A Scenario Generation for Model-based Testing of TLS using Maude Strategy*

Jaehun Lee and Kyungmin Bae†

Pohang University of Science and Technology
{thkighie1224,kmbae}@postech.ac.kr

Abstract

This paper introduces a systematic approach to model-based testing of the Transport Layer Security (TLS) protocol by leveraging Maude strategies. We present an encoding that simplifies the use of Maude Strategy, enabling the formalization of counterexample situations derived from the requirements of RFC 5246 and RFC 8446. With this encoding, only the rewrite rules corresponding to the given strategies are executed, and the resulting state sequences are systematically transformed into a scenario language that expresses executable test cases. Our case study demonstrates how our method can represent requirement-violating scenarios, thereby confirming the effectiveness of the proposed method in capturing deviations from the expected protocol behavior. This work contributes to the rigorous evaluation of TLS by providing a structured method for generating counterexamples, and it opens opportunities for future extensions of the encoding to test a broader range of RFC violation cases.

Keywords: Model-based Testing, TLS protocol, Maude Strategy

1 Introduction

Transport Layer Security (TLS) [1, 2] is a security protocol designed to ensure data protection and integrity between two parties. TLS is widely used in various applications, such as HTTPS, email, instant messaging, and VoIP. In particular, TLS plays a critical role in mobile Internet environments, where smartphones and IoT devices rely on it to secure communications over potentially untrusted wireless networks. Since its specification was released, numerous security vulnerabilities in the TLS protocol have continuously been discovered [3]. These flaws are difficult for programmers to detect, and successful exploitation can result in significant damage [4]. Therefore, the importance of verification techniques to thoroughly analyze and prevent security vulnerabilities in advance is increasingly emphasized.

Formal methods for rigorously verifying TLS are being actively researched. For example, tools like TAMARIN [5] and AVISPA [6] provide model-checking capabilities that specify security protocols at the specification level and automatically verify security vulnerabilities caused by man-in-the-middle attacks. Fuzzing-based TLS testing [7, 8, 9, 10, 11] and combinatorial testing [12] automatically generate test cases to detect various vulnerabilities found at the code level, successfully identifying attacks such as Heartbleed [13] and POODLE [14]. Model-based testing with ProVerif [15] has also been introduced to detect logical errors in processing TLS message sequences in various TLS implementations. Model-based testing with model learning [16, 17, 18] infers state machine models, which capture the states of the protocol implementations.

However, the above verification methods for TLS software have inherent limitations. The formal verification of the TLS specification alone cannot detect vulnerabilities present in the TLS implementation, such as Heartbleed [13]. Moreover, existing testing tools for TLS libraries face two key limitations: they fail to systematically bridge the gap between the specification and the implementation, and they

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 66, December 16-18, 2025, Sapporo , Japan.

†Corresponding author

are unable to detect complex vulnerabilities arising from man-in-the-middle attack scenarios, such as triple handshake attacks [19]. The existing model-based testing approach is constrained by its reliance on a single formal specification tool rather than utilizing various formal specification tools. As a result, it becomes difficult to leverage predefined specifications tailored to different requirements across multiple tools. Furthermore, due to the limitations inherent in each tool, conducting a comprehensive range of model-based testing becomes challenging.

In our previous work [20], we proposed a scenario language to express the behavior of formal models and applied it to model-based testing of TLS implementations. By defining the scenario language for the Maude formal verification tool, we developed a prototype framework in which state sequences from the formal model are transformed into test programs and executed against real TLS libraries such as OpenSSL and WolfSSL. The scenario language consists of a meta-language, which abstracts general communication behaviors (e.g., `send`, `recv`), and an expression language, which can be tailored to specific domains such as TLS (e.g., generating a master secret). This design enables flexible applicability across distributed systems in different domains, and in the case of TLS, it allowed us to validate message ordering requirements specified in RFC 5246. However, while the advantages of this approach in bridging formal models and implementation testing are clear, it also faces critical limitations. In particular, generating scenarios that intentionally violate RFC specifications requires exploring an exponentially increasing collection of state sequences, where the search space expands combinatorially with the complexity of the model. This state space explosion problem makes the automated generation of meaningful test scenarios highly challenging and calls for more advanced techniques to manage the inherent combinatorial growth.

To address the state space explosion problem in generating scenario languages, we introduce a methodology that systematically restricts the exploration to only those paths relevant for producing testable scenarios. Specifically, we leverage Maude's strategy language to guide the exploration process, encoding strategies that represent targeted behaviors such as RFC violations. By expressing these strategies, we can precisely direct the search toward paths that are both meaningful and practically useful for generating scenario programs. This approach enables us to generate scenario language instances that focus on specific protocol deviations, thereby reducing unnecessary exploration and improving scalability. To evaluate the effectiveness of our method, we encoded 4 violation cases from RFC 5246 and 6 violation cases from RFC 8446, and successfully generated the corresponding scenario language programs automatically.

2 Preliminaries

TLS. The TLS protocol facilitates encrypted communication using symmetric and asymmetric encryption algorithms. Symmetric key algorithms employ a single key for both encryption and decryption, with AES and DES as notable examples. Asymmetric key algorithms utilize a public key and a private key, allowing messages encrypted with the public key to be decrypted using the private key; RSA, ECDHE, and DHE are examples of such algorithms. Encrypted communication via the TLS protocol is primarily divided into two stages: the key negotiation (handshake) phase, where encryption keys are exchanged between the server and client, and the data exchange phase, where encrypted data is transmitted based on the agreed-upon keys. During the key negotiation process, the server and client generate and share a symmetric key for encrypted communication using asymmetric methods based on the mutually agreed-upon encryption algorithm. The resulting symmetric key is then used to encrypt the data for communication. Various software implementations of the TLS protocol exist, such as OpenSSL and WolfSSL.

Maude. Maude is a high-level formal language based on rewriting logic, designed for specifying and verifying concurrent and distributed systems, including cryptographic protocols [21, 22, 23]. It models system states and transitions using equations and rewriting rules, enabling rigorous analysis of dynamic behaviors. Since unrestricted rewriting may introduce excessive nondeterminism, the Maude strategy language [24] provides explicit control over rule applications by separating the specification

```
// Test1: Send ciphersuite which is
          not included in ClientHello
accept(CI);
var v0 := recv(CI);
assert(v1.handshakeType == client-hello);
assert(v1.ciphers == TLS_AES256_CCM8_SHA256);
...

var v1 := serverHello(v0.TLS,
           TLS_AES128_CCM8_SHA128,
           random(SI, 0),sessionId(SI,1),
           no-compression,extensions);
send(v1, CI);
var v2 := recv(CI);
assert(v2.contentType == alert);
assert(v2.alertLevel == fatal);
assert(v2.alertDesc == illeegal-parameter);
close(CI)
// Test2: Send invalid compression
          algorithms in ClientHello
connect(SI);
var v0 := clientHello(TLS12,
           TLS_AES128_CCM8_SHA128,
           random(CI, 0),sessionId(CI,1),
           zlib-compression,extensions);

send(v0, SI);
var v1 := recv(SI);
assert(v1.contentType == alert);
assert(v1.alertLevel == fatal);
assert(v1.alertDesc == handshake-failure);
close(SI);
```

Figure 1: Abstract two test cases written by scenario language.

of rules from their execution policies. With constructs such as selective rule application, sequencing, nondeterministic choice, and iteration, strategies allow users to guide or constrain rewriting, prune irrelevant paths, and enforce goal-directed executions. This makes it possible to mitigate the state space explosion problem and to explore only those behaviors relevant to the analysis. In the context of our TLS model, strategies serve as an execution policy that selectively applies only those mutated rewrite rules relevant to RFC-violating behaviors. This allows the search to remain both focused and scalable by avoiding unnecessary exploration of standard, non-violating paths.

Scenario Language. The scenario language provides a formalized way to describe the behavior of each node in a network protocol. It consists of two parts: a meta-language and an expression language. The meta-language specifies generic protocol operations, such as sending and receiving messages, declaring and assigning variables, establishing or closing connections, and checking assertions. The expression language defines domain-specific constructs that capture the unique aspects of particular protocols. For TLS, this includes functions to construct handshake messages (e.g., `clientHello`, `serverHello`), generate cryptographic materials (e.g., master secrets, keys, random values), and perform cryptographic operations such as hashing or encryption. This layered design allows protocol behaviors to be expressed in a uniform format, while still accommodating domain-specific details needed for accurate testing. Figure 1 illustrates the two abstract test cases written by scenario language.

3 Maude Formal Model

In order to specify TLS software at the design level, we constructed a formal model using Maude's object-oriented specification framework. The system is described in terms of objects and rewrite rules, where objects correspond to software components. In particular, we defined four main classes: `Client`, `Server`, `User`, and `Link`. Each class encapsulates the attributes relevant to its role in the TLS handshake: for instance, protocol version, cipher suites, keys, buffers, and state variables for `Client` and `Server`; API-related attributes for `User`; and message transfer properties for `Link`. The overall TLS system is then represented as a multiset of interacting objects whose states evolve through rewrite rules, such as building messages.

As a concrete example, consider the construction of the `ServerHello` message. When a server initiates a connection, the rule `buildServerHello` increments the nonce counter, generates a fresh random value, and places a `ServerHello` containing the protocol version, agreed cipher suites, and nonce into the output buffer. The generic `sendMsg`/`recvMsg` rules then deliver this message over the

corresponding `Link` object to the client's input buffer.

```
rl [buildServerHello] :
  < SI : Server | connectState : V2S-INIT, nonceCtr : N, pVersion : PV, cipher : CSL,
                  hRandom : noNonce, peer : CI, outBuf : BUF > =>
  < SI : Server | connectState : V2S-CT-HLO, nonceCtr : s N,
                  hRandom : nonce(SI, N),
                  outBuf : BUF :: (serverHello(PV, CSL, nonce(SI,N)) to CI) > .
```

4 Encoding of Maude Strategies

In order to analyze the robustness of TLS implementations beyond the standard handshake behavior, we use the Maude formal model with a *strategy language*. The strategy language guides the rewriting engine to apply specific rules in some orders. We construct rewrite rules which is deviated from normal TLS server/client behaviors and use the strategy language to execute these rewrite rules. Instead of always executing the correct message construction rule (e.g., `buildServerHello`), the strategy language can instruct the system to invoke a modified version of the rule (e.g., `buildModifiedServerHello`) that produces malformed or unexpected messages. In this way, we can systematically explore protocol misbehaviors and verify whether the receiving party detects and responds to such violations in accordance with the RFC specification.

Modified Rule. To illustrate, consider the standard `buildServerHello` rule. We introduce a corresponding rule called `buildModifiedServerHello` that behaves similarly, but intentionally alters the cipher suite list before transmission. The modification is achieved by replacing the original cipher suite set with a value generated by the auxiliary operator `modifyCS`, which may add, replace, or delete entries. Auxiliary rules such as `modifyCS-add`, `modifyCS-replace`, and `modifyCS-delete` define the concrete mutation to be applied to the cipher suites. For example:

```
rl [buildModifiedServerHello] :
  < SI : Server | acceptState : V2S-GEN, nonceCtr : N, pVersion : PV, cipher : CSL,
                  hRandom : noNonce, peer : CI, outBuf : BUF >
=>
  < SI : Server | connectState : V2S-CT-HLO, nonceCtr : s N,
                  hRandom : nonce(SI, N),
                  outBuf : BUF ::
        (serverHello(PV, modifyCS(CSL, allCipherSuites), nonce(SI,N)) to CI) > .

rl [modifyCS-add]: modifyCS(CSL, (CSS, CS', CSS')) => CSL CS' .
rl [modifyCS-delete]: modifyCS(CSL CS, (CSS, CS, CSS')) => CSL .
rl [modifyCS-replace]: modifyCS(CSL CS CSL', (CSS, CS', CSS')) => CSL CS' CSL .
```

Encoding of Strategies. The above modifications are generalized in the strategy language through three operators: `add`, `replace`, and `delete`. These operators are declared as strategy actions that take a handshake type, a message content, and a target value as arguments:

```
ops add replace delete : HandshakeType MessageContent ContentValue -> StrategyAction .
```

With these operators, the strategy language can explicitly encode how the content of a handshake message should be manipulated. For instance, `replace(server-hello, cipherSuites, TLS_AES_128_GCM_SHA256)` are encoded as the following strategy, which forces the model to construct a `ServerHello` containing a cipher suite that was never proposed by the client, thereby violating the exepcted handshake behavior.

1	When a client first connects to a server, it is required to send the ClientHello as its first message.
	`replace(client-hello, handshakeType, server-hello)`
2	The This message (Server Certificate) will always immediately follow the ServerHello message.
	`replace(server-hello, handshakeType, client-hello)`
3	Recipients of Finished messages MUST verify that the contents are correct.
	`replace(finished, verifyData, nonce)`
4	The server MUST send a Certificate message whenever the agreed upon key exchagne method uses certificates for authentication
	`delete(certificate, entry, certificate-entry1)`
5	A client which receives a cipher suite that was not offered MUST abort the handshake with an "illegal_parameter" alert.
	`replace(server-hello, cipherSuite, TLS_AES_128_CCM_SHA128)`
6	In particular, MD5, SHA-224, and DSA MUST NOT be used
	`replace(server-hello, sigAlgo, MD5)`
7	Client MUST NOT offer multiple KeyShareEntry Values
	`add(client-hello, keyshareGroups, secp256r1)`
8	The server MUST NOT send a "psk_key_exchagne_modes" extension
	`add(server-hello, extensions, psk_key_exchange_modes)`
9	For every TLS 1.3 ClientHello, this vector (compression) MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method.
	`replace(client-hello, compression, zlib)`
10	All TLS 1.3 ServerHello messages MUST contain the "supported_versions" extension.
	`delete(server-hello, extension, supported-version)`

Table 1: Maude operations to check RFC specification violations

```
buildModifiedServerHello ; modifyCS-replace[CS' <- TLS_AES_128_GCM_SHA256]
```

5 Case Study

To systematically address the requirements derived from both RFC 5246 and RFC 8446, we utilized the *add*, *replace*, and *delete* operators defined in Section 4 to construct counterexample situations. Specifically, four requirements were extracted from RFC 5246 and six additional requirements from RFC 8446, forming the basis of our formalization. Each operator in Table 1 encodes a message-level modification that perturbs the protocol execution, such as inserting unexpected messages, altering expected ones, or omitting mandatory elements. Through these modifications, the rewriting paths is directed toward states that violate the RFC requirements.

Building upon these operators, we executed the Maude specification to generate test scenarios. As illustrated in Figure 1, the first test is instantiated by the fifth entry in Table 1, where the encoded strategy language—constructed through the *add*, *replace*, and *delete* operators—produces a path that is subsequently transformed into the corresponding scenario language. Similarly, the second test is generated by the ninth entry in the same table. These executions demonstrate how strategy-driven path exploration in Maude, grounded in the operator definitions of Section 4, seamlessly translates into executable scenario-based tests.

6 Conclusion

In this paper, we presented a systematic approach for generating executable test scenarios for the TLS protocol by leveraging Maude strategies. We present an encoding that simplifies the use of Maude Strategy, enabling the formalization of counterexample situations derived from the requirements of RFC 5246 and RFC 8446. Our case study confirmed that the method can effectively generate meaningful scenarios that reflect requirement violations. As future work, we plan to extend the encoding to cover a broader set of RFC violation cases, thereby allowing more comprehensive evaluation of TLS implementations.

Acknowledgments

This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation(IITP) grants funded by the Korea government(MSIT) (No. 2022-0-00103 and No. RS-2024-00439856).

References

[1] Eric Rescorla and Tim Dierks. The transport layer security (tls) protocol version 1.2 (no. rfc5246), 2008.

[2] Eric Rescorla. The transport layer security (tls) protocol version 1.3 (no. rfc8446), 2018.

[3] Ashutosh Satapathy and Jenila Livingston. A comprehensive survey on ssl/tls and their vulnerabilities. *Computer Applications*, 5:31–38, 2016.

[4] Shinichiro Matsuo, Kunihiko Miyazaki, Akira Otsuka, and David Basin. How to evaluate the security of real-life cryptographic protocols? the cases of iso/iec 29128 and cryptrec. In *Financial Cryptography and Data Security*, volume 12059, pages 182–194, 2010.

[5] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Symbolically analyzing security protocols using tamarin. In *ACM SIGLOG*, volume 4, pages 19–30, 2017.

[6] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, and P. Hankes Drielsma. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, volume 3576, pages 281–285, 2005.

[7] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *ACM SIGSAC*, pages 1492–1504, 2016.

[8] Max Ammann, Lucca Hirschi, and Steve Kremer. Dy fuzzing: Formal dolev-yao models meet cryptographic protocol fuzz testing. *IEEE Symposium on Security and Privacy*, 2024.

[9] Andreas Walz and Axel Sikora. Exploiting dissent: Towards fuzzing-based differential black-box testing of tls implementations. *IEEE Transaction on Dependable and Secure Computing*, 17:278–291, 2020.

[10] Jinsheng Ba, Marcel Bohme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. *USENIX Security Symposium*, pages 3255–3272, 2022.

[11] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. *USENIX Security Symposium*, pages 193–206, 2015.

[12] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jorg Schwenk. Tls-anvil: Adapting combinatorial testing for tls libraries. In *USENIX Security Symposium*, pages 215–232, 2022.

[13] Neel Mehta. heartbleed. https://heartbleed.com/, 2012.

[14] Bodo Moller, Thai Duong, and Krzysztof Kotowicz. Padding oracle on downgraded legacy encryption(poodle). https://nvd.nist.gov/vuln/detail/CVE-2014-3566, 2014.

[15] Zichao Zhang, Limin Jia, and Corina Pasareanu. Proinspector: Uncovering logical bugs in protocol implementations. In *European Symposium on Security and Privacy*, pages 617–632, 2024.

[16] Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailuet, and Tom Chothia. The closer you look, the more you learn: A grey-box approach to protocol state machine learning. In *ACM SIGSAC*, pages 2265–2278, 2022.

[17] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. *USENIX Security Symposium*, pages 192–206, 2015.

[18] Aina Toky Rasoamanana, Olivier Levillain, and Herve Debar. Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint tls stacks. In *Computer Security – ESORICS 2022*, volume 13556, pages 637–657, 2022.

[19] Karthikeyan Bhargavan, Autoine Delignat-Lavaud, Cedric Fournet, Alfredo Pironti, and Pierre-Y ves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. *IEEE Symposium on Security and Privacy*, pages 215–232, 2014.

[20] Jaehun Lee and Kyungmin Bae. A scenario language for model-based testing of tls, 2024.

[21] Raul Lopez-Rueda and Santiago Escobar. Canonical narrowing with irreducibility and smt constraints as a generic symbolic protocol analysis method. In *Rewriting Logic and Its Applications*, volume 13252 of *LNCS*, pages 45–64, 2022.

[22] Canh Minh Do, Adrian Riesco, Santiago Escobar, and Kazuhiro Ogata. Parallel maude-npa for cryptographic protocol analysis. In *Rewriting Logic and Its Applications*, volume 13252 of *LNCS*, pages 253–272, 2022.

[23] Santiago Escobar, Catherine Meadows, Jose Meseguer, and Sonia Santiago. State space reduction in the maude-nrl protocol analyzer. *Information and Computation*, 238:157–186, 2014.

[24] Steven Eker and Ruben Rubio Alberto Verdejo Narciso Marti Oliet, Jose Meseguer. The maude strategy language. *Journal of Logical and Algebraic Methods in Programming*, 134, 2023.