Formally Verifying the State Machine of TLS 1.3 Handshake in OpenSSL

Jingjing Guan[†], Hui Li[†‡], XiangDong Li[†], XiaoLei Wang[†], Binghan Wang[†],
Qiuye Wang[§], Shengchao Qin[¶], Mengda He[°], Md Armanuzzaman[⋆] and Ziming Zhao[⋆]
[†]Beijing University of Posts and Telecommunications, China [§]Zhejiang University, China
[¶]Xidian University, China [°]Teesside University, United Kingdom [⋆]Northeastern University, USA
Email: [†]{guaner, lihuill, lxd_dong, wxl904, biing}@bupt.edu.cn, [§]wangqyff@gmail.com,
[¶]shengchao.qin@gmail.com, [°]mengda.he@gmail.com, [⋆]{m.armanuzzaman, z.zhao}@northeastern.edu

IEEE INFOCOM 2025 - IEEE Conference on Computer Communications | 979-8-3315-4305-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/INFOCOM55648.2025.11044576

***Abstract*—The TLS handshake state machine manages the handshake messages exchanged in a session based on the parameters negotiated between the client and the server. Although the TLS 1.3 standard has undergone multiple rounds of analysis and revisions before official release to ensure its security, the process from natural language descriptions to implementations still relies on human expertise and is error-prone. In this paper, we propose a systematic method to conduct equivalence verification between the implementation of the TLS state machine and the standard. We also conduct formal verification of OpenSSL, the extensively utilized open-source TLS implementation for secure communications. Using Cryptol, we model the handshake state machine both from the standards and OpenSSL's implementation to perform its formal verification. Our automatic tool E-Verify performs equivalence verification by comparing the state transition sequences produced by the RFC model and OpenSSL model with all combinations of negotiation parameters. Guided by the verification results, we identify 640 mismatches out of a total of 1,536 scenarios and pinpoint the corresponding negotiation parameter combinations.**

***Index Terms*—Transport Layer Security, handshake state machine, SAW, Cryptol, formal methods**

I. Introduction

Network protocol implementations must conform to their standards to ensure interoperability between different systems and to avoid security vulnerabilities. This requirement is particularly vital for the Transport Layer Security (TLS) protocol [24], [25], [26], [43], which is widely used to provide secure communications on multiple platforms. The handshake protocol in TLS plays an important role in authentication, parameter negotiation, and shared keying material derivation. As a stateful security protocol, the TLS handshake protocol relies on a state machine to keep track of the protocol process and ultimately establish a secure channel with the parameters negotiated between the client and the server. The negotiation parameters of TLS handshake include the mandatory options, such as supported version, key exchange modes and symmetric encryption ciphersuites; and optional features, such as the client authentication modes, and whether to send key and initialization vector update notification. The combinations of the parameters control the sequence of messages exchanged in the handshake protocol and make the TLS handshake state machine more flexible but also more complex.

The OpenSSL stands out from various TLS implementations with its full-featured toolkit and standardized community managements and updates, which has been widely used in general-purpose cryptography and secure communications. Other TLS implementations such as Amazon's s2n [1], Mono used in Microsoft's .NET Framework [40], and wolfSSL (formally CyaSSL) used in MySQL [42] customize only part of the optional features in TLS standards according to the target scenarios, and their state machines are simpler.

Although the latest version, TLS 1.3, has learned lessons from the "attack-patch" revolution in the past years, and adopted rounds of analysis and revisions before official release [27], [34], [36], [21], [8], [7], [20], the subtle process from the natural language standards to the implementation in programming language remains dependent on the developer experience and thus error-prone. Deviations TLS handshake state machine implementations have caused serious attacks like Triple Handshakes [11], SKIP and FREAK [6], etc. However, the above attacks were discovered through manual inspection and fuzzing, which only cover part of the optional features and running traces in the TLS handshake state machines. Although effective, they are far from complete verification. Considering the popularity of OpenSSL, it is crucial to formally verify that its implementation of the TLS 1.3 handshake state machine aligns with the standard specifications.

Presently, there have been several attempts to formally verify the equivalence of TLS implementations with the standards, but most of the works only focused on verifying the implementation a single cipher suite in OpenSSL, such as SHA-3 [30], HMAC with SHA-256 [5] and pseudo random number generator [44]. Chudnov et al. [18] formalized the TLS 1.2 and 1.3 handshake state machine in Amazon's s2n implementations, providing equivalence proof with the standards. However, their formalization covered only a subset of the options TLS 1.3 handshake, and their oversimplified modeling reduced the accuracy of the verification results. Moreover, their coarse-grained verification checked only the message sequences generated by the state machines, rather than the running process of the handshake state machine, i.e.,

[‡]Hui Li is the corresponding author.

the state transition traces obtained by step-by-step execution of the state machine. The inaccurate modeling and coarse-grained verification above may allow potential vulnerabilities in the implementations of TLS handshake state machines to slip through the net.

Systematic formal verification of the equivalence between OpenSSL implementation of TLS 1.3 handshake state machine and the definitions in standards is challenging: (i) the description of TLS 1.3 handshake state machine is scattered in more than 160 pages of standards. Modeling the transitions within a single state requires taking into account multiple descriptions in the context; (ii) the implementation of OpenSSL TLS 1.3 handshake state machine takes nearly 20,000 lines of codes, which contains the transition between more than 50 read and write states. And each state transition function involves judgment on the values of more than 20 parameters; and (iii) the multiple options in TLS 1.3 handshake further increase the complexity of the handshake state machine. To improve the accuracy of verification results, both mandatory and optional steps should be included in the verification.

In this paper, we formally model the TLS 1.3 handshake state machine, covering all features defined in the standards. We also develop a set of tools that can automatically identify the mismatched state transitions in the OpenSSL implementations. The contributions of this paper are as follows:

(1) We present the formalization of the TLS 1.3 handshake state machine from the RFC standards with Cryptol [35]. Our RFC model covers all handshake modes and features defined in the standards;

(2) We present a Cryptol formal model of the TLS 1.3 handshake state machine of OpenSSL, which strikes a balance between simplifying nested function calls and preserving the complete state transition process;

(3) We verify the correctness of our OpenSSL model with SAW [16]. The proofs show that the OpenSSL model and the C implementation can produce the same state transitions for each negotiation parameter combination;

(4) We develop an automated tool E-Verify to perform equivalence verification between the RFC and the OpenSSL models, covering all negotiation parameter combinations;

(5) We discovered 640 mismatched state transition sequences out of 1,536 cases between the OpenSSL TLS 1.3 state machine implementations and the standard definitions;

(6) We propose concrete recommendations to fix the issues found in TLS 1.3 standards and OpenSSL implementations.

II. Background

In this section, we introduce the modeling language Cryptol and provide background knowledge of the verification tool SAW. We also discuss the major differences between the TLS 1.3 and the previous TLS 1.2 handshakes.

A. Cryptol

Although the well-known formal verification tools such as Tamarin [39], [4], ProVerif [12], [13], and NuXmv [17], [19] can automatically verify whether the protocol formalizations comply with expected security properties, they do not support the equivalence verification between two formal models. Therefore we resort to Cryptol, a programming language developed by Galois for the domain of cryptography [35], to formalize the TLS 1.3 handshake state machine in the OpenSSL C implementations and the standards. Cryptol has been successfully used to model the TLS handshake state machine in Amazon's s2n [18], the NIST SHA-3 compression function [30], and AES based CTR_DRBG [44].

The bit-precise type system of Cryptol supports a rich set of data types, including custom types with specified bit lengths, which is essential for accurately modeling TLS 1.3 elements such as message types and state identifiers. Users can also define structures composed of multiple types of data, such as the SSL struct in OpenSSL. The built-in type inference of Cryptol saves the developers from having to strictly specify monomorphic types for all expressions. Its size-polymorphic type system is well-suited for modeling the TLS 1.3 handshake, where the number of branches in each state is not fixed, and the total length of the state transition sequence is not the same for different combinations of negotiation parameters.

With the features above, Cryptol enables precise modeling of the logical relationships between states and tracks changes in negotiation parameters. Analysts can verify the correctness of the executable Cryptol model by observing the outputs with specific test vectors, or compare the outputs from different models to validate the functional equivalence.

B. Software Analysis Workbench

Software Analysis Workbench (SAW) is a toolkit for extracting formal representations from C and Java implementations, and verifying whether the representations meet function-level formal specifications. SAW has been successfully applied in verifying SHA-256 HMAC [5] and SHA-3 [30] implementations in OpenSSL, and the hash and equal function implementations in Java [41].

The jss and lss interfaces provided by SAW can extract formal representations called SAWCore from Java Virtual Machine (JVM) bytecode and a linked LLVM bitcode compiled with the LLVM clang compiler. Users can use the scripting language SAWScript to import the Cryptol formal model built from the implementations, which will be compiled directly into SAWCore, and specify the equivalence verification task between the implementation and the implementation model. SAW is closely integrated with Cryptol for equivalence verification, which is efficient for simple implementations but presents significant challenges for complex ones like OpenSSL TLS 1.3 handshake, resulting in lengthy verification processes. Considering large-scale code size and complex nested function calls in the OpenSSL TLS 1.3 handshake implementations, we build the Cryptol formal model for OpenSSL implementations as an intermediate representation and use SAW to verify the functional correctness of our formal modeling.

The verification tasks in SAWScript are defined as Hoare triples (*Precondition*, *Commands*, *Postcondition*). *Precondition* initializes the values and pointers in C implementations and

Cryptol models. And the *Commands* execute the implementations and Cryptol models respectively. *Postcondition* stores the final results, which generally includes the return value of the function and the changes of specified parameters. Various external theorem provers, including Z3 [22], Yices [28], abc [14], boolector [15], cvc4 [3] and cvc5 [2], can verify properties between the Cryptol models and the function implementations.

C. Updates in TLS 1.3 Handshake

Compared with TLS 1.2 and previous versions, the TLS 1.3 standards have made the following updates in the handshake protocol, making the handshake message and state machine structure significantly different from before: (i) RSA and static Diffie-Hellman key exchange cipher suites was removed; (ii) the round-trip time for basic full handshake was reduced from 2-RTT to 1-RTT, and added a 0-RTT handshake mode; (iii) renegotiation was forbidden, and any change in handshake parameters requires a new Basic full handshake; and (iv) session resumption was replaced by a new PSK handshake mode. The differences above make the TLS 1.3 handshake significantly different from previous versions.

III. The TLS 1.3 Handshake State Machine

In this section, we first discuss the negotiation parameters of the TLS 1.3 handshake state machine, followed by the discussions on the handshake message exchanged between the client and the server. In addition, we present the complete TLS 1.3 client and server state machine.

A. Neogotiation Parameters

The choice of TLS handshake state transition traces depends on the features negotiated between the client and server, which are recorded as the negotiation parameter value in the handshake state machine. We summarize all the 10 negotiation parameters that affect the operation of the state machine. The value of a certain parameter being set to `On` indicates this mechanism is enabled, otherwise it is disabled in this handshake. The impact of these negotiation parameters on the state transition is shown in Table I.

B. Handshake Messages

Figure 1 depicts the message flights of the TLS 1.3 handshake with all negotiated parameters enabled. When middlebox_compatibility_mode is enabled, both the client and server send `ChangeCipherSpec` after `ClientHello` and `ServerHello`. If the client provides unacceptable DHE or ECDHE groups in the previous `ClientHello`, hello_retry_request will be enabled. The server expects a new `ClientHello` with matched groups and sends a `ServerHello` message again. With psk_mode enabled, the PSK-based authentication skips the exchange of certificates and signatures, directly hashing the handshake context in `Finished`. Enabling key_update requires both parties to actively send `KeyUpdate` message to remind each other to update the sending cryptographic keys. If the server chooses to enable new_session_ticket, it sends `NewSessionTicket` messages to the client. The number of this type of messages is not specified in the specifications and is configured as needed in the implementations. As for post-handshake authentication, the server sends `CertificateRequest` message after the main handshake, and the client responds with a `Certificate` message. If the client chooses to send an empty certificate, that is, empty_cert is enabled, the `CertificateVerify` message, which contains the signature of the corresponding private key, will not be sent.

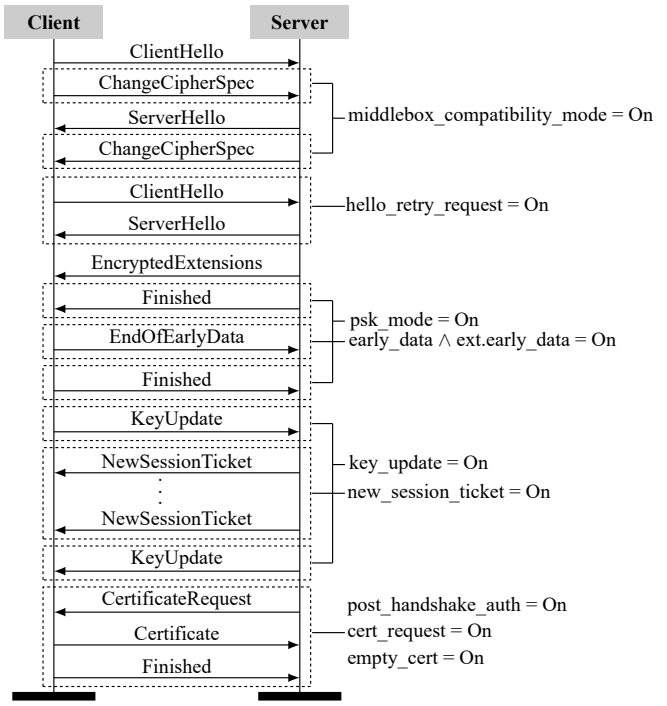


Fig. 1: TLS 1.3 handshake message sequence between the client and the server with all negotiation parameters enabled.

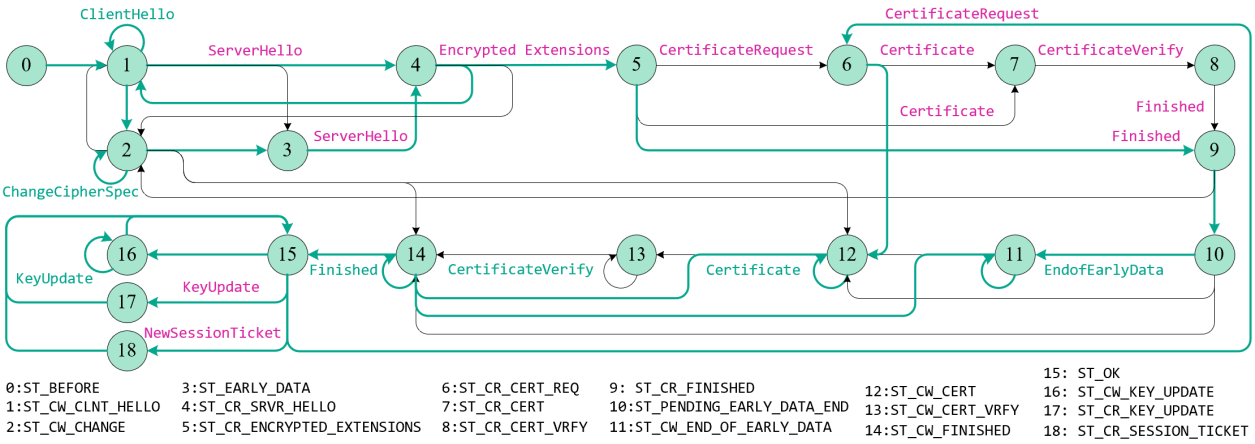
C. State Machine

The TLS 1.3 state machines of the client and server are shown in Figures 2a and 2b, respectively. Each state machine is a directed graph $\mathcal{D}$. The set of nodes $\mathcal{N}$ in the graph are the handshake states of client and server. The set of edges $\mathcal{E}$ between the nodes are valid state transitions in TLS 1.3 handshake, with the source being the starting state and the end being the successor state. The type of handshake message corresponding to a specific state transition is recorded as an attribute on this edge. The messages from the client are indicated in purple text, and those from the server are indicated in green text. For nodes with multiple outgoing edges, the specific edge selected at the execution is determined by the value of the negotiation parameter in the current state.

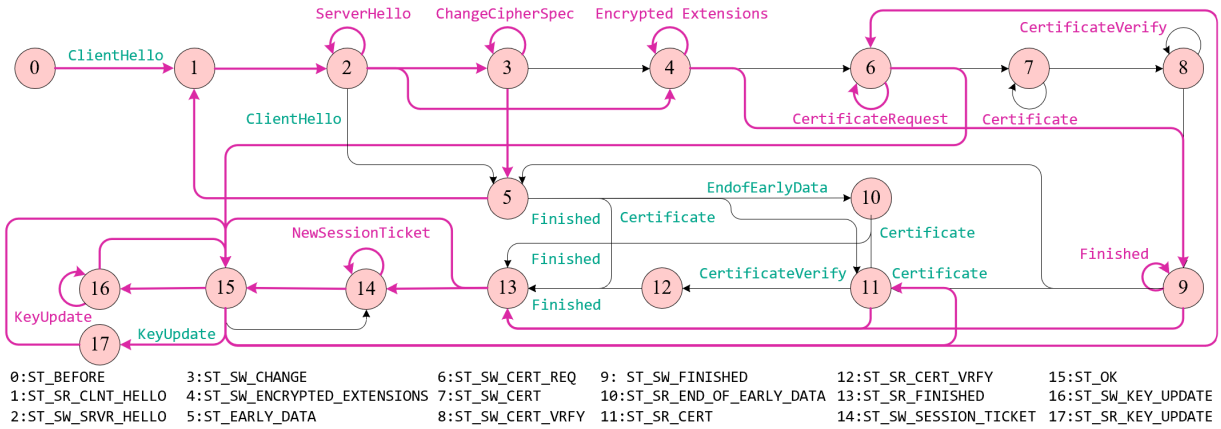
The execution of the TLS 1.3 state machine is given a specific value combination of the handshake negotiation parameters, starting from the initial handshake state `ST_BEFORE`, iteratively selecting valid edges from all the outgoing edges

TABLE I: TLS 1.3 Negotiation Parameters.

Full name	Description
hello_retry_request	For clients, if it receives `HelloRetryRequest` first, this field is On. Otherwise, this field is Off. For servers, it indicates whether to send a `HelloRetryRequest` to enable client retransmission of the `ClientHello`. If `HelloRetryRequest` is required, it is On. Otherwise, it is Off.
middlebox_compatibility_mode	Indicates whether the client and server need to send a `ChangeCipherSpec` for compatibility (although it is no longer a handshake message in TLS 1.3).
psk_mode	Indicates whether the client and server choose the PSK-based handshake mode, where the client and server do not have to exchange `Certificate` and `CertificateVerify`.
early_data	If the client and server choose to enable the transmission of 0-RTT early data, it is On. Otherwise it is Off.
ext.early_data	On the client side, this field is On if the server accepts 0-RTT early data and requires an `EndOFEarlyData`; otherwise, it is Off, and no `EndOFEarlyData` should be sent. On the server side, it indicates whether 0-RTT early data is accepted, determining if `EndOFEarlyData` from the client is expected.
cert_request	On the client side, this field is On if the `CertificateRequest` is received, triggering the sending of `Certificate`; otherwise, it is Off, indicating no client authentication is required. On the server side, this field is On if client authentication is required, prompting the sending of `CertificateRequest`; otherwise, it is Off, indicating no client certificate is required.
empty_cert	On the client side, this field is On if an empty `Certificate` is sent, omitting `CertificateVerify`; otherwise, it is Off, and the client sends its certificate and `CertificateVerify`. For the server, this field indicates whether an empty `Certificate` was received, determining if `CertificateVerify` is expected.
key_update	Indicates whether the client and server should send `KeyUpdate` to each other. If so, it is On; otherwise, it is Off.
post_handshake_auth	For the client, if it agrees to use the post-handshake client authentication mode, it is allowed to receive the `CertificateRequest` and send the `certificate` after the main handshake process. For the server, it indicates whether it accepts the choice of post-handshake client authentication mode. If so, it will send the `CertificateRequest` after the main handshake process ends.
new_session_ticket	For the server, it indicates whether to send `NewSessionTicket` to the client, which contains a token derived from the key material of this session. The token will be used in the derivation of PSK for subsequent sessions.



(a) The client state machine.



(b) The server state machine.

Fig. 2: TLS 1.3 state machines.

of the current state and finally reaching the ST_OK state. The above execution process completes the sending and receiving of handshake messages as expected on client and server side. And the sequence with all edges that have been gone through on the trace in order is the state transition sequence corresponding to this combination of handshake negotiation parameters. The bold edges in the figure are the state transition traces that the client and server experience when the negotiation parameters are all turned on.

IV. Verification Methodology

In this section, we first illustrate the methodology for verifying the equivalence between the state machine described by the standard and the state machine implemented by OpenSSL. Then, we present our modeling of the RFC state machine and the OpenSSL implementations, respectively. Finally, we introduce our automatic equivalence verification tool E-Verify.

A. Overview

Our verification strategy for the equivalence between OpenSSL implementation of TLS 1.3 handshake state machine and the descriptions in standards is shown in Figure 3. To bridge the gap between the natural language standards and the C code implementations, we formalize them into Cryptol models respectively. We extract the negotiation parameters controlling the execution of the state machine from the RFC standards and OpenSSL implementations to cover all the realistic scenarios. The equivalence between the RFC model and the standard is manually inspected. And SAW is used to automatically verify that the OpenSSL model and implementations generate the same state transitions. Our tool E-Verify conducts the equivalence verification between the RFC model and the OpenSSL model by comparing their state transition sequences generated with each set of negotiation parameter instantiations in an automated way.

B. Modeling of the Standards

The client and server TLS 1.3 state machines shown in Figures 2a and 2b can be modeled as directed graphs $\mathcal{D}_C$ and $\mathcal{D}_S$ shown in Figure 4. The internal structure of each node n in the graph of client and server TLS 1.3 state machines, i.e., a single handshake state, is shown in Figure 5. Each node is instantiated by the triple (*cur_id*, *cur_params*, *actions*), where *cur_id* identifies the current state, *cur_params* keeps the the current negotiation parameter values, and *actions* preserves the relationships between the current state and all potential successor states. Each action consists of two terms: a *condition*, which is a boolean expression about *cur_params*, indicating the conditions that need to be met for transitioning to each successor state; a state transition tuple (*cur_id*, *message*, *next_id*, *new_params*), which records the source state, the type of message received or sent in this transition, the identifier of the successor state, and the changes of negotiation parameters. The structure of a single state is shown in Figure 5.

With the definitions above, the initial state of the TLS handshake is instantiated by a given combination of negotiation parameter value, and the execution of the state machine can be modeled as a process of iteratively searching for and constructing successor states starting from the initial state. The search for the successor state is: calculating the condition in each action of the current state with the negotiation parameter *cur_params* of the current state, until the first action with a true condition is found. The successor state is then constructed with the *next_id* and *new_params* of this state transition tuple.

Unlike previous modeling of TLS handshake state machine, we add *cur_params* to keep the values of negotiation parameters in the current state, and involve the *new_params* to each action to save the changes of negotiation parameter values for each state transition. While enabling features such as post_handshake_auth and new_session_ticket in TLS 1.3, the traces will return to the state that has been experienced before (state backtracking) and resend the same type of message once or more times (state loop). With the negotiation parameter values in the state and the changes of parameter in each state transition, backtracking and loops can be accurately controlled in our modeling, making our modeling methodology can be used not only for the modeling of the TLS 1.3 handshake protocol, but also for the modeling of other complex protocol state machines.

C. Modeling of the OpenSSL Implementations

The OpenSSL implementations of client and server handshake state machines each maintain an SSL struct, which records the current state, the negotiation parameters of this handshake and the current read and write state msg_flow_state of the state machine. The handshake controller state_machine function determines the next state transition based on the value of msg_flow_state and calls the corresponding function. The function read_transition performs a read state transition operation and function write_transition performs a write state transition operation.

Both the read and write state transition functions are structured as a large switch-case statement. Each case corresponds to a single state, and the if-else statements within the case specify the allowed transitions and corresponding conditions. Once an if condition inside the case is hit, the function returns value_1 and changes the hand_state in the SSL structure to the next handshake state, indicating a successful state transition. Otherwise, the function returns value_0 indicating that no valid state transition is found, and the error handling process will be triggered later. For cases with overlapping state transitions and the same conditions, the fall-through feature of the C code can avoid repeated code blocks. The previous case without a break will automatically execute the operations within the following case. Since there are no built-in features switch-case and fall-through in Cryotol, these statements should be converted into if-then-else statements in Cryptol model, and the conditional blocks reused in the fall-through mechanism need to be restored to the corresponding case.

SAW verifies the equivalence between our formal model of the state transition functions and the C code implementations.

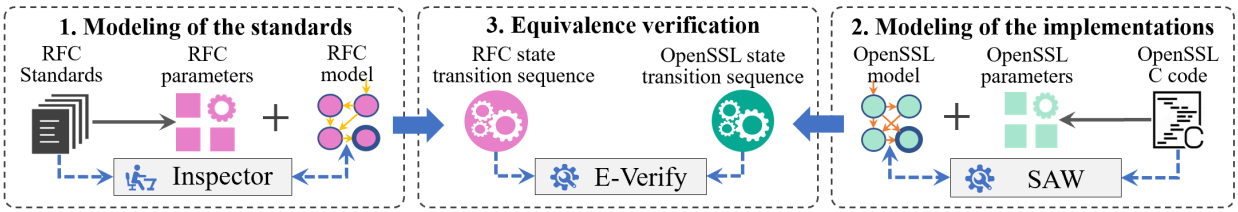


Fig. 3: Equivalence verification for the standards and the OpenSSL implementations of TLS 1.3 state machine.

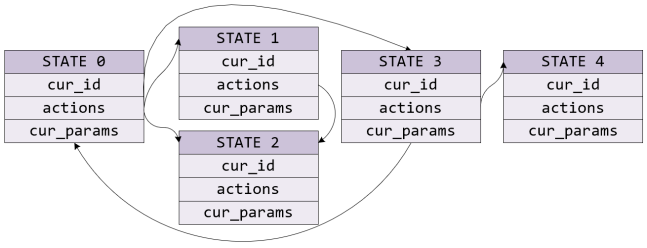


Fig. 4: Formal abstraction of state machine.

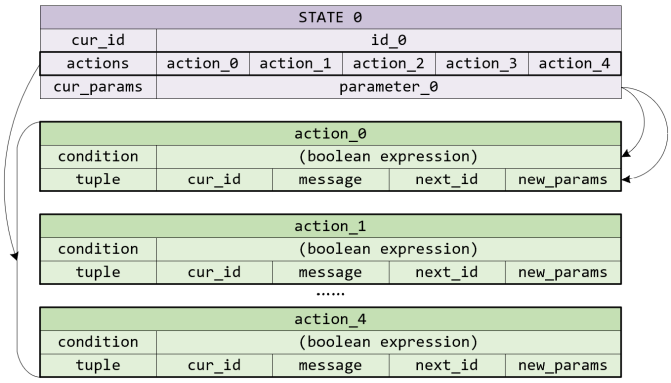


Fig. 5: The structure of a single state.

As shown in Figure 6, we import the LLVM bitcode of the state machine implementations generate by the Clang compiler and our OpenSSL model into SAW, converting both into SAWCore intermediate representation. SAW also accepts a SAWScript with verification tasks in the form of hoare triples. The number of tasks is determined by the number of enumeration parameter combinations involved in the function.

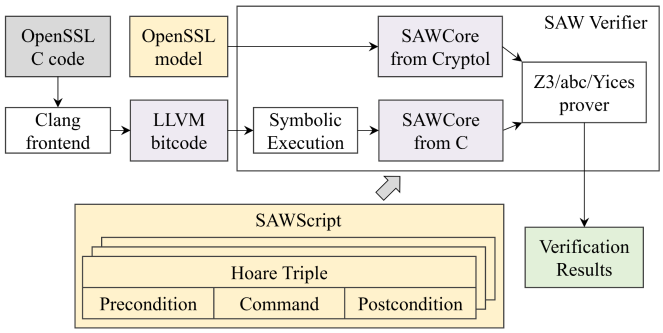


Fig. 6: SAW verification.

The `setup_ssl_statem` function instantiates the *Precondition*, including the definition of all parameters required for the C code and the OpenSSL model in Cryptol.

```
let setup_ssl_statem OHS SPS chosen_pha_null = do{
    s <- llvm_alloc (llvm_struct "struct.ssl_st");
    let hand_state = llvm_term {{ OHS }};
    llvm_points_to(llvm_field(llvm_field s "statem")"
        hand_state")(hand_state);
    s_hit <- llvm_fresh_var "hit" (llvm_int 32);
    llvm_precond{{ s_hit == (0 : [32]) \/ s_hit == (1 :
        [32])}};
    llvm_points_to(llvm_field s "hit")(llvm_term s_hit);
    return (s, {{ ssl_struct_for_cryptol_model }});
};
```

The *Commands* calls the C code SAWCore with `llvm_execute_func` instruction, and the Cryptol SAWCore is called in the `let` statement, specifying the function name `ossl_statem_client13_read_transition` in the OpenSSL Cryptol model.

```
llvm_execute_func [s,llvm_term mt];
let conn' = {{ossl_statem_client13_read_transition conn mt
    }};
```

The *Postcondition* captures the results of the above two commands. We not only capture the return value of the function to verify the execution status of the function (whether a valid state transition is hit); but also the changes of the parameter `hand_state` of this function to check whether the same state transition is generated.

```
llvm_points_to (llvm_field(llvm_field s "statem")"
    hand_state")(llvm_term{{conn'.1.statem.hand_state}});
llvm_return (llvm_term {{conn'.0}});
```

The `verify` command invokes an external prover to verify the equivalence between the SAWCore from OpenSSL model and C code. If the prover determines that the OpenSSL model and the OpenSSL C code produce equivalent results "Proof succeeded!" will be printed. Otherwise, the *Precondition* of the counterexample will be given. We traverse all enumeration types involved in function execution and call the verify command for each combination of the variables.

```
for mt (\mt ->
    for ossl_handshake_state (\ohs ->
        for ssl_pha_state (\sps ->
            clientread_spec mt ohs sps )));
```

D. Equivalence Verification between the Formal Models

Figure 7 shows our approach of equivalence verification between the RFC model and OpenSSL model. We instantiate both models with each combination of the negotiation parameter, and record the state transition sequences generated by the

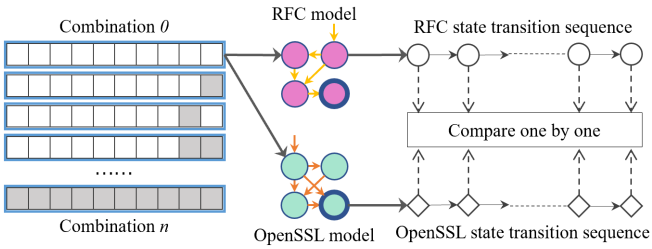


Fig. 7: Equivalence verification of RFC and OpenSSL models.

models respectively. The equivalence verification between the two models is converted into the comparison of state transition sequences. With the same negotiation parameter combination, matching state transition sequences means that the two models can produce the same execution results, while mismatches are counterexamples. These mismatches are further analyzed based on their locations to assess whether they could lead to potential security vulnerabilities.

The full set of negotiation parameters combinations and state transition sequences are denoted as $\mathcal{P}$ and $\mathcal{T}$. The handshake state machine formal model is abstracted as a mapping relationship $\mathcal{R}$ from $\mathcal{P}$ to $\mathcal{T}$, with $\mathcal{R}_1$ for the RFC model and $\mathcal{R}_2$ for the OpenSSL mode. The equivalence of RFC model and OpenSSL model can be converted into the equivalence proof of mapping relationships $\mathcal{R}_1$ and $\mathcal{R}_2$. That is: for each element p in $\mathcal{P}$, $\mathcal{R}_1(p) \in \mathcal{T}$, $\mathcal{R}_2(p) \in \mathcal{T}$ and $\mathcal{R}_1(p) = \mathcal{R}_2(p)$. For all combinations of negotiation parameters, if the formal models produce the same state transition sequences, indicating that the OpenSSL machine conforms to the standards. Otherwise, the values of negotiation parameters and the mismatched state transition sequences reveal the deviations in OpenSSL handshake state machine implementation.

Our tool E-Verify automatically generates all negotiation parameter combinations, instantiates the RFC and OpenSSL model templates with these combinations, and compares the state transition sequences between the formal models for each combination. E-Verify reads a configuration file defining the negotiation parameters and values, and the Cryptol model templates of the RFC standards and OpenSSL implementations. Decoupling parameter configurations from formal models facilitates the adding or removing and modifying the configuration items and the state machine operations subsequently.

(1) Parameter Combination Generator: There are 10 negotiation parameters in the TLS 1.3 handshake state machine, each with two options `On` and `Off`. As for the 0-RTT Early Data, `early_data_state` indicates whether the client and server negotiate to enable the transmission of 0-RTT Early Data, while `ext.early_data` indicates whether the server accepts the 0-RTT early data from the client. We exclude invalid cases of `early_data_state = Off` and `ext.early_data = On` that is, the Early Data mechanism is not enabled but the server receives and accepts the data. Therefore, there are $\frac{3}{4} \times 2^{10} = 768$ valid combinations of negotiation parameters in $\mathcal{P}$ and each combination corresponds to a scenario to be evaluated. Considering that the formal models of client and server state machines should be verified separately, the total evaluated scenarios are $768 \times 2 = 1,536$.

(2) Cryptol Model Instance Constructor and Executor: Each valid negotiation parameter combination will be converted into the variable assignment Cryptol code, and inserted into the templates of the formal models of the client and server state machine respectively to complete the instantiation of the Cryptol model. With the operations the above, all elements of the set $\mathcal{P}$ are provided as inputs for the RFC client and server formal models $\mathcal{R}_{RFC-C}$ and $\mathcal{R}_{RFC-S}$, as well as the OpenSSL formal models $\mathcal{R}_{OSSL-C}$ and $\mathcal{R}_{OSSL-S}$. These Cryptol models produce four groups, each with 768 results.

(3) Result Processing and Comparison: We convert the results of client formal models $\mathcal{R}_{RFC-C}$ and $\mathcal{R}_{OSSL-C}$, and server formal models $\mathcal{R}_{RFC-S}$ and $\mathcal{R}_{OSSL-S}$ with each combination of negotiation parameters into a list of state transition tuples, and compare the elements in the list orderly. If there are mismatches in the sequence, the negotiation parameter values and the first mismatched state identifier are recorded.

V. Results

In this section, we present the formal verification results of the OpenSSL TLS 1.3 handshake state machine implementations. Our verification is performed on a dell-Precision-7920-Tower with 80 Intel(R) Xeno(R) CPUs and 512G RAM Using the SAW tool with the w4_unint_z3 prover, the verification of the OpenSSL model is completed in 28 hours, while equivalence verification between the RFC and OpenSSL models takes less than 2 minutes.

A. SAW Results

We develop Cryptol formal models for read and write state machines functions of OpenSSL TLS 1.3 client and server, and verify the correctness of our modeling with SAW. Our proof shows that with the same *Precondition*, the formal models and C implementations of the four functions yield matching execution status (return value) and state transition (the state identifier) in *Postcondition*. Table II summarizes the cost of the verification in running time and LOCs for these functions.

The number of enumeration parameter combinations in the *Precondition* of each function determines the number of verification tasks in SAW. Our results indicate that more verification tasks lead to longer proof time. The read state function, which additionally evaluates handshake message types, requires more verification operations than the write state function. As for the server read state function, combinations of enumeration variables `early_data_state` and `post_handshake_auth` further increase verification operations, making the verification time tens of times more intensive than others.

B. Equivalence Verification Results

We formalize the client and server TLS 1.3 state machines as $\mathcal{R}_{RFC-C}$, $\mathcal{R}_{RFC-S}$, $\mathcal{R}_{OSSL-C}$, and $\mathcal{R}_{OSSL-S}$. Each model is instantiated with 768 valid negotiation parameters value combinations, expressed as logical formulas about the

TABLE II: Proof runtimes and code sizes of OpenSSL state machine functions.

Function	Total Time	Single Time	Total Ttasks	C LoC	Cryptol LoC	SAW LoC
client13_read	55min 24s	0.698s	4750	1011	92	72
client13_write	37min 17s	0.686s	3250	953	89	72
server13_read	26h 10min 7s	1.526s	61750	947	118	79
server13_write	37min 17s	0.686s	3250	1008	131	94

ten negotiation parameters. The equivalence verification between the RFC model and the OpenSSL model on the client side is performed by comparing the state transition sequences generated by $\mathcal{R}_{RFC-C}$ and $\mathcal{R}_{OSSL-C}$ with each value negotiation parameters combination. Similarly, The equivalence verification on the server side is to compare the state transition sequence generated by $\mathcal{R}_{RFC-S}$ and $\mathcal{R}_{OSSL-S}$. We record the logical formulas corresponding to the negotiation parameter combinations of mismatched state transition sequences, and merge the adjacent minimum terms to eliminate redundant factors to obtain the parameters related to the discrepancies. With the simplifications above, the discrepancies in the handshake state machine will fall into several different categories.

Table III summarizes the discrepancies in client and server and the corresponding negotiation parameter values. The RFC and OpenSSL models produce different state transition sequences for 192 parameter combinations on the client side and 448 parameter combinations on the client side. The remaining scenarios produce identical state transition sequences.

(1) Differences in Client Models: The RFC model and OpenSSL model behave differently on the client side when `psk_mode = On` and `new_session_ticket = On` in the negotiation parameters, that is, the client and the server negotiate to use the PSK-based handshake mode and agree with the use of the New Session Ticket mechanism. This mismatch affects $\frac{1}{4}$ of all the negotiation parameter combinations, that is, $\frac{1}{4} \times 768 = 192$ cases in total.

Enabling PSK-banse handshake mode allows the client and server to reuse the initial key material previously negotiated in a basic full handshake. If the client and server agree to use the `new_session_ticket` mechanism, at the end of handshake, the server sends the `NewSessionTicket` message to the client, which contains: 1) the nonce `ticket_nonce` for the derivation of the PSK used in subsequent handshakes; 2) the `ticket_lifetime` to limit the lifetime of this token, which can be up to 604800 seconds, about 7 days; and 3) `max_early_data_size` to define the maximum amount of 0-RTT data can be protected using this token.

However, the above restrictions in the standards only specify the lifetime and protection scope of a single ticket. The absence of limitations on the upper limit of the number of `NewSessionTicket` messages that a client can receive in a handshake may result in a Denial of Service attack on the client implementation. A malicious server can trick the client into establishing a TLS connection and make it agree to enable the `new_session_ticket` mechanism. And then the server sends a large number of `NewSessionTicket` messages in the handshake session. We check the source code of OpenSSL TLS 1.3 server implementation and find that the upper limit of the number of `NewSessionTicket` messages that can be sent is the maximum of an unsigned long integer.

(2) Differences in Server Models: There are two scenarios where the RFC and OpenSSL state machines behave differently. The first mismatch occurs in the cases with `psk_mode = On` and `new_session_ticket = On`, where the server allows the use of the `new_session_ticket` mechanism while performing a PSK-based handshake. The standard does not explicitly limit the number of `NewSessionTicket` messages that a server is allowed to send if the `new_session_ticket` mechanism is enabled again in a PSK-based handshake, where the initial key material has been reused. Different from the client, the server is the sender of the `NewSessionTicket` message. The lack of a limit on the number of `NewSessionTicket` messages sent in a PSK-based handshake does not affect the server's functionality and therefore does not pose a flaw in OpenSSL.

The second mismatch corresponds to the cases where `early_data = Off`, or `early_data = On` and `ext_early_data = Off` in the negotiation parameters. That is, the client and the server negotiate to turn off the 0-RTT Early data option, or they both agree to turn on 0-RTT Early data but the server refuses to accept the data due to reasons including parsing errors. In the server-side implementation of OpenSSL, for convenience, after receiving the `Finished` message from the client, the state machine will be uniformly transferred from `ST_SW_FINISHED` state to `ST_EARLY_DATA` state. However, according to the description in the standards, if `END_OF_EARLY_DATA` is not required, the state machine should keep the current state `ST_SW_FINISHED` to process the `CERTIFICATE` or `FINISHED` messages from the client, instead of transferring to the state `ST_EARLY_DATA`. For these affected scenarios, the OpenSSL server implementation goes through one more state transition, which may increase the time consumption of the handshake protocol process. However this mismatch does not affect the functionality of the OpenSSL TLS 1.3 handshake and will not cause security flaws.

C. Recommendations

We propose several concrete recommendations to enhance the security of the TLS handshake state machine.

(1) Explicit Definitions in Standards: We recommend the standards clearly limit the number of `NewSessionTicket` sent at once and reused within a connection. The ambiguous definition in the standards leads to unclear guidelines for state machine implementations. The `NewSessionTickets` issued by the server cryptographically ties the security context of the new connection to the original connection, but the

TABLE III: The scenarios where RFC model and OpenSSL model produce different state transitions.

Location	Negotiation Parameter Value	Number of Affected Scenes
Client	psk_mode = On ∧ new_session_ticket = On	192 (1/4 cases)
Server	psk_mode = On ∧ new_session_ticket = On	64
	early_data = Off ∨ (early_data = On ∧ ext.early_data = Off)	384 cases (128 overlap cases)

standards only restrict token lifetime to 7 days, which may be too lenient for high-security applications. We suggest adding restrictions on the number of `NewSessionTickets` issued at once and the total number issued per connection to avoid potential risks associated with unreasonably numerous tickets.

(2) Specific Limit of Loops in Implementations: We recommend the developers should carefully check the scenarios where the same type of message is repeatedly received or sent in the implementation of the state machine and set a limit for it. As the core component of controlling the order and logic of the handshake operation, it is responsible for the state machine to clearly define the upper limit of the number of loops, rather than leaving it to the application layer to decide. Although the resources and computing capabilities of the client are sufficient to support caching and processing thousands or even tens of thousands of handshake messages, in practical applications, it is possible to clearly define the upper limit of the number of messages such as NST in the state machine.

VI. Discussions and Limitations

Limitations: This paper acknowledges two limitations. Firstly, since there is no mature method to directly extract state machines from natural language and structured data in specifications, we resort to manual modeling of the state machine implementation, which is experience-dependent and time-consuming. Analysts have to repeat the process of manually establishing formal models for the verification of other state machine implementations. Considering that the specifications are stable in months or even years, the overhead of manual formal modeling is acceptable. Secondly, our RFC formal model is manually checked, as there are no explicit test vectors or test cases in TLS 1.3 specifications, and the expected operations of the state machine are implicit in the descriptions of processing different types of handshake messages.

Future Work: We aim to improve the efficiency for continuous verification during development and extend our methodology to verify other TLS state machine implementations. It is worthwhile to develop a tool to automatically generate formal models from the state machine implementations. And further research is required to accurately extract the descriptions of operations from the specifications and automatically convert them into test vectors for the state machine.

VII. Related Work

The design of handshake protocol went through multiple rounds of verification and improvement before its release [27], [34], [36], [21], [8], [7], [20], and the cipher suites in it were also formally verified [31], [33], [37]. There are also several efforts to formally verify whether the TLS 1.0 and 1.2 handshake implementations written in F# satisfy the expected security requirements in the standards [9], [10]. The IoT binary analysis system SAMBA [38] only focused on the implementations of insecure certificate verification and the deprecated SSL/TLS version calls. Although state machine fuzzing is effective in discovering vulnerabilities in the handshake [6], [23], [32], these works cover neither all misbehaviours nor all the features in TLS handshake state machines.

Equivalence verification between OpenSSL cipher suite implementations and the standards: Beringer et al. proved that the OpenSSL implementation of HMAC with SHA-256 complies with the FIPS functional specification and satisfies the expected cryptographic properties [5]. Hanson et al. verified that OpenSSL C implementation of SHA-3 matches a Cryptol specification derived from the FIPS 202 standard by NIST [30]. They only verified the implementation of some cipher suites in OpenSSL, not the handshake state machine.

Equivalence verification between OpenSSL state machine implementations and the standards: Chudnov et al. formally verified the equivalence between Amazon's s2n TLS 1.2 and 1.3 state machine implementations and the definitions in standards [18]. But the s2n TLS 1.3 state machine does not include the optional mechanisms such as post handshake client authentication, new session ticket and key update. And their SAW verification of the formal model from s2n implementations only covered valid state transitions. Fiterau-Brostean et al. offered an automated black-box technique to detect state machine bugs in SSH and DTLS implementations with a known catalogue of state machine bug patterns [29].

VIII. Conclusion

In this paper, we formalized the TLS 1.3 handshake state machine described in the standards and implemented in OpenSSL. The SAW proof showed our OpenSSL model is equivalent to the state transition functions in the actual C implementation. We generated all valid negotiation parameter combinations to cover all scenarios in the running process, where each set of negotiation parameters is a set of input to the RFC model and OpenSSL model. Guided by the verification results, we identified the mismatched state transitions between the OpenSSL state machine implementation and the expectations in standards with specific negotiation parameter combinations. Additionally, We proposed several concrete recommendations to improve the security of TLS 1.3 state machine implementations.

Acknowledgements

The research of Beijing University of Posts and Telecommunications was supported by the National Natural Science Foundation of China (Grant No. 62472045) and the BUPT Excellent Ph.D. Students Foundation (CX2023121).

REFERENCES

[1] I. Amazon.com, "s2n," 2017. [Online]. Available: https://github.com/aws/s2n-tls

[2] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A versatile and industrial-strength SMT solver," in *2022 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2022, pp. 415–442.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "cvc4," in *2011 International Conference on Computer Aided Verification (CAV)*. Springer, 2011, pp. 171–177.

[4] D. Basin, C. Cremers, J. Dreier, and R. Sasse, "Tamarin: verification of large-scale, real-world, cryptographic protocols," *IEEE Security & Privacy*, vol. 20, no. 3, pp. 24–32, 2022.

[5] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel, "Verified correctness and security of {OpenSSL}{HMAC}," in *2015 USENIX Security Symposium*, 2015, pp. 207–221.

[6] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2015, pp. 535–552.

[7] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017, pp. 483–502.

[8] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin, "Downgrade resilience in key-exchange protocols," in *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 506–525.

[9] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu, "Verified cryptographic implementations for tls," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–32, 2012.

[10] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing tls with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013, pp. 445–459.

[11] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over tls," in *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014, pp. 98–113.

[12] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pp. 82–96.

[13] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with lemmas, induction, fast subsumption, and much more," in *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 69–86.

[14] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *2010 International Conference on Computer Aided Verification (CAV)*. Springer, 2010, pp. 24–40.

[15] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *2009 International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2009, pp. 174–177.

[16] K. Carter, A. Foltzer, J. Hendrix, B. Huffman, and A. Tomb, "Saw: the software analysis workbench," in *2013 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*, 2013, pp. 15–18.

[17] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The NuXmv symbolic model checker," in *2014 International Conference on Computer Aided Verification (CAV)*. Springer, 2014, pp. 334–342.

[18] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran *et al.*, "Continuous formal verification of amazon s2n," in *2018 International Conference on Computer Aided Verification (CAV)*. Springer, 2018, pp. 430–446.

[19] A. Cimatti, A. Griggio, E. Magnago, M. Roveri, and S. Tonetta, "Extending nuxmv with timed transition systems and timed temporal properties," in *2019 International Conference on Computer Aided Verification (CAV)*. Springer, 2019, pp. 376–386.

[20] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. Van Der Merwe, "A comprehensive symbolic analysis of tlsTLS 1.3," in *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1773–1788.

[21] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 470–485.

[22] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *2008 International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[23] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *2015 USENIX Security Symposium*, 2015, pp. 193–206.

[24] T. Dierks and C. Allen, "RFC 2246: The transport layer security (TLS) protocol version 1.0," 1999.

[25] T. Dierks and E. Rescorla, "RFC 4346: The transport layer security (TLS) protocol version 1.1," 2006.

[26] ——, "RFC 5246: The transport layer security (TLS) protocol version 1.2," 2008.

[27] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 1197–1210.

[28] B. Dutertre, "Yices 2.2," in *2014 International Conference on Computer Aided Verification (CAV)*. Springer, 2014, pp. 737–744.

[29] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations," in *2023 Network and Distributed Systems Security Symposium (NDSS)*, 2023.

[30] P. Hanson, B. Winters, E. Mercer, and B. Decker, "Verifying the sha-3 implementation from openssl with the software analysis workbench," in *2022 International Symposium on Model Checking Software (SPIN)*. Springer, 2022, pp. 97–113.

[31] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of tls-dhe in the standard model," in *2012 Annual Cryptology Conference (CRYPTO)*. Springer, 2012, pp. 273–293.

[32] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation," in *2015 USENIX Security Symposium*, 2015, pp. 223–238.

[33] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the tls protocol: A systematic analysis," in *2013 Annual Cryptology Conference (CRYPTO)*. Springer, 2013, pp. 429–448.

[34] H. Krawczyk and H. Wee, "The OPTLS protocol and TLS 1.3," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 81–96.

[35] J. Lewis, "Cryptol: specification, implementation and verification of high-grade cryptographic applications," in *2007 ACM Workshop on Formal Methods in Security Engineering (FMSE)*, 2007, pp. 41–41.

[36] X. Li, J. Xu, Z. Zhang, D. Feng, and H. Hu, "Multiple handshakes security of TLS 1.3 candidates," in *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 486–505.

[37] Y. Li, S. Schäge, Z. Yang, F. Kohlar, and J. Schwenk, "On the security of the pre-shared key ciphersuites of tls," in *Public-Key Cryptography–PKC 2014: 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings 17*. Springer, 2014, pp. 669–684.

[38] K. Liu, M. Yang, Z. Ling, Y. Zhang, C. Lei, L. Luo, and X. Fu, "SAMBA: Detecting SSL/TLS API misuses in IoT binary applications," in *2024 International Conference on Computer Communications (INFOCOM)*, 2024.

[39] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The Tamarin prover for the symbolic analysis of security protocols," in *2013 International Conference on Computer Aided Verification (CAV)*. Springer, 2013, pp. 696–701.

[40] Microsoft, "Mono," 2017. [Online]. Available: https://www.mono-project.com/

[41] K. Okano, S. Harauchi, T. Sekizawa, S. Ogata, and S. Nakajima, "Consistency checking between java equals and hashcode methods using software analysis workbench," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 8, pp. 1498–1505, 2019.

[42] T. Ouska, "wolfssl," 2006. [Online]. Available: https://www.wolfssl.com/

[43] E. Rescorla, "RFC 8446: The transport layer security (TLS) protocol version 1.3," 2018.

[44] D. Selvakumar, J. Mervin, S. Pattanshetty, and D. Vivian, "Formal verification and analysis of a pseudo random number generator," in *2021 International Symposium on VLSI Design and Test (VDAT)*. IEEE, 2021, pp. 1–6.