, it also introduces additional computational overhead. We believe that this extra computational effort is justified due to the exhaustive nature of the search it provides. Although no new nonconformances were found, we validated that all concretisations belonging to the paths tested conform to the specification.

The method of using assume statements ensures that all possible concretisations are explored. However, it can be redundant when testing certain requirements in certain implementations, as demonstrated by the results of this study. In such cases, the additional computational effort does not provide meaningful benefits, as the results are the same while testing performance is up to nine times slower. Thus, the flexibility offered by exploring all possible concretisations with assume statements does not always provide improvements in identifying deeper or more subtle edge cases within this context. This is, however, difficult to determine before considering all concretisations, and the argument could be made that it is always better to test more exhaustively. As noted in their original research, Wilson and Asplund suggested that their selected test cases were sufficiently representative for the requirements being tested. Our findings confirm this, as our more exhaustive approach did not uncover any additional results beyond those identified using their test-case-only strategy. In other scenarios, it may not be that test cases are sufficiently representative, in which case our methods guarantee that all concretisations are considered.

It is possible that our method of inserting assume statements is more appropriate in other contexts and remains a viable approach for other requirements or implementations. Given additional time and resources, we would have liked to extend our analysis to other TLS 1.3 configurations, other implementations, or requirements that are less trivial, in order to better demonstrate the potential advantages of our methodology. However, due to the constraints of this study, we were only able to apply our approach to the same scenario used by Wilson and Asplund [26].

One key contribution of this study is the demonstration of an alternative approach that could serve as a "silver bullet" for requirement checking. We show that it is possible to evaluate requirements after a single symbolic execution run by inserting those requirements di-

rectly into the KQuery file generated by KLEE and then feeding the file to a solver. This approach is highly promising. It avoids introducing additional computational overhead, since the symbolic execution is performed only once to generate the KQuery files, but still ensures that all concretisations are accounted for. Therefore, it surpasses both the assume statement approach taken in this study and the test-case-only approach taken by Wilson and Asplund [26]. It achieves exhaustive coverage while keeping execution times comparable to a single symbolic run, with the added advantage of exploring all concretisations post-execution. A summarized comparison of Wilson and Asplunds approach, modifying KQuery files, and the assume statement approach can be found in Table 7.2.

The biggest limitation of this technique is its reliance on the tester being comfortable with the KQuery language, which may be a barrier for adoption. Also, some requirements may be difficult to express only using the structures and constructs available within KQuery. For instance, requirements concerning fields where a larger number of bytes are to be checked result in larger queries, as showcased in Appendix B. This suggests that, for more complex requirements, the KQuery modification approach could potentially be impractical. This remains relatively speculative, however, as our study did not extend to those scenarios. As such, we leave the evaluation of this limitation and the broader exploration of this method for future work.

Table 7.2: Comparison of verification approaches discussed in the thesis

Dimension	**Assume Statement Approach**	**KQuery Modification Approach**	**Wilson & Asplund's Approach [26]**
Description of Method	Insert negated constraints directly in code using `klee_assume`	Post-execution query modification in generated KQuery files	Test-case–based analysis using symbolic execution traces
Placement of Constraints	Early (socket), before internal validation, or after message acceptance	Post symbolic execution	Post symbolic execution
Execution Time	Medium–High (depends on placement)	Low (one run of symbolic execution)	Low (one run of symbolic execution)
Considers All Concretisations	Yes	Yes	No
Generalisability	Low-Medium-High (depends significantly on placement strategy)	High (Not SUT specific)	High (Not SUT specific)
Complexity of Implementation	Medium-High (manual constraint insertion required)	Medium–High (requires KQuery manipulation)	Medium (requires trace analysis setup)

7.5 Method Limitations and Threats to Validity

While both approaches outlined in this paper offer good insight into the conformance testing of TLS 1.3 through symbolic execution, there are some limitations that may affect the generalisability, replicability, and interpretability of the results.

7.5.1 Assumptions and Internal Validity

This study relies on several assumptions that may affect the internal validity of the results. First, the correctness of the evaluation depends on our manual translations of requirements from RFC 8446 [19] into symbolic constraints written in C code. These translations may fail to fully capture the intended semantics of the specification. Furthermore, constraint placement, particularly for the "Before Validation Logic" strategy, requires detailed understanding of the

implementation at hand. Incorrect placement can result in relevant checks being bypassed, which in turn affects both accuracy and coverage.

We also assume that klee_assume behaves as intended. However, available documentation on the function is limited, and its internal workings are not fully understood. As a result, the precise effects of these statements remain a bit unclear, which may impact the reliability of the results. One unintended consequence of using klee_assume arose when we initially applied short-circuit operators (e.g., &&, ||) within assume statements, which led to a significant and unexpected increase in explored paths. This issue, and the corrective use of bitwise operators instead, is described in more detail in Section 7.5.4.

Lastly, the study builds upon modifications made to both KLEE and WolfSSL (described briefly in Chapter 3), made by Wilson and Asplund. Our evaluation assumes that their changes are both correct and complete. Such reliance reduces replicability: to recreate our findings, others need not only to replicate our approach but also accurately mimic the changes introduced by Wilson and Asplund. For newer versions of WolfSSL or completely different implementations, considerable rework would probably be required.

7.5.2 External Validity

The study is limited in scope to a single protocol (TLS 1.3), a certain implementation of that protocol (WolfSSL), and one specific message (ServerHello) within that implementation. This narrow scope restricts the generalisability of the findings. There is no evidence that the proposed method would work effectively on other TLS messages, different protocol versions, or unrelated protocols altogether.

Although we argue that the "When the Message is Received" placement strategy and the KQuery modification method are most probably generalizable to other implementations and versions of the protocol, we have not empirically shown this to be the case. These methods were only ever tested against WolfSSL and the TLS 1.3 ServerHello message. Thus, while the underlying principles themselves might be applied more broadly, we cannot make absolute guarantees of correctness, scalability, or effectiveness upon other implementations, message formats, or protocol variations without further testing.

7.5.3 Construct and Conclusion Validity

We use the number of paths explored as a central metric to measure symbolic execution performance and input space coverage. This metric does not directly reveal the test case completeness. It may provide a rough indication of symbolic complexity or path explosion, but it does not necessarily provide any insights into, for instance, code coverage. Hence, our results on the performance of the methods need to be viewed with this in mind, and other metrics could provide a more advanced view of how well the methods work.

7.5.4 Impact of Logical Operators on Path Explosion

While developing our methodology, we encountered unexpected behaviour related to the use of klee_assume statements. Initially, we used short-circuit operators, such as && and ||, within the assume statements without further consideration. However, during testing, we observed that this setup resulted in a significantly higher number of explored paths compared to running symbolic execution without augmenting the test suite with assume statements. This outcome was puzzling at first, as it seemed counterintuitive.

We reviewed the KLEE documentation and discovered that the use of short-circuit operators in klee_assume statements can indeed lead to unexpected behaviour, including that KLEE might produce more symbolic paths than expected [12]. This happens because short-circuit operators introduce additional branching in symbolic execution, which causes KLEE to explore paths that may not align with the intention of the test.

To address this issue, we revised our implementation to use bitwise comparison operators (& and |) within the assume statements. This resulted in a significantly reduced and more reasonable number of paths generated during symbolic execution. For example, for the "Requires Version" requirement, the number of explored paths decreased from approximately 9000 to 2594, a result that aligns better with our expectations. This change improved both the precision and performance of our testing methodology.

7.6 The Work in a Wider Context

Testing implementations of critical protocols such as TLS is essential for better cybersecurity. TLS protocols are the foundation of modern digital communication, enabling secure exchanges in areas such as banking, healthcare systems, and ecommerce platforms. Research efforts in this area play an important role in identifying and mitigating vulnerabilities that could expose sensitive data, including personal, financial, and medical information. Ensuring strict adherence to protocol standards creates trust among users and reinforces the reliability of secure communication, which is vital to the functioning of modern society. Furthermore, maintaining robust cybersecurity through exhaustive testing and disclosing vulnerabilities found in them reduces the risk of attacks on critical infrastructure, thereby protecting societal and economic stability.

If not exhaustively verified, implementations of critical protocols could cause massive damage as illustrated by the Heartbleed vulnerability[10][7] or the POODLE attack[16] discussed in the introduction of this thesis. In these cases, millions of websites were affected and a lot of sensitive data was exposed. Public trust in cybersecurity may fall if more vulnerabilities like these are exploited by malicious actors. Therefore, research efforts like those presented in this thesis are vital to prevent the occurrence of similar vulnerabilities and to ensure the security and reliability of critical protocol implementations. It is of course imperative that researchers responsibly disclose any vulnerabilities identified in such research. While the knowledge gained through such research has the potential to improve security, there is always a risk that it could be misused to develop exploits. Consequently, researchers have a professional and ethical obligation to prioritise enhancing security and reporting any discovered issues, rather than exploiting the vulnerabilities for malicious or unauthorised purposes. Additionally, ethical considerations are essential to protecting data privacy, particularly when testing systems that may expose sensitive or private information during the debugging or verification processes.

Also, the balance between privacy and security remains a topic of ongoing debate. Encryption protocols, including TLS, are frequently examined in the context of discussions surrounding government surveillance and the protection of individual privacy rights. Research on TLS inherently intersects with these broader societal and ethical debates, as it contributes to the development and refinement of encryption technologies that play a pivotal role in both protecting individual privacy and addressing national security concerns.

Furthermore, fostering collaboration between industrial contributors and academia is crucial for the development of secure software. Academic research, such as the work presented in this study, can provide valuable insights to support open-source developments and advance the state of cybersecurity. To ensure robust and secure infrastructure, it is essential for academia, industry, and policymakers to work together. This collaboration enables research efforts to influence the development of future secure software and drive the adoption of new secure design policies. Additionally, researchers play a key role in sharing their findings in ways that inform the development of secure systems, promote better coding practices, and advance education on secure standards.

8 Conclusion

In this thesis, we developed a method using KLEE assume statements to verify a TLS implementation against its specification, aiming to identify nonconformances. We investigated the placement of these assume statements to understand their impact on the time required for program analysis using KLEE and the variation in state space exploration. Our method was compared to the approach by Wilson and Asplund[26] to evaluate robustness and execution time. Additionally, we explored modifications to KQuery files to test requirements from the RFC[19], facilitating quicker and more thorough testing.

The placement of KLEE assume statements significantly affects both the analysis time and the number of paths explored. Placing checks within the socket is more generalisable to other TLS implementations but increases the analysis time. Conversely, placing checks before the validation logic reduces the analysis time but requires a deeper understanding of the code and is specific to the particular TLS implementation. Placing checks after the message acceptance is more thorough, as no code is executed post-validation, but it increases the analysis time due to the lack of path pruning.

Compared to Wilson and Asplund's[26] approach, our method is more time-consuming because we check each requirement separately. However, our method examines all possible values of a path rather than a single concretisation. To expedite the process, we modified KQuery files to check requirements, which significantly reduced the time to a few seconds per requirement after running KLEE once.

8.1 Research Questions

Below, the research questions introduced in Section 1.3 are summarised based on our findings from the literature and the experiments carried out by the authors.

How can TLS 1.3 requirements from RFC 8446, specifically those concerning the ServerHello message, be translated into KLEE assume statements for symbolic execution?

To derive the requirements from the RFC, one can follow the approach described by Wilson and Asplund[26]. Once the requirements are derived and it is clear what you wish to test for, these requirements need to be converted into C-code to be inserted into a KLEE assume

statement. The requirements can then be negated to test the paths that do not conform to the specification.

How do different placement strategies for assume statements: early placement, before validation, and after message acceptance, affect the symbolic execution performance and ability to detect nonconformances?

The placement of KLEE assume statements can have different effects, where both the number of paths and the execution time vary, as presented in Section 6.2.1 and Section 6.2.2. The placement strategies are further discussed in Section 7.1. For all three strategies investigated, the known nonconformances associated with the respective WolfSSL versions were successfully detected.

How can post-execution modification of KQuery files be used to verify multiple TLS requirements efficiently, and what are the benefits and limitations of this approach compared to in-code instrumentation?

The KQuery files generated when using KLEE to analyse the ServerHello message can be modified. Using Kleaver, these files can be solved for the satisfiability of a query expression. This query expression can be adjusted to test any specific requirement, and the SMT solver will check if the expression is valid. A single run of the program using KLEE is necessary to generate the KQuery files with the program's logic, without any KLEE assume statements. Running Kleaver with these KQuery files only takes a few seconds, while still verifying if the requirement always holds.

However, this approach still requires a full run with KLEE and knowledge of the KQuery format to manually insert the requirements into each KQuery file, which can be time-consuming since every single KQuery file needs to be modified.

How does the proposed method, including both in-code assume statements and KQuery file modifications, compare to Wilson and Asplund's test-case–based method in terms of thoroughness and time overhead?

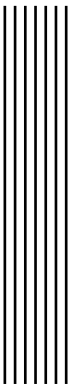
Wilson and Asplund use a test-case–based method where a single concretisation of a test case is analysed to see if it conforms to the specification. This can potentially miss concretisations that would violate the requirement. The assumption-based method, on the other hand, negates the requirement being tested to obtain all paths that lead to a nonconformance, which in theory is more thorough. Running KLEE once without any assumptions and modifying the KQuery file allows for checking requirements using Kleaver, which will check for the satisfiability of the requirements based on the logical path of the program, which is also a robust way of testing the requirement. The performance and test coverage are presented in Section 6.2.1 and Section 6.2.2, and discussed in Section 7.4 and Section 7.1.

8.2 Future work

Since our approach did not yield different results compared to the research this thesis is based on, likely due to the nature of the tested requirements, future work could explore the methodology on requirements that are less trivial to better demonstrate its potential. Additionally, more generalisable strategies, such as the early placement of assume statements or the modification of KQuery files, could be applied to testing other TLS 1.3 implementations. Expanding this work in such directions would address limitations arising from the time and resource constraints of this study.

Future research should also explore the modification of KQuery files in greater depth to better understand how this approach can streamline the validation process, while also examining its limitations and benefits more comprehensively. Although this method shows

promise, as indicated by the findings in this thesis, further investigation is required to fully assess its potential, particularly by applying it to more complex and advanced requirements. Furthermore, future work could focus on automating this process by developing techniques to analyse a KQuery file and automatically generate testable requirements in the KQuery language for insertion into the KQuery files. Achieving such automation could significantly reduce the time and effort required to conduct exhaustive conformance testing of protocol implementations.



Bibliography

[1] https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/. [Accessed 24-02-2025].

[2] Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, and Konstantinos Sagonas. "Monitor-based Testing of Network Protocol Implementations Using Symbolic Execution". In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*. 2024, pp. 1–12.

[3] Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. "Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification". In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022, pp. 70–81.

[4] Hooman Asadian, Paul Fiterău-Broştean, Bengt Jonsson, and Konstantinos Sagonas. "Testing IoT Protocol Requirements Using Fuzzing and Symbolic Execution: Application to CoAP". In: *2024 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE. 2024, pp. 48–54.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.

[6] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. "EXE: Automatically generating inputs of death". In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), pp. 1–38.

[7] Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. "Heartbleed 101". In: *IEEE security & privacy* 12.4 (2014), pp. 63–67.

[8] Sze Yiu Chau, Omar Chowdhury, 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 (SP)*. IEEE. 2017, pp. 503–520.

[9] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[10] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. "The matter of heartbleed". In: *Proceedings of the 2014 conference on internet measurement conference*. 2014, pp. 475–488.

[11] Jean Goubault-Larrecq and Fabrice Parrennes. "Cryptographic protocol analysis on real c code". In: *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 363–379.

[12] *Intrinsics . KLEE — klee-se.org*. `https://klee-se.org/docs/intrinsics/`. [Accessed 22-05-2025].

[13] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* (1976), pp. 385–394.

[14] KLEE. *Intrinsics*. `https://klee-se.org/releases/docs/v2.3/docs/intrinsics/`. Accessed: 2025-05-28. 2023.

[15] *KQuery · KLEE — klee-se.org*. `https://klee-se.org/docs/kquery/`. [Accessed 07-05-2025].

[16] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. "This POODLE bites: exploiting the SSL 3.0 fallback". In: *Security Advisory* 21 (2014), pp. 34–58.

[17] Ricardo Peña, Jaime Sánchez-Hernández, Miguel Garrido, and Javier Sagredo. "SMT-based test-case generation and validation for programs with complex specifications". In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems: Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*. Springer, 2023, pp. 188–205.

[18] Andrew Reynolds and Viktor Kuncak. "Induction for SMT solvers". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2015, pp. 80–98.

[19] *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3 — rfc-editor.org*. `https://www.rfc-editor.org/rfc/rfc8446`. [Accessed 24-02-2025].

[20] *Solver Chain. KLEE — klee-se.org*. `https://klee-se.org/docs/solver-chain/`. [Accessed 27-05-2025].

[21] Juraj Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2016, pp. 1492–1504.

[22] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. "SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications". In: *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 695–709.

[23] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. "Specification-Based Symbolic Execution for Stateful Network Protocol Implementations in IoT". In: *IEEE Internet of Things Journal* 10.11 (2023), pp. 9544–9555.

[24] Sean Turner. "Transport layer security". In: *IEEE Internet Computing* 18.6 (2014), pp. 60–63.

[25] Shameng Wen, Qingkun Meng, Chao Feng, and Chaojing Tang. "A model-guided symbolic execution approach for network protocol implementations and vulnerability detection". In: *PloS one* 12.11 (2017). Article e0188229.

[26] Johannes Wilson and Mikael Asplund. "Analysing TLS Implementations Using Full-Message Symbolic Execution". In: *Nordic Conference on Secure IT Systems*. Springer. 2024, pp. 283–302.

[27] WolfSSL. *WolfSSL – Embedded SSL/TLS Library*. `https://www.wolfssl.com/`. [Accessed 23-01-2025].

[28] Julek - WolfSSL. *Additional TLS checks by julek-wolfssl · Pull Request #7029 · wolfSSL/wolfssl*. Accessed: 2023-10-11. 2023. URL: https://github.com/wolfSSL/wolfssl/pull/7029.

[29] Li Zhang and Vrizlynn LL Thing. "A hybrid symbolic execution assisted fuzzing method". In: *TENCON 2017-2017 IEEE Region 10 Conference*. IEEE. 2017, pp. 822–825.

A Strict TLS

Table A.1: Strict TLS Requirements

Requirement	Explanation and Reasoning
Use Only TLS 1.3	Enforce the use of only TLS 1.3, disabling all prior versions. This ensures utilisation of the latest security features and mitigates downgrade attacks, which can force communication to use less secure older versions. Servers supporting only older versions will be incompatible.
Single Cipher Suite	Restrict the client to select only one cipher suite, providing maximal control over cryptographic parameters and simplifying analysis. This reduces variability, making security verification more straightforward and ensuring the cipher's security properties are maintained.
Support and Generate Keys for a Single Key Group	Limit key exchange to a specific, supported key group, determining the security strength of the Diffie-Hellman exchange. Servers supporting only other groups will be rejected, ensuring consistent security assumptions.
Predefine Extension Order	Require that extensions follow a predefined order during parsing, reducing state space explosion during symbolic execution and ensuring predictable processing order.
Support Versions Extension First	Ensure that the SupportedVersions extension is the first in both ClientHello and ServerHello messages. This guarantees that version negotiation occurs prior to other extension processing, simplifying the handshake logic and verification.

Requirement	Explanation and Reasoning
Disable Record Fragmentation	Enforce that records are not fragmented unless exceeding size limits, allowing complete messages to be read directly. This prevents the combinatorial complexity associated with message fragmentation during symbolic analysis.
No Use of PSK	Disallow the use of pre-shared keys, which can introduce additional security considerations. PSK handshakes are handled separately, outside the primary TLS 1.3 handshake flow.
Disable DTLS	Do not support DTLS, the datagram version of TLS, due to fundamental differences such as message reordering and retransmission mechanisms. Treat DTLS as an entirely separate protocol to avoid incorrect assumptions during analysis.
No Early Data (0-RTT)	Disable 0-RTT (early data) to avoid weaker security guarantees, since early data is authenticated after transmission, posing potential replay and downgrade risks.
Reject Unknown Extensions	Any received extension that is not explicitly supported or recognised must result in connection rejection, ensuring strict compliance with the protocol specification and preventing unintended behaviours.

B Requirements in KQuery Format

BOUNDARY ALIGNS

```
(Eq
(ZExt w32
(Add w16
(Shl w16 (ZExt w16 (Read w8 3 sym_read)) 8)
(ZExt w16 (Read w8 4 sym_read)))
)
(Add w32 4
(Add w32
(Shl w32 (ZExt w32 (Read w8 1 sym_read_1)) 16)
(Add w32
(Shl w32 (ZExt w32 (Read w8 2 sym_read_1)) 8)
(ZExt w32 (Read w8 3 sym_read_1)))
)
)
)
```

LEGACY VERSION 1.2

```
(And(Eq 3 (Read w8 4 sym_read_1)) (Eq 3(Read w8 5 sym_read_1)))
```

NOT RETRY

```
(Eq false
(And (And (And (And (And (And (And (And (And (And
(And (And (And (And (And (And (And (And (And (And
(And (And (And (And (And (And (And (Eq 156 (Read w8 37 sym_read_1))
(Eq 51 (Read w8 36 sym_read_1)))
(Eq 168 (Read w8 35 sym_read_1)))
(Eq 200 (Read w8 34 sym_read_1)))
(Eq 226 (Read w8 33 sym_read_1)))
(Eq 9 (Read w8 32 sym_read_1)))
(Eq 158 (Read w8 31 sym_read_1)))
(Eq 7 (Read w8 30 sym_read_1)))
```

```
(Eq 94 (Read w8 29 sym_read_1)))
(Eq 140 (Read w8 28 sym_read_1)))
(Eq 187 (Read w8 27 sym_read_1)))
(Eq 122 (Read w8 26 sym_read_1)))
(Eq 22 (Read w8 25 sym_read_1)))
(Eq 17 (Read w8 24 sym_read_1)))
(Eq 162 (Read w8 23 sym_read_1)))
(Eq 194 (Read w8 22 sym_read_1)))
(Eq 145 (Read w8 21 sym_read_1)))
(Eq 184 (Read w8 20 sym_read_1)))
(Eq 101 (Read w8 19 sym_read_1)))
(Eq 30 (Read w8 18 sym_read_1)))
(Eq 2 (Read w8 17 sym_read_1)))
(Eq 140 (Read w8 16 sym_read_1)))
(Eq 29 (Read w8 15 sym_read_1)))
(Eq 190 (Read w8 14 sym_read_1)))
(Eq 17 (Read w8 13 sym_read_1)))
(Eq 97 (Read w8 12 sym_read_1)))
(Eq 154 (Read w8 11 sym_read_1)))
(Eq 229 (Read w8 10 sym_read_1)))
(Eq 116 (Read w8 9 sym_read_1)))
(Eq 173 (Read w8 8 sym_read_1)))
(Eq 33 (Read w8 7 sym_read_1)))
(Eq 207 (Read w8 6 sym_read_1))))
```

SESSION ID MATCH

```
(Eq 0 (Read w8 38 sym_read_1))
```

CIPHER MATCH

```
(And(Eq 19 (Read w8 39 sym_read_1)) (Eq 1 (Read w8 40 sym_read_1)))
```

EMPTY COMPRESSION

```
(Eq 0 (Read w8 41 sym_read_1))
```

REQUIRES VERSION

```
(And(Eq 0 (Read w8 44 sym_read_1)) (Eq 43 (Read w8 45 sym_read_1)))
```

VERSION 1.3

```
(And
(Eq 3 (Read w8 48 sym_read_1))
(Eq 4 (Read w8 49 sym_read_1))
)
```

REQUIRES KEY SHARE

```
(And(Eq 0 (Read w8 50 sym_read_1)) (Eq 51(Read w8 51 sym_read_1)))
```